A GENETIC ALGORITHM APPROACH FOR CONSTRUCTING RAILROAD OPERATING PLANS

By

HONGJIANG WU

Master of Science

Oklahoma State University

Stillwater, Oklahoma

1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1996

A GENETIC ALGORITHM APPROACH FOR

CONSTRUCTING RAILROAD

OPERATING PLANS

Thesis Approved:

J. Chandler

Thesis Adviser

Blayne E. Mayfield

H. Lu

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Dr. John P. Chandler, my principal adviser, for his guidance, dedication, patience, and invaluable instructions. He made a great deal of effort to improve this thesis, both in content and in English. My appreciation is also extended to Dr. H. Lu, Dr. K. M. George, and Dr. B. E. Mayfield, the other committee members, for their helpful advisement and suggestions.

I would also like to thank to the staff of OSU Library and Department of Computer Science for their cordial assistance.

I am grateful to my parents Ding-An Wu and Xiang-Ying Liang, for their consistent support and encouragement. My deep thanks are also dedicated to my wife, Kelly, my sisters, Hongwei and Hongli, and brother, Liping, for their support and understanding.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

During the last thirty years there has been a growing interest in problem-solving systems based on principles of evolution and heredity: such systems maintain a population of potential solutions, they have some selection processes based on fitness of individuals, and they evolve according to some "genetic" operators [Michalewicz, 1]. The typical representatives of these systems are the Genetic Algorithms (GAs, or GA for Genetic Algorithm) that have been developed by John Holland, his colleagues, and his students at the University of Michigan [Goldberg, 2].

Genetic algorithms are adaptive search algorithms based on the principles and mechanics of natural selection and natural genetics, 'survival of the fittest' from natural evolution [Goldberg, 2]. They operate as iterative procedures on a fixed size population or pool of candidate solutions. The candidate solutions represent an encoding of the problem into a form that is analogous to the chromosomes of biological systems. Each chromosome represents a possible solution for a given objective function. Associated with each chromosome is a fitness value, which is found by evaluating the chromosome with the objective function. It is the fitness of a chromosome which determines its ability to survive and produce offspring. Each chromosome is made up of a string of genes (whose values are called alleles). The chromosome is typically represented in the GA as a string of bits. However, integers and floating point numbers can easily be used [Corcoran, 10], but we will use a binary matrix as our chromosome, where each element of the matrix is a gene, to represent each potential solution of an ROP (Railroad Operating Plan) instance (see CHAPTER II).

The traditional search algorithms mainly fall into three categories: calculus-based, enumerative, and random.

The current main calculus-based search methods assume a smooth search space and the existence of its derivatives, and most of them use the gradient-following technique (also called hill-climbing). It comes as no surprise that methods depending upon the restrictive requirements of continuity and derivative existence are unsuitable for all but a very limited problem domain [Goldberg, 2].

Many enumerative schemes start searching by looking at objective function values at every point in the space, one at a time. This is attractive when the search space is finite. But many practical spaces are simply too large to search one at a time [Goldberg, 2] (Note: not all enumerative schemes look at every point, for instance, alpha-beta pruning.).

Random search algorithms start searching by random walking. In the long run, they can be expected to do no better than enumerative schemes [Goldberg, 2] (in some cases, they may be much faster).

A genetic algorithm (GA) is different from conventional optimization methods in several ways: A genetic algorithm is a parallel and global search that searches multiple points by maintaining and evolving the population, so it is more likely to obtain the global solution. The advantage of a GA is that of genetic diversity: at each step in the search process a set of candidates, instead of a single candidate, is considered, and more candidates are simultaneously involved in the creation of new candidates [Eiben, 11]. A genetic algorithm makes no assumption on the search space and only requires the payoff values (objective function values) associated with individual chromosomes, so it is simple and can be applied to a wide class of problems [Jeong, 3]. GAs belong to the class of probabilistic algorithms, yet, they are very different from random algorithms as they combine elements of directed and stochastic search. GAs use probabilistic transition rules (not just a simple random walk) as a tool to guide a search toward regions of the search space with likely improvement. Because of this, GAs are also more robust than many directed search methods, and more efficient than existing stochastic search methods [Michalewicz, 1].

The object in a railroad operating plan problem is to optimize the configuration of the network structure of a railroad system according to the conditions of traffic flow and cost. It belongs to the class of combinatorial optimization problems. Methods to solve difficult combinatorial problems can be divided into two types. The first type includes those methods which try to find optimal solutions through an 'intelligent' exhaustive search. This includes techniques such as backtracking, branch and bound, implicit enumeration, and dynamic programming. Such techniques are only useful for solving combinatorial problems with small sizes. The other type of method for solving combinatorial problems relies on approximate optimization. That is, rather than finding the absolute optimal solution, a 'good' solution is desired within an acceptable time period. These are known as combinatorial optimization techniques. These methods usually employ heuristic algorithms which are problem-specific. The true optimal solution is often unknown and impossible to determine.

Combinatorial optimization methods have as their goal the minimization or maximization of a function. They are composed of three parts. First, there is a set of problem instances. Second, for each problem instance, there is a finite set of candidate solutions. Finally, there is a function which assigns to each instance and candidate solution a positive real number called the solution value for the candidate solution. Notice how these elements correspond to those found in genetic algorithms. For a particular problem instance, the GA maintains a set of candidate solutions which are evaluated by the problem-specific evaluation function. The solution value returned by the function is used by the GA to measure the relative fitness of that candidate solution. This information is used with the idea of 'survival of the fittest' to conduct the genetic search. As a result GAs are very successful in finding good near-optimal solutions for combinatorial optimization problems [Corcoran, 10]. In this paper we will discuss the implementation of the genetic algorithms integrated with the simulated annealing algorithms (SA), which is used to approach the optimal solution of a railroad operating plan, and will analyze the performance of the implementation based on the results.

This thesis is organized as follows:

In Chapter I, a general introduction to the problem and algorithms we are going to implement and investigate is given.

In Chapter II, a brief review will be given of genetic algorithm basic concepts (selection, crossover, mutation, replacement), simulated annealing algorithm basic concepts, and a description of the railroad operating plan problem.

Chapter III will be dedicated to the study of pure genetic algorithms, and various hybrid modes of genetic algorithms and simulated annealing algorithms.

In Chapter VI, we will give the results of the test runs of the algorithms, and make some conclusions on the comparison of performance of these algorithms for approaching the optimal solutions of one railroad operating plan instance.

Chapter V, conclusions and recommendations.

Finally, the source program which implemented the hybrid algorithms and the minimization cost routines for the railroad operating plan will be put into Appendix A.

CHAPTER II

BACKGROUND

Basic Concepts of Genetic Algorithms

A genetic algorithm is a probabilistic algorithm that maintains a *population* (also called *generation* or *pool* sometimes) of *individuals* (also called *chromosome* sometimes), $P(t) = \{x^t_1, \ldots, x^t_n\}$ for each *iteration* (sometimes called *generation*) t. Each individual represents a potential solution to the problem at hand, and, in any genetic algorithm, is implemented as some (possibly complex) data structure S. Each candidate solution x^t_i is evaluated to give some measure of its "fitness". Then, a new population (or pool, or generation) (iteration $t+1$) is formed by selecting the more fit individuals (this is the selection step). Some members of the new population (pool, generation) undergo transformations (this is the alteration step) by means of "genetic" operators to form new solutions (individuals, chromosomes). The "genetic" operators are referred to as *selection, crossover, mutation,* and *replacement* operators. Mutation m is an unary transformation that creates a new individual by a small change in a single individual (m: S $\rightarrow$ S); crossover c is a higher order transformation that creates new individuals by combining parts from several (two or more) individuals (c: S $\times \cdots \times$ S $\rightarrow$ S); and replacement r is an unary transformation that carries out some policies to replace one old individual with a newborn individual (see Figure 2.1.). We usually define the genetic operators by incorporating the constraints and the specific knowledge of the problems. These three operators evolve the population from generation to generation. After some number of generations (iterations) the program usually converges. It is hoped that the best individual represents a near-optimum solution.

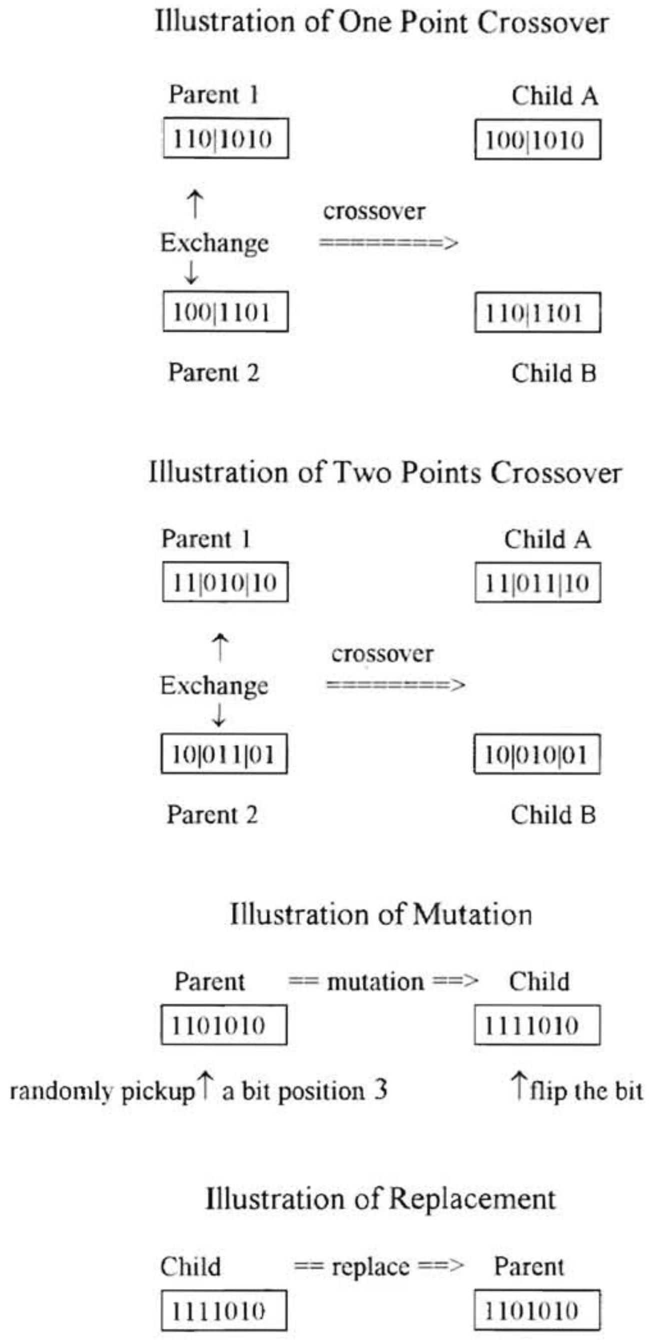


Figure 2.1 Illustrations of Genetic Operators

The structure of a genetic algorithm is shown in Figure 2.2.

```
procedure GA

begin
    t = 0;
    *initialize* P(t);
    *evaluate* structures in P(t);
    while termination condition not satisfied do
    begin
        t = t+1;
        P(t) = *select* from P(t-1);
        *alter* structures in P(t);
        *evaluate* structures in P(t);
    end
end.
```

Figure 2.2 Genetic Algorithm

Figure 2.2 illustrates a 'canonical' genetic algorithm. The GA begins by generating an initial population (pool), $P(t = 0)$, and evaluating each of its members with the objective function. While the termination condition is not satisfied (reaching the maximal iteration limit, or the convergence of the pool), a portion (some chromosomes) of the population (pool) is probabilistically selected according to the emulation of natural selection rules (uniform random, fitness biased, rank biased selections, etc.) to participate in the competition for crossover (mating), and some successfully win the chance to crossover to generate offspring of the next generation according to the crossover rate (there are numerous ways to implement the crossover operator and refine the probability of the crossover rate, as we will discuss in Chapter III), some mutation operations are applied to offspring according to the mutation rate (the mutation operator insures against a bit loss and can be a source of new bits or diversity. Since mutation is a random walk through the search space, it must be used sparingly [Jeong, 3]), the offspring are placed back into pool by applying a replacement operator according to various replacement policies; that is, perhaps replacing other members of the pool or simply appending them to the pool. The new pool (also called population or generation) is generated by finishing one iteration.

In this paper we have implemented three models of pure genetic algorithm: traditional model [Michalewicz, 1;Goldberg, 2], generational model [Holland, 4], and steady-state model [Whitley & Kauth, 5], and we have also incorporated with a simulated annealing algorithm to result in a variety of hybrid modes of a SAGA algorithm [Adler, 6].

Traditional GA model: it maintains two pools at each iteration. In one iteration, first, it selects candidates from the old pool to form a *reproduction pool* probabilistically by using a selection operator; second, it picks parents from the *reproduction pool* to form a temporary *mating pool* probabilistically according to the crossover rate; third, it randomly pairs two parents chosen from the *mating pool* and carries out a crossover operation to generate two offspring; after all new offspring have been born, and places them into the *reproduction pool*; finally, it mutates the *reproduction pool*; the new generation (pool, or population) of this iteration is formed in the *reproduction pool*.

Generational GA model: it maintains two pools at each iteration. In one iteration, it selects two parents from the old pool probabilistically by using a selection operator, crosses these two parents to generate two children according to a given probability, mutates the two children according to a given probability, then appends the two children to the new pool (*reproduction pool*) until it is full up to the pool size. The new generation of this iteration is formed in the new pool (*reproduction pool*).

Steady-state GA model: it maintains only one pool at each iteration. In one iteration, it only generates two offspring; that is, selects two parents from the old pool probabilistically by using the selection operator, crosses these two parents to generate two children according to a given probability, mutates the two children according to a given probability, then places the two children into the old pool (current generation) to form the new generation (old pool) of this iteration.

Basic Concepts of Simulated Annealing

Simulated Annealing (SA) is another algorithm that is popular in heuristic optimization. SA belongs to a class of algorithms called probabilistic hill-climbing algorithms that dynamically alter the probability of accepting inferior solutions. The SA algorithm is especially popular in the field of VLSI design where it has been successfully applied to the optimization of extremely high-dimensional problems such as placement and global routing of interconnect layers in VLSI chips which contain tens or hundreds of thousands of parameters to be optimized [Adler, 6].

A simulated annealing algorithm might be specified as follows:

$\mathrm{SA} = (P(t),\ \tau,\ f_{eval},\ accept)$,

where $P(t)$ is the t^{th} generation, $\tau = T_0, T_1, \ldots, T_k$ is the *annealing schedule* (or called *temperature*) which may be problem-specified in order for SA to achieve the best performance for each specific application, or may also be simply viewed as the function of the generation number t (in this paper we set $T_t = f(t) = 1/t$ if there is no maximal iteration limit for SAGA processing; $T_t =$ *maximally allowed iteration* - t otherwise; where t is the current iteration number), f_{eval} is the evaluation function which is identical to the evaluation function used by the GA to find the relative fitness of a solution, and *accept* is the accepting function of SA which gives the adapted probability of accepting a new-born child.

The structure of the adaptive simulated annealing algorithm is shown in Figure 2.3.

```
procedure SA:

T = $T_0$;
for(i=0; i<=k; i++)
{  repeat
   {  for each s in P(t)
      {   s' = reproduce(s, T);
          if(accept(s', s, T)) P(t+1) ← s';
          else P(t+1) ← s;
      }
      P(t) ← P(t+1);
   } until "local convergence" (or "iteration limit reached")
   T = adapt($T_i$);
}
```

Figure 2.3. Simulated Annealing Algorithm.

Where in Figure 2.3 the *reproduce* function in SA is the mutation or perturbation operation of a solution point *s*, it aims to search around *s* at temperature *T* for better solutions, it is a problem-specific heuristic. The most important, and intuitively appealing, aspect of the SA algorithms is the conditional acceptance function, where the probability of accepting an inferior solution *s'* over *s* is given by: $exp(-\Delta f / T)$, where *T* is the current temperature (annealing schedule) and Δf is the difference between $f_{eval}(s')$ *and* $f_{eval}(s)$ in the uphill direction (for a global minimum problem). This allows the SA algorithm to escape from local extrema at the early stages of the search, and to hill-climb efficiently as the temperature approaches zero [Adler, 6].

Description of Railroad Operating Plans

Railroad operating plans consider how to send freight cars from terminals to terminals within the railroad network efficiently. In this thesis, only single direction problems are considered. There is a fixed cost associated with operating a direct train, and there is a transfer cost for a car to change trains at intermediate terminals. If every pair of terminals is provided with a direct train, then all cars could be sent to their destinations without changing trains. In this case, however, the fixed cost is very high. If only every pair of adjacent terminals is provided with a direct train, all cars could also be sent to their destinations and the fixed cost is minimized. But notice that every car being sent to a destination beyond the adjacent terminal has to change trains at each intermediate terminals, so the transfer cost is very significant. Thus, an operating plan needs to decide which pairs of terminals are to be provided with direct trains, but also to give a scheme that shows how cars change trains at intermediate terminals through the available configuration of direct trains.

Finding an optimal operating plan is a very important problem in the railroad industry. Many researchers have developed various methods to solve the problem. In 1988, the problem was modeled as a network with fixed cost, and an optimal solution for a 12-terminal problem was reported [Shi, 15]. Guangping Lei, an OSU mathematics master's degree student, reported an optimal solution in her thesis by applying the Compressing Branch and Bound Method for up to 14-terminal problems in 1992. Here I try to use genetic algorithms to approach the best possible solutions of the problem by realizing data structure representation and genetic operators, tuning, and analyzing the effect of the system parameters on the performance of the algorithm.

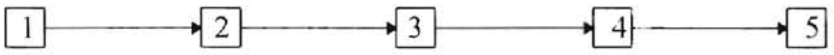
Mathematical Model

Consider a directed network $(\boldsymbol{N},\boldsymbol{B}_0)$, consisting of a set of nodes $N = \{1,2,\cdots,n\}$ and a maximal connect set of directed arcs $\boldsymbol{B}_0 = \{a_{ij} \mid i < j,\ i, j \in N\}$ (called a maximal feasible arc set). With each arc a_{ij}, we associate three known numbers c_i, t_j and f_{ij} which are fixed cost, length and flow of arc a_{ij} respectively. Let $\boldsymbol{A}_0 = \{ a_{i,i+1} \mid i = 1,2,\cdots,n - 1\}$ be the least connected set of directed arcs (called the least feasible arc set) (see Figure 2.4). Let $\boldsymbol{E}(\boldsymbol{A},\boldsymbol{B}) = \{ \boldsymbol{D} \mid \boldsymbol{A} \subseteq \boldsymbol{D} \subseteq \boldsymbol{B}\}$, where $\boldsymbol{A}_0 \subseteq \boldsymbol{A} \subseteq \boldsymbol{B} \subseteq \boldsymbol{B}_0$. Any element $\boldsymbol{D}$ in $\boldsymbol{E}(\boldsymbol{A}_0,\boldsymbol{B}_0)$ is called a feasible arc set. For any feasible arc set $\boldsymbol{D}$, in the subnetwork $(N, \boldsymbol{D})$, there is always a path connecting any two points i and j (since $\boldsymbol{A}_0 \subseteq \boldsymbol{D}$). Denote $\boldsymbol{L}_{ij}(\boldsymbol{D})$ as the arc set forming the shortest path from i to j in $(N, \boldsymbol{D})$, and $\boldsymbol{l}_{ij}(\boldsymbol{D})$ as the length of the shortest path, i.e.

$$\boldsymbol{l}_{ij}(\boldsymbol{D}) = \sum_{a_{uv} \in L_{ij}(\mathbf{D})} t_v$$

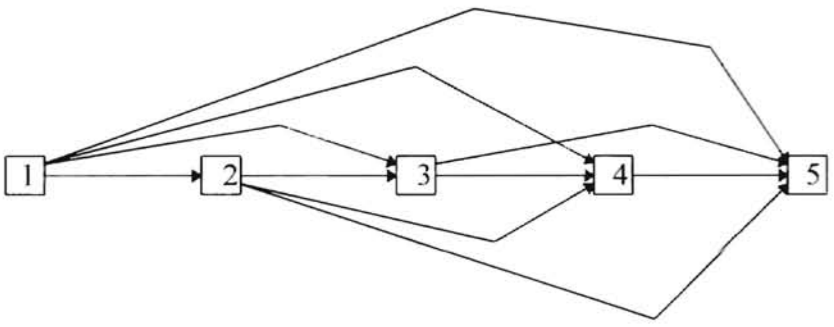
In this model, the set of nodes N corresponds to the set of n terminals. A directed arc a_{ij} corresponds to a direct train from terminal i to terminal j. The set of directed arcs $\boldsymbol{A}_0$ corresponds to the set of direct trains connecting adjacent terminals. The set of directed arcs $\boldsymbol{B}_0$ corresponds to the set of all possible direct trains. A feasible arc set $\boldsymbol{D}$ corresponds to a set of direct trains or a feasible direct train scheme which can send all freight cars to their destinations since $\boldsymbol{A}_0 \subseteq \boldsymbol{D} \subseteq \boldsymbol{B}_0$. Moreover, c_i is the fixed cost to operate one direct train at terminal i. t_j is the transfer cost for each car to change trains at intermediate terminal j. f_{ij} is the number of cars to be sent from terminal i to terminal j. $\boldsymbol{l}_{ij}(\boldsymbol{D})$ is the least transfer cost to send one car from i to j.

Illustration of $\boldsymbol{A}_0$



The least feasible arc set $\boldsymbol{A}_0$ with 5 nodes

Illustration of $\boldsymbol{B}_0$



The maximal feasible arc set $\boldsymbol{B}_0$ with 5 nodes

Figure 2.4.

Under a feasible direct train scheme associated with $\boldsymbol{D}$, the total fixed cost to send all freight cars to their destinations is

$$\sum_{a_{ij} \in \mathbf{D}} c_i$$

the least transfer cost to send f_{ij} cars from i to j is $f_{ij}\boldsymbol{l}_{ij}(\boldsymbol{D})$, so the total least transfer cost to send all freight cars to their destinations is

$$\sum_{a_{ij}\in\mathbf{B}_0} f_{ij} l_{ij}(\mathbf{D})$$

The total cost for a feasible set $\boldsymbol{D}$ is

$$\sum_{a_{ij}\in\mathbf{D}} c_i \quad + \quad \sum_{a_{ij}\in\mathbf{B}_0} f_{ij} l_{ij}(\mathbf{D})$$

As arcs a_{ij} are added to the set $\boldsymbol{D}$, the transfer cost decreases, but the fixed cost increases. Similarly, as arcs a_{ij} are subtracted from the set $\boldsymbol{D}$, the fixed cost decreases, but the transfer cost increases. So we need to trade between them. Our job is to find a feasible direct train scheme that minimizes the total cost. that is to find $\boldsymbol{D}^* \in \boldsymbol{E}(\boldsymbol{A}_0,\boldsymbol{B}_0)$ such that

$$\sum_{a_{ij}\in\mathbf{D}^*} c_i \; + \; \sum_{a_{ij}\in\mathbf{B}_0} f_{ij} l_{ij}(\mathbf{D}^*) = \min_{D\in\mathrm{E}(\mathrm{A}_0,\mathrm{B}_0)} \left\{ \sum_{a_{ij}\in\mathbf{D}} c_i \; + \; \sum_{a_{ij}\in\mathbf{B}_0} f_{ij} l_{ij}(\mathbf{D}) \right\}.$$

Finding $\boldsymbol{D}^*$, we get an optimal direct train scheme; and finding $\{ \boldsymbol{L}_{ij}(\boldsymbol{D}^*) \}$, the set of shortest paths between any two nodes in $(\boldsymbol{N}, \boldsymbol{D}^*)$, we get an optimal transfer scheme. The optimal direct train scheme along with the optimal transfer scheme constitutes the optimal railroad operating plan.

CHAPTER III

IMPLEMENTATION

Chromosome Representation

Genetic Algorithms (GA) are a new approach to solving many optimization problems. It approaches the optimal solution by maintaining and evolving a population (pool or generation) of legal potential solutions of the problem. It evolves the population by applying bias in selecting the parents, according to the evaluation of the fitness of the solutions, and then applying the genetic operators: crossover and mutation randomly, in such a way as to produce the next generation. Several things play a vital role in the algorithm.

First of all is the representation of the potential solution (called a chromosome) of the problem. In the classic genetic algorithm the potential solution of the problem is usually represented by a binary bit string, as Goldberg [2] pointed out: "the binary alphabet offers the maximum number of schemata per bit of information of any coding." Unfortunately, it isn't suitable for us to code all problems in binary strings due to the great diversity of problems. As Michalewicz [1] pointed out: "Classical genetic algorithms, which operate on binary strings, require a modification of an original problem into appropriate (suitable for GA) form; this would include mapping between potential solutions and binary representation, taking care of decoders or repair algorithms, etc. This is not usually an easy task." Therefore, in recent years, many researchers have attempted to select the data structure of the problem according to the specific structure of the problems. Different kinds of data structure representations: binary string, integer string, matrix, appear suitable for different problems.

We use a Boolean matrix to represent the chromosome, a feasible arc set $\boldsymbol{D}$ of the problem; it is also a potential solution of problem. The dimension of the matrix is the number of the nodes in the network $(\boldsymbol{N},\boldsymbol{B}_0)$ (the number of stations of railroad system), and the elements d_{ij} of the matrix $\boldsymbol{D}$ are defined as:

$$d_{ij} = \begin{cases} 1 & \textit{if there is an arc from } i \textit{ to } j \ ; \\ 0 & \textit{otherwise.} \end{cases}$$

So, the matrix of the arc set $\boldsymbol{A}_0$ is in the form of, for example,

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and the matrix of the arc set $\boldsymbol{B}_0$ is in the form of, for instance,

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Generally, the chromosome representation will be a strictly upper-triangular matrix for a railroad scheduling problem.

Evaluation of Fitness

Here the fitness is the total cost. In order to find the least total cost of the feasible arc set ***D***, the key is to find the least total transfer cost, that is, the length of the shortest path of each pair of nodes in the subnetwork (*N*, ***D***) should be known. We have built a subroutine named EV_fun() in the file "main.c" to do that and to store the values in corresponding cells of a two-dimensional array named "LTC" and defined in the "main.c" file. The biased selection of parents is based on the value of fitness, and there are several policies which we can adopt in biased selection. For example, we can choose the best fit, or above average, or we can even choose some bad chromosomes for the good of the diversity of the population.

Data Structures

We have defined and maintained three data structures in the implementation of the SAGA algorithm: structures of chromosome, pool, and GA information center (also called the GA control center).

The structure of a chromosome denoted as "chrom" in the implementation is defined in the header file "gah.h" and maintained by the routines of the file "chrom.c". The structure contains the chromosome gene matrix and chromosome-related data such as the matrix dimension, denoted as "dim", the fitness, the percentage of total fitness of pool (generation), denoted as "ptf", which is used by the fitness-biased selection operator, and a weighted value of sorted rank denoted as "rank_prob", one kind of scaling of percentage of total fitness, which is used by the rank-biased selection operator, and the chromosome' pool index denoted "index", parents indices denoted as "parent1, parent2", etc.. The segment of related code is shown below.

```
/*--- A chromosome ---*/
typedef struct
{   GENE_PTR* gene;        /*the address of the 1st gene of chrom. matrix*/
    int     dim;           /*the dimension of the chromosome matrix*/
    double fitness;        /*the fitness of the chromosome*/
    float   ptf;           /*the percentage of the total fitness of pool*/
    float   rank_prob;     /*the weighted value of sorted rank for rank-biased selection*/
    int     index;         /*my index in the current pool chromosome array*/
    int     parent1,parent2;  /*the parents indices used for identifying and replacing*/
} CHROM_TYPE, *CHROM_PTR;
```

The structure of the pool (generation), denoted as "pool" in the implementation, is defined in the header file "gah.h" and maintained by routines of the file "pool.c". The structure contains the array of chromosomes which represents the population of chromosomes of the pool, and pool-related data such as the maximally-allowed pool size denoted as "max_size", the pool actual size denoted as "size", the total fitness of the pool denoted as "tot_fitness", the average fitness of the pool denoted as "ave", the minimal fitness of the pool denoted as "min", the maximal fitness of the pool denoted as "max", the best fitness of the pool denoted as "best", the variance and standard deviation of the pool denoted as "var" and "dev", and two pool status flags to tell if the current pool is sorted (by the "sorted" flag) (sort pool means: sequence chromosome array of the pool from best (0th cell) to worst (array tail) according to the fitness of the chromosomes) and updated (by the "updated" flag"), etc.. The segment of related code is shown below.

OKLAHOMA STATE UNIVERSITY

```
/*--- A pool ---*/
typedef struct
{   CHROM_PTR* chrom;        /*the address of an array of chromosomes*/
    int    max_size;         /*maximally allowed pool size (must be at least 4)*/
    int    size;             /*actual pool size(must be at most max_size-2)*/
    double tot_fitness;      /*the total fitness of the pool*/
    double min, max, ave;    /*current pool fitness states*/
    double var, dev;         /*variance and devariance of fitness of pool*/
    int best_index;          /*index of the best chromosome*/
    int sorted;              /*flag to tell if pool is sorted or not after last change*/
    int updated:             /*flag to tell if pool stats is updated after last change*/
} POOL_TYPE, *POOL_PTR;
```

The structure of the GA information center (also called the GA control center), denoted as "ga_center" in the implementation, is defined in the header file "gah.h" and maintained by routines of the file "gac.c". The structure contains the two pools: the old pool denoted as "old_pool", being used for the old generation; the new pool denoted as "new_pool", being used for the new generation reproduced from the old pool. It also contains the history list array of the best chromosome of each generation during the GA processing denoted as "best"; and all system control (or configuration) parameters such as the random seed, denoted as "rand_seed", the pool maximally-allowable size denoted as "pool_max_size", the chromosome matrix dimension denoted as "chrom_dim", the maximally-allowed iterations for GA to run denoted as "max_iter", the rank-biased selection pressure denoted as "bias" (used to compute the weighted rank value "rank_prob" of each chromosome of the pool, and the rank_prob is used by rank-biased selection), the generation gap denoted as "gap", the crossover rate denoted as "x_rate", the mutation rate denoted as "mu_rate", the invoking SAGA's SAM probability denoted as "sam", the invoking SAGA's SAC probability denoted as "sac"; several policy flags such as the selection elitism flag denoted as "se_elitist" (0: disable; 1: two copies of the

OKLAHOMA STATE UNIVERSITY

best; 2: transfer ELITIST percent of top bests), the replacement elitism flag denoted as "re_elitist", and the mutation method flag denoted as "mu_flag"; and all GA operators registered for running such as the GA function operator "GA_fun", the selection operator "SE_fun", the crossover operator "X_fun", the mutation operator "MU_fun", the replace operator "RE_fun"; several run time counters and flags to tell the system status such as the iteration counter "iter", the mutation number counter "num_mu", and the convergence flag "converged"; etc.. The segment of related code is shown below.

```
/*--- GA control (configuration info) center ---*/
typedef struct              /*cfg means the value is given at config time*/
{  /*--- Basic info ---*/   /*run means the value is maintained in run time*/
   int  rand_seed;          /*cfg: seed for random number generator*/
   int pool_max_size;        /*cfg: pool max. allowable size, at least 4.*/
   int chrom_dim;            /*cfg: chrom. matrix dimension*/
   int max_iter;             /*cfg: max. allowable number of iteration for ga*/
   int iter;                 /*run: counter value of actual number of iteration*/
   int converged;           /*run: has ga converged? 1/0 TRUE/FALSE */
   int se_elitist;           /*cfg: selection elitism flag:
   int re_elitist;           /*cfg: replacement elitism flag:0/1: dis/enable*/
   float gap;                /*cfg: generation gap used by traditional. generational GA*/
   float bias;              /*cfg: rank-biased pressure, used  to compute rank_prob of chrom.*/
   float x_rate;             /*cfg: crossover rate*/
   float mu_rate;            /*cfg: mutation rate,it has two differ kind of value: see appendix A*/
   int mu_flag;              /*cfg: mutation selection method flag: see appendix A*/
   float sam;                 /*cfg: probability to invoke the SA mutation  operator (SAM) */
   float sac;                /*cfg: probability to invoke the SA crossover operator (SAC) */
   /*--- GA operators ---*/
   FN_PTR   GA_fun;      /*cfg: ga function*/
   FN_PTR   SE_fun;      /*cfg: selection operator*/
   FN_PTR   X_fun;       /*cfg: crossover operator*/
   FN_PTR   MU_fun;      /*cfg: mutation operator*/
   FN_PTR   RE_fun;      /*cfg: replacement operator*/
   FN_PTR   EV_fun;      /*cfg: evaluation function*/
   /*--- Pools ---*/
   POOL_PTR old_pool;   /*run: used for old generation*/
   POOL_PTR new_pool;   /*run: used for producing new generation*/
   /*--- Stats ---*/
   CHROM_PTR* best;      /*run: the array of best history list of chrom. of each generation*/
   int num_mut, tot_mut;    /*run: counter value of actural and total  number of mutation*/
}GA_CENTER_TYPE, *GA_CENTER_PTR;
```

GA Operators

Basically, one *single basic GA operation* consists of four different types of *atomic genetic operations: selection, crossover, mutation, and replacement,* and generates and places back two (or sometime only one) newborn children. One *GA iteration* consists of several (sometimes only one, for instance, in a steady-state GA) *single basic GA operations* and generates the next new generation. One *GA run* consists of several *GA iterations* and evolves to an approximation to the best possible generation of the chromosomes. Here the core part is the four different types of *atomic genetic operations*. We have implemented them in the four different files "select.c", "cross.c", "mutation.c", and "replace.c".

The *"selection"* operator, to carry out the first *atomic genetic operation*, is implemented in three different ways in the file "select.c".

Random selection: select one chromosome from the pool randomly.

Fitness-biased selection: select one chromosome from the pool by using standard roulette and fitness-biased values (the "ptf" value, see Appendix A). According to the definition of the "ptf" of a chromosome in the pool: when the optimization is to minimize fitness, then the chromosome with smaller fitness will have bigger "ptf", so it will have a bigger probability to be selected by fitness-biased selection. When the optimization is to maximize fitness, then the chromosome with bigger fitness will have a bigger "ptf", so it will have a bigger probability to be selected by fitness-biased selection.

Rank-biased selection: select one chromosome from the pool by using standard roulette and rank-biased (rank-prob) values (the weighted value of "rank_prob" is computed based on the rank-biased selection pressure "bias" and the rank index of the chromosome in the sorted pool; see Appendix A). According to the definition of "rank_prob" of a chromosome in the sorted pool, the chromosome with smaller pool index will have a bigger value of "rank_prob", so it will have a bigger probability to be selected by rank-biased selection; the chromosome with bigger pool index will have a smaller value of "rank_prob", so it will have a smaller probability to be selected by rank-biased selection.

The choice of one out of three above to be registered with the GA control center structure (denoted as "ga_center") is specified by the configuration file (named "gaccfg.h") (see Appendix A) which is provided and modified according to the user need. We also implement the *selection* interface "seRunPool()" to stand between the GA function operator and the GA control center registered *selection* operator (see Figure 3.1)

```
procedure seRunPool()
begin
    obtain valid pool index range [lo,hi] from current pool;
    call GA registered selection operator to select a chromosome from the valid range;
    return;
end
```

Figure 3.1. Procedure of the Selection Interface "seRunPool()"

The *"crossover"* operator, to carry out the second *atomic genetic operation*, is implemented in six different ways in the file "cross.c":

Crossover by 1 cross point: cross two parents to produce two children by using 1 crossover point.

OKLAHOMA STATE UNIVERSITY

Crossover by 2 cross points: cross two parents to produce two children by using 2 crossover points.

Crossover by 3 cross points: cross two parents to produce two children by using 3 crossover points.

Crossover by 4 cross points: cross two parents to produce two children by using 4 crossover points.

Crossover by random flip: cross two parents to produce two children by copying each gene from a parent based on a random flip of a fair coin.

Crossover by asexual reproduction: to produce one child from one parent by swapping blocks of the parent.

The choice of one out of six above to be registered with the GA control center structure (denoted "ga_center") is specified by the configuration file (named "gaccfg.h") which is provided and modified according to the user need. We also implement two *crossover* interfaces "xRunPair()" and "xRunPool()" to stand between the GA function operator and the GA control center registered *crossover* operator (see Figure 3.2, 3.3).

procedure xRunPair()

```
xRunPair(p1,p2,c1,c2)
{  initialize child chromosomes c1, c2 and record parents p1, p2 index;
   if (random probability <= x_rate)
       c1,c2 = crossover(p1,p2) by GA registered crossover operator;
   else
   {  c1=p1;
      c2=p2;
   }
}
```

Figure 3.2 Procedure of the Crossover Interface "xRunPair()"

The *crossover* interface "xRunPair()" is designed to call the GA control center registered *crossover* operator to cross two passed-in parents to generate two children if a randomly generated probability meets the *crossover* rate. The two children are put in global variables "child1", and "child2" and passed out. This is used by generational and steady-state GA, and called by generational and steady-state GA's *single basic GA operation*: function "ga_SE_X_MU_RE_2CHILD()" (which selects, then crosses and mutates two parents from the old pool to generate two children, and replaces the two children back to the new pool) or function "ga_SE_X_RE_2CHILD()" (which selects, then crosses two parents from old pool to generate two children, and replaces the two children back to the new pool); both of these are implemented in the file "ga.c".

The *crossover* interface "xRunPool()" is designed to be used by a traditional GA, called within a traditional GA iteration loop after the new pool (*reproduction pool*) has been selected by applying the GA control center registered-selection operator. We select the *mating pool* from the new pool according to the *crossover* rate. All new pool chromosomes have an equal right to participate in the competition of being parents; that is, for each chromosome we examine the randomly generated probability against the *crossover* rate. If it is no more than the *crossover* rate, that chromosome will be put into the *mating pool* and wait for a random pairing later. After the *mating pool* is selected (it must contain an even number of chromosomes), we do random pairing and call the GA control center registered *crossover* operator to do crossover by randomly selecting crossover point positions (the number of cross positions in each crossover is determined by the GA control center registered crossover operator) to produce an array of newly born children who reside in the array "child", which is passed in as a parameter. It is possible for the *mating pool* to be empty if the crossover rate is relatively small; then there is no newborn child. But We choose to "go" back to re-select the *mating pool* until it is not empty. This can be easily changed later for different trials according to needs.

procedure xRunPool()

```
xRunPool(pool, child)
{  mating pool = randomly select from pool according to x_rate;
   for(; mating pool is not empty;)
   {  p1, p2 = randomly pairing parents from mating pool;
      c1, c2 = GA registered crossover operator ( p1,p2);
      copy c1.c2 into child array;
   }
}
```

Figure 3.3 Procedure of the Crossover Interface "xRunPool()"

The "*mutation*" operator, to carry out the third *atomic genetic operation,* is implemented in three different ways and handles three different policies (the mutation method flag "mu_flag" is configured in the GA control center "ga_center") in the file "mutation.c".

The three mutation operators:

Mutation by flip bit: randomly choose a bit to do a flip mutation for a chromosome chosen by the mutation interface "muRunChromLevel()" based on the mutation rate, or a chromosome passed into the mutation interface "muRunSingleChrom()" and which met the mutation rate by checking a randomly generated probability.

Mutation by random bit: randomly choose a bit to do a random flip for a chromosome chosen by the mutation interface "muRunChromLevel()" based on the mutation rate, or a chromosome passed into the mutation interface "muRunSingleChrom()" and which met the mutation rate by checking a randomly generated probability.

Mutation by swapping two genes: randomly choose two rows to do a swap mutation for a chromosome matrix chosen by the mutation interface "muRunChromLevel()" based on the mutation rate, or a matrix passed into the mutation interface "muRunSingleChrom()" and which met the mutation rate by checking a randomly generated probability.

The three mutation policies:

Mutate child chromosome immediately: ("mu_flag" = "MU_CHILD" in the GA control center "ga_center") Only the mutation interface "muRunSingleChrom()" can be called and mutaten a bit of that newborn child immediately; the mutation rate "mu_rate" in the GA control center "ga_center" is the probability of that child to undergo a bit mutation within it (mutation interface "muRunSingleChrom()" to check the mutation rate and invoke the GA control center "ga_center" registered mutation operator to do a mutation for that child).

Mutate pool on the chromosome level: ("mu_flag" = "MU_CHROM" in the GA control center "ga_center") Only the mutation interface "muRunPool()" can be called (it in turn calls the mutation interface "muRunChromLevel()"); mutation selection is on the chromosome level of the whole pool; the mutation rate "mu_rate" in the GA control center "ga_center" is the probability for a chromosome of the pool to undergo a bit mutation within it (the mutation interface "muRunChromLevel()"to check the mutation rate and invoke the GA control center "ga_center" registered mutation operator to do a mutation for every chromosome member of the pool).

Mutate pool on the bit level: ("mu_flag" = "MU_BIT" in the GA control center "ga_center") Only the mutation interface "muRunPool()" can be called (it in turn calls the mutation interface "muRunBitLevel()"); mutation selection is on the bit level of the whole pool; the mutation rate "mu_rate" in the GA control center "ga_center" is the probability for a bit of the pool to undergo a bit mutation (mutation interface "muRunBitLevel()" to check the mutation rate and invoke the GA control center "ga_center" registered mutation operator to do a mutation for every bit of the pool).

The choice of one out of three *mutation* operators above to be registered with the GA control center structure (denoted "ga_center"), and the one out of three mutation flags, are specified by the configuration file (named "gaccfg.h" see Appendix A) which is provided and modified according to user needs. We also implement four mutation interfaces "muRunChromLevel()", "muRunBitLevel()", "muRunPool()", and "muRunSingleChrom()" to stand between the GA function operator and the GA control center "ga_center" registered mutation operator (see Figure 3.4, 3.5, 3.6. 3.7).

```
procedure muRunChromLevel()

muRunChromLevel(pool)
{   for each chromosome s of pool
    {  if (randomly generated probability <= mu_rate)
           s' = GA registered mutation operator (s );
       else
           s' = s;
       repair(s');
       s=s';
    }
}
```

Figure 3.4 Procedure of the Mutation interface "muRunChromLevel()"

The mutation interface "muRunChromLevel()" is designed as a submanager to do mutations on a whole pool by selecting chromosomes from the whole pool. For each chromosome of the whole pool the interface is randomly tested to see if it can perform a mutation on the chromosome based on the mutation rate; if it can, it will call the GA control center "ga_center" registered mutation operator to do a mutation on that chromosome.

```
procedure muRunBitLevel()

muRunBitLevel(pool)
{   for each bit  of pool
    {  if (randomly generated probability <= mu_rate)
       {  computing bit position to find host chromosome s of that bit.
          s'= mutation s by  flipping that bit;
          repair(s');
          s=s';
       }
       else {}
    }
}
```

Figure 3.5 Procedure of the Mutation interface "muRunBitLevel()"

The mutation interface "muRunBitLevel()" is designed as a submanager to do mutations on a whole pool by selecting bits from the whole pool. For each bit of the whole pool the interface is randomly tested to see if it can perform mutation on that bit based on the mutation rate; if it can, it will flip that bit. It doesn't use the GA control center "ga_center" registered mutation operator.

The mutation interface "muRunPool()" is designed as a manager to do mutation on a whole pool by calling the submanager, either "muRunChromLevel()" or "muRunBitLevel()", according to the value of the mutation flag "mu_flag": either

"MU_CHROM" or "MU_BIT". Note: outside routines (i.e. GA function operaor) can only call the interface "muRunPool()" to perform mutation when it is needed to do mutation on a whole pool.

```
procedure muRunPool()

muRunPool(pool)
{  if (mutation flag "mu_flag" = ="MU_CHROM")
       call muRunChromLevel(pool);
   else if (mutation flag "mu_flag"= = "MU_BIT")
       call muRunBitLevel(pool);
   else { error on mutation flag value, exit!}
}
```

Figure 3.6 Procedure of the Mutation interface "muRunPool()"

```
procedure muRunSingleChrom()

muRunSingleChrom(chrom)
{  if (randomly generated probability <= mu_rate)
   {  s' = GA registered mutation operator (chrom );
       repair(s');
       chrom=s';
   }
   else
   {}
}
```

Figure 3.7 Procedure of the Mutation interface "muRunSingleChrom()"

The mutation interface "muRunSingleChrom()" is designed as a manager to do one mutation on a single chromosome. If the randomly generated probability of that chromosome is no more than the mutation rate ("mu_rate"), the interface will call the GA control center registered mutation operator (one of "muByFlipBit()", "muByRandomBit()" and "muBySwap()") to perform mutation for a single chromosome that was already selected and crossed. The interface is called by "ga_SE_X_MU_RE_2CHILD()", the *single basic GA operation* of generational or steady-state GA, and also called by the GA function operator "gaByTraditional()" (the traditional GA function operator) when the mutation flag "mu_flag" = "MU_CHILD", to mutate the child immediately, in three GA models.

The *"replace"* operator, to carry out the fourth *atomic genetic operation,* is implemented in six different ways and is handled by two different policies (the *replace elitism* flag "re_elitist" is configured in the GA control center "ga_center") in the file "replace.c":

Replace by append: simply attach two children to the tail of the current chromosome array of the pool. This can only be used by generational GA to fill a new pool since the new pool is formed by filling in two children once, so it cannot be used by traditional or steady-state GA.

Replace parents: simply replace the two parents by the two children in the pool. This can be used by traditional and steady-state GA, but cannot be used by generational GA.

Replace by rank: in a sorted pool replace the worst chromosome (at tail) with one of two children one time, then move this new member leftwise to a suitable rank position. If the weakest chromosome is better than the child, no replacement happens when the replace elitism flag "re_elitist" is "on", otherwise replacement of the weakest is carried out anyway. This can be used by traditional and steady-state GA, but cannot be used by generational GA.

Replace first weaker: in a pool, replace the first hit weaker with one of two children one time; if every chromosome in pool is better than the child, then no replacement happens. This can be used by traditional and steady-state GA, but cannot be used by generational GA.

Replace the weakest: in a pool, replace the weakest member with one of two children one time; if the pool is sorted it is the same operator as *replace by rank* ("reByRank()") except for ranking movement. If the weakest chromosome is better than the child, no replacement happens when the replace elitism flag "re_elitist" is on; otherwise replacement of the weakest is carried out anyway. This can be used by traditional and steady-state GA, but cannot be used by generational GA.

Replace Random: replace two children into pool by randomly picking victims from the pool.

The choice of one out of six *replace* operators above to be registered with the GA control center structure (denoted "ga_center") is specified by the configuration file (named "gaccfg.h"), which is provided and modified according to user needs. We also implement two replacement interfaces "reRunPair()" and "reRunPool()", to stand between the GA function operator and the GA control center registered replacement operator (see Figure 3.8, 3.9).

```
procedure reRunPair()

reRunPair(p1,p2,c1,c2)
{  validating parents p1, p2 are two parents of twined children c1,c2;
   if (replace elitist flag "re_elitist" == 1)
       c1,c2 = the best two of two parents p1,p2 and twined children c1,c2;
   else
   {}
    call GA registered replace operator to put back c1and c2;
}
```

Figure 3.8 Procedure of the Replacement Interface "reRunPair()"

The *replace* interface "reRunPair()" is designed to call the GA control center "ga_center" registered *replace* operator to replace two passed-in children (passed into the interface as parameters) into the pool. It is used by generational and steady-state GA, and called by generational and steady-state GA's *single basic GA operation:* "ga_SE_X_MU_RE_2CHILD()" or "ga_SE_X_RE_2CHILD()".

procedure reRunPool()

```
reRunPool(pool, child)
{  for each pair of twined children c1,c2 from child array
   {  validating twined children c1,c2 by their parents indeices;
      if (replace elitist flag "re_elitist" == 1)
         c1,c2 = the best two of two parents p1,p2 and twined children c1,c2;
      else
      {}
       call GA registered replace operator to put back c1and c2 into pool;
   }
}
```

Figure 3.9 Procedure of the Replacement Interface "reRunPool()"

The *replace* interface "reRunPool()" is designed to be used by traditional GA, called within the traditional GA iteration loop after the new pool (*reproduction pool*) has been selected by applying the GA control center registered *selection* operator, the *mating pool* has been selected from the *reproduction pool* according to the *crossover* rate, the crossovers have been performed to give birth of all children by randomly pairing parents from the *mating pool*, the mutations of all newborn children have been carried out if it is requested by the GA control center to mutate the children immediately when the mutation method flag "mu_flag" has value of "MU_CHILD" in the GA control center "ga_center". All new children reside in the array "child" (declared in the traditional GA function operator "gaByTraditional()") are passed into the interface as parameter. We replace all new children into the new pool by passing twinned new children to the GA control center registered *replace* operator (except the operator "reByAppend()") to perform replacement until finished.

GA Function Operators

Genetic algorithms are implemented in three different ways in the file "ga.c". They are traditional GA, generational GA, and steady-state GA (see Figure 3.10, 3.11, 3.12, 3.13).

procedure gaByTraditional()

```
gaByTraditional()
{   t = 0;
    initialize the population (pool) P(t) to be the starting pool;
    while( t is less than max. allowed iteration limit and P(t) is not converged)
    {   initialize P(t+1) by handling selection elitism and generation gap;
        P(t+1) = select from P(t) by calling selection interface seRunPool();
        mating pool = select from P(t+1) according to the crossover rate;
        call xRunPool() to randomly pairing parents from mating pool and crossover;
        if(GA requests mutation on child immediately)
        {   mutation all children before replace them by calling muRunSingleChrom();
            replace all children into P(t+1) by calling reRunPool();
        }
        else if (GA requests to mutate pool)
        {   replace all children into P(t+1) by calling reRunPool();
            mutate P(t+1) by calling muRunPool();
        }
        evaluate and update P(t+1);
        P(t) ← P(t+1);
    }
}
```

Figure 3.10 Procedure of the Traditional GA Function Operator

Traditional GA is implemented by the function "gaByTraditional()" in the file "ga.c". It is the GA that maintains two pools. In each GA iteration, it first handles *selection* *elitism*, if provided, which transfers several copies of the top best chromosomes from the old pool to the new pool, and handles the *generation gap*, if provided, which transfers directly several chromosomes from the old pool to the new pool randomly. These two handlings are enabled or disabled by the *selection elitism* flag "se_elitist" and *generation gap* "gap" being "on" or "off" in the GA control center "ga_center" respectively. Here *selection elitism* means that we make sure at least two copies of the top best chromosomes are put into the new pool, that is the *reproduction pool,* such that it will enhance the chance for the best to win a crossover to contribute their genetic genes to the next generation when they participate in the crossover competition with the rest of *reproduction pool*. *Generation gap* means that a percentage of the population of the old pool is copied (cloned) to the new pool at the pool initialization stage of each iteration of traditional GA and generational GA. This only makes sense in a GA with two pools, as in the traditional and generational models. Actually "gap" plays no part in traditional GA because handling "gap" becomes a part of the selection operation to select the *reproduction pool* of each iteration of traditional GA. After handling the *selection elitism* "se_elitist" and *generation gap* "gap" flags, the traditional GA function operator calls the *selection* operation interface "seRunPool()" to use the GA control center "ga_center" registered *selection* operator to randomly select the rest of the chromosomes forming the *reproduction pool* from the old pool until the pool is full up to the pool size. Then, the traditional GA function operator calls the *crossover* operation interface "xRunPool()" to form the *mating pool* by randomly choosing chromosomes from the whole *reproduction pool* according to the *crossover* rate "x_rate" and to use the GA control center "ga_center" registered *crossover* operator to cross all randomly paired parents from the *mating pool* until the *mating pool* is empty. After all children have been produced, the GA has two ways to deal with bit mutations and replacements: In the first way, the traditional GA function operator does *mutation* to all children first then does *replacement*. At this time the GA control center "ga_center" data of the mutation method flag "mu_flag" has the value: "MU_CHILD" (GA requests to mutate every child immediately before replacing him), so the GA performs *mutation* interface "muRunSingleChrom()" to each new child before it calls the *replacement* interface "reRunPool()" to replace all new children into the new pool. After all new children have been mutated, the traditional GA function operator calls the *replacement* interface "reRunPool()" to replace all these new mutated children into the new pool (*reproduction pool)*. In the second way, the traditional GA function operator does *replace* children into the new pool first, then does *mutation* for the whole new pool. At this time the GA control center "ga_center" data of the mutation method flag "mu_flag" has a value either "MU_CHROM" or "MU_BIT" (the GA requests to mutate the whole pool by checking every chromosome or every bit of the pool). After all new children have been born, the traditional GA function operator does *replace* first, calling the *replace* interface "reRunPool()" to replace all children into the new pool, then the traditional GA function operator calls the *mutation* interface "muRunPool()" to do *mutation* on the whole new pool by using the GA control center registered *mutation* operator. The new generation is produced in the new pool (*reproduction pool)* by finishing one iteration of traditional GA. The GA function operator continues on until GA converges or a maximum number of allowed GA iterations has been reached. The following are the *atomic genetic operators* that the traditional GA function operator can use.

Traditional GA:	selection	: any
	crossover	: any
	mutation	: any
	replace	: any except "append"

The generational and steady-state GA's *single basic GA operation* is implemented by the function "ga_SE_X_MU_RE_2CHILD()" (which selects, then crosses and mutates two parents from the old pool to produce two children, and replaces the two children back to the new pool) or the function "ga_SE_X_RE_2CHILD()" (which selects, then crosses two parents from the old pool to produce two children, and replaces the two children back to the new pool). Both of these are implemented in the file "ga.c" (see Figure 3.11).

procedure ga_SE_X_MU_RE_2CHILD()

```
ga_SE_X_MU_RE_2CHILD()
{
   p1,p2 = select two parents from old pool by calling selection interface seRunPool();
   c1,c2 = crossover p1,p2 by calling crossover interface xRunPair();
   c1,c2 = mutate c1,c2 by calling mutation interface muRunSingleCrom();
   replace c1,c2 into new pool by calling replace interface reRunPair();
}
```

procedure ga_SE_X_RE_2CHILD()

```
ga_SE_X_RE_2CHILD()
{
   p1,p2 = select two parents from old pool by calling selection interface seRunPool();
   c1,c2 = crossover p1,p2 by calling crossover interface xRunPair();
   replace c1,c2 into new pool by calling replace interface reRunPair();
}
```

Figure 3.11 Procedure of Single Basic GA Operations

The outlines of these two functions are as follows: call the *selection* operation interface "seRunPool()" to use the GA registered *selection* operator to randomly select two parent chromosomes from the old pool, then call the *crossover* operation interface "xRunPair()" to check the random probability against the *crossover* rate "x_rate" to determine if they need *crossover* and to use the GA control center registered *crossover* operator to cross the two parents. After two children have been produced, we have two ways to deal with bit mutations and replacements. The first way, done by "ga_SE_X_MU_RE_2CHILD()", it does *mutation* to two children first then does *replacement*; at this time the mutation method flag "mu_flag" has the value: "MU_CHILD" (GA requests to mutate every child immediately before replacing it), calls the *mutation* interface "muRunSingleChrom()" to each new child before it calls the *replace* interface "reRunPair()" to replace two children into the new pool. After two new children have been mutated, it calls the *replace* interface "reRunPair()" to replace these two mutated children into the new pool. The second way, done by "ga_SE_X_RE_2CHILD()", performs *replace* children into the new pool then does *mutation* for the whole new pool later; at this time the mutation method flag "mu_flag" has a value either "MU_CHROM" or "MU_BIT" (GA requests to replace all new children into the new pool, then mutate the new pool). After two new children have been born, it does *replace* first, replacing the two children into the new pool. Until the new pool is full or just before the end of this iteration, the GA function operator (generational ,or steady-state) calls the *mutation* interface "muRunPool()" to do *mutation* on the whole new pool by using the GA control center registered mutation operator.

procedure gaByGenerational()

```
gaByGenerational()
{  t = 0;
   initialize the population (pool) P(t) to be the starting pool;
   while( t is less than max. allowed iteration limit and P(t) is not converged)
   {  initialize P(t+1) by handling selection elitism and generation gap;
      if(GA requests mutation on child immediately)
      {   while(P(t+1) is not full)
             perform single basic GA operation by calling ga_SE_X_MU_RE_2CHILD();
      }
      else if (GA requests to mutate pool)
      {  while(P(t+1) is not full)
             perform single basic GA operation by calling ga_SE_X_RE_2CHILD();
         mutate P(t+1) by calling muRunPool();
      }
      evaluate and update P(t+1);
      P(t) ← P(t+1);
   }
}
```

Figure 3.12 Procedure of the Generational GA Function Operator

Generational GA is implemented by the function "gaByGenerational()" in the file "ga.c". It is the other GA which maintains two pools. In each GA iteration, it first handles *selection elitism*, if provided, which transfers several copies of the best chromosomes from the old pool to the new pool, and handles the *generation gap*, if provided, which transfers several copies of chromosomes from the old pool to the new pool randomly. These two handlings are enabled or disabled by the *selection elitism* flag "se_elitist" and the *generation gap* "gap" being "on" or "off" in the GA control center "ga_center" respectively. Here the *selection elitism* means that we make sure at least two copies of the best chromosomes are put into the new pool, such that the best survive through the next generation. *Generation gap* means that a percentage of the population of the old pool is copied (cloned) to the new pool at the pool initialization stage of each iteration of traditional and generational GA. This only makes sense in a GA with two pools, as in the traditional and generational models. Here "gap" does play some part in generational GA because handling "gap" directly transfers a portion of the old generation to the new generation randomly. After handling the *selection elitism* flag "se_elitist" and *generation gap* "gap", the generational GA function operator enters the loop of *single basic GA operation* which will make up the rest of the new generation by applying the *single basic GA operation* cyclically. In each cycle, the GA produces two children; the task is carried out by the function "ga_SE_X_MU_RE_2CHILD()" or the function "ga_SE_X_RE_2CHILD()" according to the value of the mutation method flag "mu_flag" in the GA control center "ga_center" (see Figure 3.11). The generational GA function operator keeps performing the *single basic GA operation* until the new pool is full and the new generation is produced in the new pool and the current iteration is finished and begins preparing the next iteration. The generational GA function operator continues on until GA converges or the maximum number of allowed GA iterations has been reached. The following are the *atomic genetic operators* that the generational GA function operator can use.

generational GA:	selection	: any
	crossover	: any
	mutation	: any
	replace	: only "append"

procedure gaBySteady_state()

```
gaBySteady_state()
 {  t = 0;
    initialize the population (pool) P(t) to be the starting pool;
    while( t is less than max. allowed iteration limit and P(t) is not converged)
    {
        if(GA requests mutation on child immediately)
        { perform single basic GA operation by calling ga_SE_X_MU_RE_2CHILD();
        }
        else if (GA requests to mutate pool)
        { perform single basic GA operation by calling ga_SE_X_RE_2CHILD();
          mutate P(t) by calling muRunPool();
        }
        evaluate and update P(t);
        t=t+1;
    }
}
```

Figure 3.13 Procedure of the Steady-state GA Function Operator

Steady-state GA is implemented by the function "gaBySteady_state()" in the file "ga.c". This is the GA that only maintains one pool in each iteration. In each GA iteration, it does not handle *selection elitism* and *generation gap*, though they may be provided, due to only one pool being maintained. Each GA iteration only produces two children and replaces them back into the pool; that is, each iteration consists of randomly selecting two parents from the pool to perform *crossover* and *mutation*. If the *replace elitism* flag is on, choose the best two out of four to replace back in the pool. Each iteration is carried out by only one *single basic GA operation:* function "ga_SE_X_MU_RE_2CHILD()" or function "ga_SE_X_RE_2CHILD()" according to the value of the mutation method flag "mu_flag" in the GA control center "ga_center". Then we have finished one iteration of steady-state GA. The steady-state GA function operator continues on until GA converges or the maximum number of allowed GA iterations has been reached. The following are the *atomic genetic operators* that the steady-state GA function operator can use.

steady-state GA:	selection	:	any
	crossover	:	any
	mutation	:	any
	replace	:	any except "append"

SAGA Modes

We adopt *simulated annealing* (SA) by incorporating its operators: SAM (simulated annealing mutation) and SAC (simulated annealing crossover) into a standard GA environment. These operators use the SA stochastic acceptance function internally to limit adverse moves. This is shown to solve two key problems in GA optimization: populations can be kept small, and hill-climbing in the later phase of the search is facilitated. We introduce the invoking probabilities "sam" and "sac" in the GA control center "ga_center" to control the uses of SAM and SAC, such that the system operates as pure SA (or iterated SA) when sam = 1.0 and sac = 1.0, pure GA when sam = 0.0 and sac = 0.0, or in various hybrid modes when 0.0 < sam <1.0 and 0.0 < sac <1.0. The hybrid algorithm is seen to improve on pure GA in two ways: better solutions for a given number of evaluations, and more consistency over many runs (see Chapter IV).

The method we have used to combine SA with GA resulting in SAGA is by combining GA mutation with SA mutation (SAM) and GA crossover with SA crossover (SAC). We switch between them by a stochastic statement installed within the GA functions which will check random probabilities against the invoking probabilities "sam" and "sac" at run time to determine whether or not we invoke the SA operators.

The SAM Operator

The SAM operator works exactly like a standard GA mutation operator: it gets a solution as input, mutates it and returns a solution as output. The difference is that internally the SA operator can call the evaluation function and use the result to decide whether to accept the mutated solution, or just stay with the previous solution:

```
SAM(s, T)
{   s'=mutate(s, T);
    if(accept(s', s, T)) return s';
    else return s;
}
```

After *s'* is getten from a mutation of *s* at temperature *T*, we evaluate *accepting function* to determine the probability of accepting the *s'*. The *annealing temperature T* is lowered through generations. Thus, every one (or more) generations, the temperature is lowered, making the SA operators more selective about the mutations it accepts.

We implement the SAM operator by introducing *accepting function* into the GA mutation interfaces after the call of the GA control center "ga_center" registered mutation operator, resulting in four functions in the file "mutation.c": "samRunChromLevel()", "samRunBitLevel()", "samRunPool()", "samRunSingleChrom()".

```
procedure samRunChromLevel()

samRunChromLevel(pool)
{   for each chromosome s of pool
    {   if (randomly generated probability <= mu_rate)
        {   s' = GA registered mutation operator (s );
            repair(s');
            if(accept(s', s, T)) s = s';
            else s = s;
        }
    }
}
```

Figure 3.14 Procedure of the SAM interface "samRunChromLevel()"

The SAM interface "samRunChromLevel()" is a simulated annealing genetic algorithm mutation submanager to do mutation on a whole pool (see Figure 3.14). For each chromosome of the pool we generate a random number to see if it needs mutation based on the mutation rate, then call the GA control center "ga_center" registered mutation operator to do mutation. After mutation we call the *accepting function* to calculate the accepting probability based on the fitness and current *annealing temperature* T, then we check the random probability against the accepting probability to determine if the mutated solution can be accepted.

The SAC Operator

The SAC operator works exactly like a standard GA crossover operator: it gets two solutions as input, crosses them and returns two children as output. The difference is that internally, the SA operator can call the evaluation function, and use the result to decide whether to accept the crossed children, or just stay with the previous solutions:

```
SAC($s_1$, $s_2$, T)
{  $s_1$' = crossover ($s_1$, $s_2$, T);
   $s_2$' = crossover ($s_1$, $s_2$, T);
   s = best of ($s_1$, $s_2$);
   if( accept($s_1$', s, T) ) return $s_1$';
   else return s;
   if( accept($s_2$', s, T)) return $s_2$';
   else return s;
}
```

After s_i' (i=1,2) is getten from crossover of s_1 and s_2 at temperature T, we evaluate an *accepting function* (located in the file "ga.c") to determine the probability of accepting the s_i'. The *annealing temperature* T is lowered through generations. Thus, every one (or more) generations, the temperature is lowered, making the SA operators more selective about the crossovers it accepts.

We implement the SAC operator by introducing an *accepting function* into the GA crossover interfaces after the call of the GA control center "ga_center" registered crossover operator, resulting in two functions in the file "cross.c": "sacRunPair()", "sacRunPool()".

```
procedure sacRunPair()

sacRunPair(p1,p2,c1,c2)
{  initialize child chromosomes c1, c2 and record parents p1, p2 index;
   if (random probability <= x_rate)
   {  c1,c2 = crossover(p1,p2) by GA registered crossover operator;
      p = best of ($p_1$, $p_2$);
      if( accept(c1, p, T) ) c1 = c1;
      else c1 = p;
      if( accept(c2, p, T)) c2 = c2;
      else c2 = p;
   }
   else
   {  c1=p1;
      c2=p2;
   }
}
```

Figure 3.15 Procedure of the SAC Interface "sacRunPair()"

The SAC interface "sacRunPair()" is a simulated annealing genetic algorithm crossover manager to call the GA control center "ga_center" registered crossover operator to cross two passed-in parents to produce two children if a randomly generated probability meets the crossover rate. Two children are put in global variables "child1" and "child2" and passed out. After crossover we call the *accepting function* to calculate the accepting probability based on the fitness and current *annealing temperature* T, then we check the random probability against the accepting probability to determine if the crossed solutions can be accepted.

CHAPTER IV

RESULTS AND ANALYSIS

In a GA, the search or optimization process consists of initializing the population and then breeding new individuals until the termination condition is met. There can be several goals for the search process, one of which is to find the global optima. With the types of models that GA's work well with, this can never be assured. There is always the chance that the next iteration in the search will produce a better solution. Alternatively, the search could run for years and not produce any better solution than it did in the first five iterations [Bartlett, 7].

Another goal is quick convergence. When the objective function is expensive to run, quick convergence is desirable; however, the chance of converging on a local, and possibly quite substandard optima, is increased [Bartlett, 7].

In this chapter, we will discuss the system parameters settings, in order to achieve the above two goals, by comparing the performance of three GA models and integrated SAGA with a variety of parameter values when applied to one railroad operating plan instance. The three GA models are traditional GA, generational GA, and steady-state GA. SAGA is the GA integrated with simulated annealing operators: SAM (simulated annealing mutation) and SAC (simulated annealing crossover). The railroad operating plan problem used is the problem of 14 terminals.

Test Data Preparation

To do the comparison of performance mentioned above, we use the configuration data of 14 stations of a railroad operating plan problem presented in Ms. Lei's Master's degree thesis of Oklahoma State University Mathematics Department (1992).

The fixed cost array:

630 590 600 600 600 600 600 650 650 650 620 620 140 0.0

where c_i is the fixed cost at station i.

The transfer cost array:

0.0 3.0 3.0 3.0 3.0 2.8 2.9 3.2 3.4 3.8 3.5 4.0 3.1 5.0

where t_i is the transfer cost at station i.

The flow matrix:

$$\begin{pmatrix}
0 & 1 & 37 & 25 & 33 & 17 & 27 & 10 & 1 & 1 & 9 & 1 & 4 & 13 \\
0 & 0 & 1 & 42 & 29 & 8 & 4 & 3 & 1 & 1 & 3 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 56 & 22 & 28 & 7 & 1 & 1 & 8 & 1 & 3 & 3 \\
0 & 0 & 0 & 0 & 1 & 10 & 9 & 18 & 4 & 14 & 18 & 3 & 4 & 17 \\
0 & 0 & 0 & 0 & 0 & 1 & 42 & 35 & 2 & 5 & 13 & 3 & 3 & 19 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 132 & 11 & 12 & 33 & 6 & 17 & 26 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 10 & 4 & 15 & 0 & 4 & 26 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 16 & 0 & 6 & 22 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 36 & 12 & 19 & 48 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 10 & 14 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 79 & 104 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 43 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

where f_{ij} is the number of cars to be sent from terminal i to terminal j.

The optimal solution [Lei, 1992] is the feasible arc set:

$\boldsymbol{D}^* = A_0 \cup \{ a_{3,6}, a_{6,8}, a_{6,11}, a_{9,11}, a_{11,14}\}$

The minimal cost $\boldsymbol{F}(\boldsymbol{D}^*) = 8470.4$

Ms. Lei (1992) has used the compressing branch and bound method to find the above optimal solution.

General Comparison of Three GAs

In general, regarding convergence, traditional GA is faster than steady-state GA, and generational GA is faster than traditional GA. Since steady-state GA only applies crossover to two selected parents during each iteration, the genetic searching and hill climbing is the slowest among three GAs. Since traditional GA selects *reproduce pool* and *mating pool* before crossover operations are applied, it is more likely that traditional GA will lose more genetic diversity than generational GA during genetic evolution, and that will limit the search scope of traditional GA and slow down convergence. Therefore traditional GA is slower than generational GA.

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of three GA models on the ROP (Railroad Operating Plan) instance of 14 terminals by the following configuration:

```
------------------begin GA system configuration------------------
GA control center configuration information:
-------------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     :  randomly
allowed max. pool size   : 80
actual pool size         : 66
defined chrom. matrix dim: 14
```

```
allowed max. trials         : 220 iterations, ignore convergence
minimize optimization       : yes
select elitism policy       : transfer two copies of the best
replace elitism policy      : yes
GA generation gap           : 0.1
GA rank-biased pressure     : 0
GA crossover rate           : 0.7
GA mutation rate            : 0.5
GA mutation method          : mutation whole pool from chrom. level
probability to invoke SAM : 0
probability to invoke SAC  : 0
GA registered operations:
GA function                 : generational, traditional, steady-state
GA selection                : uniform-random
GA crossover                : random-flip
GA mutation                 : simple-flip
GA replace                  : append (using by generational GA), replace-parent
GA reports :
report type                 : Minimal
report-interval             : 10
--------------------end GA system configuration-----------------------
```

Comparison of Three GAs
(the average best fitness among ten runs)

# OF ITER	GENERATIONAL	TRADITIONAL	STEADY-STATE
0	20927.7	20927.7	21032.8
10	15283,6	16162.8	16047.9
20	12740.7	13059.5	14534.5
30	11033.4	11031.7	14922.0
40	9932.7	9849.9	15056.6
50	9232.9	9214.3	14761.7
60	8922.1	8858.0	14588.7
70	8692.5	8757.8	14219.3
80	8678.1	8738.0	14426.8
90	8642.8	8721.7	14967.8
100	8630.3	8721.7	15253.6
110	8630.3	8710.1	15103.0
120	8630.3	8710.1	14577.5
130	8630.3	8710.1	14373.7
140	8630.3	8710.1	14790.2
150	8630.3	8710.1	14915.5
160	8630.3	8710.1	15397.6
170	8630.3	8700.2	15177.1
180	8624.1	8691.7	15853.9
190	8624.1	8676.0	14327.4
200	8624.1	8676.0	14748.2
210	8624.1	8675.6	15105.7
220	8624.1	8675.6	14917.8

Table 4.1

Comparison of Three GAs
(the result of first turn among ten runs)

# OF ITER	GENERATIONAL	TRADITIONAL	STEADY-STATE
0	19622.2	19622.2	19622.2
10	16216.3	16801.6	16590.8
20	12519.1	14121.3	16175.7
30	10870.5	12181.4	15157.3
40	9389.3	11146.9	15808.3
50	8796.9	10194	16569.9
60	8676.8	9293.8	14644
70	8676.8	8676.8	16217.3
80	8595.6	8604.8	14820.9
90	8470.4	8595.6	14575.2
100	8470.4	8470.4	16678.8
110	8470.4	8470.4	15383.3
120	8470.4	8470.4	12533.7
130	8470.4	8470.4	16134.8
140	8470.4	8470.4	15457.6
150	8470.4	8470.4	15135.4
160	8470.4	8470.4	15383.9
170	8470.4	8470.4	17139.9
180	8470.4	8470.4	14879.5
190	8470.4	8470.4	15690.7
200	8470.4	8470.4	15189.9
210	8470.4	8470.4	15009.2
220	8470.4	8470.4	14828.4

Table 4.2

In the typical run (the first turn among ten runs) (see Table 4.2), the first appearance of the global optimum 8470.4 is in the 87th iteration in generational GA and 96th iteration in traditional GA, and does not occur in steady-state GA even if it runs to the 30000th iteration! As for the records of steady-state GA, it takes roughly pool-size/2 (here pool-size = 66, 66/2 = 33) iterations of the steady-state model to equal one iteration of the generational model and traditional model. Thus we record the results of iterations 0, 330, 660, 990, ..., 7260 of steady-state GA to correspond to and compare with the results of iterations 0, 10, 20, ..., 220 of generational GA and traditional GA. We have noticed that the best fitness of each iteration of steady-state GA is not monotonically decreasing. This is because replace elitism does not take into account effects on mutation operations if mutation operations are applied to the whole pool level when the mutation flag is "MU_BIT" or "MU_CHROM", since in this situation we do *replace* before *mutation*. It is possible that the mutation operator may blindly mutate the best fit chromosome of that generation and degenerate the pool. Another reason may be that the number of crossover operations is not much more than the number of mutations in steady-state GA, compared with other GAs.

Population Size and Convergence

In our implementation, the maximum allowed pool size is given by the system configuration file specified by the user to suit the requirements of the particular model. The actual pool size is between the maximum allowed pool size and half of it, depending on the actually successful reading of the chromosome matrices either from a file or by a random generator. We have code segment to make sure that the actual pool size is an even number.

Our implementation of GA to dealing with matrix representation of chromosomes can simulate large models but can be expensive in terms of time, so a smaller population may be desirable, but if the population is too small then the loss of genetic diversity may compromise the search. This may prevent the appearance of the fittest chromosome in future generations. Genetic diversity in GA's is important when the solution space is topographically rugged or convoluted [Bartlett, 7]. A small population would be more likely to converge quickly on what may be a local optimum. A comparatively fit individual in a small starting population will be selected for, and it and its descendants will quickly come to dominate, further limiting genetic diversity. Also with the sparse spread of initial points, better optima may never be visited before the search converges on a poor local optimum, or even if they are visited, the point may be comparatively unfit, and will not be given adequate opportunity to reproduce and start hill climbing. At the other extreme, the problem with excessively large populations is that the search may flounder in the overabundance of genetic diversity. In a large population there may be many fit individuals; a problem arises when there are several local optima. Our implementation has no methods to assess genetic distance when selecting parents. If there are several groups of fit individuals clustering around different optima, the union of individuals from two distinct optima may not preserve the genetically 'good' part of either of the parents. At worst, this type of miscegenation could depopulate the areas around the various optima. More likely is that the hill climbing around the optima will be retarded, which further increases the time cost. If the objective function is cheap, or the cost of failing to find the global optima is high, it may be worth bearing the extra time to increase the chance of finding the best possible solution.

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of three GA models on the ROP (Railroad Operating Plan) instance of 14 terminals.

The following is the test run result of generational GA that will illustrate the above discussion. Here is the GA configuration:

```
------------------begin GA system configuration------------------
GA control center configuration information:
----------------------------------------------------
basic parameters:
random-seed                : 1
Initial pool entered       :  randomly
allowed max. pool size     : test on 20, 50, 80, 100, 130
actual pool size           : 16, 40, 66, 82,108 correspondingly
defined chrom. matrix dim: 14
allowed max. trials        : 220 iterations, ignore convergence
minimize optimization      : yes
select elitism policy      : transfer two copies of the best
replace elitism policy     : yes
GA generation gap          : 0.1
GA rank-biased pressure  : 0
GA crossover rate          : 1.0
GA mutation rate           : 0.4
GA mutation method         : mutation whole pool from chrom. level
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function                : generational
GA selection               : uniform-random
GA crossover               : 2xp-crossover
GA mutation                : simple-flip
GA replace                 : append
GA reports :
report type                : Minimal
report-interval            : 10
-------------------end GA system configuration----------------------
```

We run the GA as a generational model on the test data of 14 stations railroad problem, set the selection elitist flag equal to 1 (transfer two copies of the best of old to new pool), set the replace elitist flag on (always choose two best among two parents and two children to place back into the pool), GA crossover rate is 1.0, GA mutation rate is 0.4, mutation method flag is "MU_CHROM" (mutate the whole pool at the chromosome level), GA function is generational, GA selection is "uniform-random", GA crossover is "2xp-crossover", GA mutation is "simple-flip", GA replace is "replace by append". We have tested by running generational GA with pool sizes of 20, 50, 80, 100, 130, and the other parameters have been kept unchanged. The results are shown on Table 4.3 and 4.4.

P-size and Convergence (generational GA)
(the average best fitness among ten runs)

# of iter	p-size 20	p-size 50	p-size 80	p-size 100	p-size 130
0	22927.2	21673.3	20703.6	20778.9	20744.3
10	19392.4	17017.2	16007.7	15146.1	16266.5
20	17701.6	14798.4	13246.4	12708.0	13656.3
30	15857.5	13051.3	11363.9	1135.8	11667.2
40	14595.5	11520.8	10099.8	9911.2	10613.3
50	13427.3	10557.4	9275.8	9164.7	9786.6
60	12686.8	9922.5	8972.3	8833.8	9188.0
70	11617.2	9418.6	8865.5	8740.9	8981.7
80	10869.7	9114.0	8707.8	8674.9	8872.1
90	10318.3	8982.4	8673.8	8638.1	8820.7
100	9919.8	8878.8	8673.8	8628.5	8793.7
110	9563.8	8764.9	8673.8	8625.2	8743.3
120	9381.9	8723.5	8573.2	8589.6	8731.3
130	9207.1	8710.9	8573.2	8589.6	8731.3
140	9169.7	8707.1	8573.2	8567.2	8731.3
150	9083.7	8701.5	8573.2	8567.2	8731.3
160	8909.1	8701.5	8573.2	8567.2	8731.3
170	8882.5	8693.6	8573.2	8516.3	8731.3
180	8827.0	8693.6	8556.5	8516.3	8731.3
190	8826.1	8693.6	8556.5	8516.3	8731.3
200	8824.1	8687.6	8556.5	8488.2	8724.7
210	8813.6	8687.6	8556.5	8488.2	8697.4
220	8813.6	8687.6	8556.5	8488.2	8697.4

Table 4.3

P-size and Convergence (generational GA)
(the result of first turn among ten runs)

# of iter	p-size 20	p-size 50	p-size 80	p-size 100	p-size 130
0	23637.6	22460.2	19622.2	19622.2	19622.2
10	21170.9	16145.8	15761.6	15707.1	15733.5
20	19732.3	15021.1	14009.3	13331.4	13419.8
30	17582.5	13030.8	12134.1	11086.8	11627.5
40	15943.1	10894.4	10483.6	9846.7	10639.6
50	14773.3	10781.9	9632.2	9151.3	10043.3
60	14108.9	9676.1	9125.9	8933.5	9056.8
70	11971.3	9000.4	8860.9	8727.6	8827.4
80	11579.7	8691.7	8657.6	8595.6	8641.4
90	10589.5	8654.4	8547.9	8470.4	8641.4
100	10537.5	8625.9	8547.9	8470.4	8641.4
110	9877.8	8625.9	8547.9	8470.4	8641.4
120	9644.2	8625.9	8547.9	8470.4	8641.4
130	9644.2	8625.9	8547.9	8470.4	8641.4
140	9548.2	8625.9	8547.9	8470.4	8641.4
150	9548.2	8625.9	8547.9	8470.4	8641.4
160	9548.2	8625.9	8547.9	8470.4	8641.4

170	9051.9	8625.9	8547.9	8470.4	8641.4
180	8923.8	8625.9	8547.9	8470.4	8641.4
190	8857.8	8625.9	8547.9	8470.4	8615.6
200	8857.8	8625.9	8547.9	8470.4	8564.8
210	8857.8	8625.9	8547.9	8470.4	8470.4
220	8803.8	8625.9	8547.9	8470.4	8470.4

Table 4.4

From Table 4.3 and 4.4 we can see that when the pool size is small, the pool lacks of genetic diversity, and this prevents the appearance of the global optimum in the future generations, The pool quickly converges at so-called local optima. As the pool size increases to a suitably large value, say pool size = 100, GA successfully converges at the global optima 8470.4 (see Table 4.4), but if the pool size is too large, say pool size = 130, the search may flounder in the over abundance of genetic diversity. It is more likely that the hill climbing around the optima will be retarded (it is late until 210th iteration the global optima 8470.4 appears (see Table 4.4)). This may be overcome by bearing extra time to increase the chance of finding the best possible solution.

The following is a test run result of traditional GA that will also illustrate the above discussion. Here is the GA configuration:

```
------------------begin GA system configuration------------------
GA control center configuration information:
----------------------------------------------------
basic parameters:
random-seed               : 1
Initial pool entered      :  randomly
allowed max. pool size    : test on 20, 60, 80, 100, 110, 120
actual pool size          : 16, 50, 66, 82, 90, 98 correspondingly
defined chrom. matrix dim: 14
allowed max. trials       : 220 iterations, ignore convergence
minimize optimization     : yes
select elitism policy     : transfer two copies of the best
replace elitism policy    : yes
GA generation gap         : 0.1
GA rank-biased pressure   : 0
GA crossover rate         : 0.7
GA mutation rate          : 0.5
GA mutation method        : mutation whole pool from chrom. level
```

```
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function            : traditional
GA selection           : uniform-random
GA crossover           : random-flip
GA mutation            : simple-flip
GA replace             : replace-parent
GA reports :
report type            : Minimal
report-interval         : 10
--------------------end GA system configuration-----------------------
```

We have tested traditional GA on pool sizes of 20, 60, 80, 100, 110, 120 and the other parameters have been kept unchanged. The results are in Table 4.5 and 4.6.

P-size and Convergence (traditional GA)
(the average best fitness among ten runs)

# OF ITER	P-SIZE 20	P-SIZE 60	P-SIZE 80	P-SIZE 100	P-SIZE 110	P-SIZE 120
0	22951.7	21079.9	20830.9	20865.2	20076.3	20927.7
10	18527.0	15987.0	15055.9	15482.0	14829.6	15283.6
20	16710.9	13537.9	12600.5	12949.3	12323.4	12740.7
30	15043.2	11865.2	10811.2	11189.6	10652.2	11033.4
40	13434.0	10332.7	9672.6	9906.2	9482.1	9932.7
50	12065.8	9506.4	9087.7	9303.1	8997.6	9232.9
60	11088.7	9018.7	8735.0	8916.8	8848.1	8922.1
70	10425.2	8804.5	8668.1	8746.4	8694.9	8692.5
80	9929.0	8762.2	8639.1	8653.9	8660.9	8678.1
90	9667.4	8703.0	8639.1	8644.8	8632.6	8642.8
100	9455.3	8677.6	8639.1	8621.7	8632.6	8630.3
110	9230.2	8671.1	8623.0	8593.5	8632.6	8630.3
120	9207.2	8667.8	8610.8	8593.5	8632.6	8630.3
130	9084.6	8647.1	8572.0	8593.5	8632.6	8630.3
140	9035.1	8647.1	8572.0	8593.5	8632.6	8630.3
150	9007.3	8647.1	8565.3	8593.5	8632.1	8630.3
160	8951.3	8647.1	8556.8	8593.5	8623.7	8630.3
170	8940.4	8647.1	8556.8	8593.5	8623.2	8630.3
180	8938.8	8647.1	8556.8	8593.5	8623.2	8624.1
190	8931.2	8647.1	8556.8	8593.5	8623.2	8624.1
200	8924.0	8647.1	8556.8	8593.5	8623.2	8624.1
210	8916.4	8647.1	8556.8	8593.1	8622.3	8624.1
220	8896.3	8647.1	8556.8	8593.1	8581.2	8624.1

Table 4.5

P-size and Convergence (traditional GA)
(the result of first turn among ten runs)

# OF ITER	P-SIZE 20	P-SIZE 60	P-SIZE 80	P-SIZE 100	P-SIZE 110	P-SIZE 120
0	23637.6	22460.2	19622.2	19622.2	19622.2	19622.2
10	19007.1	17736.4	16801.6	15514.7	16152.1	15171.1
20	16915.9	13991.1	14121.3	12865.5	13018.4	12188.3
30	14925.8	12684	12181.4	11656.1	11215.6	11193.7
40	13486.5	10937.3	11146.9	10672.7	9593.9	9840.3
50	12268.7	10071.5	10194	10000.1	9274.4	9077.7
60	10275.2	9599.9	9293.8	8684.8	8839.1	8861.7
70	9794.8	9323.6	8676.8	8470.4	8604.8	8705.6
80	9449.5	9247.4	8604.8	8470.4	8485.9	8659.2
90	9449.5	8894.9	8595.6	8470.4	8485.9	8659.2
100	9035.9	8851.5	8470.4	8470.4	8485.9	8659.2
110	9035.9	8845.5	8470.4	8470.4	8485.9	8659.2
120	9035.9	8827.9	8470.4	8470.4	8485.9	8659.2
130	8801	8803.8	8470.4	8470.4	8485.9	8659.2
140	8801	8621.5	8470.4	8470.4	8485.9	8659.2
150	8801	8621.5	8470.4	8470.4	8485.9	8659.2
160	8801	8621.5	8470.4	8470.4	8485.9	8659.2
170	8801	8621.5	8470.4	8470.4	8485.9	8659.2
180	8801	8621.5	8470.4	8470.4	8485.9	8659.2
190	8801	8621.5	8470.4	8470.4	8485.9	8659.2
200	8801	8621.5	8470.4	8470.4	8485.9	8659.2
210	8801	8621.5	8470.4	8470.4	8485.9	8659.2
220	8801	8621.5	8470.4	8470.4	8485.9	8659.2

Table 4.6

From Table 4.5 and 4.6 we can see that when pool sizes are relatively small, traditional GA converges very quickly on so-called local optima due to lack of genetic diversity. When pool sizes are too big, traditional GA retards convergence or flounders around some local optima. Here once again we have illustrated the discussion about the relation between pool size and pool convergence.

The following is the test run results of steady-state GA that will also illustrate the discussion at the beginning of this section. Here is the GA configuration:

```
------------------begin GA system configuration------------------
GA control center configuration information:
-----------------------------------------------------
basic parameters:
random-seed              : 1
```

```
Initial pool entered          :  randomly
allowed max. pool size        : test on 20, 40,60, 80, 90, 110,120
actual pool size              : 16, 32, 50, 66, 74, 90,98 correspondingly
defined chrom. matrix dim     : 14
allowed max. trials           : run until convergence
minimize optimization         : yes
select elitism policy  : 0 (steady GA has no mechanics to handle select elitism)
replace elitism policy: yes
GA generation gap    : 0 (steady GA has no mechanics to handle gap)
GA rank-biased pressure      : 0
GA crossover rate            : 0.9
GA mutation rate             : 0.6
GA mutation method           : mutation single chrom. immediately
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function                 : steady-state
GA selection                : uniform-random
GA crossover                : 2xp-crossover
GA mutation                 : simple-flip
GA replace                  : replace-parent
GA reports :
report type                 : Minimal
report-interval             : 330
-------------------end GA system configuration-----------------------
```

We have tested steady-state GA on pool sizes of 20, 40, 60, 80, 90, 110, 120 and the other parameters have been kept unchanged. The results are in Table 4.7 and 4.8.

P-size and Convergence (Steady-State GA)
(the average best fitness among ten runs)

# OF ITER	P-SIZE 20	P-SIZE 40	P-SIZE 60	P-SIZE 80	P-SIZE 90	P-SIZE 110	P-SIZE 120
0	22869.0	21472.1	20467.8	21484.7	21328.4	20835.0	20162.7
330	12028.6	14414.0	14272.8	14646.5	12822.5	14765.6	15308.6
660	9309.5	11360.0	11898.6	11801.9	9770.3	11980.1	12896.3
990	9309.5	9770.8	10127.0	9987.3	8917.3	10357.9	10866.2
1320	9309.5	9051.3	9401.6	9089.2	8475.7	9332.3	10184.7
1650	9309.5	8680.9	8975.7	8899.1	8475.7	8903.6	9263.6
1980	9309.5	8603.3	8716.7	8697.8	8475.7	8711.5	8729.4
2310	9309.5	8603.3	8666.1	8604.6	8475.7	8601.9	8623.9
2640	9309.5	8603.3	8592.1	8537.4	8475.7	8576.2	8585.7
2970	9309.5	8603.3	8592.1	8470.4	8475.7	8612.6	8639.7
3300	9309.5	8603.3	8592.1	8470.4	8475.7	8570.8	8716.5

Table 4.7

P-size and Convergence (Steady-State GA)
(the result of first turn among ten runs)

# OF ITER	P-SIZE 20	P-SIZE 40	P-SIZE 60	P-SIZE 80	P-SIZE 90	P-SIZE 110	P-SIZE 120
0	23637.6	22460.2	22460.2	19622.2	19622.2	19622.2	19622.2
330	11873.4	13267.6	14891.4	15111.2	15009.2	15250.7	14382
660	9098.3	10160.6	12330.7	11849.4	12163.1	11924.6	12191.4
990	9098.3	9196.4	10291.7	10636.9	10382.6	10243.3	10924.2
1320	9098.3	8475.7	9038.7	9332.9	9343.9	9357.4	10232.4
1650	9098.3	8475.7	8687.5	8877.8	8864.7	8703.5	9212.8
1980	9098.3	8475.7	8536.4	8719.4	8576.7	8691.7	8838.6
2310	9098.3	8475.7	8536.4	8470.4	8576.7	8660.5	8516.8
2640	9098.3	8475.7	8536.4	8470.4	8576.7	8654.4	8475.7
2970	9098.3	8475.7	8536.4	8470.4	8576.7	8654.4	8475.7
3300	9098.3	8475.7	8536.4	8470.4	8576.7	8654.4	8475.7

Table 4.8

From Table 4.7 and 4.8 we can see that when pool sizes are relatively small, steady-state GA converges very quickly on so-called local optima due to lack of genetic diversity. When pool sizes are too big, steady-state GA retards convergence or flounders around some local optima. Here once again we have illustrated the discussion about the relation between pool size and pool convergence.

We conclude that, from the above six tables and the discussions, the ideal pool size of 14 terminal ROP problem is range from 80 to 100.

Selection Operator and Convergence

The results of the above discussions showed that, in general, maintaining genetic diversity of the of GA is extremely important, especially at the early evolution stage (genetic searching stage), for facilitating the appearance of the global optima in the future. Thus the population size must not be too small; it should be balanced between accommodating enough genetic diversity of the pool on the one hand and also not causing too much slowness in converging of pool on the other hand. Random selection is preferred over fitness-biased or rank-biased selection at the early searching stage when the initial population is randomly initialized, since it is not wise to apply bias too early and too often to curb the random search around the solution space at the early searching stage. Due to the matrix representation of solution there is no way to guarantee that "good" chromosomes produce "good" children or not; perhaps "bad" chromosomes can produce "better" children too. So the pool should have room to accommodate some "weak" chromosomes. Actually we suggest using random operations at an early stage to do random searching around the solution space and tuning GA system parameters such as pool size, crossover rate and mutation rate, etc., then apply bias on searching by biased selection and replacement, thus accelerating the convergence of the pool. We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of three GA models on the ROP (Railroad Operating Plan) instance of 14 terminals.

The following is the test run result of generational GA that will illustrate the above discussion. The GA configuration is as follows.

```
-------------------begin GA system configuration-------------------
GA control center configuration information:
--------------------------------------------------
basic parameters:
random-seed                 : 1
Initial pool entered        :  randomly
allowed max. pool size      : 100
actual pool size            : 82
defined chrom. matrix dim   : 14
allowed max. trials         : 220 iterations, ignore convergence
minimize optimization       : yes
select elitism policy       : transfer two copies of the best
replace elitism policy      : yes
```

```
GA generation gap          : 0.1
GA rank-biased pressure    : 0.4 (used by rank-biased selector)
GA crossover rate          : 0.9
GA mutation rate           : 0.6
GA mutation method         : mutation single chrom. immediately
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function                : generational
GA selection               : uniform-random, fitness-biased, rank-biased
GA crossover               : 2xp-crossover
GA mutation                : simple-flip
GA replace                 : append
GA reports :
report type                : Minimal
report-interval            : 10
-------------------end GA system configuration-----------------------
```

We first use random selection and other registered genetic operators to tune pool size, crossover rate, and mutation rate to be 100, 0.9 and 0.6. The global optimum 8470.4 first appears in the 81st iteration (see Table 4.10). Then we change from random selection to biased selection. Fitness-biased selection makes the GA converge to a local optimum, but rank-biased selection facilitates the global optimum 8470.4 first appearance a little bit earlier, in the 71st iteration (see Table 4.10), by applying rank-biased selection pressure 0.4. It is interesting that when the pressure is small (< 0.5) or big (>0.7) the rank-biased selection accelerates convergence of the pool to the global optimum and also the same situation happened to all three models.

Selection Operator and Convergence (generational GA)
(the average best fitness among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (0.4)
0	20767.8	20767.8	20767.8
10	15837.1	16044.6	15672.8
20	13814.8	13573.9	13037.9
30	11930.4	11724.0	10964.5
40	10341.4	10495.9	9595.9
50	9526.2	9584.2	9046.9
60	9130.3	9029.1	8803.8
70	8888.9	8832.6	8640.0
80	8816.3	8768.6	8603.7

90	8712.4	8710.0	8599.1
100	8677.4	8708.8	8596.2
110	8666.3	8703.9	8596.2
120	8660.9	8703.9	8596.2
130	8660.9	8703.9	8596.2
140	8660.9	8703.9	8596.2
150	8660.9	8703.9	8596.2
160	8660.9	8703.9	8596.2
170	8660.9	8703.9	8596.2
180	8660.9	8703.9	8596.2
190	8660.9	8703.9	8596.2
200	8660.9	8703.9	8596.2
210	8660.9	8703.9	8596.2
220	8660.9	8703.9	8596.2

Table 4.9.

Selection Operator and Convergence (generational GA)
(the result of first turn among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (0.4)
0	19622.2	19622.2	19622.2
10	16320.9	15474.8	15749.3
20	13845.3	12846.3	12684.5
30	11725.9	11655.4	10578.4
40	9919.5	10314.2	9870
50	8966.2	9288.9	9213.4
60	8754.6	8821.7	8835.9
70	8595.6	8760.2	8554.4
80	8485.9	8760.2	8470.4
90	8470.4	8760.2	8470.4
100	8470.4	8760.2	8470.4
110	8470.4	8760.2	8470.4
120	8470.4	8760.2	8470.4
130	8470.4	8760.2	8470.4
140	8470.4	8760.2	8470.4
150	8470.4	8760.2	8470.4
160	8470.4	8760.2	8470.4
170	8470.4	8760.2	8470.4
180	8470.4	8760.2	8470.4
190	8470.4	8760.2	8470.4
200	8470.4	8760.2	8470.4
210	8470.4	8760.2	8470.4
220	8470.4	8760.2	8470.4

Table 4.10.

We show the effects of the rank-biased selection pressure (also called the bias pressure) on convergence of the pool below by running the generational GA on the same GA configuration shown above.

Bias Pressure and Convergence (generational GA)
(the average best fitness among ten runs)

# OF ITER	BIAS=0.2	BIAS=0.4	BIAS=0.6	BIAS=0.7	BIAS=0.9
0	20767.8	20767.8	20767.8	20767.8	20767.8
10	15102.0	15672.8	15068.1	15212.4	14620.4
20	12711.7	13037.9	12324.9	12173.8	11690.5
30	10891.1	10964.5	10387.2	10327.2	9806.9
40	9692.3	9595.9	9328.5	9404.9	9033.1
50	8954.8	9046.9	8879.5	8832.9	8821.7
60	8745.3	8803.8	8783.8	8717.5	8730.8
70	8708.4	8640.0	8701.6	8646.0	8692.4
80	8634.9	8603.7	8693.5	8643.2	8692.4
90	8634.9	8599.1	8682.5	8638.6	8692.4
100	8613.5	8596.2	8682.5	8638.6	8692.4
110	8613.5	8596.2	8682.5	8638.6	8692.4
120	8613.5	8596.2	8682.5	8638.6	8692.4
130	8613.5	8596.2	8682.5	8638.6	8692.4
140	8613.5	8596.2	8682.5	8638.6	8692.4
150	8613.5	8596.2	8682.5	8638.6	8692.4
160	8613.5	8596.2	8682.5	8638.6	8692.4
170	8613.5	8596.2	8682.5	8638.6	8692.4
180	8613.5	8596.2	8682.5	8638.6	8692.4
190	8613.5	8596.2	8682.5	8638.6	8692.4
200	8613.5	8596.2	8682.5	8638.6	8692.4
210	8613.5	8596.2	8682.5	8638.6	8692.4
220	8613.5	8596.2	8682.5	8638.6	8692.4

Table 4.11.

Bias Pressure and Convergence (generational GA)
(the result of first turn among ten runs)

# OF ITER	BIAS=0.2	BIAS=0.4	BIAS=0.6	BIAS=0.7	BIAS=0.9
0	19622.2	19622.2	19622.2	19622.2	19622.2
10	15918.1	15749.3	14040.4	14871.8	15352.6
20	13694	12684.5	11450.5	11433.2	12163
30	11039.5	10578.4	9439.6	9205.2	9648.8
40	9520.7	9870	8979.8	8816.2	8831.9
50	9086.6	9213.4	8547.9	8592.1	8705.6
60	9086.6	8835.9	8532.4	8592.1	8705.6
70	9034.1	8554.4	8532.4	8592.1	8705.6
80	8684.8	8470.4	8532.4	8592.1	8705.6
90	8684.8	8470.4	8532.4	8592.1	8705.6
100	8470.4	8470.4	8532.4	8592.1	8705.6

110	8470.4	8470.4	8532.4	8592.1	8705.6
120	8470.4	8470.4	8532.4	8592.1	8705.6
130	8470.4	8470.4	8532.4	8592.1	8705.6
140	8470.4	8470.4	8532.4	8592.1	8705.6
150	8470.4	8470.4	8532.4	8592.1	8705.6
160	8470.4	8470.4	8532.4	8592.1	8705.6
170	8470.4	8470.4	8532.4	8592.1	8705.6
180	8470.4	8470.4	8532.4	8592.1	8705.6
190	8470.4	8470.4	8532.4	8592.1	8705.6
200	8470.4	8470.4	8532.4	8592.1	8705.6
210	8470.4	8470.4	8532.4	8592.1	8705.6
220	8470.4	8470.4	8532.4	8592.1	8705.6

Table 4.12.

We see from Table 4.12 that when the bias pressure is less than 0.5, the pool converges to the global optimum very quickly (the first appearance of the global optimum 8470.4 is in the 100th, and 71st iterations respectively).

The following is the test run result of traditional GA that will illustrate the above discussion also. The GA configuration is as follows.

```
------------------begin GA system configuration------------------
GA control center configuration information:
--------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     :  randomly
allowed max. pool size   : 100
actual pool size         : 82
defined chrom. matrix dim: 14
allowed max. trials      : 220 iterations, ignore convergence
minimize optimization    : yes
select elitism policy    : transfer two copies of the best
replace elitism policy   : yes
GA generation gap        : 0.1
GA rank-biased pressure  : 1.0 (used by rank-biased selector)
GA crossover rate        : 0.7
GA mutation rate         : 0.5
GA mutation method       : mutation whole pool from chrom. level
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function              : traditional
GA selection             : uniform-random, fitness-biased, rank-biased
GA crossover             : random-flip
GA mutation              : simple-flip
GA replace               : replace-parent
GA reports :
```

```
report type          : Minimal
report-interval      : 10
--------------------end GA system configuration-----------------------
```

We first use random selection and other registered genetic operators to tune the pool size, crossover rate, and mutation rate to be 100, 0.7 and 0.5 to achieve a result of the global optimum 8470.4 first appearance being in the 63rd iteration (see Table 4.14). Then we change from random selection to biased selection: fitness-biased or rank-biased. Fitness-biased selection makes GA converge to a local optimum, but rank-biased selection facilitates the global optimum 8470.4 first appearance much earlier, in the 36th iteration (see Table 4.14), by applying rank-biased selection pressure 1.0. An interesting fact is that when the pressure is small (< 0.5) or big (>0.7) rank-biased selection accelerates the convergence of the pool on the global optimum and also the same situation happened to all three models.

Selection Operator and Convergence (traditional GA)
(the average best fitness among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (1.0)
0	20865.2	20865.2	20526.6
10	14833.1	15482.0	12959.2
20	12259.1	12949.3	10385.2
30	10703.0	11189.6	9233.4
40	9667.9	9906.2	8599.6
50	9074.1	9303.1	8560.9
60	8781.7	8916.8	8541.9
70	8644.7	8746.4	8541.9
80	8622.8	8653.9	8541.9
90	8620.1	8644.8	8541.9
100	8613.4	8621.7	8541.9
110	8601.6	8593.5	8541.9
120	8599.8	8593.5	8541.9
130	8599.8	8593.5	8541.9
140	8571.2	8593.5	8541.9
150	8568.3	8593.5	8541.9
160	8568.3	8593.5	8541.9
170	8568.3	8593.5	8541.9
180	8568.3	8593.5	8541.9
190	8554.2	8593.5	8541.9
200	8554.2	8593.5	8541.9

210	8553.8	8593.1	8541.9
220	8553.8	8593.1	8541.9

Table 4.13.

Selection Operator and Convergence (traditional GA)
(the result of first turn among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (1.0)
0	19622.2	19622.2	19622.2
10	15514.7	16107.7	13674.2
20	12865.5	11941.2	10707.6
30	11656.1	10902.2	9285.8
40	10672.7	9847.2	8470.4
50	10000.1	8948.5	8470.4
60	8684.8	8676.8	8470.4
70	8470.4	8544.8	8470.4
80	8470.4	8544.8	8470.4
90	8470.4	8544.8	8470.4
100	8470.4	8544.8	8470.4
110	8470.4	8544.8	8470.4
120	8470.4	8544.8	8470.4
130	8470.4	8544.8	8470.4
140	8470.4	8544.8	8470.4
150	8470.4	8544.8	8470.4
160	8470.4	8544.8	8470.4
170	8470.4	8544.8	8470.4
180	8470.4	8544.8	8470.4
190	8470.4	8544.8	8470.4
200	8470.4	8544.8	8470.4
210	8470.4	8544.8	8470.4
220	8470.4	8544.8	8470.4

Table 4.14.

We show the effects of rank-biased selection pressure (also called the bias pressure) on the convergence of the pool below by running the traditional GA on the same GA configuration shown above.

Bias Pressure and Convergence (traditional GA)
(the average best fitness among ten runs)

# OF ITER	BIAS=0.0	BIAS=0.3	BIAS=0.5	BIAS=0.6	BIAS=0.7	BIAS=1.0
0	20526.6	20829.9	20829.9	20829.9	20829.9	20526.6
10	16455.4	14573.9	14475.5	13541.3	13957.9	12959.2
20	13369.8	11892.6	11868.8	11037.8	10840.7	10385.2
30	11686.5	10114.7	10081.0	9492.3	9477.7	9233.4
40	9753.4	9129.8	9097.0	8847.4	8757.0	8599.6

50	9103.3	8732.1	8853.7	8760.1	8645.1	8560.9
60	8708.4	8626.9	8755.9	8699.7	8621.3	8541.9
70	8666.6	8580.4	8734.9	8683.5	8618.2	8541.9
80	8666.6	8580.4	8734.9	8673.3	8618.2	8541.9
90	8666.6	8579.6	8724.1	8662.5	8618.2	8541.9
100	8666.6	8579.6	8724.1	8662.5	8618.2	8541.9
110	8666.6	8579.6	8724.1	8662.5	8618.2	8541.9
120	8666.6	8579.6	8724.1	8662.5	8581.8	8541.9
130	8666.6	8579.6	8724.1	8662.5	8581.8	8541.9
140	8666.6	8579.6	8724.1	8662.5	8581.8	8541.9
150	8666.6	8579.6	8724.1	8662.5	8581.8	8541.9
160	8666.6	8579.6	8724.1	8662.5	8581.8	8541.9
170	8640.0	8579.6	8724.1	8662.5	8581.8	8541.9
180	8633.4	8579.6	8724.1	8662.5	8581.8	8541.9
190	8623.8	8579.6	8724.1	8662.5	8581.8	8541.9
200	8623.1	8579.6	8724.1	8662.5	8542.0	8541.9
210	8623.1	8579.6	8724.1	8662.5	8542.0	8541.9
220	8585.1	8579.6	8724.1	8662.5	8542.0	8541.9

Table 4.15

Bias Pressure and Convergence (traditional GA)
(the result of first turn among ten runs)

# OF ITER	BIAS=0.0	BIAS=0.3	BIAS=0.5	BIAS=0.6	BIAS=0.7	BIAS=1.0
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
10	16737.3	14708.5	15315.5	15231.2	14878.8	13674.2
20	13462.9	11730.4	12228	11373.2	10765.7	10707.6
30	11583.6	10438.8	10227.4	9674.9	9490.1	9285.8
40	9844.2	8729.5	9099.7	8993.8	8738.3	8470.4
50	9313.5	8696.8	8861.9	8993.8	8604.8	8470.4
60	8595.6	8615.6	8673.4	8792.8	8485.9	8470.4
70	8470.4	8470.4	8586.4	8711.6	8470.4	8470.4
80	8470.4	8470.4	8586.4	8660.8	8470.4	8470.4
90	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
100	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
110	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
120	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
130	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
140	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
150	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
160	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
170	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
180	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
190	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
200	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
210	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4
220	8470.4	8470.4	8532.4	8606.8	8470.4	8470.4

Table 4.16

We see from Table 4.15 and 4.16 that when the bias pressure is less than 0.5 or bigger than 0.7, pool converges on the global optimum very quickly (the first appearance of the global optimum 8470.4 is in the 67th, 67th, 63th, 36th iterations respectively).

The following is the test run result of steady-state GA that will illustrate the above discussion also. The GA configuration is as follows.

```
-------------------begin GA system configuration-------------------
GA control center configuration information:
------------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     :  randomly
allowed max. pool size   : 80
actual pool size         : 66
defined chrom. matrix dim: 14
allowed max. trials      : run until convergence
minimize optimization    : yes
select elitism policy  : 0 (steady GA has no mechanics to handle select elitism)
replace elitism policy: yes
GA generation gap  : 0 (steady-state GA has no mechanics to handle gap)
GA rank-biased pressure  : 0.3 (used by rank-biased selection)
GA crossover rate        : 0.9
GA mutation rate         : 0.6
GA mutation method       : mutation single chrom. immediately
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function              : steady-state
GA selection             :  uniform-random, fitness-biased, rank-biased
GA crossover             : 2xp-crossover
GA mutation              : simple-flip
GA replace               : replace-parent
GA reports :
report type              : Minimal
report-interval          : 330
--------------------end GA system configuration------------------------
```

We first use random selection and other registered genetic operators to tune the pool size, crossover rate, and mutation rate to be 80, 0.9 and 0.6 to achieve a result of the global optimum 8470.4 first appearance being in the 2255th iteration (see Table 4.18). Then we change from random selection to biased selection: fitness-biased or rank-biased.

Fitness-biased selection makes GA converge to a local optimum, but rank-biased selection facilitates the global optimum 8470.4 first appearance much earlier, in the1514th iteration, by applying rank-biased selection pressure 0.3. An interesting fact is that when the pressure is small (< 0.5) or big (>0.7) the rank-biased selection accelerates the convergence of the pool to the global optimum and also the same situation happened to all three models.

Selection Operator and Convergence (steady-state GA)
(the average best fitness among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (0.3)
0	21484.7	20991.8	20429.1
330	14646.5	14386.6	14652.8
660	11801.9	11795.6	11566.4
990	9987.3	10158.8	9806.7
1320	9089.2	9001.8	8761.3
1650	8899.1	8886.4	8618.8
1980	8697.8	8768.5	8587.9
2310	8604.6	8768.5	8485.9
2640	8537.4	8768.5	8485.9
2970	8470.4	8768.5	8485.9
3300	8470.4	8768.5	8485.9

Table 4.17

Selection Operator and Convergence (steady-state GA)
(the result of first turn among ten runs)

# OF ITER	UNIFORM-RANDOM	FITNESS-BIASED	RANK-BIASED (0.3)
0	19622.2	19622.2	19622.2
330	15111.2	14923.6	15420
660	11849.4	12219.5	12311.6
990	10636.9	10700.8	10020.4
1320	9332.9	8808.6	8911.5
1650	8877.8	8808.6	8470.4
1980	8719.4	8768.5	8470.4
2310	8470.4	8768.5	8470.4
2640	8470.4	8768.5	8470.4
2970	8470.4	8768.5	8470.4
3300	8470.4	8768.5	8470.4

Table 4.18

We show the effects of rank-biased selection pressure (also called the bias pressure) on the convergence of the pool below by running the steady-state GA on the same GA configuration shown above.

Bias Pressure and Convergence (steady-state GA)
(the average best fitness among ten runs)

# OF ITER	BIAS=0.1	BIAS=0.3	BIAS=0.5	BIAS=0.6	BIAS=0.7	BIAS=0.9
0	20698.7	20429.1	20430.2	20013.4	21132.4	19985.6
330	13866.9	14652.8	14122.3	13534.4	13276.7	13033.3
660	11482.1	11566.4	11343.7	10770.5	10471.0	10146.6
990	9812.3	9806.7	9646.2	9494.9	9238.1	9134.4
1320	9017.5	8761.3	8782.8	9106.4	8813.4	8872.3
1650	8750.9	8618.8	8679.6	8953.3	8737.6	8675.9
1980	8580.4	8587.9	8677.3	8923.6	8766.8	8645.7
2310	8690.5	8485.9	8547.9	8729.6	8672.0	8645.7
2640	8690.5	8485.9	8547.9	8619.9	8470.4	8538.2
2970	8539.4	8485.9	8547.9	8619.9	8470.4	8470.4
3300	8539.4	8485.9	8547.9	8619.9	8470.4	8470.4

Table 4.19

Bias Pressure and Convergence (steady-state GA)
(the result of first turn among ten runs)

# OF ITER	BIAS=0.1	BIAS=0.3	BIAS=0.5	BIAS=0.6	BIAS=0.7	BIAS=0.9
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
330	14688.7	15420	14981.6	14516	14109.5	14202.5
660	12197.7	12311.6	11734.3	11757.8	11799.1	11392.5
990	1007	10020.4	9334.3	10618.3	9944.2	9647.6
1320	8934	8911.5	8871.4	9900.1	9317.9	8902.1
1650	8621.5	8470.4	8503.4	9293.1	8935.7	8621.5
1980	8470.4	8470.4	8503.4	9203.9	8935.7	8470.4
2310	8470.4	8470.4	8503.4	8729.6	8684.8	8470.4
2640	8470.4	8470.4	8503.4	8619.9	8470.4	8470.4
2970	8470.4	8470.4	8503.4	8619.9	8470.4	8470.4
3300	8470.4	8470.4	8503.4	8619.9	8470.4	8470.4

Table 4.20

We see from Table 4.19 and 4.20 that when the bias pressure is less than 0.5 or bigger than 0.7, the pool converges to the global optimum very quickly (the first appearance of the global optimum 8470.4 is in the 1794th, 1514th, 2625th, and 1794th iterations respectively).

After this discussion our suggestion is that when we have little knowledge about the solution space and the convergence of pool we had better use non-biased random mechanics to search and probe around to tune the system parameters such as pool size, crossover rate, mutation rate, etc.. After we get some knowledge of the solution space, we can resort to biased mechanics to cause GA to converge on the global optimum quickly.

Elitism and Convergence

In our implementation of GA, we have distinguished the selection elitism from the replacement elitism. The main reasons are for flexibility of GA choices and no conflicts on integrating SA into GA by introducing an accepting function into the mutation and crossover interfaces.

The prime ideal of selection elitism is to make sure that the best fit chromosome can survive through genetic selection, and the prime ideal of replacement elitism is to make sure that the best fit chromosome does not die out in the next generation due to the genetic operations of crossover and mutation. Both elitisms are very important to make sure that once the global optimum appears in pool, it will never disappear.

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of generational GA model on the ROP (Railroad Operating Plan) instance of 14 terminals. The GA configuration is as follows.

```
------------------begin GA system configuration------------------
GA control center configuration information:
------------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered      :  randomly
allowed max. pool size   : 100
actual pool size         : 82
defined chrom. matrix dim: 14
allowed max. trials      : 220 iterations, ignore convergence
minimize optimization     : yes
GA generation gap        : 0.1
GA rank-biased pressure  : 0.0
GA crossover rate        : 1.0
GA mutation rate         : 0.4
GA mutation method       : mutation whole pool from chrom. level
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function              : generational
GA selection             : uniform-random
GA crossover             : 2xp-crossover
GA mutation              : simple-flip
GA replace               : append
GA reports :
report type              : Minimal
report-interval           : 10
-------------------end GA system configuration-----------------------
```

The following test results show how bad the performance of GA if it lacks at least one of two elitisms.

If the selection elitism flag is off (the GA has no selection elitism), we are not sure if the best fit chromosome can be selected for the crossover or not, so there is no guarantee for the best fit chromosome to distribute its genetic genes through the next generation. That will be a disadvantage in genetic search, as is shown on the column 3 and 5 of Table 4.22.

If the replacement elitism flag is off (the GA has no replacement elitism), it may happen that the worse child replaces the better parent, there is no guarantee that the next iteration in the search will produce a better solution, and it is possible that the fitness of next iteration is even worse, as is perfectly shown in the column 4 and 5 of Table 4.21 and 4.22.

Elitists and Convergence (generational GA)
(the average best fitness among ten runs)

# of iter	se_elit=1, re_elit=1	se_elit=0, re_elit=1	se_elit=1, re_elit=0	se_elit=0, re_elit=0
0	20517.0	20472.0	20517.0	20472.0
10	16147.5	16202.4	18435.7	21727.5
20	13429.5	13643.9	17361.9	19754.7
30	11347.8	11898.4	16205.6	21268.0
40	10218.2	10652.1	15496.8	21109.2
50	9390.3	9880.2	14930.4	20809.7
60	9043.8	9371.5	14340.5	20626.4
70	8920.8	9131.8	13686.6	21173.4
80	8641.5	8907.6	12761.5	21273.7
90	8616.4	8811.1	12464.1	21561.4
100	8580.4	8804.7	11860.5	20715.3
110	8580.4	8749.4	11411.7	20916.7
120	8580.4	8738.6	10998.7	20822.5
130	8580.4	8737.4	10493.8	21411.2
140	8580.4	8724.8	10098.8	21021.2
150	8580.4	8676.4	9752.88	21501.1
160	8580.4	8676.4	9673.2	20931.9
170	8580.4	8676.4	9717.1	21443.3
180	8580.4	8676.4	9388.6	21171.9
190	8580.4	8676.4	9353.1	20986.7
200	8580.4	8676.4	9380.2	21376.7
210	8580.4	8641.6	9361.0	20981.1
220	8580.4	8641.6	9341.4	20783.9

Table 4.21

Elitists and Convergence (generational GA)
(the result of first turn among ten runs)

# of iter	se_elit=1, re_elit=1	se_elit=0, re_elit=1	se_elit=1, re_elit=0	se_elit=0, re_elit=0
0	19622.2	19622.2	19622.2	19622.2
10	15707.1	15743.6	16333	22199.3
20	13331.4	14311.9	14069.6	20873.9

30	11086.8	12110.5	12100.2	20688
40	9846.7	10803.4	10766.3	21288.8
50	9151.3	10481.2	10273	21333.8
60	8933.5	9286.5	9539.6	19713.8
70	8727.6	9138.2	9539.6	20528.2
80	8595.6	8831.2	9029.2	20078.5
90	8470.4	8831.2	8974.2	19942.8
100	8470.4	8831.2	9029.2	18694
110	8470.4	8831.2	9029.2	20113.6
120	8470.4	8831.2	9029.2	18843
130	8470.4	8831.2	9029.2	20721.6
140	8470.4	8831.2	9029.2	19616.2
150	8470.4	8831.2	9029.2	19838.3
160	8470.4	8831.2	9029.2	18862.6
170	8470.4	8831.2	9029.2	19695.8
180	8470.4	8831.2	9029.2	20393.1
190	8470.4	8831.2	8974.2	18576.7
200	8470.4	8831.2	8974.2	18608.3
210	8470.4	8831.2	8974.2	19290.8
220	8470.4	8831.2	8974.2	18028.6

Table 4.22

The graph of column 2 and 5 as follow:

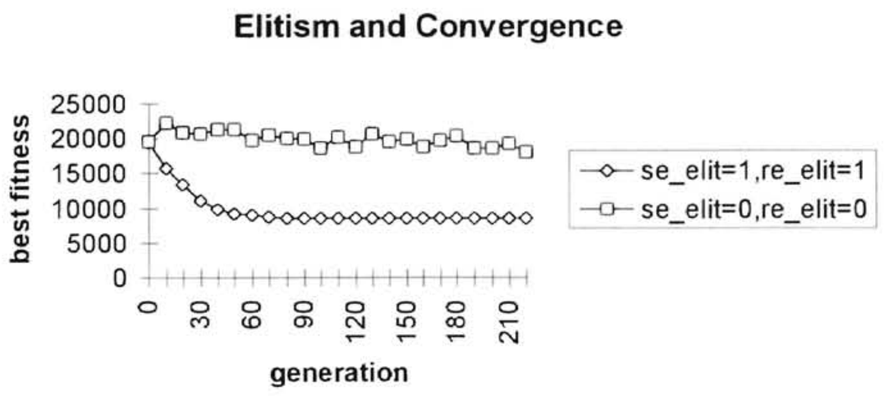


Figure 4.1

Crossover Rate and Convergence

The crossover rate is the probability that controls the frequency of crossover operations during the genetic processing. The GA mainly depends on crossover operations to distribute genetic genes, spread out the diversity of pool, and bring up new chromosomes to the next generation. Therefore the crossover rate is usually very high compared with the mutation rate. In our test run, it shows that the best performance is achieved when the crossover rate is 1.0.

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of generational GA model on the ROP (Railroad Operating Plan) instance of 14 terminals. The GA configuration is as follows.

```
------------------begin GA system configuration------------------
GA control center configuration information:
-----------------------------------------------------
basic parameters:
random-seed                 : 1
Initial pool entered        :  randomly
allowed max. pool size      : 100
actual pool size            : 82
defined chrom. matrix dim:  14
allowed max. trials         : 220 iterations, ignore convergence
minimize optimization       : yes
select elitism policy       : transfer two copies of the best
replace elitism policy      : yes
GA generation gap           : 0.1
GA rank-biased pressure     : 0.0
GA crossover rate           : test on 1.0, 0.8, 0.5, 0.2
GA mutation rate            : 0.5
GA mutation method          : mutation whole pool from chrom. level
probability to invoke SAM:  0
probability to invoke SAC :  0
GA registered operations:
GA function                 : generational
GA selection                : uniform-random
GA crossover                : 2xp-crossover
GA mutation                 : simple-flip
GA replace                  : append
GA reports :
report type                 : Minimal
report-interval             : 10
-------------------end GA system configuration----------------------
```

Crossover rate and Convergence (generational GA)
(the average best fitness among ten runs)

# of iter	x_rate=1.0	x_rate=0.8	x_rate=0.5	x_rate=0.2
0	20758.0	19755.5	20373.5	20248.5
10	15844.9	15488.7	16527.7	17360.2
20	12988.2	12982.4	13898.2	15218.2
30	11424.0	11047.2	12003.5	13213.9
40	10326.7	9845.9	10447.2	11581.1
50	9700.3	9047.0	9284.9	10535.9
60	9180.6	8782.5	9023.6	9695.7
70	9088.5	8654.2	8862.2	9278.6
80	8983.8	8654.2	8812.5	9060.0
90	8832.6	8654.2	8800.2	8806.1
100	8673.0	8654.2	8689.9	8806.1
110	8640.9	8654.2	8689.9	8806.1
120	8640.9	8654.2	8689.9	8806.1
130	8640.9	8654.2	8689.9	8806.1
140	8636.1	8654.2	8689.9	8786.6
150	8636.1	8654.2	8689.9	8766.9
160	8636.1	8654.2	8656.2	8766.9
170	8636.1	8654.2	8656.2	8736.6
180	8636.1	8654.2	8656.2	8728.0
190	8636.1	8654.2	8656.2	8728.0
200	8636.1	8649.2	8656.2	8713.8
210	8636.1	8649.2	8653.1	8713.8
220	8636.1	8649.2	8653.1	8685.7

Table 4.23

Crossover rate and Convergence (generational GA)
(the result of first turn among ten runs)

# of iter	x_rate=1.0	x_rate=0.8	x_rate=0.5	x_rate=0.2
0	19622.2	19622.2	19622.2	19622.2
10	15929.2	14981.3	15328.3	17531.8
20	13784.5	11736.2	13627.9	15612.4
30	12417.1	10236.1	11994.9	13347.3
40	10647.8	9291.7	10479.3	11252.5
50	9520.7	8880.5	9381.6	9902.7
60	9277.5	8730.8	8868.1	9098.1
70	9088.0	8730.8	8868.1	8818.9
80	8913.4	8730.8	8868.1	8818.9
90	8835.4	8730.8	8806.6	8818.9
100	8470.4	8730.8	8806.6	8818.9
110	8470.4	8730.8	8806.6	8818.9
120	8470.4	8730.8	8806.6	8818.9
130	8470.4	8730.8	8806.6	8818.9
140	8470.4	8730.8	8806.6	8818.9
150	8470.4	8730.8	8806.6	8818.9
160	8470.4	8730.8	8806.6	8818.9

170	8470.4	8730.8	8806.6	8818.9
180	8470.4	8730.8	8806.6	8818.9
190	8470.4	8730.8	8806.6	8818.9
200	8470.4	8705.6	8806.6	8818.9
210	8470.4	8705.6	8806.6	8818.9
220	8470.4	8705.6	8806.6	8818.9

Table 4.24

From Table 4.23 and 4.24 we can see that when the crossover rate is too small, it is hard for crossover operations to take effect to produce and bring new individuals into the pool, so the pool is quick to converge on a local optimum. As the crossover rate increases, the crossover operations bring up more new individuals and new solution space regions are being searched, and this facilitates the appearance of the global optimum.

Mutation Rate and Convergence

The mutation operator flips bits and can be a source of new bits or diversity. Since mutation is a random walk through the search space, the heavy use of mutation will degrade the effect of crossover. It must be used sparingly; that is, the mutation rate should be small when compared with the crossover rate, but it cannot be too small, as our test run results illustrate.

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of generational GA model on the ROP (Railroad Operating Plan) instance of 14 terminals. The GA configuration is as follows..

```
------------------begin GA system configuration------------------
GA control center configuration information:
------------------------------------------------------
random-seed              : 1
Initial pool entered     :  randomly
allowed max. pool size   : 100
actual pool size         : 82
```

```
defined chrom. matrix dim: 14
allowed max. trials       : 220 iterations, ignore convergence
minimize optimization     : yes
select elitism policy     : transfer two copies of the best
replace elitism policy    : yes
GA generation gap         : 0.1
GA rank-biased pressure   : 0.0
GA crossover rate         : 1.0
GA mutation rate          : test on 0.05, 0.1, 0.2, 0.3, 0.4, 0.6
GA mutation method        : mutation whole pool from chrom. level
probability to invoke SAM: 0
probability to invoke SAC : 0
GA function               : generational
GA selection              : uniform-random
GA crossover              : 2xp-crossover
GA mutation               : simple-flip
GA replace                : append
report type               : Minimal
report-interval           : 10
--------------------end GA system configuration------------------------
```

Mutation rate and Convergence (generational GA)
(the average best fitness among ten runs)

# of iter	mu_rate=0.05	mu_rate=0.1	mu_rate=0.2	mu_rate=0.3	mu_rate=0.4	mu_rate=0.6
0	19928.4	20542.7	20583.4	20517.0	20389.2	19560.6
10	15851.6	16114.4	15789.1	16147.5	15432.8	15583.2
20	13910.0	14117.4	13423.2	13429.5	12492.4	13026.1
30	12992.3	13150.7	12011.2	11347.8	10844.6	11509.4
40	12995.5	12231.2	11012.9	10218.2	9846.8	10320.7
50	11895.4	11133.0	9994.9	9390.3	9067.2	9579.5
60	11483.9	10604.2	9312.8	9043.8	8805.3	9080.7
70	11217.7	10248.8	9013.9	8920.8	8703.7	8898.3
80	10601.8	9709.3	8905.4	8641.5	8545.2	8818.3
90	10288.2	9476.7	8890.7	8616.4	8541.8	8709.5
100	10019.0	9358.6	8785.9	8580.4	8509.2	8709.5
110	9700.2	9313.2	8685.4	8580.4	8509.2	8692.1
120	9529.2	9173.8	8685.4	8580.4	8509.2	8692.1
130	9334.3	9053.0	8685.4	8580.4	8509.2	8692.1
140	9188.6	9053.0	8685.4	8580.4	8509.2	8692.1
150	9121.5	9039.8	8685.4	8580.4	8509.2	8692.1
160	9092.5	8968.6	8685.4	8580.4	8509.2	8692.1
170	9090.6	8902.0	8685.4	8580.4	8509.2	8692.1
180	9064.6	8875.6	8685.4	8580.4	8509.2	8692.1
190	9042.7	8841.3	8685.4	8580.4	8509.2	8692.1
200	9042.7	8841.3	8685.4	8580.4	8509.2	8692.1
210	8996.6	8841.3	8685.4	8580.4	8509.2	8692.1
220	8970.0	8830.5	8685.4	8580.4	8509.2	8692.1

Table 4.25

Mutation rate and Convergence (generational GA)
(the result of first turn among ten runs)

# of iter	mu_rate=0.05	mu_rate=0.1	mu_rate=0.2	mu_rate=0.3	mu_rate=0.4	mu_rate=0.6
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
10	16095.3	17384.6	15505.6	16115.8	15707.1	15702.5
20	14765.3	13798.4	13051.7	12672.5	13331.4	12178.9
30	13952.2	12844.7	11884.9	11218.0	11086.8	10701.2
40	12905.7	11694.7	10717.5	10106.4	9846.7	9702.2
50	12137.1	10412.3	9294.8	9619.4	9151.3	9465.4
60	11773.7	10050.1	8777.2	9038.5	8933.5	9006.1
70	11427.1	9853.4	8777.2	8841.6	8727.6	8892.1
80	10774.1	8760.2	8777.2	8826.7	8595.6	8621.5
90	10774.1	8760.2	8748.5	8684.9	8470.4	8621.5
100	10263.7	8760.2	8748.5	8684.9	8470.4	8621.5
110	10263.7	8760.2	8748.5	8684.9	8470.4	8621.5
120	9765.5	8760.2	8748.5	8684.9	8470.4	8621.5
130	9445.2	8760.2	8748.5	8684.9	8470.4	8621.5
140	9445.2	8760.2	8748.5	8684.9	8470.4	8621.5
150	9445.2	8760.2	8748.5	8684.9	8470.4	8621.5
160	9445.2	8760.2	8748.5	8684.9	8470.4	8621.5
170	9445.2	8760.2	8748.5	8684.9	8470.4	8621.5
180	9315.6	8760.2	8748.5	8684.9	8470.4	8621.5
190	9315.6	8760.2	8748.5	8684.9	8470.4	8621.5
200	9315.6	8760.2	8748.5	8684.9	8470.4	8621.5
210	9085.2	8760.2	8748.5	8684.9	8470.4	8621.5
220	8952.0	8760.2	8748.5	8684.9	8470.4	8621.5

Table 4.26

From Table 4.25 and 4.26 we can see that when the mutation rate is too small, it is hard for mutation operations to take effect to introduce new genes and diversity into the pool and bring up new individuals into the pool, so the pool is retarded around local optima (see columns 2, 3, 4, 5). As the mutation rate increases, the mutation operations introduce some new genes sparsely into the pool to conduct a genetic search which may accidentally reach some rare and unusual region of solution space, and this facilitates the appearance of the global optimum (see column 6). But if the mutation rate is too high, and mutation operations are applied too often, they will (perhaps) degenerate the results achieved by the crossover operations (see column 7).

Replacement Operator and Convergence

Replacement operators are a counterpart of selection operators in the GA processing. They are divided into two categories of biased or not biased. If replace elitism is off, the *replace by append, replace parent,* and *replace random* are all non-bias replacement operators; the *replace by rank, replace first weaker, replace weakest* are bias replacement operators. The discussions here are similar to those about selection operators; in order to maintain genetic diversity of pool during the early searching stage, bias-free replacement operators are preferred over biased replacement operators. It is not wise to apply bias too early and in too big a hurry to sweep "weak" individuals out of the pool and destroy the diversity of pool at an early searching stage. Due to the matrix representation of solutions, there is no way to guarantee that "good" chromosomes produce "good" children or not; perhaps "bad" chromosomes can produce "better" children. So the pool should have room to accommodate some "weak" chromosomes and let them stay alive for a while at the early search stage. Actually we suggest the use of bias-free operations at an early stage to utilize maximally the diversity of pool to do random searching around the solution space and tuning system parameters such as pool size, crossover rate and mutation rate, etc.. Then apply bias on searching by biased selection and replacement, thus accelerating the convergence of the pool. We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of traditional GA and steady-state GA on the ROP (Railroad Operating Plan) instance of 14 terminals.

The following is the test run result of traditional GA that will illustrate the above discussion. The GA configuration is as follows.

```
------------------begin GA system configuration------------------
GA control center configuration information:
-------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     : randomly
allowed max. pool size   : 130
actual pool size         : 108
defined chrom. matrix dim: 14
allowed max. trials      : 220 iterations, ignore convergence
minimize optimization    : yes
select elitism policy    : transfer two copies of the best
replace elitism policy   : yes
GA generation gap        : 0.1
GA rank-biased pressure  : 0
GA crossover rate        : 0.7
GA mutation rate         : 0.5
GA mutation method       : mutation whole pool from chrom. level
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function              : traditional
GA selection             : uniform-random
GA crossover             : random-flip
GA mutation              : simple-flip
GA replace               : replace-parent, random, rank, weaker, weakest
GA reports :
report type              : Minimal
report-interval          : 10
-------------------end GA system configuration----------------------
```

Replace Operators and Convergence (traditional GA)
(the average best fitness among ten runs)

# OF ITER	PARENT	RANDOM	RANK	WEAKER	WEAKEST
0	20155.8	20301.2	20155.8	20155.8	20155.8
10	15108.7	16351.3	14771.8	16244.6	14403.3
20	12320.9	13699.3	11757.7	13610.4	11315.4
30	10870.1	12347.4	9982.7	11767.0	9730.6
40	9563.9	11244.5	9033.4	10213.4	9149.1
50	9029.7	10236.4	8769.9	9134.1	8841.9
60	8824.3	9360.4	8705.4	8844.3	8649.7
70	8673.4	9043.1	8705.4	8761.9	8606.9

80	8639.1	8800.4	8705.4	8738.5	8606.9
90	8599.5	8718.3	8705.4	8738.5	8606.9
100	8599.5	8654.2	8705.4	8738.5	8606.9
110	8599.5	8597.9	8705.4	8738.5	8606.9
120	8599.5	8593.9	8705.4	8738.5	8606.9
130	8599.5	8568.8	8705.4	8738.5	8606.9
140	8599.5	8568.8	8705.4	8738.5	8606.9
150	8598.7	8568.8	8705.4	8738.5	8606.9
160	8586.9	8561.4	8705.4	8738.5	8606.9
170	8586.9	8561.4	8705.4	8738.5	8606.9
180	8586.9	8561.4	8704.6	8738.5	8606.9
190	8586.9	8561.4	8704.6	8738.5	8606.9
200	8586.9	8561.4	8704.6	8738.5	8606.9
210	8586.9	8561.4	8704.6	8738.5	8606.9
220	8560.6	8561.4	8704.6	8738.5	8573.2

Table 4.27

Replace Operators and Convergence (traditional GA)
(the result of first turn among ten runs)

# OF ITER	PARENT	RANDOM	RANK	WEAKER	WEAKEST
0	19622.2	19622.2	19622.2	19622.2	19622.2
10	13546.7	15706.3	13695.1	15939.6	12837.4
20	11136.4	13747.4	10654.6	13989.8	10719.5
30	10185.4	12238.9	9412.8	11631.2	9332.3
40	9380.7	11195.4	8576.7	9717.3	9159.8
50	9179.5	10208.3	8576.7	9129.5	9146
60	8920.3	9567.3	8576.7	8948.9	9146
70	8687.5	9058.5	8576.7	8851.5	9146
80	8536.4	9058.5	8576.7	8797.1	9146
90	8536.4	8913.4	8576.7	8745.9	9146
100	8536.4	8913.4	8576.7	8745.9	9146
110	8536.4	8718.4	8576.7	8745.9	9146
120	8470.4	8604.4	8576.7	8745.9	9146
130	8470.4	8584.4	8576.7	8745.9	9146
140	8470.4	8539.4	8576.7	8745.9	9146
150	8470.4	8536.4	8576.7	8745.9	9146
160	8470.4	8536.4	8576.7	8745.9	9146
170	8470.4	8536.4	8576.7	8745.9	9146
180	8470.4	8536.4	8576.7	8745.9	9146
190	8470.4	8536.4	8576.7	8745.9	9146
200	8470.4	8532.4	8576.7	8745.9	9146
210	8470.4	8532.4	8576.7	8745.9	9146
220	8470.4	8470.4	8576.7	8745.9	9146

Table 4.28

From Table 4.27 and 4.28 we can see that when bias-free replacement operators: *replace parent,* and *replace random* are used, the diversity of the pool is maintained, and the global optimum 8470.4 appears within the first 220 iterations. When biased replace operators: *replace rank, replace first weaker, replace weakest* are used, they destroy the diversity of the pool by always sweeping "weak" individuals out of the pool; thus the search scope is limited and it is hard for the global optimum to appear, and the pool is quick to converge on some local optimum.

The same discussions are also true for the steady-state GA; we just give the test run result below.

```
------------------begin GA system configuration------------------
GA control center configuration information:
--------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     : randomly
allowed max. pool size   : 80
actual pool size         : 66
defined chrom. matrix dim: 14
allowed max. trials      : run until convergence
minimize optimization    : yes
select elitism policy : 0 (steady GA has no mechanics to handle select elitism)
replace elitism policy: yes
GA generation gap : 0 (steady-state GA has no mechanics to handle gap)
GA rank-biased pressure  : 0
GA crossover rate        : 0.9
GA mutation rate         : 0.6
GA mutation method       : mutation single chrom. immediately
probability to invoke SAM: 0
probability to invoke SAC : 0
GA registered operations:
GA function              : steady-state
GA selection             : uniform-random
GA crossover             : 2xp-crossover
GA mutation              : simple-flip
GA replace               : replace-parent, random, rank, weaker, weakest
GA reports :
report type              : Minimal
report-interval          : 330
-------------------end GA system configuration----------------------
```

OKLAHOMA STATE UNIVERSITY

Replace Operators and Convergence (steady-state GA)
(the average best fitness among ten runs)

# OF ITER	PARENT	RANDOM	RANK	WEAKER	WEAKEST
0	21484.7	20890.2	19795.1	20591.3	20597.2
330	14646.5	16028.5	14670.7	16639.9	13772.2
660	11801.9	13471.0	11412.6	13561.9	10361.3
990	9987.3	12301.6	9255.7	11926.4	9761.3
1320	9089.2	10742.1	9255.7	10603.5	9761.3
1650	8899.1	10050.7	9255.7	9588.0	9761.3
1980	8697.8	9459.1	9255.7	9588.0	9761.3
2310	8604.6	9204.2	9255.7	9588.0	9761.3
2640	8537.4	8942.8	9255.7	9588.0	9761.3
2970	8470.4	8756.3	9255.7	9588.0	9761.3
3300	8470.4	8641.2	9255.7	9588.0	9761.3
3630	8470.4	8641.2	9255.7	9588.0	9761.3

Table 4.29

Replace Operators and Convergence (steady-state GA)
(the result of first turn among ten runs)

# OF ITER	PARENT	RANDOM	RANK	WEAKER	WEAKEST
0	19622.2	19622.2	19622.2	19622.2	19622.2
330	15111.2	15905.8	15169.8	16225.7	12480.9
660	11849.4	13819.3	11345.9	13623.3	9374.8
990	10636.9	13226.7	10243.6	12188.7	9104.9
1320	9332.9	11559.8	10243.6	11635.0	9104.9
1650	8877.8	10178.6	10243.6	11635.0	9104.9
1980	8719.4	9384.6	10243.6	11635.0	9104.9
2310	8470.4	9011.9	10243.6	11635.0	9104.9
2640	8470.4	8914.7	10243.6	11635.0	9104.9
2970	8470.4	8877.8	10243.6	11635.0	9104.9
3300	8470.4	8746.1	10243.6	11635.0	9104.9
3630	8470.4	8746.1	10243.6	11635.0	9104.9

Table 4.30

SAM and SAC and Convergence

SAGA is actually pure GA integrated with the simulated annealing mutation (SAM) and crossover (SAC). The difference between SAM (or SAC) and GA mutation (or crossover) operator is that there is an *accepting function* to decide if the child can be accepted or not, depending on the *scheduled annealing* (temperature). If the pure GA replace elitism flag is on, GA always chooses the best two from two parents and two children; that is, the inferior children are always denied acceptance in the new generation. If SAM and/or SAC are invoked (simultaneously, pure GA replace elitism is disabled), the inferior children may be accepted in the new generation depending on the *scheduled annealing* and a randomly generated probability. Therefore SAM and/or SAC are the alternative of pure GA's replace elitism, and they give more benefit than pure GA's replace elitism to maintain the genetic diversity of pool and facilitate the appearance of the global optimum.

A variety of hybrid modes of SAGA is implemented in our program and is tuned by the values of the SAM invoking probability *sam* and the SAC invoking probability *sac*. If we want to run the algorithm in a SAGA mode, the algorithm should invoke SAM and/or SAC by checking the randomly generated probabilities against the invoking probabilities *sam* and/or *sac* specified by the system configuration file. If the values of the invoking probabilities *sam* and/or *sac* are too small (< 0.5), it is very hard for the algorithm SAGA to invoke SAM and/or SAC to replace pure GA mutation and crossover operators and at this time pure GA replacement elitism is also disabled. Thus SAGA is running most of time under the circumstance of no replacement elitism, and there is a high risk that the pool is retarded and degenerated by "good" parents being replaced by the "bad" children very often (we just allow a few of these cases to happen for the good of the diversity of pool, but cannot tolerate a lot), so the pool is expected either to be retarded or to converge on some local optimum. But if the invoking probabilities *sam* and/or *sac* are too

OKLAHOMA STATE UNIVERSITY

big (>0.7), it will be very easy for the algorithm SAGA to invoke SAM and/or SAC, so SAGA mainly depends SAM and/or SAC for genetic evolution. The *accepting function* and *scheduled annealing* will play a major role in improving the performance of the SAGA; therefore the problem-related designing of the *accepting function* and *scheduled annealing* become extremely important in the genetic searching and converging of pool. That is a quite difficult task if we only have a little knowledge about the solution space of the specified problem. Therefore our suggestion is that the invoking probabilities *sam* and/or *sac* should be set to be not too small (< 0.5) and also not too big (> 0.7) for the best possible performance of SAGA based on the current available, not question specifically designed *accepting function* and *scheduled annealing* (temperature).

We will illustrate the discussion above by showing the average result of ten runs and making an explanation on the first turn of ten runs of three SAGA models on the ROP (Railroad Operating Plan) instance of 14 terminals.

The following is the test run result of generational SAGA that will illustrate the above discussion. The SAGA configuration is below.

```
------------------begin GA system configuration------------------
GA control center configuration information:
--------------------------------------------------------
basic parameters:
random-seed               : 1
Initial pool entered      :  randomly
allowed max. pool size    : 130
actual pool size          : 108
defined chrom. matrix dim : 14
allowed max. trials       : 220 iterations, ignore convergence
minimize optimization     : yes
select elitism policy     : transfer two copies of the best
replace elitism policy    : yes (sam=sac=0), no(sam>0, or sac>0)
GA generation gap         : 0.1
GA rank-biased pressure   : 0.0
GA crossover rate         : 1.0
GA mutation rate          : 0.4
GA mutation method        : mutation whole pool from chrom. level
```

```
probability to invoke SAM:  test on 0.0, 0.1, 0.3, 0.5, 0.7, 0.9
probability to invoke SAC :  test on 0.0, 0.1, 0.3, 0.5, 0.7, 0.9
GA registered operations:
GA function              : generational
GA selection             : uniform-random
GA crossover             : 2xp-crossover
GA mutation              : simple-flip
GA replace               : append
GA reports :
report type              : Minimal
report-interval          : 10
-------------------end GA system configuration----------------------
```

SAM, SAC and Convergence (generational GA)
(the average best fitness among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.1, sac=0.1	sam=0.3, sac=0.3	sam=0.5, sac=0.5	sam=0.7, sac=0.7	sam=0.9, sac=0.9
0	20741.9	20563.7	19920.3	20603.2	20174.0	20326.8
10	14944.9	17578.9	14484.6	15583.7	15513.7	15620.3
20	12566.5	15822.4	11734.3	13212.6	12560.5	13550.9
30	10987.5	14700.6	10268.8	11732.7	10710.5	12324.7
40	9768.7	14008.7	9576.8	10729.1	9391.3	11530.3
50	9103.4	12909.4	9179.6	9932.7	9022.8	10551.9
60	8806.0	12416.3	8898.7	9252.0	8903.0	9859.3
70	8688.9	11630.2	8749.1	9089.0	8861.4	9065.0
80	8621.5	10998.3	8600.1	8895.6	8761.9	9038.3
90	8621.5	10293.7	8588.7	8739.6	8738.8	8823.4
100	8621.5	10061.4	8588.7	8729.9	8698.1	8743.4
110	8621.5	9567.3	8588.7	8657.3	8651.9	8689.6
120	8621.5	9442.8	8588.7	8657.3	8651.9	8689.6
130	8621.5	9194.6	8588.7	8645.7	8651.9	8681.1
140	8621.5	9121.3	8588.7	8629.0	8651.9	8681.1
150	8621.5	9049.6	8588.7	8571.9	8651.9	8681.1
160	8621.5	8979.1	8588.7	8560.5	8651.9	8681.1
170	8621.5	8904.7	8588.7	8560.5	8651.9	8681.1
180	8621.5	8848.5	8588.7	8560.5	8651.9	8677.6
190	8616.4	8731.9	8588.7	8560.5	8651.9	8671.1
200	8606.2	8647.7	8586.6	8560.5	8634.0	8661.2
210	8587.3	8642.1	8586.6	8560.5	8603.7	8661.2
220	8587.3	8642.1	8586.6	8560.5	8603.7	8661.2

Table 4.31

SAM, SAC and Convergence (generational GA)
(the result of first turn among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.1, sac=0.1	sam=0.3, sac=0.3	sam=0.5, sac=0.5	sam=0.7, sac=0.7	sam=0.9, sac=0.9
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
10	15733.5	17441.2	13848.0	16914.3	14076.0	13953.3

OKLAHOMA STATE UNIVERSITY

20	13419.8	15543.1	12322.0	13910.8	11605.0	10143.5
30	11627.5	15407.1	11222.7	12221.5	10273.6	9177.5
40	10639.6	15407.1	10736.0	11088.8	9341.5	9129.3
50	10043.3	13680.8	9811.7	10370.7	9042.1	8895.1
60	9056.8	12791.7	8849.5	9541.2	8911.3	8596.1
70	8827.4	12172.7	8495.7	9386.4	8911.3	8596.1
80	8641.4	11308.8	8495.7	8999.5	8659.2	8596.1
90	8641.4	10829.6	8475.7	8676.8	8659.2	8596.1
100	8641.4	10829.6	8475.7	8650.0	8659.2	8596.1
110	8641.4	10175.4	8475.7	8604.8	8659.2	8596.1
120	8641.4	9564.8	8475.7	8604.8	8659.2	8596.1
130	8641.4	9155.0	8475.7	8595.6	8659.2	8596.1
140	8641.4	9117.0	8475.7	8595.6	8659.2	8596.1
150	8641.4	9117.0	8475.7	8470.4	8659.2	8596.1
160	8641.4	9117.0	8475.7	8470.4	8659.2	8596.1
170	8641.4	9104.9	8475.7	8470.4	8659.2	8596.1
180	8641.4	8826.7	8475.7	8470.4	8659.2	8596.1
190	8615.6	8684.9	8475.7	8470.4	8659.2	8596.1
200	8564.8	8684.9	8475.7	8470.4	8659.2	8596.1
210	8470.4	8684.9	8475.7	8470.4	8659.2	8596.1
220	8470.4	8684.9	8475.7	8470.4	8659.2	8596.1

Table 4.32

From Table 4.31 and 4.32 we can see that when *sam=0, sac=0*, SAGA is in the pure GA mode and replacement elitism does take effect on the genetic evolution, and the global optimum finally appears on the 208th iteration. But when *sam* and *sac* are set to be nonzero and too small (< 0.5), SAGA hardly invokes SAM and SAC and also pure GA replacement elitism is disabled, so SAGA's processing is almost carried out under the circumstance of no replacement elitism supposed to be provided by SAM and SAC, the pool is very likely to be retarded or converged on some local optima. When *sam* and *sac* are too big (>0.7), SAGA' processing may be heavily depended on the SAM and SAC, that is, dependent on the problem-specified *accepting function* and *scheduled annealing*, this is another source of problem to affect the performance of SAGA, the pool is very likely to be retarded and converged on some local optima too. When *sam*=0.5, *sac*=0.5, they are not too small, and also not too big, SAGA uses SAM and SAC in a fair frequency, such that the SAGA has achieved a better performance by the pool converging on the global optimum 8470.4 whose first appearance is in the 145th iteration.

The following is the test run result of traditional SAGA that will illustrate the above discussion. The GA configuration is below.

```
------------------begin GA system configuration------------------
GA control center configuration information:
--------------------------------------------------
basic parameters:
random-seed                : 1
Initial pool entered       : randomly
allowed max. pool size     : 100
actual pool size           : 82
defined chrom. matrix dim: 14
allowed max. trials        : 220 iterations, ignore convergence
minimize optimization      : yes
select elitism policy      : transfer two copies of the best
replace elitism policy     : yes (sam=sac=0), no(sam>0, or sac>0)
GA generation gap          : 0.1
GA rank-biased pressure    : 0
GA crossover rate          : 0.7
GA mutation rate           : 0.5
GA mutation method         : mutation whole pool from chrom. level
probability to invoke SAM: test on 0.0, 0.2, 0.4, 0.7, 0.8, 1.0
probability to invoke SAC : test on 0.0, 0.2, 0.4, 0.7, 0.8, 1.0
GA registered operations:
GA function                : traditional
GA selection               : fitness-biased
GA crossover               : random-flip
GA mutation                : simple-flip
GA replace                 : replace-parent
GA reports :
report type                : Minimal
report-interval            : 10
-------------------end GA system configuration----------------------
```

SAM, SAC and Convergence (traditional GA)
(the average best fitness among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.2, sac=0.2	sam=0.4, sac=0.4	sam=0.7, sac=0.7	sam=0.8, sac=0.8	sam=1.0, sac=1.0
0	20829.9	20873.2	20959.1	19787.6	20625.1	20306.5
10	15253.8	16735.1	15946.1	14298.1	15031.0	13796.1
20	12023.2	14067.8	12870.0	11999.9	12487.8	11574.4
30	10717.0	12572.8	11313.7	10553.0	10775.2	9980.4
40	9494.6	11381.1	9890.9	9730.0	9836.6	9084.8
50	8951.5	10517.6	9483.5	9196.2	9143.7	8869.6
60	8640.6	9989.3	9050.8	8930.4	8967.0	8726.0

70	8592.2	9553.3	8855.6	8621.2	8832.8	8655.0
80	8592.2	9309.8	8818.9	8556.8	8746.7	8618.9
90	8592.2	9089.2	8756.9	8529.4	8696.8	8618.9
100	8592.2	8854.0	8661.3	8529.4	8640.8	8618.9
110	8579.0	8661.6	8628.1	8529.4	8587.6	8618.9
120	8575.3	8627.6	8564.7	8529.4	8587.6	8618.9
130	8575.3	8599.4	8564.7	8529.4	8587.6	8618.9
140	8545.3	8599.4	8564.7	8529.4	8587.6	8618.9
150	8539.6	8583.5	8564.7	8529.4	8587.6	8618.9
160	8539.6	8576.6	8564.7	8529.4	8587.6	8605.7
170	8539.6	8576.6	8564.7	8529.4	8587.6	8605.7
180	8539.6	8554.1	8564.7	8529.4	8587.6	8605.7
190	8511.3	8554.1	8564.7	8529.4	8587.6	8605.7
200	8511.3	8548.6	8564.7	8529.4	8587.6	8605.7
210	8510.5	8534.4	8564.7	8529.4	8587.6	8605.7
220	8510.5	8519.6	8564.7	8529.4	8587.6	8605.7

Table 4.33

SAM, SAC and Convergence (traditional GA)
(the result of first turn among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.2, sac=0.2	sam=0.4, sac=0.4	sam=0.7, sac=0.7	sam=0.8, sac=0.8	sam=1.0, sac=1.0
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
10	16107.7	16986.1	15951.9	15079.5	15079.5	14044.2
20	11941.2	15646.4	13655.6	13042.4	12909.6	11654.3
30	10902.2	13225.3	10874.3	11172.3	11313.2	10184.9
40	9847.2	11787.8	9783.5	10221.7	10187.4	9155.8
50	8948.5	10434.1	9236.1	9551.4	9295.7	8947.5
60	8676.8	10028.1	8887.4	8983.5	9040.6	8780.5
70	8544.8	9842.1	8887.4	8470.4	8852.2	8703.5
80	8544.8	9679.5	8782.4	8470.4	8852.2	8673.9
90	8544.8	9471.4	8782.4	8470.4	8782.4	8673.9
100	8544.8	9164.3	8736.6	8470.4	8736.6	8673.9
110	8544.8	8613.8	8736.6	8470.4	8684.9	8673.9
120	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
130	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
140	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
150	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
160	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
170	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
180	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
190	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
200	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
210	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9
220	8544.8	8539.4	8684.9	8470.4	8684.9	8673.9

Table 4.3

OKLAHOMA STATE UNIVERSITY

From Table 4.33 and 4.34 we can see that when *sam=0, sac=0*, SAGA is in the pure GA mode and replacement elitism does have effect on the genetic evolution, but the global optimum doesn't appear before the 220th iteration. When *sam* and *sac* are set to be nonzero and too small (< 0.5), SAGA hardly invokes SAM and SAC and also pure GA replacement elitism is disabled, so SAGA's processing is almost carried out under the circumstance of no replacement elitism supposed now to be provided by SAM and SAC. The pool is very likely to be retarded or converged on some local optima. When *sam* and *sac* are too big (>0.7), SAGA processing may be heavily dependent on the SAM and SAC, that is, dependent on the problem-specified *accepting function* and *scheduled annealing*. This is another source of problem to affect the performance of SAGA. The pool is very likely to be retarded and converge on some local optimum. When *sam*=0.7, *sac*=0.7, they are not too small, and also not too big, SAGA resorts to SAM and SAC in a fair frequency, such that the SAGA has achieved a better performance by the pool converging on the global optimum 8470.4 whose first appearance is in the 62th iteration.

The following is the test run result of steady-state SAGA that will illustrate the above discussion also. The GA configuration is below.

```
------------------begin GA system configuration------------------
GA control center configuration information:
------------------------------------------------
basic parameters:
random-seed              : 1
Initial pool entered     : randomly
allowed max. pool size   : 90
actual pool size         : 74
defined chrom. matrix dim: 14
allowed max. trials      : run until convergence
minimize optimization    : yes
select elitism policy : 0 (steady GA has no mechanics to handle select elitism)
replace elitism policy     : yes (sam=sac=0), no(sam>0, or sac>0)
GA generation gap : 0 (steady-state GA has no mechanics to handle gap)
GA rank-biased pressure  : 0
GA crossover rate        : 0.9
GA mutation rate            : 0.6
GA mutation method          : mutation single chrom. immediately
probability to invoke SAM:  test on 0.0, 0.2, 0.4, 0.6, 0.8, 1.0
probability to invoke SAC : test on 0.0, 0.2, 0.4, 0.6, 0.8, 1.0
GA registered operations:
GA function                 : steady-state
GA selection                : uniform-random
GA crossover                : 2xp-crossover
GA mutation                 : simple-flip
GA replace                  : replace-parent
GA reports :
report type                 : Minimal
report-interval             : 330
-------------------end GA system configuration------------------------
```

SAM, SAC and Convergence (steady-state GA)
(the average best fitness among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.2, sac=0.2	sam=0.4, sac=0.4	sam=0.6, sac=0.6	sam=0.8, sac=0.8	sam=1.0, sac=1.0
0	20467.8	19887.4	20849.6	20064.5	20298.1	20510.9
330	14272.8	18175.5	16990.3	15927.0	14542.8	13793.8
660	11898.6	15803.4	13895.0	12668.0	11515.0	10651.1
990	10127.0	14856.1	12501.3	11073.4	10070.5	9494.1
1320	9401.6	13603.7	11100.5	9753.8	9231.4	8973.7
1650	8975.7	12842.2	9946.1	9262.4	8809.7	8760.4
1980	8716.2	11932.8	9454.9	8844.1	8722.9	8760.4
2310	8666.1	11163.2	9086.6	8635.6	8684.1	8760.4
2640	8592.1	10835.0	8869.6	8618.0	8674.9	8760.4
2970	8592.1	10261.9	8701.4	8616.8	8655.0	8760.4
3300	8592.1	10055.4	8641.9	8591.5	8662.7	8760.4
3630	8592.1	10055.4	8641.9	8591.5	8662.7	8760.4

Table 4.35

SAM, SAC and Convergence (steady-state GA)
(the result of first turn among ten runs)

# OF ITER	sam=0.0, sac=0.0	sam=0.2, sac=0.2	sam=0.4, sac=0.4	sam=0.6, sac=0.6	sam=0.8, sac=0.8	sam=1.0, sac=1.0
0	19622.2	19622.2	19622.2	19622.2	19622.2	19622.2
330	15009.2	18715.9	17950.8	15808.8	15294.6	13137.6
660	12163.1	16918.3	15038.8	12675.1	11573.7	10428.0
990	10382.6	15858.9	13250.8	10808.1	10151.3	9253.8
1320	9343.9	14641.4	11811.5	9665.4	9511.4	8828.2
1650	8864.7	13837.3	10148.4	8897.9	8925.3	8828.2
1980	8576.7	12170.3	9708.7	8621.5	8797.3	8828.2
2310	8576.7	12809.9	8947.5	8470.4	8721.6	8828.2
2640	8576.7	11772.5	8683.5	8470.4	8721.6	8828.2
2970	8576.7	11118.0	8532.4	8470.4	8693.1	8828.2
3300	8576.7	10413.3	8532.4	8470.4	8693.1	8828.2

3630	8576.7	10212.0	8532.4	8470.4	8693.1	8828.2

Table 4.36

From Table 4.35 and 4.36 we can see that when *sam=0, sac=0*, SAGA is in pure GA mode and replacement elitism does have an effect on the genetic evolution, but the global optimum doesn't appear before the 3630th iteration; actually the pool converges on the 2310th iteration to a local optimum 8576.7. When *sam* and *sac* are set to be nonzero and too small (< 0.5), SAGA hardly invokes SAM and SAC and also pure GA replacement elitism is disabled, so SAGA processing is almost carried out under the circumstance of no replacement elitism supposed now to be provided by SAM and SAC, and the pool is very likely to be retarded or converged on some local optimum (see column 3 and 4). When *sam* and *sac* are too big (>0.7), SAGA processing may be heavily dependent on the SAM and SAC, that is, dependent on the problem-specified *accepting function* and *scheduled annealing*. This is another source of problem to affect the performance of SAGA. The pool is very likely to be retarded and converge on some local optimum (see column 6, 7). Now when *sam*=0.6, *sac*=0.6, they are not too small, and also not too big, SAGA resorts to SAM and SAC in a fair frequency, such that the SAGA has achieved a better performance by the pool converging on the global optimum 8470.4 whose first appearance is about or before the 2310th iteration.

From the discussion and the analysis of the test data of this section, we suggest that the invoking probabilities *sam* and/or *sac* should be set in the range of 0.4 to 0.7 for the best possible performance of the SAGA.

Now we have finished the discussion and analysis of the test run of the implementation of the SAGA.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

The results and discussions in CHAPTER IV have shown that the steady-state SAGA is the slowest of all methods regarding convergence, traditional SAGA is faster than steady-state SAGA, and generational SAGA is faster than traditional SAGA in general. The results also showed that, in general, maintaining genetic diversity of the generation of SAGA is extremely important, especially at the early searching stage, for facilitating the appearance of the global optimum in the future. Thus the population size cannot be too small; it should be balanced between accommodating enough genetic diversity of pool on the one hand and also not causing too much slowness in converging of the pool on the other hand. Random selection is preferred over fitness-biased or rank-biased selection at an early searching stage when the initial population is randomly initialized since it is not wise to apply bias too early and too often to curb the random search around the solution space at the early searching stage other than hill climbing stage. Due to the matrix representation of the solution there is no way to guarantee that "good" chromosomes produce "good" children. Perhaps "bad" chromosomes can produce "better" children. Our suggestion is that when we have little knowledge about the solution space and the convergence of pool we had better not use biased random mechanics to search and probe around to tune the system parameters such as pool size, crossover rate, mutation rate, etc.. After we get some knowledge of the solution space, we can resort to biased

OKLAHOMA STATE UNIVERSITY

mechanics to expedite the GA to converge on the global optimum quickly. The crossover rate must be much higher than the mutation rate since SAGA depends mainly on crossover to create new individuals and bring up better solutions. The mutation rate usually is far smaller than crossover rate because mutation is only a second source to introduce unusual genetic genes into the pool, and in order to avoid mutation degenerating the result of crossover the mutation operator is used sparsely. Both selection and replacement elitisms are also very important for keeping the generation from degenerating and increasing the chance to distribute "hopeful" genetic genes through generations. SAM and SAC operators are together to implement the scheduled bias replacement mechanics, an alternative of replacement elitism, which contributes to the schema of maintaining genetic diversity of pool at early searching stages to escape from local optima and concentrating on hill-climbing, intending to reject inferior children at late stages to expedite the converging of the pool. SAGA is a very efficient algorithm to obtain the best possible optimal solution of a railroad operating plan in a rather large size of problem.

It is recommended that future study can be conducted to introduce some mechanics into SAGA such that it has the ability to adjust itself during run-time according to the performance and status of the system. In the current implementation, the GA function, genetic operators, and their invoking probabilities are all set at the beginning of program execution according to the system configuration file provided by user, and are used by the system until the termination of algorithm. It will be a good ideal for the system to use different genetic operators and their invoking probabilities during the different stages of searching and hill climbing processes in order to achieve the best performance by random searching as broad as possible during the early stage and focusing on hill climbing to accelerate convergence as quickly as possible during the late stages to make system more robust.

REFERENCES

[1] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 1994.

[2] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley Publishing company, Inc., Reading, MA, 1989.

[3] Jeong, I. K. and Lee, J. J., *Adaptive Simulated Annealing Genetic Algorithm for Control Applications,* International Journal of Systems Science, Vol. 27, No. 2, pp. 241-153, 1996.

[4] Holland, J. H., *Adaptation in Natural and Artificial Systems,* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[5] Whitley, D. and Kauth, J., *GENITOR: A Different Genetic Algorithm,* Proceedings of the Rocky Mountain Conference on Artificial Intelligence, pp. 118-130, Denver, 1988.

[6] Adler, D., *Genetic Algorithms and Simulated Annealing: A Marriage Proposal,* IEEE International Conference on Neural Networks, pp. 1104-1109, 1993.

[7] Bartlett, G., *Genie: A First GA,* Practical Handbook of Genetic Algorithms, Vol.1, pp.31-56, CRC Press, Boca Raton, FL, 1995.

[8] Aarts, E. and Korst, J., *Simulated Annealing and Boltzmann Machines,* John Wiley & Sons, Chichester, 1989.

[9] Abuali, F. N., *Terminal Assignment in a Communications Network Using Genetic Algorithms,* Proceedings of 22nd annual ACM Computer Science Conference, Phoenix, Arizona, 1994.

[10] Corcoran, A. L., *Using LibGA to Develop Genetic Algorithms for Solving Combinatorial Optimization Problems,* Practical Handbook of Genetic Algorithms, Vol.1, pp.143-172, CRC Press, Boca Raton, FL, 1995.

[11] Kirkpatrick, S., *Optimization by Simulated Annealing,* Science, Vol. 220, pp. 671-680, May, 1983.

[12] Bertsimas, D., *Simulated Annealing,* Probability and Algorithm, pp. 17-30, National Academy Press, Washington, D.C., 1992.

[13] Lagarias Jeffrey C., *Pseudorandom Numbers,* Probability and Algorithm, pp. 65-86, National Academy Press, Washington, D.C., 1992.

[14] Ramachandran, V., *Randomization in Parallel Algorithms,* Probability and Algorithm, pp. 149-160, National Academy Press, Washington, D.C., 1992.

[15] Shi, F., *Study on Optimization of Unconstrained Railroad Operating Plans in Single Direction.* Journal of The China Railway Society, Vol. 10, No. 2, 1988.

[16] Lawler, E. L., *The Traveling Salesman Problem,* John Wiley & Sons, Chichester, 1985.

[17] Lawler, E. L., *Combinatorial Optimization: Networks and Matriods,* Holt, Rinehart and Winston, New York, 1976.

[18] Zimmermann,U., *Linear and Combinatorial Optimization in Ordered Algebraic Structures,* North-Holland Publishing Company, Amsterdam, 1981.

[19] Martello, S., *Surveys in Combinatorial Optimization,* North-Holland Publishing Company, Amsterdam, 1987.

[20] Bachem, A., *Bonn Workshop on Combinatorial Optimization,* North-Holland Publishing Company, Amsterdam, 1982.

[21] Padberg, M. W., *Mathematical Programming Study 12: Combinatorial Optimization,* North-Holland Publishing Company, Amsterdam, 1980.

[22] Rayward-Smith, V. J., *Mathematical Programming Study 13: Combinatorial Optimization II,* North-Holland Publishing Company, Amsterdam, 1980.

APPENDIX A

A COMBINATORIAL OPTIMIZATION PROGRAM FOR GENETIC ALGORITHMS INTEGRATED WITH SIMULATED ANNEALING TO FIND THE BEST POSSIBLE SOLUTION FOR CONSTRUCTING RAILROAD OPERATING PLANS

BY

HONGJIANG WU

COMPUTER SCIENCE DEPARTMENT

OKLAHOMA STATE UNIVERSITY

STILLWATER, OK 74078

JUNE, 1996

Header file: gah.h

```
/*------------------------------------------------------------------------------------------------------------
| Header files
------------------------------------------------------------------------------------------------------------*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<ctype.h>
#include<string.h>
/*------------------------------------------------------------------------------------------------------------
| Constants only reset in gah.h, no  modification after assign
------------------------------------------------------------------------------------------------------------*/
#define FALSE       0
#define TRUE        !(FALSE)
#define OK          0
#define ERR         !(OK)
#define UNSPECIFIED   -1
/*--- Magic card for validation ---*/
#define NL_CARD       0x0000  /*NULL card*/
#define GA_CARD       0x1111  /*ga_center card*/
#define PL_CARD       0x2222  /*pool card*/
#define CH_CARD       0x3333  /*chrom card*/
/*------------------------------------------------------------------------------------------------------------
| system constants, only reset in gah.h, no  modification after assign
------------------------------------------------------------------------------------------------------------*/
/*------------------------------------------------------------------------------------------------------------
| system constants, after default assigning, may be modified by cfg file.
------------------------------------------------------------------------------------------------------------*/
/*--- Method to initialize the pool ---*//*hex number by leading 0x*/
#define IP_NONE         0x00  /*flag to tell not initialize pool*/
#define IP_FILE         0x01  /*flag to tell initialize pool from file*/
#define IP_RANDOM       0x02  /*flag to tell initialize pool randomly*/
#define IP_INTERACT     0x03  /*flag to tell initialize pool from stdin*/
/*--- Type of output report ---*/
#define RP_NONE         0     /*flag to tell not do report*/
#define RP_MINI         1     /*flag to tell report only stats info*/
#define RP_SHORT        2     /*flag to tell report stats info and best chrom*/
#define RP_LONG         3     /*flag to tell report stats info and generation*/
/*--type of mutation--*/
#define MU_CHILD  0x4444   /*flag: mutation on child immediately  for a single chromosome*/
#define MU_CHROM 0x5555    /*flag: mutate whole pool by selecting chrom. from whole pool */
#define MU_BIT        0x6666  /*flag: mutation whole pool by selecting bits from whole pool*/
/*traditional, generational, and steady_state ga can use all three flags*/
/*--system default configuration  constants in ga_cneter used by gacReset()---*/
#define PLSIZE        100      /*default pool_max_size*/
#define CHROMDIM 10            /*default chromosome matrix dimension*/
#define ELITIST       0.10     /*the percentage of top bests to be transferred from the old  gneration
                                  to the new generation, used by gaElitist1() when se_elitist=2*/
#define GAP           0.0      /*percent of old pool directly copy to new pool*/
/*traditional, generational ga can use gap and steady_state ga can not use */
#define CRITIA  0.000000001  /*pool convergence critia*/
```

OKLAHOMA STATE UNIVERSITY

```
#define BIAS      0.0           /*rank_biased selection pressure*/
/*BIAS=0.0 disable rank_biased select, same as select randomly.
  BIAS=1.0 apply the heaviest rank_bias selection.
The range of ranked_bias pressure: BIAS from 0.0 to 1.0. such that scalared ranked_bias r ranges from
0.0 to 1/((PLSIZE-1)*PLSIZE), such that rk_prb(i)=1/n for any i when BIAS=0.0, and rk_prb(1)=2/n
more heavier than rk_prb(n)=0 when BIAS=1.0.
for reference:
bias=ga_center->bias;              /*ranked_bias pressure on selection
r=(float)((float)2/(n*(n-1)))*bias;   /*scalar ranked_bias
q=(float)((float)r*(n-1))/2+(float)1/n;
/*if bias=0 there is no rked bias at all, rk_prb(i)=(q-i*r)=q=1/n
/*if bias=1 thare is heaviest rked_bias, rk_prb(0)=q-0*r=2/n; rk_prb(n-1)=q-(n-1)*r=0;
/*traditional, generational, and steady_state ga can all use ranked_bias*/
#define XRATE      1.0          /*the prob. for a chrom to be parent to crossover*/
#define MURATE   0.01           /*threefolds: depends on mu_flag;
1) if mu_flag=MU_CHILD, only muRunSingleChrom() can be called and mutation a bit of that new born
child immediately, MURATE is the prob. of that child to undergo a bit mutation of it.
2) if mu_flag=MU_CHROM, only muRunPool() can be called ( it turns to call muRunChromLevel()).
mutation selecting is on chromosome level of whole pool, MURATE is the prob. for a chrom of pool to
undergo a bit mutation of it.
3) if mu_flag=MU_BIT, only muRunPool() can be called ( it turns to call muRunBitLevel()). mutation
selecting is on bit level of whole pool, MURATE is the prob. for a bit of pool to undergo a bit mutation
*/
/*traditional, generational, and steady_state ga can use all three mutation flags*/
/*--------------------------------------------------------------------------------------------------------------
| Type definitions
--------------------------------------------------------------------------------------------------------------*/
/*--- A function pointer ---*/
typedef int  (*FN_PTR)(); /*int type function, return an int pointer */
                          /* FN_PTR is the pointer that points to the entry addr of a function block */
/*--- A general function table ---*/
typedef struct
{  char *name;                    /*function name as a character string*/
   FN_PTR fun;                    /*the address of the function modular entry point*/
} FN_TABLE,*FN_TABLE_PTR;
/*--- A gene ---*/
typedef int GENE_TYPE;            /*each cell of chrom matrix is gene value of int*/
typedef GENE_TYPE* GENE_PTR;  /*each row of int array is pointed by GENE_PTR*/

/*--- A chromosome ---*/
typedef struct
{ int       magic_card;       /*for validation*/
  GENE_PTR* gene;             /*the address of the 1st gene of chrom matrix*/
  int       dim;              /*the dimension of the chromosome matrix*/
  double  fitness;            /*the fitness of the chromosome*/
  int       eva;              /*the flag to tell if chrom need evaluated*/
                              /*only EV_fun() set eva=1, chromReset(), chromRepair(), s.t. any x
                                ops and mu ops set eva=0; check eva flag before call EV_fun()*/
 float    ptf;                /*the percentage of the total fitness*/
 float    rank_prob;          /*the weight for rank-biased selection, assigned by poolRank()
                                according to the rank in pool*/
 int      index;              /*my index in the current pool chromosome array*/
 int      idx_min,idx_max;
 int      parent1,parent2;
 int      xp1,xp2;
} CHROM_TYPE, *CHROM_PTR;

/*--- A pool ---*/
typedef struct
{ int      magic_card;            /*for validation*/
  CHROM_PTR* chrom;               /*the address of an array of chromosomes*/
  int      max_size;              /*max allowed pool size (must be at least 4)*/
  int      size;                  /*actual pool size(must be at most max_size-2)*/
  double tot_fitness;             /*the total fitness of the pool*/
  double min, max, ave;           /*current pool fitness states*/
  double var, dev;                /*variance and devariance of fitness*/
                                  /*a=(a1+...+a9)/9; var=((a1-a)^2+...+(a9-a)^2)/9; dev=sqrt(var);*/
 int min_index, max_index;        /*index of min/max chromosomes*/
 int best_index;                  /*index of the best chromosome*/
/*04/08/96 int minimize;/*flag to tell if minimize optimization [y/n][1/0]*/
/*only ga_center handle wether or not minimize optimization*/
 int sorted;                      /*flag to tell if pool is sorted or not after last change*/
/*sort pool means: seqence chromosome array of pool  from best (0th cell) to worest (array taill). default
set FALSE , any change of pool turns it to FALSE(=0), only poolSort() turns it to TRUE(=1), and only
poolRank() turns it to 2 */
 int updated;                     /*flag to tell if pool stats is updated after last change*/
/*default set FALSE(=0) ,any change of pool turns it to FALSE, only poolStats() turns it to TRUE(=1)*/
} POOL_TYPE, *POOL_PTR;

/*--- GA configuration info center ---*/
typedef struct            /*cfg means the value is given at config time*/
{ /*--- Basic info ---*/  /*run means the value is maintained in run time*/
 int magic_card;     /*for validation*/
 int  rand_seed;     /*cfg: seed for random number generator*/
 int  ip_flag;       /*cfg: initial pool generation method flag*/
 char ip_file[80];   /*cfg: initial pool user data file*/
 int pool_max_size; /*cfg: pool max size for IP_RAND & poolAlloc()
                          pool_max_size at least 4. actural pool size at most pool_max_size-2*/
 int chrom_dim:      /*cfg: chrom matrix dim for IP_RAND & chromAlloc*/
 int max_iter;       /*cfg: of max number of iteration for ga*/
 int iter;           /*run: counter value of actual number of iteration*/
 int minimize;       /*cfg: flag to tell if minimize pool [y/n][1/0]*/
 int converged;      /*run: has ga converged? 1/0 TRUE/FALSE */
 int use_converge;   /*cfg: use convergence critia? 1/0 TRUE/FALSE*/
 int se_elitist;  /*cfg: selection elitism flag:
                         0: disable;  1: two copies of the best; 2: transfer ELITIST percent of top bests*/
 int re_elitist;/*cfg: replacement elitism flag:0/1: dis/enable*/
 float elitist;  /*cfg: percentage of top bests transfered.[0.0,1.0) used by gaElitist2() when se_elitist=2*/
 float gap;         /*cfg: generation gap used by tradi, gener ga*/
 double critia;     /*cfg: small positive number if pool->dev<critia,then pool is converged*/
 float bias;        /*cfg: rank_biased pressure, it is used by poolRank() to compute rank_prob of each
                          chromosome of pool, and rank_prob is used by rank_biased selection*/
 float x_rate;      /*cfg: crossover rate*/
 float mu_rate;  /*cfg: mutation rate,it has two different kind of value:
1) rate for selecting chromosome from pool undergoing mutation, used by muRunChromLevel(),and
muRunSingleChrom(), when mu_flag = MU_CHROM, or MU_CHILD.
2) rate for select bit from pool undergoing mutation. used by muRunBitLevel() when mu_flag=
MU_BIT*/
```

```
 int mu_flag;      /*cfg: mutation selection method flag:
1) MU_CHROM: mutation selection is on chromosome level when mutation a pool.
2) MU_BIT: mutation selection is on bit level when mutation a pool.
3) MU_CHILD: mutation on child immidiately when call mutation interface to a single child chrom.*/
/*traditional, generational, and steady_state ga can use all three flags*/
 float sam;        /*probability to invoke the integrated simulated annealing mutation operator (SAM) to
replace pure GA mutation operator. Inside GA_fun, whenever need mutation, we check random
probability against sam value to determine if invoke SAM or not. sam=0.0: disable SAM, use standard
GA mutation operator. sam=1.0: use SAM, disable standard GA mutation operator. 0.0<sam<1.0: a
variety of hybrid modes. */
 float sac;        /*probability to invoke the integrated simulated annealing crossover operator (SAC) to
replace pure GA crossover operator. inside GA_fun, whenever need xover, we check random probability
against sac value to determine if invoke SAC or not. sac=0.0: disable SAC, use standard GA crossover
operator. sac=1.0: use SAC, disable standard GA crossover operator. 0.0<sac<1.0: a variety of hybrid
modes. */
 /*--- Function ---*/
 FN_PTR  GA_fun;    /*cfg: ga function*/
 FN_PTR  SE_fun;    /*cfg: selection operator*/
 FN_PTR  X_fun;     /*cfg: crossover operator*/
 FN_PTR  MU_fun;    /*cfg: mutation operator*/
 FN_PTR  RE_fun;    /*cfg: replacement operator*/
 FN_PTR  EV_fun;    /*cfg: evaluation function*/

 /*--- Report ---*/
 int  rp_type;            /*cfg: type of output report*/
 int  rp_interval;        /*cfg: output report interval*/
 FILE* rp_fid;            /*cfg: output report fid*/
 char  rp_file[80];       /*cfg: output report file name*/
/*in gacReset() default rp_fid=stdout; rp_file='\0';in gacRead() cfgfile set rp_fid=fopen(...) and
rp_file=...*/
 /*--- Pools ---*/
 POOL_PTR old_pool, new_pool;/*run: old pool & new pool setup in gaSetup(), maintain by gaStats()*/
 /*--- Stats ---*/
 CHROM_PTR* best;     /*run: an array of CHROM_PTR to record the best history list, that array setup
in gaSetup() and after gaConfig() that is after max_iter is read; the length of array is depend on max_iter.
if max_iter=-1(no limit), best has 2 cells; 1st for best, 2nd for NULL. if max_iter>1, array
lenght=max_iter+2; keep best of each iter from 0th to max_iter, max_iter+1 th cell is for NULL
terminator, the array is maintained by gaStats()*/
 int num_mut, tot_mut;   /*run: counter value of actural number of mutation*/
} GA_CENTER_TYPE, *GA_CENTER_PTR;
/*pool_max_size, chrom_dim given by cfg file to specify the max_size of  pool and the dim of chrom
matrix. Both values are used by IP_RAND and IP_FILE flags. pool_max_size is also used by poolAlloc()
before initial pool. initpool.dat does not provide pool_size and chrom_dim, just provide the raw data of
chrom matrix and actual size is recorded according to actually how many chrom matrix is successfully
readed into the pool.*/
/*the ga_center has three kind of data should be maintained:
1) the cfg data, the following should be write once when gaConfig(): rand_seed, ip_flag, ip_file[],
pool_max_size, chrom_dim, max_iter, minimize, se_elitist, re_elitist, elitist, use_converge, critia, bias,
gap, x_rate, mu_rate, sam, sac, mu_flag, GA_fun, SE_fun, X_fun, MU_fun, RE_fun, EV_fun, rp_type,
rp_interval, rp_fid, rp_file[];
2) counter data, the following should be updated during run:  iter, num_mut, tot_mut;
3) status data of ga, the following should be updated after each iteration by gaStats(): converged, best,
old_pool, new_pool */
```

OKLAHOMA STATE UNIVERSITY

```
/*bias: it is the rank-biased selection pressure or called selection pressure used by selection-by-rank other
than selection-by-fitness to avoid supper  chromosome to have a large number of offspring which is likely
happened in selection-by-fitness.*/
/*-----------------------------------------------------------------------------------------------------------
| Pseudo functions
-----------------------------------------------------------------------------------------------------------*/
/*--- Random number in the domain [0..1] ---*/
/*04/15/96 #if defined (_BORLANDC_) 04/15/96*/
#define SRAND(seed) (srand(seed))
#define RANDFRAC()  ((double)rand()/RAND_MAX)
/*04/15/96
#else
#define SRAND(seed) (srandom(seed))
#define RANDFRAC()  ((double)random()*(1.0/2147483647.0))
04/15/96*/
/*--- Random number in the domain [lo..hi] ---*/
#define RANDDOM(lo,hi) ((int)floor(RANDFRAC()*(((hi)-(lo))+0.999999))+(lo))

/*--- Random bit ---*/
#define RANDBIT()  ((RANDFRAC()>=0.5)? 1:0)

/*--- min and max ---*/
#define MIN(a,b)    ((a<b)? (a):(b))
#define MAX(a,b)    ((a>b)? (a):(b))
#define SWAP(a,b)   {int temp; temp=*(a); *(a)=*(b); *(b)=temp;}

/*--- warning and error messages ---*/
/*
#define WARN(message)  {fprintf(stderr,"WARNING: %s\n",message);}
#define ERROR(message) {fprintf(stderr,"ERROR: %s\n",message); exit(1);}
*/
#define WARN(message)  {fprintf(stdout,"WARNING: %s\n",message);}
#define ERROR(message) {fprintf(stdout,"ERROR: %s\n",message); exit(1);}




/*======== end of file: gah.h =========*/
```

OKLAHOMA STATE UNIVERSITY

file: chrom.c

```
/*=====================================================================
| Chromosome operations:
|
| Functions:
|  /*--- chromosome memory setup and default initialization-----------------------------------------------
|  chromValid()  -check to see if a chromosome is valid
|  chromReset()  -reset a chromosome, reset flag eva=0;
|  chromAlloc()  -allocate a chromosome
|  chromResize()-resize a chromosome
|  chromFree()    -free the space of a valid chrom. which passes the validation check chromValid()
|  chromKill()    -free the space of a invalid chromosome which doesn't pass the validation check
|  /*--- chromosome munipulation-------------------------------------------------------------------------------
|  chromCopy()  -copy chromosome over to the another if in pool must recompute pool related data
|                       such  as pool index, ptf, rank_prb, etc. after the call chromCopy().
|  chromComp() -compare two chromosome if they are the same or not.
|  chromPrint()   -print a chromosome
|  chromRepair() -modify chrom matrix cells such that it is a feasible solution; reset flag eva=0;
|  chromVerify() -make sure the chromosome make sense
=====================================================================*/
#include "gah.h"
/*=====================================================================
| function prototype
=====================================================================*/
int chromValid(CHROM_PTR );
void chromReset(CHROM_PTR);
CHROM_PTR chromAlloc(int);
void chromResize(CHROM_PTR, int);/*never be used*/
void chromFree(CHROM_PTR );
void chromKill(CHROM_PTR );
void chromCopy(CHROM_PTR , CHROM_PTR);
int chromComp(GA_CENTER_PTR, CHROM_PTR , CHROM_PTR );
void chromPrint(GA_CENTER_PTR,CHROM_PTR );
void chromRepair(CHROM_PTR );
void chromVerify(GA_CENTER_PTR ga_center,CHROM_PTR chrom );

extern int gacValid(GA_CENTER_PTR);

/*=====================================================================
| chromosome memory setup and default initialization
=====================================================================*/
/*---------------------------------------------------------------------------------------------------------------
| chromValid()    -check to see if a chromosome is valid
---------------------------------------------------------------------------------------------------------------*/
int chromValid(CHROM_PTR chrom)
{  /*--- check for null pointers ---*/
   if(chrom==NULL) return FALSE;
   if(chrom->gene==NULL) return FALSE;
   /*--- check for magic card ---*/
   if(chrom->magic_card != CH_CARD) return FALSE;
   /*--- otherwise valid ---*/
   return TRUE;
}
```

```
/*--------------------------------------------------------------------------------------------------------------
| chromReset()   -reset a chromosome, reset eva=0.
--------------------------------------------------------------------------------------------------------------*/
void chromReset(CHROM_PTR chrom)
{  int i,j;
   /*--- check the validation of chromosome ---*/
   if(!chromValid(chrom))ERROR("chromReset():chromValid() check fails,exit");
   /*--- initialize the gene matrix of chromosome ---*/
   for(i=0;i<=chrom->dim-1;i++)
   {  if(chrom->gene[i]==NULL)/*can also alloc a new array instead of exit*/
        ERROR("chromReset(): try to reset a null gene row, exit");
      for(j=0;j<=chrom->dim-1;j++) chrom->gene[i][j]=0;
   }
   /*--- initialize the parameters of chromosome ---*/
   chrom->fitness  =0.0;
   /*==set flag eva=0 due to chrom gene changes, chromReset(), chromRepair(), any X ops and MU ops
        reset eva=0 FALSE, only EV_fun() set eva=1 TRUE, if eva=1 not need to call EV_fun() again.*/
   chrom->eva        =0;                    /*every time check eva flag first before eva*/
   chrom->ptf        =0.0;
   chrom->rank_prob=0.0;
   chrom->index      =-1;                   /*invalid index in pool at the begainning*/
   chrom->idx_min =0;                       /*the lowest index of gene matrix column*/
   chrom->idx_max=chrom->dim-1; /*the highest index of gene matrix column*/
   chrom->parent1   =-1;
   chrom->parent2   =-1;
   chrom->xp1        =-1;
   chrom->xp2        =-1;
}
/*--------------------------------------------------------------------------------------------------------------
| chromAlloc() -allocate a chromosome
--------------------------------------------------------------------------------------------------------------*/
CHROM_PTR chromAlloc(int length) /*length=dim*/
{  int i;
   CHROM_PTR chrom;
   /*--- error check ---*/
   if(length <= 0) ERROR("chromAlloc(): invalid chromosome length, exit!");
   /*--- allocate the memory for chromosome ---*/
   if((chrom=(CHROM_PTR)calloc(1, sizeof(CHROM_TYPE)))==NULL)
       ERROR("chromAlloc(): alloc chrom fails!");
   chrom->dim=length;
   /*--- allocate the memory for genes ---*/
   if((chrom->gene=(GENE_PTR*)calloc(length, sizeof(GENE_PTR)))==NULL)
      ERROR("chromAlloc(): alloc gene fails!");
   for(i=0;i<=length-1;i++)
      if((chrom->gene[i]=(GENE_PTR)calloc(length, sizeof(GENE_TYPE)))==NULL)
         ERROR("chromAlloc(): alloc gene fails!");
   /*--- put magic card ---*/
   chrom->magic_card=CH_CARD;
   /*--- initialize the chromosome by default reset---*/
   chromReset(chrom);
   return chrom;
}
```

OKLAHOMA STATE UNIVERSITY

```
/*--------------------------------------------------------------------------------------------------------------
| chromResize()  -resize a chromosome we intend not use it because ga_center chrom_dim is determied by
cfg file, never change, agree in whole program
--------------------------------------------------------------------------------------------------------------*/
/*acturally in my program, it is never realloc to expand chrom matrix, because chrom_dim is like
pool_max_size to be frozen in ga_center by cfg file and we choose never to change it. so we never use
chromResize() and poolResize()*/
void chromResize(CHROM_PTR chrom, int length)
{ int i;
 /*--- check the validation ---*/
 if(!chromValid(chrom)) ERROR("chromResize(): invalid chromosome, exit");
 if(length <0) ERROR("chromResize(): invalid chromosome length, exit!");
 /*--- reallocate the memory for the chromosome ---*/
 if((chrom->gene=(GENE_PTR*)realloc(chrom->gene, length*sizeof(GENE_PTR)))==NULL)
   ERROR("chromResize(): realloc gene fails!");
 for(i=0;i<=length-1;i++)
   if((chrom->gene[i]=(GENE_PTR)realloc(chrom->gene[i], length*sizeof(GENE_TYPE)))==NULL)
      ERROR("chromResize(): alloc gene fails!");
 chrom->dim=length;
 /*--- Reset chromosome ---*/
 chromReset(chrom);
}
/*--------------------------------------------------------------------------------------------------------------
| chromFree()-Free the space allocated to a chromosome  which passes the validation check chromValid()
--------------------------------------------------------------------------------------------------------------*/
void chromFree(CHROM_PTR chrom)
{  int i;
   /*--- check the validation ---*/
   if(!chromValid(chrom))ERROR("chromFree():chromValid() check fails, exit");
   /*--- free memory of genes ---*/
   if(chrom->gene!=NULL)              /*double check for safety of free()*/
   { for(i=0;i<=chrom->dim-1;i++)     /*free each row of gene matrix*/
     { if(chrom->gene[i]!=NULL)       /*each row is a integer array*/
       { free(chrom->gene[i]);chrom->gene[i]=NULL;
       }
     }
     free(chrom->gene);chrom->gene=NULL;
   }
   /*--- put in NULL magic card ---*/
   chrom->magic_card=NL_CARD;
   /*--- free memory of chromosome ---*/
   if(chrom!=NULL) { free(chrom);chrom=NULL; } /*double check for safety of free()*/
}
/*--------------------------------------------------------------------------------------------------------------
| chromKill()free the space of a invalid chromosome which doesn't pass the validation check chromValid()
--------------------------------------------------------------------------------------------------------------*/
void chromKill(CHROM_PTR chrom)
{  int i;
   /*--- check the validation ---*/
   if(chrom==NULL)ERROR("chromKill():chrom ptr null, exit");
   /*--- free memory of genes ---*/
   if(chrom->gene!=NULL)              /*double check for safety of free()*/
   { for(i=0;i<=chrom->dim-1;i++)     /*free each row of gene matrix*/
     { if(chrom->gene[i]!=NULL)       /*each row is a integer array*/
```

OKLAHOMA STATE UNIVERSITY

```
        { free(chrom->gene[i]); chrom->gene[i]=NULL;
        }
      }
      free(chrom->gene); chrom->gene=NULL;
   }
   /*--- put in NULL magic card ---*/
   chrom->magic_card=NL_CARD;
   /*--- free memory of chromosome ---*/
   if(chrom!=NULL) { free(chrom); chrom=NULL;}  /*double check for safety of free()*/
}
/*=====================================================================
| chromosome manipulation
=====================================================================*/
/*---------------------------------------------------------------------
| chromCopy() -copy one chromosome to another; if in pool must recompute pool related data such
|                   as pool index, ptf, rank_prb, etc. after the call chromCopy().
---------------------------------------------------------------------*/
void chromCopy(CHROM_PTR src, CHROM_PTR dst)
{  int i;
   GENE_PTR* gene;
   /*--- Error checking ---*/
   if(!chromValid(src)) ERROR("chromCopy(): invalid chromosome, exit");
   if(!chromValid(dst)) ERROR("chromCopy(): invalid chromosome, exit");
   /*--- save destination chrom gene ptr ---*/
   gene=dst->gene;
   /*--- copy chromosome ---*/
   memcpy(dst, src, sizeof(CHROM_TYPE));
   /*--- restore memory pointed to by gene ---*/
   dst->gene=gene;
   /*--- copy gene one row by one row---*/
   for(i=0;i<=src->dim-1;i++) memcpy(dst->gene[i], src->gene[i], src->dim*sizeof(GENE_TYPE));
}
/*---------------------------------------------------------------------
| chromComp()- compare two chromosomes chrom1, chrom2,;
|                   chrom1 better return -1,| chrom2 better return  1, same  return  0
---------------------------------------------------------------------*/
int chromComp(GA_CENTER_PTR ga_center, CHROM_PTR chrom1, CHROM_PTR chrom2)
{ /*--- check validation ---*/
  if(!gacValid(ga_center)) ERROR("chromComp(): invalid ga_center, exit");
  if(!chromValid(chrom1)) ERROR("chromComp(): invalid chromosome, exit");
  if(!chromValid(chrom2)) ERROR("chromComp(): invalid chromosome, exit");
  /*--- compare ---*/
  if(ga_center->minimize)
  { if(chrom1->fitness < chrom2->fitness) return -1;
    else if(chrom1->fitness > chrom2->fitness) return  1;
    else return  0;
  }
  else
  { if(chrom1->fitness > chrom2->fitness) return -1;
    else if(chrom1->fitness < chrom2->fitness) return  1;
    else return  0;
  }
}
```

OKLAHOMA STATE UNIVERSITY

```
/*----------------------------------------------------------------------------------------------------------
| chromPrint()  print a chromosome
----------------------------------------------------------------------------------------------------------*/
void chromPrint(GA_CENTER_PTR ga_center,CHROM_PTR chrom)
{  int i,j;
   FILE* fid;
  /*===check the validation==*/
  if(!gacValid(ga_center))ERROR("chromPrint:gacValid() check fails,exit");
  if(!chromValid(chrom)) ERROR("chromPrint(): invalid chromosome, exit");
  if(ga_center->rp_fid==NULL)ERROR("chromPrint:invalid rp_fid in ga,exit");
  /*===print chrom info to rp_fid==*/
  fid=ga_center->rp_fid;
  /*===print header line===*/
  fprintf(fid,"\n");
  fprintf(fid,"-------------------begin chrom print---");
  fprintf(fid,"---------------------------------------");
  fprintf(fid,"\n");
  /*==print chrom matrix==*/
  fprintf(fid,"chrom matrixm:\n");
  fprintf(fid,"--------------");
  for(i=0;i<chrom->dim;i++)
  { fprintf(fid,"\n");
    for(j=0;j<chrom->dim;j++) fprintf(fid,"%d, ", chrom->gene[i][j]);
  }
  fprintf(fid,"\n\n");
  /*==print fitness info==*/
  fprintf(fid,"fitness=%G, ptf=%G, rank_prob=%G, index=%d\n",
               chrom->fitness,chrom->ptf,chrom->rank_prob,chrom->index);
  /*===print footer line===*/
  fprintf(fid,"\n");
  fprintf(fid,"-------------------end chrom print-----");
  fprintf(fid,"---------------------------------------");
  fprintf(fid,"\n");
  /*==print onto fid immidiately==*/
  fflush(fid);
}
/*----------------------------------------------------------------------------------------------------------
| chromRepair()   -modify chrom matrix cell such that feasible solution
----------------------------------------------------------------------------------------------------------*/
void chromRepair(CHROM_PTR chrom )
{  int i,j;
   /*===check validaion===*/
   if(!chromValid(chrom))ERROR("chromRepair(): invalid chromosome, exit");
   /*===make sure chrom is feasible===*/
   for(i=0;i<=chrom->dim-2;i++) chrom->gene[i][i+1]=1;
   for(i=0;i<=chrom->dim-1;i++) for(j=0;j<=i;j++) chrom->gene[i][j]=0;
/*==set flag eva=0 due to chrom gene changes, chromReset(), chromRepair(), any X ops and MU ops
reset flag eva=0 FALSE, only EV_fun() set eva=1 TRUE, if eva=1, not call EV_fun() again==*/
   chrom->eva=0;     /*every time check eva flag first before eva*/

}
/*----------------------------------------------------------------------------------------------------------
| chromVerify()   -make sure the chromosome make sense
----------------------------------------------------------------------------------------------------------*/
```

```
void chromVerify(GA_CENTER_PTR ga_center,CHROM_PTR chrom )
{ int i,j;
 /*==check validaion==*/
 if(!gacValid(ga_center))ERROR("chromVerify(): invalid ga_center, exit");
 if(!chromValid(chrom))ERROR("chromVerify(): invalid chromosome, exit");
 /*==check chrom dim==*/
 if(ga_center->chrom_dim!=chrom->dim)
   ERROR("chromVerify(): chrom dim not agree w/ ga_center, exit");
 /*==check chrom fitness==*//*defaut reset fitness=0*/
 if(chrom->fitness<0.0)
  ERROR("chromVerify(): invalid chromosome fitness<0.0 , exit");
 /*==check chrom ptf==*//*defaut reset ptf=0*/
 if(chrom->ptf<0.0||chrom->ptf>1.0)
  ERROR("chromVerify(): invalid chromosome ptf<0.0||>1.0, exit");
 /*==check chrom rank_prob==*//*defaut reset rank_prob=0*/
 if(chrom->rank_prob<0.0||chrom->rank_prob>1.0)
  ERROR("chromVerify(): invalid chromosome rank_prob<0.0||>1.0, exit");
 /*==check chrom pool index==*//*defaut reset index=-1*/
 if(chrom->index<-1||chrom->index>=ga_center->pool_max_size)
  ERROR("chromVerify(): invalid chromosome index<-1||>=pl_max_size,exit");
 /*==check min and max index ==*//*defaut reset idx_min=0, idx_max=dim-1*/
 if(chrom->idx_min<0||chrom->idx_min>=chrom->dim)
  ERROR("chromVerify(): invalid chrom idx_min<0||>=dim , exit");
 if(chrom->idx_max<0||chrom->idx_max>=chrom->dim)
  ERROR("chromVerify(): invalid chrom idx_max<0||>=dim, exit");
 if(chrom->idx_min>chrom->idx_max)
  ERROR("chromVerify(): invalid chrom idx_min>idx_max, exit");
 /*==check chrom gene matrix, if it is feasible?==*/
 for(i=0;i<=chrom->dim-2;i++)
   if(chrom->gene[i][i+1]!=1) ERROR("chromVerify(): chromosome matrix diagonal invalid , exit");
 for(i=0;i<=chrom->dim-1;i++)
   for(j=0;j<=i;j++)
     if(chrom->gene[i][j]!=0) ERROR("chromVerify:chrom matrix left low triangle invalid, exit");
}




/*=========== end of file: chrom.c ===========*/
```

OKLAHOMA STATE UNIVERSITY

file: pool.c

```
/*========================================================================
| Pool management
|
|  /*---pool memory setup and pool default initialization-----------------------------------------------------
|  poolValid()---check the validation of a pool
|  poolReset()-default initialize paramaters of the pool, default reset chromosomes if exist, to be spare
|               chroms for future reuse. after poolReset(), pool size=0 even there may exist some
|               spare chroms resided in pool waiting for future reuse. call poolValid(), chromValid(),
|               chromReset(),chromKill().
|  poolAlloc()---allocate a new pool,  poolReset()
|  poolResize() -resize a existed old pool for new use
|  poolFree()   -free a health pool(pass poolValid()).
|  poolKill()   -free a illness pool(pool!=NULL, but fail on poolValid()).
|  /*---pool chromosomes manipulattion-----------------------------------------------------------------------
|  poolRemove()-remove a chrom. from the pool by providing index, if null chrom meeted, abort
|              op, otherwise free the space  and reduce actual pool size by 1, this is the only place to
|              reduce pool size. called by poolClean(),call poolValid(),chromValid(),chromFree(),etc.
|  poolClean()-clean out chromosomes of pool from (including) minidx to maxidx by calling
|               poolRemove() to free spaces & reduce size.
|  poolMove()-move a chrom. in the pool: free chrom at dst if exists,and reduce pool size by 1 by
|             calling poolRemove(). move chrom at src to dst, and NULL src cell.
|  poolSwap()---swap two health chromosomes in the pool
|  poolAlign()---bubble any holes of chrom array of pool to the right side.
|  poolInsert()--insert a chromosome chrom into the pool at location idx,
|                copy=1 TRUE: copy chrom into pool to the old one if exits.
|                copy=0 FALSE: move chrom into pool, free old one if exits.
|  poolAppend() -------append a chromosome to the pool, call poolInsert() to do that.
|  poolCompareMin()-compare two chromosome fitness when minimize pool
|  poolCompareMax()-compare two chromosome fitness when maximize pool
|  poolSort()------------seqence chromosom array of pool from best (0th cell)to worest (array tail)
|  /*--maintaining pool datum---------------------------------------------------------------------------------
|  poolIndex()----update index of each chromosome in the pool.
|  poolFitness()-update fitness of each chrom. of pool, check the eva flag first before call EV_fun().
|  poolPtf()-------update ptf(percent of total fitness) of each chromosome.
|  poolRank()----update rank_prob of each chromosomes in a sorted pool.
|  poolStats()-----update all pool parameters.
|  /*--pool initialization by reading from initial file or randomly--------------------------------------------
|  getNum()-------read num digit string from initpool data file
|  poolRead()-----initial pool by reading chrom matrices from initpool file
|  poolRand()-----initial pool by randomly creating chrom matrices
|  poolInit()--------initial pool manager
========================================================================*/
#include "gah.h"
/*========================================================================
| Data structures
========================================================================*/
/*========================================================================
| function prototype declarations
========================================================================*/
int poolValid(POOL_PTR pool);
void poolReset(POOL_PTR pool);
POOL_PTR poolAlloc(int max_size);
void poolResize(POOL_PTR pool, int new_size);/*never be used*/
```

OKLAHOMA STATE UNIVERSITY

```
void poolFree(POOL_PTR pool);
void poolKill(POOL_PTR pool);
void poolRemove(POOL_PTR pool, int index);
void poolClean(POOL_PTR pool,int minidx,int maxidx);
void poolMove(POOL_PTR pool, int src_idx, int dst_idx);
void poolSwap(POOL_PTR pool, int idx1, int idx2);
void poolAlign(POOL_PTR pool);
void poolInsert(POOL_PTR pool, CHROM_PTR chrom, int idx, int copy);
void poolAppend(POOL_PTR pool, CHROM_PTR chrom, int copy);
/*int poolCompareMin(CHROM_PTR* a, CHROM_PTR* b);
  int poolCompareMax(CHROM_PTR* a, CHROM_PTR* b); */
static int poolCompareMin(const void* a, const void* b);
static int poolCompareMax(const void* a, const void* b);
void poolSort(GA_CENTER_PTR ga_center,POOL_PTR pool);
void poolIndex(POOL_PTR pool);
void poolFitness(GA_CENTER_PTR ga_center,POOL_PTR pool);
void poolPtf(GA_CENTER_PTR ga_center,POOL_PTR pool);
void poolRank(GA_CENTER_PTR ga_center,POOL_PTR pool);
void poolStats(GA_CENTER_PTR ga_center,POOL_PTR pool);
char* getNum(FILE* fid);
void poolRead(POOL_PTR pool,int chrom_dim,FILE* fid);
void poolRand(POOL_PTR pool, int chrom_dim);
void poolInit(GA_CENTER_PTR ga_center, POOL_PTR pool);

extern int chromValid(CHROM_PTR chrom);
extern void chromReset(CHROM_PTR chrom);
extern CHROM_PTR chromAlloc(int length);
extern void chromResize(CHROM_PTR chrom, int length);
extern void chromFree(CHROM_PTR chrom);
extern void chromKill(CHROM_PTR chrom);
extern void chromCopy(CHROM_PTR src, CHROM_PTR dst);
extern void chromRepair(CHROM_PTR chrom);

extern int gacValid(GA_CENTER_PTR ga_center);
/*========================================================================
| pool memory setup and pool default initialization
========================================================================*/
/*------------------------------------------------------------------------------------------------------------
| poolValid() -check the validation of a pool
------------------------------------------------------------------------------------------------------------*/
int poolValid(POOL_PTR pool)
{ /*==check for null point==*/
 if(pool==NULL) return FALSE;
 if(pool->chrom==NULL) return FALSE;
 /*==check for magic card==*/
 if(pool->magic_card != PL_CARD) return FALSE;
 /*==otherwise valid==*/
 return TRUE;
}
/*------------------------------------------------------------------------------------------------------------
| poolReset() -default initialize paramaters of the pool. default reset chromosomes if exist, to be spare
chroms for future reuse. after poolReset(), pool size=0 even there may exist some spare chroms resided in
pool waiting for future reuse. this is the only place to reset pool size to 0. note: pool size is only reset to 0
by poolReset(), reduced by 1 by poolRemove(), increased by 1 by poolAppend(), no any other routions can
```

OKLAHOMA STATE UNIVERSITY

```
modify pool size. set off sorted and updated flags. any change of pool turn them to FALSE, only
poolSort() turn sorted to TRUE(1), only poolRank() set sorted = 2, only poolStats() turn updated to TRUE.
for convenience, move all spare chroms to left side by call poolAlign().
------------------------------------------------------------------------------------------------*/
void poolReset(POOL_PTR pool)
{ int i;
 /*===check the validation of pool===*/
 if(!poolValid(pool))ERROR("poolReset(): poolValid() check fails , exit");
 /*===initialize chromosomes===*/
 /*---initializing every cell of chrom array of a new pool to NULL--*/
 /*---initializing every no null cell(zeroing gene) of chrom array of a old pool--*/
 for(i=0;i<=pool->max_size-1;i++)
 { if(pool->chrom[i]!=NULL)
   { if(chromValid(pool->chrom[i]))
        chromReset(pool->chrom[i]);/*reset a health chrom to be spare chrom*/
     else
     { chromKill(pool->chrom[i]); /*kill(free) a illness chromosome*/
       pool->chrom[i]=NULL;
       /*pool->size-=1*/;/*reduce pool size (counter of spare chroms) by 1*/
       /*pool->size isn't accurate(may be garbage value)for a newly alloc pool*/
       /*pool size will be reset to 0 even there may be some spare chroms*/
     }
   }
 }
 pool->sorted=FALSE;/*false at begin, any change of pool turn it to FALSE
                          only poolSort() turn it to TRUE*//*TRUE=1, FALSE=0*/
 pool->updated=FALSE;/*false at begin, any change of pool turn it to FALSE
                          only poolStats() turn it to TRUE*/
 /*move sorted=updated=false here, in order poolAlign() may execuate*/
 /*===bubble any holes of chrom aaray to the right side===*/
 /*if there be at least one spare chrom, move it to left*/
 for(i=0;i<=pool->max_size-1;i++)
 { if(pool->chrom[i]!=NULL)
   { poolAlign(pool);/*for convenience, move all spare chroms to left side*/
     break;
   }
 }
 /*--make sure chrom pool index is correct in case of bubble moving--*/
 /*poolIndex(pool);*//*no need index for spare chroms. acturally already poolIndex() inside poolAlign()*/
 /*===initialize the parameters of pool===*/
 pool->size=0;            /*actual pool size,max allowed pool size set at poolAlloc()*/
 pool->tot_fitness=0.0;   /*total of all fitness of chromosomes of pool*/
 pool->min=0.0;           /*minimal fitness value among chroms of pool*/
 pool->max=0.0;           /*maximal fitness value among chroms of pool*/
 pool->ave=0.0;           /*average fitness value among chroms of pool*/
 pool->var=0.0;           /*variance of fitness value among chroms of pool*/
 pool->dev=0.0;           /*devariance of fitness value among chroms of pool*/
                          /*a=(a1+...+a9)/9; var=((a1-a)^2+...+(a9-a)^2)/9; dev=sqrt(var);*/
 pool->min_index=-1;      /*cell index of chromosome w/ min fitness in pool*/
 pool->max_index=-1;      /*cell index of chromosome w/ max fitness in pool*/
 pool->best_index=-1;     /*cell index of chromosome w/ best fitness in pool*/
/*4/8/96pool->minimize=TRUE;default to do minimize optimization of fitness*/
}
```

OKLAHOMA STATE UNIVERSITY

```
/*-----------------------------------------------------------------------------------------------------------
| poolAlloc()  -allocate a new pool. call poolValid(),poolReset()
-----------------------------------------------------------------------------------------------------------*/
POOL_PTR poolAlloc(int max_size)
{  int i;
   POOL_PTR pool;
  /*=== error check ===*/
  if(max_size <= 0) ERROR("poolAlloc(): invalid pool max_size, exit!");
  /*=== allocate the memory for pool ===*/
  if((pool=(POOL_PTR)calloc(1, sizeof(POOL_TYPE)))==NULL)ERROR("plAllc: alloc pool fails!");
  pool->max_size=max_size;
  /*=== allocate the memory for chrom array ===*/
  if((pool->chrom=(CHROM_PTR*)calloc(max_size, sizeof(CHROM_PTR)))==NULL)
    ERROR("poolAlloc(): alloc chrom array fails!");
  /*===NULL each cell of chrom array===*/
  for(i=0;i<=max_size-1;i++) pool->chrom[i]=NULL;
  /*=== put magic card ===*/
  pool->magic_card=PL_CARD;
  /*=== initialize pool ===*/
  poolReset(pool);/*for new alloc pool, no any spare chroms,just defaut*/
  return pool;   /*reset the parameters of pool*/
}
/*-----------------------------------------------------------------------------------------------------------
| poolResize() -resize a pool, set off sorted and updated flags. any change of pool turns them to FALSE,
only poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2, only poolStats() turns updated to
TRUE. call poolValid(),poolRemove(),poolClean()
-----------------------------------------------------------------------------------------------------------*/
 /*acturally in my program, it is never realloc to expand pool, because pool_max_size is like chrom_dim
to be frozen in ga_center by cfg file and we choose never to change it. so we never use chromResize() and
poolResize()*/
void poolResize(POOL_PTR pool, int new_size)
{ int old_size;
  int i;
  /*=== check the validation ===*/
  if(!poolValid(pool))ERROR("poolResize():poolValid() check failed, exit");
  if(new_size <= 0)ERROR("poolResize(): invalid pool new_size, exit!");
  old_size=pool->max_size;
 /*===align chrom array in pool===*/
 poolAlign(pool);/*bubble holes to the right side*/
 /*=== free any extra chromosomes if exists against new_size===*/
 if(new_size < old_size)/*keep 0,...,new_size-1;free new_size,...,old_size-1*/
          poolClean(pool,new_size,old_size-1);
 /*=== reallocate the memory for the pool upto the new_size===*/
 if((pool->chrom=(CHROM_PTR*)realloc(pool->chrom,new_size*sizeof(CHROM_PTR)))==NULL)
   ERROR("poolResize(): realloc chrom fails!");
 /* for(i=0;i<=new_size-1;i++)
      if(!(pool->chrom[i]=(CHROM_PTR)realloc(pool->chrom[i],sizeof(CHROM_TYPE))))
        ERROR("poolResize(): alloc chrom fails!");
    pool->max_size=new_size;
    /*--- Reset pool ---
    poolReset(pool);  */
 /*===update pool size ===*/
 pool->max_size = new_size;
```

OKLAHOMA STATE UNIVERSITY

```
 /*===make any new chromosomes null ===*/
 if(new_size > old_size)
 { for(i=old_size;i<=new_size-1;i++) pool->chrom[i]=NULL;
 }
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;  /*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*----------------------------------------------------------------------------------------------------------
| poolFree()--Free the space of a pool which pass poolValid(). call chromFree(),chromKill(), poolValid()
----------------------------------------------------------------------------------------------------------*/
void poolFree(POOL_PTR pool)
{  int i;
   /*=== check the validation ===*/
    if(!poolValid(pool))ERROR("poolFree(): poolValid() check fails, exit");
   /*=== free memory of chroms ===*/
   if(pool->chrom!=NULL)            /*double check for safety of free()*/
   {  for(i=0;i<=pool->max_size-1;i++)  /*free each cell of chrom array*/
      {  if(pool->chrom[i]!=NULL)
         {  if(chromValid(pool->chrom[i]))
            { chromFree(pool->chrom[i]); pool->chrom[i]=NULL;  /*free each health chromosome*/
            }
           else{ chromKill(pool->chrom[i]); pool->chrom[i]=NULL;}/*kill a illness chromosome*/
         }
      }
      free(pool->chrom); pool->chrom=NULL;
   }
   /*=== put in NULL magic card ===*/
   pool->magic_card=NL_CARD;
   /*=== free memory of pool ===*/
   if(pool!=NULL){ free(pool); pool=NULL; }
}
/*----------------------------------------------------------------------------------------------------------
| poolKill()   -free a illness pool (pool!=NULL, but fail on poolValid()), call poolRemoval(),poolClean()
----------------------------------------------------------------------------------------------------------*/
void poolKill(POOL_PTR pool)
{ int i;
 /*=== error check ===*/
 if(pool==NULL)ERROR("poolKill():pass in pool ptr points to NULL,exit");
 /*=== free memory of chroms ===*/
 if(pool->chrom!=NULL)                 /*double check for safety of free()*/
 {  for(i=0;i<=pool->max_size-1;i++)  /*free each cell of chrom array*/
    {   if(pool->chrom[i]!=NULL)
        {  if(chromValid(pool->chrom[i]))
           {  chromFree(pool->chrom[i]); pool->chrom[i]=NULL; /*free each health chromosome*/
           }
           else{ chromKill(pool->chrom[i]); pool->chrom[i]=NULL;}/*kill a illness chromosome*/
        }
    }
    free(pool->chrom); pool->chrom=NULL;
 }
 /*=== put in NULL magic card ===*/
 pool->magic_card=NL_CARD;
```

OKLAHOMA STATE UNIVERSITY

```
 /*=== free memory of pool ===*/
 if(pool!=NULL){ free(pool); pool=NULL; }
}/*the same as poolFree, only differ error check, enable to kill ill pool*/
/*=====================================================================
| pool chromosomes manipulattion
=====================================================================*/
/*---------------------------------------------------------------------
| poolRemove() -remove a chromosome from the pool by providing index. If null chrom met, aborts op
otherwise frees the space and reduces actual pool size by 1, this is the only place to reduce pool size. note:
pool size is only reset to 0 by poolReset(), reduced by 1 by poolRemove(), increased by 1 by poolAppend(),
no any other routions can modify pool size. set off sorted and updated flags. any change of pool turns them
to FALSE, only poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2, only poolStats()
turns updated to TRUE. called by poolClean(). call poolValid(), chromValid(), chromFree(), ChromKill()
---------------------------------------------------------------------*/
void poolRemove(POOL_PTR pool, int index)
{ /*===error check===*/
 if(!poolValid(pool))ERROR("poolRemove(): poolValid() check fails, exit");
 if(index<0||index>=pool->max_size)ERROR("plRemove(): invalid index,exit");
 /*===removing...===*/
 if(pool->chrom[index]!=NULL)
 { if(chromValid(pool->chrom[index])) chromFree(pool->chrom[index]); /*free a health chrom.*/
   else chromKill(pool->chrom[index]); /*kill(free) a illness chromosome*/
   pool->chrom[index]=NULL;
   /*===reduce pool size by 1, this is only place to reduce pool size==*/
   pool->size-=1;                    /*reduce actural pool size by 1*/
 }
 else
 { WARN("poolRemove(): try to remove a null chromosome, abort operation!");
   pool->chrom[index]=NULL;         /*make sure it points to NULL*/
 }
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*---------------------------------------------------------------------
| poolClean() -clean out chromosomes of pool from (including) minidx to maxidx by calling poolRemove()
to free spaces & reduce size. set off sorted and updated flags. any change of pool turns them to FALSE,
only poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2, only poolStats() turns updated to
TRUE.
---------------------------------------------------------------------*/
void poolClean(POOL_PTR pool,int minidx,int maxidx)
{ int i;
 /*===error check===*/
 if(!poolValid(pool))ERROR("poolClean(): poolValid() check fails, exit");
 if(minidx<0||minidx>=pool->max_size)ERROR("plClean:invalid minidex,exit");
 if(maxidx<0||maxidx>=pool->max_size)ERROR("plClean:invalid maxidex,exit");
 if(minidx>maxidx)ERROR("poolClean(): minidx > maxidex, exit");
 /*===removing...===*/
 for(i=minidx;i<=maxidx;i++)
 {  poolRemove(pool,i); pool->chrom[i]=NULL;  /*if chrom[i]!=NULL, free it & reduce size by 1*/
 }
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-------------------------------------------------------------------------------------------------------------
| poolMove() -move a chromosome in the pool: free chrom at dst and reduce pool size by 1, if exists, by
calling poolRemove(). move chrom at src to dst, and NULL src cell. set off sorted and updated flags. any
change of pool turns them to FALSE, only poolSort() turns sorted to TRUE(1), only poolRank() sets
sorted = 2, only poolStats() turns updated to TRUE.
-------------------------------------------------------------------------------------------------------------*/
void poolMove(POOL_PTR pool, int src_idx, int dst_idx)
{ /*===error check===*/
 if(!poolValid(pool))ERROR("poolMove(): poolValid() check fails, exit");
 if(src_idx<0||src_idx>=pool->max_size)ERROR("plMv:invalid src_idex,exit");
 if(dst_idx<0||dst_idx>=pool->max_size)ERROR("plMv:invalid dst_idex,exit");
 if(src_idx==dst_idx) ERROR("poolMove(): src_idx = dst_idex, exit");
 /*===remove dst_idx chromosome...===*/
 poolRemove(pool,dst_idx);/*if dst exists, free it & reduce size by 1*/
 /*===change the index of chromosome from src_idx to dst_idx===*/
 pool->chrom[dst_idx]=pool->chrom[src_idx];
 pool->chrom[src_idx]=NULL;
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-------------------------------------------------------------------------------------------------------------
| poolSwap() -swap two health chromosomes in the pool. set off sorted and updated flags. any change of
pool turns them to FALSE, only poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2, only
poolStats() turns updated to TRUE.
-------------------------------------------------------------------------------------------------------------*/
void poolSwap(POOL_PTR pool, int idx1, int idx2)
{ CHROM_PTR tmp;
 /*===error check===*/
 if(!poolValid(pool))ERROR("poolSwap(): poolValid() check fails, exit");
 if(idx1<0||idx1>=pool->max_size)ERROR("poolSwap(): invalid idx1, exit");
 if(!chromValid(pool->chrom[idx1])) ERROR("poolSwap(): chrom[idx1] check fails, exit");
 if(idx2<0||idx2>=pool->max_size)ERROR("poolSwap(): invalid idx2, exit");
 if(!chromValid(pool->chrom[idx2])) ERROR("poolSwap(): chrom[idx2] check fails, exit");
 if(idx1==idx2){WARN("poolSwap(): idx1 = idx2, abort!");return;}
 /*===exchange the indies of chromosomes at idx1 and idx2===*/
 tmp=pool->chrom[idx2];
 pool->chrom[idx2]=pool->chrom[idx1];
 pool->chrom[idx1]=tmp;
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE; /*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-------------------------------------------------------------------------------------------------------------
| poolAlign()--bubble holes of chrom array of pool to the left side, do not change the order of chroms in
chrom array. re-indexing the chroms in array by calling poolIndex(). check the pool size but do not reset
it, pool size is only reset to 0 by poolReset(), reduced by 1 by poolRemove(),increased by 1 by
poolAppend(), no any other routions can modify pool size.set off sorted and updated flags, due to index
and position changes in pool. any change of pool turns them to FALSE, only poolSort() turns sorted to
```

```
TRUE(1), only poolRank() sets sorted = 2, only poolStats() turns updated to TRUE. execution poolAlign
depends flags of sorted and updated; if either one of them TRUE, means no holes between chroms, simply
return, otherwise exe poolAlign(); if sorted=TRUE(may updated=FALSE), have exe poolSort(); if
updated=TRUE(may sorted=FALSE), have exe poolStats(); in both cases, there are no holes.
-----------------------------------------------------------------------------------------------------------*/
void poolAlign(POOL_PTR pool)
{ int i,k;
 /*===check the validation===*/
 if(!poolValid(pool))ERROR("poolAlign(): poolValid() check fails, exit");
 /*===is pool aligned ?===*/
 if(pool->sorted || pool->updated){ WARN("poolAlign(): pool is already aligned, return."); return;}
 /*===bubble any hole to the far right end of array==*/
 for(i=1;i<=pool->max_size-1;i++)/*keep the order of chroms in the array*/
 {  if(pool->chrom[i]!=NULL)
    {  k=i;
       while(k>0 && (pool->chrom[k-1]==NULL))
       {  pool->chrom[k-1]=pool->chrom[k]; pool->chrom[k]=NULL; k=k-1;
       }
    }
 }
 /*=== reindexing ===*/
 poolIndex(pool);
 /*==recounter and exam the pool size but not modify just print result==*/
 k=pool->size;
 i=0;
 while(pool->chrom[i]!=NULL) i++;
 if(i!=k)WARN("poolAlign:actural # of chrom not agree w/ pl size,not reset");
 /*pool->size=i;*//*not modify by poolAlign()*/
 printf("\npoolAlign():plsize=%d,actual # of chrom=%d\n",k,i);
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE; /*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*as long as updated=TRUE or sorted=TRUE, poolAlign() will not be executed, but once poolAlign() be
executed, it will set updated =FALSE due to (min_,max_,best_)index changes of pool, it will also set
sorted=FALSE due rank position changes such that rank_probs of chrom are outdated. It is expected that
either poolStats() which sets updated=TRUE or poolSort() which sets sorted=TRUE or poolRank() which
sets sorted=2 will go to disable poolAlign() execution right.
 */
/*----------------------------------------------------------------------------------------------------------
| poolInsert()-insert a chromosome chrom into the pool at location idx, copy=1 TRUE: copy chrom into
pool, to the old one if exits. copy=0 FALSE: move chrom into pool, free old one if exits. do not modify
pool->size. note: pool size is only reset to 0 by poolReset(), reduced by 1 by poolRemove(), increased by 1
by poolAppend(), no any other routions can modify pool size. set off sorted and updated flags. any change
of pool turns them to FALSE, only poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2,
only poolStats() turns updated to TRUE. called by poolAppend(), and replace operations. if called by
poolAppend(), old one means a spare chromosome in pool. if called by replace, old one means the parent
need replaced.
-----------------------------------------------------------------------------------------------------------*/
void poolInsert(POOL_PTR pool, CHROM_PTR chrom, int idx, int copy)
{ /*===error check===*/
 if(!poolValid(pool))ERROR("poolInsert(): poolValid() check fails, exit");
 if(!chromValid(chrom))ERROR("poolInsert(): chromValid check fails, exit");
```

OKLAHOMA STATE UNIVERSITY

```
 if(idx<0||idx>pool->size-1)ERROR("poolInsert(): invalid idx, exit");
  /*note:we always insert between (including) 0 and pool->size-1;0<=idx<=pool->size-1, if called by
poolAppend(), idx=pool->size-1 since pool->size just increase 1 in poolAppend(); if called by replace op,
idx is the index of parent, it also between 0 and pool->size-1 though not increase pool->size at this time.
*/
 if(copy!=TRUE&&copy!=FALSE)ERROR("poolInsert(): invalid copy key, exit");
 /*===insert chromosome===*/
 if(copy)/*--copy but not move chrom into pool, so prepare target chrom--*/
 { if(pool->chrom[idx]==NULL) /*thare is no target chrom so alloc new one*/
   { pool->chrom[idx]=chromAlloc(chrom->dim);/*pool->size+=1;*//*not modify by poolInsert()*/
   }
    else                                /*thare is a target chrom*/
   { if(chromValid(pool->chrom[idx])) /*thare is a health target chrom*/
       chromReset(pool->chrom[idx]); /*reset for reuse*/
     else                                /*thare is a illness target chrom*/
     { chromKill(pool->chrom[idx]);   /*so kill the ill target chrom*/
       pool->chrom[idx]=chromAlloc(chrom->dim); /*and alloc a new one*/
     }
   }
    /*--inserting, by copying chrom into the target chrom --*/
   chromCopy(chrom,pool->chrom[idx]);/*chromCopy(src, dst)*/
   pool->chrom[idx]->index=idx;/*make sure pool index is correct*/
 }
 else/*--copy=FALSE, move original chrom into pool to replace old one--*/
 {   /*--removing old chromosome at cell idx if exists, clean the cell--*/
     if(pool->chrom[idx]!=NULL)
     {   if(chromValid(pool->chrom[idx])) chromFree(pool->chrom[idx]); /*free a health chrom.*/
         else chromKill(pool->chrom[idx]);  /*kill(free) a illness chromosome*/
          pool->chrom[idx]=NULL;
     }
/*this segment is equivalent to a statement: poolRemove(pool,idx); we do not call poolRemove() directly is
due to avoiding reducing pool size by 1 since free a spare chrom when poolAppend() copy = FALSE call
poolInsert(), or free a parent chrom when replace op copy=FALSE call poolInsert(). in both case we do
not allow pool size modified inside execution of poolInsert().*/
     /*--inserting, by moving chrom into the target position---*/
      pool->chrom[idx]=chrom; chrom=NULL;
      pool->chrom[idx]->index=idx;/*make sure pool index is correct*/
 }
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE; /*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-----------------------------------------------------------------------------------------------------------
| poolAppend()-append a chromosome to the tail of chrom array in the pool. copy=1 TRUE: copy chrom
into pool, to the spare one if exits. copy=0 FALSE: move chrom into pool, free spare one if exits. increase
pool size by 1, this is the only place to increase pool size by 1. note: pool size is only reset to 0 by
poolReset(), reduced by 1 by poolRemove(), increased by 1 by poolAppend(), no any other routions can
modify pool size. set off sorted and updated flags. any change of pool turns them to FALSE, only
poolSort() turns sorted to TRUE(1), only poolRank() sets sorted = 2, only poolStats() turns updated to|
TRUE.
-----------------------------------------------------------------------------------------------------------*/
void poolAppend(POOL_PTR pool, CHROM_PTR chrom, int copy)
{ int idx;
```

OKLAHOMA STATE UNIVERSITY

```
 /*===error check===*/
 if(!poolValid(pool))ERROR("poolAppend(): poolValid() check fails, exit");
 if(!chromValid(chrom))ERROR("poolAppend(): chromValid check fails, exit");
 if(copy!=TRUE&&copy!=FALSE)ERROR("poolAppend(): invalid copy key, exit");
 /*===realloc space if needed===*/
/*if(pool->size==pool->max_size){ poolResize(pool,pool->max_size+1);pool->max_size+=1;}*/
 /*acturally in my program, it is never realloc to expand pool, because pool_max_size is like chrom_dim
to be frozen in ga_center by cfg file and we choose never to change it. so we never use chromResize() and
poolResize()*/
 /*===make sure pool still has room to accommodate a new chrom===*/
 if(pool->size==pool->max_size) ERROR("poolAppend(): try to append to a full pool, exit");
 /*===set appending index to the tail of current chrom array*/
 idx=pool->size;
 /*===increase pool size by 1, this is only place to increase pool size==*/
 pool->size+=1;/*we do not allow  poolInsert() to modify pool size*/
 /*===appending by inserting to the end of the chromosome array===*/
 poolInsert(pool,chrom,idx,copy);/*insert chrom to idx, not modify pl size*/
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE; /*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*---------------------------------------------------------------------------------------------------------------
| poolCompareMin(a,b)-compare two chromosome fitness when minimize pool. If a is better than b (a<b)
return -1, so a is on the left of b. If a is worse than b (a>b) return 1, so a is on the right of b. If a is equal to
b (a=b) return 0, seq of a,b nochange
---------------------------------------------------------------------------------------------------------------*/
static int poolCompareMin(const void* a, const void* b)
{ /*---error check---*/
 if(!chromValid(*(CHROM_PTR*)a)) ERROR("plCompMin():chromValid(a) check fails, exit");
 if(!chromValid(*(CHROM_PTR*)b)) ERROR("plCompMin():chromValid(b) check fails, exit");
 /*---comparing---*/
 if((*(CHROM_PTR*)a)->fitness < (*(CHROM_PTR*)b)->fitness) return -1;/*a is better than b*/
 else if((*(CHROM_PTR*)a)->fitness > (*(CHROM_PTR*)b)->fitness) return 1; /*a is worse than b*/
 else return 0;
}
/*---------------------------------------------------------------------------------------------------------------|
poolCompareMax(a,b)-compare two chromosome fitness when maximize pool. If a is better than b (a>b)
return -1, so a is on the left of b. If a is worse than b (a<b) return 1, so a is on the right of b. If a is equal to
b (a=b) return 0, seq of a,b nochange
---------------------------------------------------------------------------------------------------------------*/
static int poolCompareMax(const void* a,const void* b)/*CHROM_PTR* a in real*/
{ /*---error check---*/
 if(!chromValid(*(CHROM_PTR*)a)) ERROR("plCompMax():chromValid(a) check fails, exit");
 if(!chromValid(*(CHROM_PTR*)b)) ERROR("plCompMax():chromValid(b) check fails, exit");
 /*---comparing---*/
 if((*(CHROM_PTR*)a)->fitness > (*(CHROM_PTR*)b)->fitness) return -1;/*a is better than b*/
 else if((*(CHROM_PTR*)a)->fitness < (*(CHROM_PTR*)b)->fitness) return 1;/*a is worse than b*/
 else return 0;
}
/*---------------------------------------------------------------------------------------------------------------
| poolSort()   -seqence chromosomes from best (0th cell)to worest (tail), turn on sorted flag, let sorted = 1.
only poolSort() to set sorted = 1,  only poolRank() to set sorted = 2, and any chroms changes in pool will
turn the flag off(FALSE = 0). If sorted flag on(1, or 2), means no any change of chroms of pool since last
```

OKLAHOMA STATE UNIVERSITY

```
call of poolSort(), we simply return, otherwise, if sorted = 0, continue on sorting the pool. turn off updated
flag, since change of (min_)index of pool. any change of pool turns updated to FALSE, only poolStats()
turns updated to TRUE. called by poolRank(), and reByRank() operations
-----------------------------------------------------------------------------------------------------------*/
void poolSort(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("poolSort():gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("poolSort(): poolValid() check fails, exit");
 /*---is current pool sorted?---*/
 if(pool->sorted)/*if sorted=1 or 2, not exe and return; if sorted=0, exe*/
 { WARN("poolSort(): current pool sorted flag on, return!"); return;
 }
 /*---seqence chromosomes from best (0th cell)to worest---*/
if(ga_center->minimize) qsort((void*)pool->chrom,pool->size,sizeof(CHROM_PTR),poolCompareMin);
 else qsort((void*)pool->chrom,pool->size,sizeof(CHROM_PTR),poolCompareMax);
 /*---update chromosome index in pool--*/
 poolIndex(pool);
 /*---set sorted = 1, this is only place to set sorted=1--*/
 pool->sorted=TRUE;/*TRUE=1, FALSE=0*/
 /*---set off updated flag, since change of (min_)index of pool--*/
 pool->updated=FALSE;/*due to chromosome changes in positions in pool*/
}
/*=========================================================================
| maintaining pool datum
=========================================================================*/
/*---------------------------------------------------------------------------------------------------------
| poolIndex()  -update index of each chromosome in the pool after calls of poolAlign(), and / or
poolSort().If updated=TRUE, means no any chromosome change in pool (including: reset, resize, remove,
clean, move, swap, align, insert, append, sort) since last call of poolStats(), simply return, otherwise if
updated = FALSE, continue poolIndex().If sorted=FALSE && updated=FALSE, make sure also call
poolAlign() first outside to align pool before indexing.
-----------------------------------------------------------------------------------------------------------*/
void poolIndex(POOL_PTR pool)
{ int i;
 /*--- check the validation of pool ---*/
 if(!poolValid(pool)) ERROR("poolIndex(): poolValid() check fails , exit");
 /*--- if pool updated ?--*/
 if(pool->updated){ WARN("poolIndex(): current pool updated flag on, return!"); return; }
 /*---set index for each chromosomes---*/
 for(i=0;i<=pool->size-1;i++)pool->chrom[i]->index=i;/*5-23-96 the following while loop more safe*/
 /*i=0;
 while(i<pool->max_size && pool->chrom[i]!=NULL) { pool->chrom[i]->index=i; i++;}
 *//*the while loop also counter spare chroms, so will have mistake*/
}
/*---------------------------------------------------------------------------------------------------------
| poolFitness()-update fitness of each chromosome in the pool, check the updated flag first before call
poolFitness(), after calls of poolAlign() and/or poolSort(), poolIndex(). If updated = TRUE, means no any
chromosome change in pool (including:reset,resize,remove,clean,move,swap,align,insert,append,sort)
since last call of poolStats(), simply return, otherwise if updated = FALSE, continue poolFitness(), but
make sure first call poolIndex()) outside since the change of pool. If updated=FALSE && sorted=FALSE,
make sure also call poolAlign() first outside to align pool before poolIndex().
-----------------------------------------------------------------------------------------------------------*/
void poolFitness(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ int i;
```

OKLAHOMA STATE UNIVERSITY

```
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("poolFitness():gacValid() check fails,exit");
 if(!poolValid(pool))ERROR("poolFitness(): poolValid() check fails, exit");
 /*--- if pool updated ?--*/
 if(pool->updated){ WARN("poolFitness(): current pool updated flag on, return!"); return; }
 /*---evaluate each chromosomes in the pool---*/
 for(i=0;i<=pool->size-1;i++)
 { if(pool->chrom[i]->eva==0){ga_center->EV_fun(pool->chrom[i]); }
 }
}
/*-----------------------------------------------------------------------------------------------------------
| poolPtf()  -update ptf(percent of total fitness) of each chromosome after calls of poolAlign(), and/or
poolSort(), poolIndex(), poolFitness(). the value ptfs are used by the fitness_biased selection. If updated =
TRUE, means no any chromosome change in pool (including: reset, resize, remove, clean, move, swap,
align, insert, append, sort, mu, re) since last call of poolStats(),simply return, otherwise if updated =
FALSE, continue poolPtf(), but make sure first call poolIndex()) and poolFitness() outside since the
change of pool. if updated=FALSE && sorted=FALSE, make sure also call poolAlign() first outside to
align pool before poolIndex().
-----------------------------------------------------------------------------------------------------------*/
void poolPtf(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ int i;
 float tot_ptf=0.0;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("poolPtf(): gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("poolPtf(): poolValid() check fails, exit");
 /*--- if pool updated ?--*/
 if(pool->updated) { WARN("poolPtf(): current pool updated flag on, return!"); return; }
 /*---find total fitness---*/
 pool->tot_fitness=0.0;
 for(i=0;i<=pool->size-1;i++) pool->tot_fitness+=pool->chrom[i]->fitness;
 /*---update ptf for each chromosome---*/
 if(ga_center->minimize)        /*--update ptf when minimize--*/
 { for(i=0;i<=pool->size-1;i++) /*--smaller fitness has bigger ptf--*/
   { pool->chrom[i]->ptf=(float)pool->tot_fitness/pool->chrom[i]->fitness;
     tot_ptf+=pool->chrom[i]->ptf;
   }
   /*--normalize ptf to percentage--*/
   for(i=0;i<=pool->size-1;i++) pool->chrom[i]->ptf=(float)pool->chrom[i]->ptf/tot_ptf;
 }
 else                           /*--update ptf when maximize--*/
 { for(i=0;i<=pool->size-1;i++) /*--biger fitness has bigger ptf--*/
     pool->chrom[i]->ptf=(float)pool->chrom[i]->fitness/pool->tot_fitness;
 }
}
/*-----------------------------------------------------------------------------------------------------------
| poolRank()  -using bias and rank position update rank_prob of each chromosome after the call of
poolSort(). rank_probs are used by rank_biased selection. Turn on sorted flag, let sorted = 2. only
poolRank() to set sorted = 2, only poolSort() to set sorted = 1, and any chroms changes in pool will turn
the flag off(FALSE = 0). The execution of poolRank() does depend on sorted flag: If sorted=2, means no
any change of pool(set sorted=0) including not exe poolSort()(set sorted=1) since the last call of
poolRank()(set sorted=2), so simply return; if sorted=1, means no any change of pool(set sorted=0) since
the last call of poolSort()(set sorted=1), but may be some change of pool since the last call of poolRank()
(set sorted=2), for example, after the last call of poolRank(), replace op may first change pool(set
sorted=0), then another replace op may call poolSort()(set sorted=1), so it may be current pool be sorted
```

OKLAHOMA STATE UNIVERSITY

```
but rank_probs of chroms are not up-to-date. So continue exe poolRank() with no need of call poolSort()
first inside. If sorted=0, means pool changed(set sorted=0) since the last call of poolRank()(set sorted=2)
and the last call of poolSort()(set sorted=1), so continue exe poolRank() with the need of call poolSort()
first inside.
-----------------------------------------------------------------------------------------------------------*/
void poolRank(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ int i,n;
  float q=0.0;
  float r=0.0;
  float bias=0.0;
  /*==check the validation==*/
  if(!gacValid(ga_center))ERROR("poolRank(): gacValid() check fails, exit");
  if(!poolValid(pool))ERROR("poolRank(): poolValid() check fails, exit");
  /*==is current pool changed since the last call of poolRank()?==*/
  if(pool->sorted==2)/*no any change of pool since last call poolRank()*/
  { WARN("poolRank(): pool sorted flag = 2, no change, return."); return;
  }
  /*==is current pool sorted?==*/
  if(!pool->sorted)/*if sorted=1,current pool sorted, but rank_prb outdate*/
     poolSort(ga_center,pool);/*if sorted=0, call poolSort() first*/
  /*==calculate ranked_bias parameters q,r==*/
  n=pool->size;
  if(n==0||n==1)ERROR("poolRank():pl size is 0 or 1 error, exit");
  bias=ga_center->bias; /*--ranked_bias on selection*/
  r=(float)((float)2/(n*(n-1)))*bias;/*scalar ranked_bias */
  q=(float)((float)r*(n-1))/2+(float)1/n;
  /*==update rank_prob==*/
/*if bias=0, r=0, no rked bias at all,    rk_prb(i)=q-i*r=q=1/n for any i*/
/*if bias=1,thare is heaviest rked_bias,rk_prb(0)=q-0*r=q=2/n;best; rk_prb(n-1)=q-(n-1)*r=0;weakest*/
  for(i=0;i<=pool->size-1;i++) pool->chrom[i]->rank_prob=(float)q-((float)i*r);
  /*==set sorted = 2, this is only place to do so==*/
  pool->sorted=2;
}
/*sorted=2(only set by poolRank()), disable poolRank(), disable poolSort() sorted=1(only set by
poolSort()), enable poolRank(), disable poolSort() sorted=0(set by any pool change), enable poolRank(),
enable poolSort()
*/
/*----------------------------------------------------------------------------------------------------------
| poolStats()---update all pool parameters, turn updated flag on. This is the only place to turn updated flag
on, and any chroms changes in pool will turn the flag off. If updated=TRUE, means no any chromosome
change in pool (including:reset,resize,remove,clean,move,swap,align,insert,append,sort,mu,re) since last
call of poolStats(), simply return, otherwise if updated = FALSE, continue poolStats(), but make sure call
poolIndex(),poolFitness(),poolPtf() first outside since the change of pool. If updated=FALSE &&
sorted=FALSE, make sure also call poolAlign() first outside to align pool before poolIndex().
-----------------------------------------------------------------------------------------------------------*/
void poolStats(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ unsigned i,min_index,max_index;
  double min,max,ave,tot,var;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("poolStats(): gacValid() check fails, exit");
  if(!poolValid(pool))ERROR("poolStats(): poolValid() check fails, exit");
  /*---is current pool updated?---*/
  if(pool->updated){ WARN("poolStats(): current pool updated flag on, return!"); return; }
  /*---computer statistics for a pool---*/
```

OKLAHOMA STATE UNIVERSITY

```
/*--trivial cases--*/
if(pool->size==0)  /*--empty pool--*/
{  pool->tot_fitness=0.0;
   pool->min =0.0;
   pool->max =0.0;
   pool->ave =0.0;
   pool->var =0.0;
   pool->dev =0.0;
   pool->min_index =-1;
   pool->max_index =-1;
   pool->best_index =-1;
}
else if(pool->size==1)  /*--only one chromosome in pool--*/
{  pool->tot_fitness=pool->chrom[0]->fitness;
   pool->min =pool->chrom[0]->fitness;
   pool->max=pool->chrom[0]->fitness;
   pool->ave =pool->chrom[0]->fitness;
   pool->var =0.0;
   pool->dev =0.0;
   pool->min_index  =0;
   pool->max_index  =0;
   pool->best_index =0;
}
else /*--normal case--*/
{   /*--initialize local variables--*/
    min=pool->chrom[0]->fitness;
    max=pool->chrom[0]->fitness;
    ave=0.0;
    tot=0.0;
    var=0.0;
    min_index  =0;
    max_index  =0;
    /*--computering min, max, tot, ave, var, min_index, max_index--*/
    for(i=0;i<=pool->size-1;i++)
   {   if(!chromValid(pool->chrom[i]))ERROR("plSts: chmVd check fails,exit");
       /*--reduce min as far as possible and record min_index--*/
       if(min>pool->chrom[i]->fitness){ min=pool->chrom[i]->fitness; min_index=i;}
      /*--increase max as far as possible and record max_index--*/
      if(max<pool->chrom[i]->fitness){ max=pool->chrom[i]->fitness; max_index=i;}
      /*--accumulate total--*/
      tot+=pool->chrom[i]->fitness;
      /*--make sure index of chromsome is set--*/
      pool->chrom[i]->index=i;
   }
   ave=(double)tot/pool->size;
   for(i=0;i<=pool->size-1;i++) var+=(ave-pool->chrom[i]->fitness)*(ave-pool->chrom[i]->fitness);
   var=(double)var/pool->size;
   /*---update pool parameters---*/
   pool->tot_fitness=tot;
   pool->min     =min;
   pool->max     =max;
   pool->ave      =ave;
   pool->var       =var;
   pool->dev     =sqrt(var);
```

```
    pool->min_index =min_index;
    pool->max_index =max_index;
    if(ga_center->minimize) pool->best_index=min_index;
    else pool->best_index=max_index;
    /*---check variance and deviation to see if ga converged--*/
    /*if((dev<=ga_center->critia)&&(dev>=0.0))ga_center->converged=TRUE;
      else ga_center->converged=FALSE; */
    /*ga data had better update by gaStats()*/
  }
 /*---turn pool updated flag on, this is the only place to turn it on--*/
 pool->updated=TRUE;
}
/*not write best chromosome into ga_center since if at begainning of ga, gaSetup():
1) make sure and alloc old_pool of ga,
2) initialize start working pool by poolInit(). At this moment we still  not check if ga_cneter  best history
list array is allocated or not.
Therefore it had better poolStats() only update data of pool, the ga_center data is updated by gaStats()
(only converged, best, old_pool, new_pool).*/
/*=========================================================================
| pool initialization by reading from initial file or by random generator
=========================================================================*/
/*------------------------------------------------------------------------------------------------------------
| getNum()---read digit string from initpool data file, return address of each digit string which is
interpretered as the value of each  chrom matrix cell.
------------------------------------------------------------------------------------------------------------*/
char* getNum(FILE* fid)
{   static char str[80]; int len; char ch;
    /*-- search for a digit --*/
    while(TRUE)
    {  /*--get a character--*/
        ch=fgetc(fid);
        /*--Error or Quit--*/
        if(ch==EOF||ch=='q'||ch=='Q') return NULL;
        /*--hit first charater of a digit string--*/
        if(isdigit(ch)) break;
        /*-- skip to the end of comment line--*/
        if(ch=='#') while((ch=fgetc(fid))!='\n');
    }
    /*--make sure ch is the first character of a digit string--*/
    if(!isdigit(ch))ERROR("getNum(): digit ch corrupted, exit");
   /*--now put that digit string into str[]--*/
   len=0;                    /*since digit str may contain float point*/
   while(len<80 && ch!=EOF && !isspace(ch) /*&& isdigit(ch)*/ && ch!='#')
   {   str[len++]=ch;  /*assign digit ch to the next available cell of str[]*/
       ch=fgetc(fid);
   }
  str[len]=0;       /*equal to assign '\0', null terminating the str */
   /*--return the addr of digit string stored inside str[]--*/
   return (char*)str;
}
/*------------------------------------------------------------------------------------------------------------
| poolRead()---initial pool by read in chrom matrices from initpool file, actual parameter for chrom_dim is
ga_center->chrom_dim, the pool_max_size is already inside pool, the actural pool size is depended on the
```

```
actural reading of matrix  it must be at most pool_max_size-2. pool_max_size at least 4. called by
poolInit()
--------------------------------------------------------------------------------------------------------------*/
void poolRead(POOL_PTR pool,int chrom_dim,FILE* fid)
{ CHROM_PTR chrom; int  i,j,k; int pool_size; char* sptr; GENE_TYPE gene;
 /*--error check--*/
 if(!poolValid(pool)) ERROR("poolRead(): poolValid() check fails, exit");
 if(chrom_dim<=0)ERROR("poolRead(): invalid chrom_dim check fails, exit");
 if(fid==NULL)ERROR("poolRead(): invalid fid, exit");
 /*===get pool_size===*/
 sptr=getNum(fid);
 if(sptr==NULL||sscanf(sptr,"%d",&pool_size)!=1)ERROR("plRd:actural pool_size read err, exit");
 /*===make sure pool_size is between 1 and pool_max_size -2==*/
 if(pool_size<1 || pool_size>pool->max_size-2)ERROR("plRead: actural pool_size out of range, exit");
 /*===make sure pool_size is a even number==*/
 if(pool_size%2!=0) pool_size +=1;
 /*===alloc the pool chrom space==*/
 for(k=0;k<=pool_size-1;k++) pool->chrom[k]=chromAlloc(chrom_dim);
 /*--while there is a chromosome to read--*/
 pool->size=0; /*actual size is increased after each read of chrom matrix*/
 while(pool->size<pool_size)/*reading stop at getNum() meets EOF or 'q' or 'Q'*/
 {  /*--reuse or allocate a chromsome--*/
    if(pool->chrom[pool->size]!=NULL) /*spare chrom when reset old pool*/
    {  if(!chromValid(pool->chrom[pool->size]))/*illness spare chrom*/
       {  chromKill(pool->chrom[pool->size]);  /*kill illness spare chrom*/
          pool->chrom[pool->size]=NULL;         /*NULL the current cell*/
          chrom=chromAlloc(chrom_dim);          /*alloc a new chrom*/
       }
       else                                     /*health spare chrom*/
       {  chrom=pool->chrom[pool->size];        /*give ptr to chrom for reuse*/
          pool->chrom[pool->size]=NULL;         /*NULL the current cell*/
          chromReset(chrom);                    /*chrom reset for reuse*/
       }
    }
    else                                        /*there is no spare chrom*/
    {   chrom=chromAlloc(chrom_dim);            /*alloc a new chrom*/
    }
    /*--read chrom matrix--*/
    for(i=0;i<chrom_dim;i++)
    {   for(j=0;j<chrom_dim;j++)
        {   sptr=getNum(fid);
            if(sptr==NULL||sscanf(sptr,"%d",&gene)!=1)
            {   /*either no more chrom matrix or in the middle of matrix reading*/
                /*in both cases we should abort the chromsome*/
                chromFree(chrom);
                /*if EOF in the middle of matrix reading, a warning give out*/
                if((i!=0||j!=0)&&(sptr==NULL||*sptr!='q'))
                WARN("poolRead(): premature eof reading chromosome");
                /*---end of reading--*/
                return;/*return can be apply to void funct, no use of break is due
                          to break only break up one layer of for-loop*/
            }
            /*---valid gene cell value is stored ---*/
            chrom->gene[i][j]=gene;
```

OKLAHOMA STATE UNIVERSITY

```
            }
        }
        /*--put the chromosome into the pool---*/
        /*--insert chrom in the end of the pool, then increase pool->size--*/
        poolAppend(pool,chrom,FALSE);
    }/*end while-loop, now pool->size<=pool_size*/
    /*==close input file==*/
    /* fclose(fid);  *//*fid is open and close inside of poolInit()*/
    /*==check the actural pool size==*/
    if(pool->size<pool_size)/*it should be pool->size<=pool_size*/
    {   WARN("poolRead():num of actural chrom matrix reading<pool_size,adjust");
        /*--make sure pool->size is even by adjusting--*/
        if(pool->size%2!=0)/*if not even add one more chrom*/
        {   /*--reuse or allocate a chromsome--*/
            if(pool->chrom[pool->size]!=NULL)           /*spare chrom when reset old pool*/
            {   if(!chromValid(pool->chrom[pool->size]))/*illness spare chrom*/
                {   chromKill(pool->chrom[pool->size]);  /*kill illness spare chrom*/
                    pool->chrom[pool->size]=NULL;        /*NULL the current cell*/
                    chrom=chromAlloc(chrom_dim);         /*alloc a new chrom*/
                }
                else                                     /*health spare chrom*/
                {   chrom=pool->chrom[pool->size];       /*give ptr to chrom for reuse*/
                    pool->chrom[pool->size]=NULL;        /*NULL the current cell*/
                    chromReset(chrom);                   /*chrom reset for reuse*/
                }
            }
            else                                         /*there is no spare chrom*/
            {   chrom=chromAlloc(chrom_dim);             /*alloc a new chrom*/
            }
            /*--set least feasible chrom matrix--*/
            for(i=0;i<=chrom_dim-2;i++) chrom->gene[i][i+1]=1;
            /*--append to the pool and increase pool->size by 1 --*/
            poolAppend(pool,chrom,FALSE);
        }
        /*--adjust pool actural size by freeing extra alloc chrom space--*/
        for(i=pool->size;i<=pool->max_size-1;i++)
        {   if(pool->chrom[i]!=NULL){   chromKill(pool->chrom[i]); pool->chrom[i]=NULL;}
        }
        printf("initial pool data file reading error is corrected\n");
        printf("the actural pool size: %d\n",pool->size);
    }
    else if(pool->size>pool_size)/*it should be pool->size<=pool_size*/
    { ERROR("poolRead():num of actural chrom matrix reading>pool_size,exit");
    }
    else
    {   printf("initial pool data file reading is successful\n");
        printf("the actual pool size:  %d\n",pool->size);
    }
}
/*-------------------------------------------------------------------------------------------------------------
| poolRand()----initial pool by randomly creating chrom matrices the pool_max_size is already inside pool.
need to randomly generate a actual pool size between n/2 and pool_max_size-2, need to pass in
chrom_dim value stored in ga_center. pool_max_size at least 4.
-------------------------------------------------------------------------------------------------------------*/
```

```
void poolRand(POOL_PTR pool, int chrom_dim)
{  CHROM_PTR chrom;  GENE_TYPE gene;  int  i,j,k,l,pool_size;
   i=0;j=0;k=0,l=0;  pool_size=0;
   /*--error check--*/
   if(!poolValid(pool))ERROR("poolRand(): poolValid() check fails, exit");
   if(chrom_dim<=0)ERROR("poolRand(): invalid chrom_dim check fails, exit");
   /*--random a pool actural size between n/2 and pool_max_size(>=4) - 2  --*/
   k=(int)(pool->max_size/2);
   SRAND(rand());
   pool_size=RANDDOM(k,pool->max_size-2);
   l=0;
   while(pool_size<k||pool_size>pool->max_size-2)/*make sure pool_size valid*/
   {   SRAND(rand());
       pool_size=RANDDOM(k,pool->max_size-2);  l++;
       if(l==1000)/*it should be get valid pool_size within 1000 random tries*/
           ERROR("poolRand():fail to generate valid pool_size in 1000 tries,exit");
   }
   /*===make sure pool_size is a even number==*/
   if(pool_size%2!=0) pool_size +=1;
   /*===alloc the pool chrom space==*/
   for(k=0;k<=pool_size-1;k++) pool->chrom[k]=chromAlloc(chrom_dim);
   /*pool->size=pool_size;  /*record valid actural pool size into pool*/
   /*-- random generate chrom matrix one by one --*/
   pool->size=0;/*actual size increased after each plAppend of chrom matrix*/
   while(pool->size<pool_size)
   {   /*--reuse or allocate a chromsome--*/
       if(pool->chrom[pool->size]!=NULL)           /*spare chrom when reset old pool*/
       {   if(!chromValid(pool->chrom[pool->size])) /*illness spare chrom*/
           {   chromKill(pool->chrom[pool->size]);   /*kill illness spare chrom*/
               pool->chrom[pool->size]=NULL;         /*NULL the current cell*/
               chrom=chromAlloc(chrom_dim);          /*alloc a new chrom*/
           }
           else                                      /*health spare chrom*/
           {   chrom=pool->chrom[pool->size];        /*give ptr to chrom for reuse*/
               pool->chrom[pool->size]=NULL;         /*NULL the current cell*/
               chromReset(chrom);                    /*chrom reset for reuse*/
           }
       }
       else {   chrom=chromAlloc(chrom_dim);  }/*there is no spare chrom, alloc a new chrom*/
       /*--random chrom matrix--*/
       for(i=0;i<chrom_dim;i++)
       {   for(j=0;j<chrom_dim;j++)
           {   gene=RANDBIT();
               l=0;
               while(gene!=0 && gene!=1)/*make sure gene have valid binary value*/
               {   SRAND(rand());
                   gene=RANDBIT();
                   l++;
                  if(l==1000) /*it should get valid gene within 1000 random tries*/
                  ERROR("plRand():fail to get valid gene in 1000 rand tries,exit");
               }
               /*---valid gene cell value is stored ---*/
               chrom->gene[i][j]=gene;
           }
```

```
        }
        /*--modify chrom matrix cell so that it is a feasible solution--*/
        chromRepair(chrom);
        /*--append the chromosome into the pool---*/
        /*--insert chrom in the end of the pool, then increase pool->size--*/
        poolAppend(pool,chrom, FALSE);
    }/*end random pool_size chrom matrices, now pool->size<=pool_size*/
    /*==check the actural pool size==*/
    if(pool->size<pool_size)/*it should be pool->size<=pool_size*/
    {   WARN("poolRand():num of actural chrom matrix random < pool_size,adjust");
        /*--make sure pool->size is even by adjusting--*/
        if(pool->size%2!=0)/*if not even add one more chrom*/
        {  /*--reuse or allocate a chromsome--*/
           if(pool->chrom[pool->size]!=NULL) /*spare chrom when reset old pool*/
           {   if(!chromValid(pool->chrom[pool->size]))/*illness spare chrom*/
               {   chromKill(pool->chrom[pool->size]);  /*kill illness spare chrom*/
                   pool->chrom[pool->size]=NULL;        /*NULL the current cell*/
                   chrom=chromAlloc(chrom_dim);         /*alloc a new chrom*/
               }
               else                                      /*health spare chrom*/
               {   chrom=pool->chrom[pool->size];        /*give ptr to chrom for reuse*/
                   pool->chrom[pool->size]=NULL;         /*NULL the current cell*/
                   chromReset(chrom);                    /*chrom reset for reuse*/
               }
           }
           else                                          /*there is no spare chrom*/
           {   chrom=chromAlloc(chrom_dim);              /*alloc a new chrom*/
           }
           /*--set least feasible chrom matrix--*/
           for(i=0;i<=chrom_dim-2;i++) chrom->gene[i][i+1]=1;
           /*--append to the pool and increase pool->size by 1 --*/
           poolAppend(pool,chrom,FALSE);
        }
        /*--adjust pool actural size by freeing extra alloc chrom space--*/
        for(i=pool->size;i<=pool->max_size-1;i++)
        {   if(pool->chrom[i]!=NULL){    chromKill(pool->chrom[i]);pool->chrom[i]=NULL;}
        }
        printf("initial pool random reading error is corrected\n");
        printf("the actural pool size: %d\n",pool->size);
    }
    else if(pool->size>pool_size)/*it should be pool->size<=pool_size*/
    {   ERROR("poolRand():num of actural chrom matrix random > pool_size,exit");
    }
    else
    {   printf("initial pool random reading is successful\n");
        printf("the actual pool size:  %d\n",pool->size);
    }
}
/*-----------------------------------------------------------------------------------------------------------
| poolInit()    -initial pool manager
-----------------------------------------------------------------------------------------------------------*/
void poolInit(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ FILE* fid;
  /*--check the validation--*/
```

```
 if(!gacValid(ga_center)) ERROR("poolInit(): gacValid() check fails,exit");
 if(!poolValid(pool)) ERROR("poolInit(): poolValid() check fails, exit");
 /*--switch on the type of initial pool--*/
 switch(ga_center->ip_flag)
 {   case IP_NONE: break;
     case IP_FILE:  /*--open the initpool data file--*/
                    if((fid=fopen(ga_center->ip_file,"r"))==NULL)
                      ERROR("poolInit(): open initpool data file error, exit");
                    /*--read chrom matrix one by one--*/
                    poolRead(pool,ga_center->chrom_dim,fid);
                    /*--close data file--*/
                    fclose(fid);
                    break;
     case IP_RANDOM:  /*--randomly initialize pool--*/
                        poolRand(pool,ga_center->chrom_dim);
                        break;
     case IP_INTERACT:  puts("\nEnter chrom matrix one by one('q' to quit):\n");
                         poolRead(pool,ga_center->chrom_dim,stdin);
                         break;
     default: ERROR("poolInit(): invalid ip_flag, exit");
 }
 /*--Evaluate pool--*//*because just read in new pool no need poolAlign()*/
 /*--it must be in this sequence: index,fitness,ptf,stats every time to
    evaluate the pool and every time to use poolStats()--*/
 /*this is the 1st evaluate of pool. After return to gaSetup(), may be
   add one more copy of best to make even chroms, so evaluate again there*/
 if(!pool->updated)
 {   poolIndex(pool);
     poolFitness(ga_center,pool);
     poolPtf(ga_center,pool);
     /*--update states of pool--*/
     poolStats(ga_center,pool);
 }
 /*after this step,updated=TRUE, set to FALSE by any pool changes (reset, resize, remove, clean, move,
swap, align, insert, append, sort) as long as updated=TRUE, poolAlign() will not be executed, but once
poolAlign() be executed, it will set updated=FALSE due to (min_,max_,best_) index changes of pool, it
will also set sorted=FALSE due rank position changes such that rank_probs of chrom are outdated. */
}
/*---------------end pool.c official body------*/




/*=============end of file: pool.c ========*/
```

file: gac.c
```
/*==========================================================================
| ga_center management and configuring routines:
|
| /*---ga_center memory setup and  default initialization-------------------------------------------------
| gacValid()----check to see if a ga_center is valid
| gacReset()----reset a ga_center and default initialization
| gacAlloc()----allocate a ga_center memory and default initializaion
| gacFree()------free the space of a ga-center
| /*---ga_center configuration by reading from configuration file----------------------------------------
| gacLine()------line reader of config file, subroution used by gacRead()
| gacRead()-----configuring ga_center by reading configuration file
| gacVerify()----verify ga configuration
| /*---ga_center report----------------------------------------------------------------------------------
| gacPrint()------print ga_center information to rp_fid
==========================================================================*/
#include "gah.h"
#define MAXTOK 10 /*Maximum number of tokens on a line*/
#define STRLEN 80 /*string len equal to the length of an input line*/
/*==========================================================================
| function prototypes
==========================================================================*/
int gacValid(GA_CENTER_PTR ga_center);
void gacReset(GA_CENTER_PTR ga_center);
GA_CENTER_PTR gacAlloc(void);
void gacFree(GA_CENTER_PTR ga_center);
int gacLine(char* line, char token[][STRLEN]);
int gacRead(GA_CENTER_PTR ga_center,char* cfgfile);
void gacVerify(GA_CENTER_PTR ga_center);
void gacPrint(GA_CENTER_PTR ga_center);

extern int chromValid(CHROM_PTR );
extern void chromReset(CHROM_PTR chrom);
extern void chromFree(CHROM_PTR chrom);

extern int poolValid(POOL_PTR pool);
extern void poolReset(POOL_PTR pool);
extern void poolFree(POOL_PTR pool);

extern void seSelect(GA_CENTER_PTR ga_center,char* fn_name);
extern char* seName(GA_CENTER_PTR ga_center);

extern void xSelect(GA_CENTER_PTR ga_center,char* fn_name);
extern char* xName(GA_CENTER_PTR ga_center);

extern void muSelect(GA_CENTER_PTR ga_center,char* fn_name);
extern char* muName(GA_CENTER_PTR ga_center);

extern void reSelect(GA_CENTER_PTR ga_center,char* fn_name);
extern char* reName(GA_CENTER_PTR ga_center);

extern void gaSelect(GA_CENTER_PTR ga_center,char* fn_name);
extern char* gaName(GA_CENTER_PTR ga_center);
```

```
/*=====================================================================
| ga_center memory setup and ga_center default initialization
=====================================================================*/
/*---------------------------------------------------------------------------------------------------------
| gacValid()   -check to see if a ga_center is valid
---------------------------------------------------------------------------------------------------------*/
int gacValid(GA_CENTER_PTR ga_center)
{  /*--- check for null pointers ---*/
   if(ga_center==NULL)       return FALSE;
   /*--- check for magic card ---*/
   if(ga_center->magic_card != GA_CARD) return FALSE;
   /*--- otherwise valid ---*/
   return TRUE;
}
/*---------------------------------------------------------------------------------------------------------
| gacReset()   -reset (parameters for) a ga_center
---------------------------------------------------------------------------------------------------------*/
void gacReset(GA_CENTER_PTR ga_center)
{ int i;
 /*--- check the validation of ga_center ---*/
 if(!gacValid(ga_center)) ERROR("gacReset(): invalid ga_center, exit");
 /*--- default basic parameters ---*/
 ga_center->rand_seed =1;
 ga_center->ip_flag   =IP_RANDOM;
 ga_center->ip_file[0]='\0';
 ga_center->pool_max_size=PLSIZE;    /*allow max 100 chromosomes in  pool*/
 ga_center->chrom_dim =CHROMDIM; /*assume 10 stations*/
 ga_center->max_iter  =-1;            /*no limitation on iter at initial*/
 ga_center->iter  =-1;                /*no iteration at the beginning*/
 ga_center->minimize  =TRUE;          /*default minimize optimization*/
 ga_center->converged =FALSE;         /*flag to tell if ga converge now*/
 ga_center->use_converge=TRUE;        /*stop at converged */
 ga_center->se_elitist=2;             /*default set select elitism to gaElitist1()*/
 ga_center->re_elitist=0;             /*default set replace elitism disabled*/
 ga_center->elitist   =ELITIST;       /*percentage of top bests transfered*/
 ga_center->gap       =GAP;           /*generation gap,percent of copy of old*/
 ga_center->critia    =CRITIA;        /*test variance dev of pool*/
 ga_center->bias      =BIAS;          /*rank_biased selection pressure*/
 ga_center->x_rate    =XRATE;         /*crossover rate*/
 ga_center->mu_rate   =MURATE; /*mutation rate*/
 ga_center->mu_flag   =MU_CHROM; /*mutation selection level flag*/
 ga_center->sam       =0.0;           /*default set SAM disabled*/
 ga_center->sac       =0.0;           /*default set SAC disabled*/
 /*--- default operators,only leave EV_fun for gaConfig() ---*/
 gaSelect(ga_center,"generational");
 seSelect(ga_center,"fitness_biased");
 xSelect(ga_center,"1xp_crossover");
 muSelect(ga_center,"swap");
 reSelect(ga_center,"append");

 /*---default report parameters--*/
 ga_center->rp_type=RP_LONG;
 ga_center->rp_interval=1;
 ga_center->rp_fid=stdout;
```

```
  ga_center->rp_file[0]='\0';

  /*--reset pools--*/
  /*--in gacAlloc(), set old_pool=NULL, new_pool=NULL, only gaSetup
      alloc old_pool and new_pool memory space--*/
  if(poolValid(ga_center->old_pool)) poolReset(ga_center->old_pool);
  if(poolValid(ga_center->new_pool)) poolReset(ga_center->new_pool);
  /*--reset the best chromosome--*/
  /*--in gacAlloc() set best=NULL, otherwise best is null terminated
      by the last cell, only gaSetup() alloc best array memory space--*/
  i=0;
  if(ga_center->best!=NULL)
  { while(chromValid(ga_center->best[i])){  chromReset(ga_center->best[i]);  i++; }
  }
}
/*-------------------------------------------------------------------------------------------------------------
| gacAlloc(void) -allocate a ga_center
-------------------------------------------------------------------------------------------------------------*/
GA_CENTER_PTR gacAlloc(void)
{  GA_CENTER_PTR ga_center;
   /*--- allocate the memory for ga_center ---*/
   if((ga_center=(GA_CENTER_PTR)calloc(1, sizeof(GA_CENTER_TYPE)))==NULL)
     ERROR("gacAlloc(void): alloc ga_center fails!");
   /*--- make sure to NULL several pointers in the new ga_center ---*/
   ga_center->old_pool=NULL;
   ga_center->new_pool=NULL;
   ga_center->best    =NULL;
   /*--- put magic card ---*/
   ga_center->magic_card=GA_CARD;
   /*--- initialize the ga_center ---*/
   gacReset(ga_center);
   return ga_center;
}
/*-------------------------------------------------------------------------------------------------------------
| gacFree()   -Free the space allocated to a ga_center
-------------------------------------------------------------------------------------------------------------*/
void gacFree(GA_CENTER_PTR ga_center)
{ int i;
  /*--- check the validation ---*/
  if(!gacValid(ga_center)) ERROR("gacFree():gacValid() check fails, exit");
  /*--- free memory of pools in ga ---*/
  if(poolValid(ga_center->old_pool)) poolFree(ga_center->old_pool);
  if(poolValid(ga_center->new_pool)) poolFree(ga_center->new_pool);
  ga_center->old_pool=NULL;  ga_center->new_pool=NULL;
  /*--- free memory of best chrom history array in ga ---*/
  /*--in gacAlloc() set best=NULL, otherwise best is null terminated
      by the last cell, only gaSetup() alloc best array memory space--*/
  i=0;
  if(ga_center->best!=NULL)
  {   while(chromValid(ga_center->best[i]))
      {   chromFree(ga_center->best[i]); ga_center->best[i]=NULL; i++;
      }
      free(ga_center->best); ga_center->best=NULL;
  }
```

```
 /*--- put in NULL card ---*/
 ga_center->magic_card=NL_CARD;
 /*--- free ga_center ---*/
 free(ga_center); ga_center=NULL;
}
/*=====================================================================
| ga_center configuration by reading from configuration file
======================================================================
/*------------------------------------------------------------------------------------------------------------
| gacLine()    -line reader of config file, subroution used by gacRead()
------------------------------------------------------------------------------------------------------------*/
int gacLine(char* line, char token[][STRLEN])
{ int i,j,len,toknum;
 toknum=0;
 len=strlen(line);
 for(i=0;i<=len-1; )
 { /*--find token--*/
   while(isspace(line[i])&&(i<=len-1)) i++;/*--passing white spaces--*/
   if(i>=len||line[i]=='#'||line[i]=='\n')break;/*skip remark/blank lines*/
   /*--meet one token and begin to save it into token[][] of next row --*/
   for(j=0; !isspace(line[i]) && line[i]!='\n' && i<=len-1; i++,j++ ) token[toknum][j]=line[i];
   token[toknum++][j]=0;/*null terminate this token row and move to next*/
 }
 return toknum; /*the number of tokens read from the line*/
}
/*------------------------------------------------------------------------------------------------------------
| gacRead()    -configuring ga_center by reading configuration file. MAXTOK 10 Maximum number of
tokens on a line, STRLEN 80 string len equal to the length of an input line.
------------------------------------------------------------------------------------------------------------*/
int gacRead(GA_CENTER_PTR ga_center,char* cfgfile)
{ static char str[STRLEN];
 static char token[MAXTOK][STRLEN];
 char*  file_mode; int toknum; FILE *fid;
 /*--- check the validation ---*/
 if(!gacValid(ga_center))ERROR("gacRead():gacValid() check fails, exit");
 if(cfgfile==NULL)ERROR("gacRead(): null cfg file name, exit");
 /*---open config file--*/
 if((fid=fopen(cfgfile,"r"))==NULL)ERROR("gacRead:err open cfgfile,exit");
 /*---read cfg file line by line---*/
 while(fgets(str,STRLEN,fid)!=NULL)
 { /*--convert to tokens---*/
   if((toknum=gacLine(str,token))<=0)continue;/*goto bottom of while-loop*/
   /*--reset ga_center parameters accordingly--*/
   switch(token[0][0])
   {   case 'b':
      if(!strcmp(token[0],"bias"))
      {   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->bias))==1);
          else ERROR("gacRead(): invalid bias response in cfgfile, exit");
      }
      else ERROR("gacRead(): case 'b': unknows command in cfgfile, exit");
      break;
      case 'c':
      if(!strcmp(token[0],"chrom_dim"))
      {   if(toknum>=2 && sscanf(token[1],"%d",&(ga_center->chrom_dim))==1);
```

```
    else ERROR("gacRead():invalid chrom_dim response in cfgfile,exit");
}
else if(!strcmp(token[0],"crossover"))
{   if(toknum>=2) xSelect(ga_center,token[1]);
    else ERROR("gacRead():invalid crossover response in cfgfile,exit");
}
else if(!strcmp(token[0],"critia"))
{   if(toknum>=2 && sscanf(token[1],"%lf",&(ga_center->critia))==1);
    else ERROR("gacRead(): invalid critia response in cfgfile, exit");
}
else ERROR("gacRead(): case 'c': unknows command in cfgfile, exit");
break;
case 'e':
if(!strcmp(token[0],"elitist"))
{   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->elitist))==1);
    else ERROR("gacRead(): invalid elitist response in cfgfile, exit");
}
else ERROR("gacRead(): case 'e': unknows command in cfgfile, exit");
break;
case 'g':
if(!strcmp(token[0],"gap"))
{   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->gap))==1);
    else ERROR("gacRead(): invalid gap response in cfgfile, exit");
}
else if(!strcmp(token[0],"ga"))
{   if(toknum>=2) gaSelect(ga_center,token[1]);
    else ERROR("gacRead(): invalid ga response in cfgfile, exit");
}
else ERROR("gacRead(): case 'g': unknows command in cfgfile, exit");
break;
case 'i':
if(!strcmp(token[0],"initpool"))
{   if(toknum>=2 && !strcmp(token[1],"random"))ga_center->ip_flag=IP_RANDOM;
    else if(toknum>=2 && !strcmp(token[1],"from_file"))
    {  ga_center->ip_flag=IP_FILE;
       if(toknum>=3) strcpy(ga_center->ip_file,token[2]);
       else ERROR("gacRead(): miss ip_file name in cfgfile, exit");
    }
    else if(toknum>=2 && !strcmp(token[1],"interactive"))ga_center->ip_flag=IP_INTERACT;
    else ERROR("gacRead(): invalid initpool response in cfgfile,exit");
}
else ERROR("gacRead(): case 'i': unknows command in cfgfile, exit");
break;
case 'm':
if(!strcmp(token[0],"mutation"))
{   if(toknum>=2) muSelect(ga_center,token[1]);
    else ERROR("gacRead(): invalid mutation response in cfgfile,exit");
}
else if(!strcmp(token[0],"mu_rate"))
{   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->mu_rate))==1);
    else ERROR("gacRead(): invalid mu_rate response in cfgfile, exit");
}
else if(!strcmp(token[0],"mu_flag"))
{   if(toknum>=2 && !strcmp(token[1],"MU_CHROM"))ga_center->mu_flag=MU_CHROM;
```

```
    else if(toknum>=2 && !strcmp(token[1],"MU_BIT"))ga_center->mu_flag=MU_BIT;
    else if(toknum>=2 && !strcmp(token[1],"MU_CHILD"))ga_center->mu_flag=MU_CHILD;
    else ERROR("gacRead(): invalid mu_flag response in cfgfile, exit");
}
else ERROR("gacRead(): case 'm': unknows command in cfgfile, exit");
break;
case 'o':
if(!strcmp(token[0],"objective"))
{   if(toknum>=2 && !strcmp(token[1],"minimize")) ga_center->minimize=TRUE;
    else if(toknum>=2 && !strcmp(token[1],"maximize"))ga_center->minimize=FALSE;
    else ERROR("gacRead():invalid objective response in cfgfile,exit");
}
else ERROR("gacRead(): case 'o': unknows command in cfgfile, exit");
break;
case 'p':
if(!strcmp(token[0],"pool_max_size"))
{   if(toknum>=2 && sscanf(token[1],"%d",&(ga_center->pool_max_size))==1);
    else ERROR("gacRead:invalid pl_max_size response in cfgfile,exit");
}
else ERROR("gacRead(): case 'p': unknows command in cfgfile, exit");
break;
case 'r':
if(!strcmp(token[0],"rand_seed"))
{   if(toknum>=2 && !strcmp(token[1],"my_pid"))ga_center->rand_seed=1;/*getpid();*/
    else if(toknum>=2&&sscanf(token[1],"%d",&(ga_center->rand_seed))==1);
    else ERROR("gacRead():invalid rand_seed response in cfgfile,exit");
}
else if(!strcmp(token[0],"re_elitist"))
{   if(toknum>=2 && sscanf(token[1],"%d",&(ga_center->re_elitist))==1);
    else ERROR("gacRead:invalid re_elitist response in cfgfile,exit");
}
else if(!strcmp(token[0],"replacement"))
{   if(toknum>=2) reSelect(ga_center,token[1]);
    else ERROR("gacRead:invalid replacement response in cfgfile,exit");
}
else if(!strcmp(token[0],"rp_interval"))
{   if(toknum>=2 && sscanf(token[1],"%d",&(ga_center->rp_interval))==1);
    else ERROR("gacRead:invalid rp_interval response in cfgfile,exit");
}
else if(!strcmp(token[0],"rp_type"))
{   if(toknum>=2 && !strcmp(token[1],"minimal"))ga_center->rp_type=RP_MINI;
    else if(toknum>=2 && !strcmp(token[1],"short"))ga_center->rp_type=RP_SHORT;
    else if(toknum>=2 && !strcmp(token[1],"long"))ga_center->rp_type=RP_LONG;
    else if(toknum>=2 && !strcmp(token[1],"none"))ga_center->rp_type=RP_NONE;
    else ERROR("gacRead(): invalid rp_type response in cfgfile, exit");
}
else if(!strcmp(token[0],"rp_file"))
{   if(toknum>=2)
    {   file_mode="a";
        strcpy(ga_center->rp_file,token[1]);/*save filename*/
        if(toknum>=3) file_mode=token[2];/*get file mode if provided*/
        ga_center->rp_fid=fopen(ga_center->rp_file,file_mode);/*open rpf*/
        if(ga_center->rp_fid==NULL)ERROR("gacRead(): err open rp_file in cfgfile, exit");
    }
```

```
          else ERROR("gacRead(): invalid rp_file response in cfgfile, exit");
        }
        else ERROR("gacRead(): case 'r': unknows command in cfgfile, exit");
        break;
        case 's':
        if(!strcmp(token[0],"sam"))
        {   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->sam))==1);
          else ERROR("gacRead(): invalid sam response in cfgfile, exit");
        }
        else if(!strcmp(token[0],"sac"))
        {   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->sac))==1);
          else ERROR("gacRead(): invalid sac response in cfgfile, exit");
        }
        else if(!strcmp(token[0],"se_elitist"))
        {   if(toknum>=2 && sscanf(token[1],"%d",&(ga_center->se_elitist))==1);
          else ERROR("gacRead:invalid se_elitist response in cfgfile,exit");
        }
        else if(!strcmp(token[0],"selection"))
        {   if(toknum>=2) seSelect(ga_center,token[1]);
          else ERROR("gacRead():invalid selection response in cfgfile,exit");
        }
        else if(!strcmp(token[0],"stop_after"))
        {   if(toknum==2 && !strcmp(token[1],"convergence"))
          {   ga_center->use_converge=TRUE;
              ga_center->max_iter=-1;
          }
          else if(toknum>=2&&sscanf(token[1],"%d",&(ga_center->max_iter))==1)
          {   if(ga_center->max_iter<1)ERROR("gacRead():err num for stp,exit");
              ga_center->use_converge=TRUE;
             if(toknum>2 && !strcmp(token[2],"ignore_convergence"))
                ga_center->use_converge=FALSE;
          }
          else ERROR("gacRead:invalid stop_after response in cfgfile,exit");
        }
        else ERROR("gacRead(): case 's': unknows command in cfgfile, exit");
        break;
        case 'x':
        if(!strcmp(token[0],"x_rate"))
        {   if(toknum>=2 && sscanf(token[1],"%f",&(ga_center->x_rate))==1);
          else ERROR("gacRead(): invalid x_rate response in cfgfile, exit");
        }
        else ERROR("gacRead(): case 'x': unknows command in cfgfile, exit");
        break;
        default:
        ERROR("gacRead(): no match switch cases, unknows cfg command, exit");
    }
  }
  fclose(fid);
  return OK;
}
/*------------------------------------------------------------------------------------------------------------
| gacVerify()   -verify ga configuration
------------------------------------------------------------------------------------------------------------*/
void gacVerify(GA_CENTER_PTR ga_center)
```

```
{ /*--- check the validation ---*/
 if(!gacValid(ga_center)) ERROR("gacVerify():gacValid() check fails,exit");
 /*--- verifying parameters---*/
 switch(ga_center->ip_flag)
 { case IP_NONE:       break;
   case IP_RANDOM:  break;
   case IP_INTERACT: break;
   case IP_FILE: if(ga_center->ip_file[0]=='\0')
                    ERROR("gacVerify: no file for initpool but IP_FILE flag on, exit");
                 break;
   default:  ERROR("gacVerify(): invalid ip_flag, exit");
 }
 if(ga_center->pool_max_size<=0)ERROR("gacVer:invalid pool_max_size,exit");
 if(ga_center->chrom_dim<=0)ERROR("gacVerify(): invalid chrom_dim, exit");
 if(ga_center->minimize!=TRUE && ga_center->minimize!=FALSE)
     ERROR("gacVerify(): invalid value for minimize, exit");
 if(ga_center->use_converge!=TRUE && ga_center->use_converge!=FALSE)
     ERROR("gacVerify(): invalid value for use_converge, exit");
 if(ga_center->se_elitist!=0 && ga_center->se_elitist!=1 && ga_center->se_elitist!=2)
     ERROR("gacVerify(): invalid se_elitist flag, exit");
 if(ga_center->re_elitist!=0 && ga_center->re_elitist!=1)
     ERROR("gacVerify(): invalid re_elitist flag, exit");
 if(ga_center->elitist<0.0 || ga_center->elitist>=1.0)/*elitist:[0.0,1.0)*/
     ERROR("gacVerify(): invalid value for elitist:[0.0,1.0), exit");
 if(ga_center->critia<0.0 || ga_center->critia>=1.0) /*critia:[0.0,1.0)*/
     ERROR("gacVerify(): invalid value for critia:[0.0,1.0), exit");
 if(ga_center->gap<0.0 || ga_center->gap>1.0)/*generational gap:[0.0,1.0]*/
     ERROR("gacVerify(): invalid value for gap:[0.0,1.0], exit");
 if(ga_center->bias<0.0 || ga_center->bias>1.0)/*only used for rank_b */
     ERROR("gacVerify(): invalid value for bias, exit");/*[0.0, 1.0]*/
 if(ga_center->x_rate<0.0 || ga_center->x_rate>1.0)  /*x_rate [0.0, 1.0]*/
     ERROR("gacVerify(): invalid value for x_rate:[0.0, 1.0], exit");
 if(ga_center->mu_rate<0.0 || ga_center->mu_rate>1.0)/*mu_rate [0.0, 1.0]*/
     ERROR("gacVerify(): invalid value for mu_rate:[0.0, 1.0], exit");
 if(ga_center->mu_flag!=MU_BIT&&ga_center->mu_flag!=MU_CHROM
                                &&ga_center->mu_flag!=MU_CHILD)
     ERROR("gacVerify(): invalid mu_flag, exit");
 if(ga_center->sam<0.0 || ga_center->sam>1.0)  /*sam:[0.0, 1.0]*/
      ERROR("gacVerify(): invalid value for sam:[0.0, 1.0], exit");
 if(ga_center->sac<0.0 || ga_center->sac>1.0)  /*sac:[0.0, 1.0]*/
      ERROR("gacVerify(): invalid value for sac:[0.0, 1.0], exit");
 if(ga_center->GA_fun==NULL)
      ERROR("gacVerify(): no ga function specified, exit");
 if(ga_center->SE_fun==NULL)
     ERROR("gacVerify(): no selection function specified, exit");
 if(ga_center->x_rate>0.0 && ga_center->X_fun==NULL)/*x_rat=0.0 disabl*/
     ERROR("gacVerify():x_rate>0 but no crossover function specified, exit");
 if(ga_center->mu_rate>0.0 && ga_center->MU_fun==NULL)/*mu_rat=0.0 disabl*/
     ERROR("gacVerify():mu_rate>0 but no mutation function specified, exit");
 /*if(ga_center->X_fun==NULL)
      ERROR("gacVerify(): no crossover function specified, exit"); */
 /*if(ga_center->MU_fun==NULL)
      ERROR("gacVerify(): no mutation function specified, exit");*/
 if(ga_center->RE_fun==NULL)
```

```
      ERROR("gacVerify(): no replacement function specified, exit");
  if(ga_center->EV_fun==NULL)
      ERROR("gacVerify(): no evaluation function specified, exit");
  switch(ga_center->rp_type)
  { case RP_NONE:    break;
    case RP_MINI:    break;
    case RP_SHORT:   break;
    case RP_LONG:    break;
    default:         ERROR("gacVerify(): invalid rp_type, exit");
  }
  if(ga_center->rp_interval<=0)
      ERROR("gacVerify(): invalid report interval, exit");
  if(ga_center->rp_fid==NULL)
      ERROR("gacVerify(): no rport file id specified, exit");
  if(ga_center->rp_fid!=stdout && ga_center->rp_file[0]=='\0')
      ERROR("gacVerify(): no report file name specified, exit");
}
/*=================================================================
| ga_center report
=================================================================*/
/*-----------------------------------------------------------------
| gacPrint() ----print ga_center information to rp_fid. due to report on pool actural size, this function must
be called after the poolInit(), that is, after the gaSetup() in ga function.
-----------------------------------------------------------------*/
void gacPrint(GA_CENTER_PTR ga_center)
{ FILE* fid;
  /*===check the validation==*/
  if(!gacValid(ga_center))ERROR("gacPrint():gacValid() check fails,exit");
  if(ga_center->rp_fid==NULL)ERROR("gacPrint():invalid rp_fid in ga,exit");
  /*===print ga_center info to rp_fid==*/
  fid=ga_center->rp_fid;
  /*===print header line===*/
  fprintf(fid,"\n");
  fprintf(fid,"-------------------begin gac config----");
  fprintf(fid,"---------------------------------------");
  fprintf(fid,"\n");
  /*---header---*/
  fprintf(fid,"ga_center configuration information:\n");
  fprintf(fid,"------------------------------------\n");
  /*---basic info---*/
  fprintf(fid,"basic parameters:\n");
  fprintf(fid,"rand_seed        : %d \n",ga_center->rand_seed);
  fprintf(fid,"Init pool entered:  ");
  switch(ga_center->ip_flag)
  { case IP_FILE       :fprintf(fid,"from file\n"    ); break;
    case IP_RANDOM     :fprintf(fid,"randomly\n"     ); break;
    case IP_INTERACT:fprintf(fid,"interactively\n"); break;
    default            :fprintf(fid,"unspecified\n"  ); break;
  }
  fprintf(fid,"init pool file name:  ");
  if(ga_center->ip_file[0]=='\0')fprintf(fid,"NONE\n");
  else fprintf(fid,"%s\n",ga_center->ip_file);
  fprintf(fid,"allowed max pool size  :  %d\n",ga_center->pool_max_size);
  fprintf(fid,"actural pool size      :  %d\n",ga_center->old_pool->size);
```

```
/*note:due to above, this funct must be called after pool initialing*/
fprintf(fid,"defined chrom matrix dim:  %d\n",ga_center->chrom_dim);
fprintf(fid,"allowed max trials      :  ");
if(ga_center->max_iter<0)fprintf(fid,"run until convergence\n");
else fprintf(fid,"%d iterations, %s\n",ga_center->max_iter,
                ga_center->use_converge? "or converged":"ignore converge");
fprintf(fid,"current iteration       :  %d\n",ga_center->iter);
fprintf(fid,"minimize optimazition   :  %s\n",ga_center->minimize? "yes":"no");
fprintf(fid,"ga converged ?             %s\n",ga_center->converged? "yes":"no");
fprintf(fid,"ga use converge critia ?   %s\n",ga_center->use_converge? "yes":"no");
fprintf(fid,"ga converge critia      :  %G\n",ga_center->critia);
fprintf(fid,"select elitism policy   :  %s\n",
            ga_center->se_elitist==0 ? "no":
            ga_center->se_elitist==1 ? "transfer two copies of the best":
            ga_center->se_elitist==2 ? "transfer percentage of top bests":"Unknown");
fprintf(fid,"replace elitism policy  :  %s\n",ga_center->re_elitist? "yes":"no");
fprintf(fid,"elitist percent         :  %G\n",ga_center->elitist);
fprintf(fid,"ga generation gap       :  %G\n",ga_center->gap);
fprintf(fid,"ga rank_biased pressure :  %G\n",ga_center->bias);
fprintf(fid,"ga crossover rate       :  %G\n",ga_center->x_rate);
fprintf(fid,"ga mutation rate        :  %G\n",ga_center->mu_rate);
fprintf(fid,"ga mutation method      :  ");
switch(ga_center->mu_flag)
{ case MU_CHROM: fprintf(fid,"mutation whole pool from chrom level\n");break;
  case MU_BIT    : fprintf(fid,"mutation whole pool from bit level\n");break;
  case MU_CHILD  : fprintf(fid,"mutation single chrom immidiately\n");break;
  default        : fprintf(fid,"mutation flag is invalid \n"); break;
}
fprintf(fid,"probability to invoke SAM:  %G\n",ga_center->sam);
fprintf(fid,"probability to invoke SAC:  %G\n",ga_center->sac);
/*==ga registered operations==*/
fprintf(fid,"\nga registered operations:\n");
fprintf(fid,"ga function         :  %s\n",gaName(ga_center));
fprintf(fid,"ga selection        :  %s\n",seName(ga_center));
fprintf(fid,"ga crossover        :  %s\n",xName(ga_center));
fprintf(fid,"ga mutation         :  %s\n",muName(ga_center));
fprintf(fid,"ga replace          :  %s\n",reName(ga_center));
/*===Report==*/
if(ga_center->rp_type!=RP_NONE)
{ fprintf(fid,"\nga reports:\n");
  /*---report type--*/
  fprintf(fid,"report type:  %s\n",
             ga_center->rp_type==RP_MINI? "Minimal":
             ga_center->rp_type==RP_SHORT?  "Short":
             ga_center->rp_type==RP_LONG?    "Long": "Unkown");
  /*---report interval--*/
  fprintf(fid,"rp_interval:  %d\n",ga_center->rp_interval);
  /*-- report file name--*/
  if(ga_center->rp_file[0]!=0)
  { fprintf(fid,"report file name:  %s\n",
               (!strcmp(ga_center->rp_file,"UNSPECIFIED"))? "Unspecified": ga_center->rp_file);
  }
}
```

```
 /*===print footer line===*/
 fprintf(fid,"\n");
 fprintf(fid,"--------------------end gac config-----");
 fprintf(fid,"---------------------------------------");
 fprintf(fid,"\n");
 /*==print onto fid immidiately==*/
 fflush(fid);
}



/*========== end of file: gac.c ===========*/



file: function.c
/*========================================================================
| Function table management
|
| iniTable()-------initial new created FN_Table array
| setFun()---------write user defined function into FN_TABLE and GA_CENTER
| selectFun()------get FN_PTR from FN_TABLE and write into GA_CENTER
| getFunName()--get fun name from FN_TABLE by providing FN_PTR
|
| note: FN_TABLE[] must be NULL terminated for function search to be stop
========================================================================*/
#include "gah.h"
/*========================================================================
| function prototype
========================================================================*/
void iniTable(FN_TABLE_PTR FN_Table,int cellNum);
void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                        char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                        char* fn_name,FN_PTR* rtn_ptr);
char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);
extern int gacValid(GA_CENTER_PTR ga_center);
/*------------------------------------------------------------------------
| iniTable()  -initial new created FN_Table array
------------------------------------------------------------------------*/
void iniTable(FN_TABLE_PTR FN_Table,int cellNum)
{ int i;
 /*--- error check ---*/
 if(FN_Table==NULL) ERROR("iniTable(): Null FN_Table array addr. exit");
 /*--- initializing ---*/
 for(i=0;i<=cellNum-1;i++){ FN_Table[i].name=NULL; FN_Table[i].fun =NULL;}
}
/*------------------------------------------------------------------------
| setFun() -write user defined function into FN_TABLE and GA_CENTER
------------------------------------------------------------------------*/
void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                        char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr)
```

```
{ /*--- error check ---*/
 if(!gacValid(ga_center)) ERROR("setFun(): invalid ga_center, exit");
 if(FN_Table==NULL) ERROR("setFun(): Null FN_Table array addr, exit");
 if(strlen(fn_name)==0) ERROR("setFun(): no user fun name, exit");
 if(fn_ptr==NULL) ERROR("setFun(): Null user fun ptr, exit");
 if(rtn_ptr==NULL) ERROR("setFun(): Null rtn fun ptr, exit");
 /*--- free current function name string space ---*/
 if(FN_Table[0].name!=NULL)  /*0 cell of FN_Table array is for user*/
 { free(FN_Table[0].name);      /*free current name string dynalloced*/
   FN_Table[0].name=NULL;   /*null name str ptr prepare for wrt*/
 }
 /*--- write user function into 0 th cell of FN_Table array ---*/
 /*--- allocate the memory for the function name string ---*/
 FN_Table[0].name=(char*)calloc(strlen(fn_name)+1,sizeof(char));
 if(FN_Table[0].name==NULL)ERROR("setFun(): alloc namestrspace fail,exit");
 /*--- copy user function name into 0 th cell of FN_Table array ---*/
 strcpy(FN_Table[0].name,fn_name);
 /*--- write user function ptr into 0 th cell of FN_Table array ---*/
 FN_Table[0].fun=fn_ptr;
 /*--- set user function ptr into GA_CENTER ---*/
 *rtn_ptr=fn_ptr;
 /*rtn_ptr hold the addr of variable ga_center->X_fun for example*/
}
/*------------------------------------------------------------------------------------------------------------------
| selectFun() -get FN_PTR from FN_TABLE and write into GA_CENTER
------------------------------------------------------------------------------------------------------------------*/
void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                        char* fn_name,FN_PTR* rtn_ptr)
{ int i;
 /*--- error check ---*/
 if(!gacValid(ga_center)) ERROR("selectFun(): invalid ga_center, exit");
 if(FN_Table==NULL) ERROR("selectFun(): Null FN_Table array addr, exit");
 if(strlen(fn_name)==0) ERROR("selectFun(): no selected fun name, exit");
 /***--rtn_ptr hold addr of function ptr variable of ga_center-*/
 if(rtn_ptr==NULL) ERROR("selectFun(): Null rtn fun ptr, exit");
 /*-- finding matched function in FN_Table array ---*/
 /*-- check user defined function cell --*/
 if((FN_Table[0].name!=NULL)&&(!strncmp(fn_name,FN_Table[0].name,strlen(FN_Table[0].name))))
 { if(FN_Table[0].fun==NULL)
   { printf("user function: %s is null ptr\n",FN_Table[0].name);
     ERROR("selectFun():choose a null fun ptr(user) is an error, exit");
   }
  *rtn_ptr=FN_Table[0].fun;
  return;
 }
 /*--check prepared functions in the rest of cells--*/
 for(i=1;(strlen(FN_Table[i].name)>=1)&&(FN_Table[i].fun!=NULL); i++)
 { if(!strncmp(FN_Table[i].name,fn_name,MIN(strlen(fn_name), strlen(FN_Table[i].name))))
   { /*--write selected fun ptr into ga_center--*/
     *rtn_ptr=FN_Table[i].fun;
     return;
   }
 }
 /*--- invalid selection:no match fn_name or null fun ptr ---*/
 ERROR("selectFun():invalid selection is an error, exit");
}
/*-----------------------------------------------------------------------------------------------------------------
| getFunName() -get fun name from FN_TABLE by providing FN_PTR
-----------------------------------------------------------------------------------------------------------------*/
char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table, FN_PTR fn_ptr)
{ int i;
 /*--- error check ---*/
 if(!gacValid(ga_center)) ERROR("getFunName(): invalid ga_center, exit");
 if(FN_Table==NULL) ERROR("getFunName(): Null FN_Table array addr, exit");
 if(fn_ptr==NULL) ERROR("getFunName(): pass in NULL fn_ptr, exit");
 /*-- check FN_Table array cell by cell to find matched fn_ptr--*/
 for(i=0;i==0||(FN_Table[i].fun!=NULL);i++)
 { /*-if function of this cell match?-*/
   if(fn_ptr==FN_Table[i].fun)
   { /*--function match, but null name--*/
     if(FN_Table[i].name==NULL) return "unspecified";
      /*--function match, valid name --*/
     else  return FN_Table[i].name;
   }
 }
 /*--function not found --*/
 return "function not found";
}




/*========== end of file: function.c =========*/





file: selection.c
/*===========================================================================
| Ga selection operations:
|
| /*---------Selection Operators ---------------------------------------------------------------------------------
|  seByRandom()----------select one chrmosome pool index randomly, equal prob.
|  seByRoulettePtf()------select one chrmosome pool index use standard roulette spin and fitness-
|                          biased (ptf) values.
|  seByRouletteRank()---select one chrmosome pool index use standard roulette spin and rank-
|                          biased (rank_prob) values.
| /*---------Selection Interfaces with ga_center ------------------------------------------------------------------
|  seSet()--------set user defined selection operator into SE_Table[] and register it into  ga_center.
|  seSelect()-----select and register selection operator into ga by name
|  seName()-----get current ga registered selection operator name
|  seRunPool()--select one chrom. from a whole pool and perform ga registered selection operator.
===========================================================================*/
#include "gah.h"
/*===========================================================================
| function prototypes
==========================================================================*/
int seByRandom(GA_CENTER_PTR ga_center, POOL_PTR pool);
int seByRoulettePtf(GA_CENTER_PTR ga_center, POOL_PTR pool);
int seByRouletteRank(GA_CENTER_PTR ga_center, POOL_PTR pool);
void seSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr);
void seSelect(GA_CENTER_PTR ga_center,char* fn_name);
char* seName(GA_CENTER_PTR ga_center);
CHROM_PTR seRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);

extern int gacValid(GA_CENTER_PTR ga_center);
extern int poolValid(POOL_PTR pool);
extern void poolAlign(POOL_PTR pool);
extern void poolIndex(POOL_PTR pool);
extern void poolFitness(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolPtf(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolStats(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolRank(GA_CENTER_PTR ga_center,POOL_PTR pool);

extern void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                        char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
extern void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                        char* fn_name,FN_PTR* rtn_ptr);
extern char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);

FN_TABLE SE_Table[]=
{ {NULL,                NULL},
  {"uniform_random", seByRandom},
  {"fitness_biased", seByRoulettePtf},
  {"rank_biased", seByRouletteRank},
  {NULL,                NULL},
};
/*note: FN_TABLE[] must be NULL terminated for function search to be stop*/
/*==========================================================================
| selection operators
==========================================================================*/
/*------------------------------------------------------------------------
| seByRandom()   -select one chrmosome pool index randomly
------------------------------------------------------------------------*/
int seByRandom(GA_CENTER_PTR ga_center, POOL_PTR pool)/*must be FN_PTR type*/
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("seByRandom: gacValid check fails, exit");
 if(!poolValid(pool))ERROR("seByRandom: poolValid() check fails, exit");
 /*---align chrom ptr array of pool---*/
 if(!pool->sorted && !pool->updated)
   poolAlign(pool);/*if either updated or sorted is TRUE,not exe poolAlign*/
 /*---randomly pick up a number as selected chromosome index--*/
 return RANDDOM(0,pool->size-1);
}
/*------------------------------------------------------------------------
| seByRoulettePtf()---select one chrmosome pool index by using standard roulette and fitness-biased (ptf)
values. According to poolPtf(): when minimize, smaller fitness, bigger ptf, so bigger proba when
maximize, bigger fitness, bigger ptf, so bigger probab
------------------------------------------------------------------------*/
```

```
int seByRoulettePtf(GA_CENTER_PTR ga_center,POOL_PTR pool)/*must be FN_PTR type*/
{ int i;
  float accum, spin;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("seByRoulettePtf:gacValid check err,exit");
  if(!poolValid(pool))ERROR("seByRoulettePtf:poolValid check fails, exit");
  /*---update ptf for each chromosome--*/
  if(!pool->sorted && !pool->updated)poolAlign(pool);/*if sorted=TRUE, not exe poolAlign()*/
  if(!pool->updated)
  { poolIndex(pool);  poolFitness(ga_center,pool);
    poolPtf(ga_center, pool);  /*if updated=TRUE, poolPtf() will not entered*/
    poolStats(ga_center, pool);/*set updated=1,disable poolAlign() */
  }                              /*and the whole block next time*/
  /*---randomly  roulette a chromsome index according to their ptf --*/
  accum=0.0;spin=0.0;i=0;
  /*---spin the wheel to get a random fraction value between 0.0 and 1.0 */
  spin=RANDFRAC();
  /*---find the corresponding chromosome--*/
  while(accum<spin && i<pool->size) accum+=pool->chrom[i++]->ptf;
  if(i>0) i--;
  return i;
}
/*-----------------------------------------------------------------------------------------------------------------
| seByRouletteRank()----select one chrom pool index by using standard roulette and rank_biased(
rank_prob ) values. According to poolRank(): in sorted pool, smaller pool index,bigger rank_prob,so
bigger select prob,bigger pool index,smaller rank_prob,so smaller select prob.
-----------------------------------------------------------------------------------------------------------------*/
int seByRouletteRank(GA_CENTER_PTR ga_center,POOL_PTR pool)/*must be FN_PTR type*/
{ int i;
  float accum, spin;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("seRoulRank: gacValid check fails, exit");
  if(!poolValid(pool))ERROR("seRouletteRank: poolValid check fails,exit");
  /*---update rank_prob for each chromosome--*/
  if(pool->sorted!=2) poolRank(ga_center,pool);/*if sorted=2,no exe,!=2 exe,!=1,poolsort()*/
                                                /*only exe once by set sorted=2, disable next*/
  /*---randomly  roulette a chromsome index according to its rank_prob --*/
  accum=0.0;spin=0.0;i=0;
  /*---spin the wheel to get a random fraction value between 0.0 and 1.0 */
  spin=RANDFRAC();
  /*---find the corresponding chromosome--*/
  while(accum<spin && i<pool->size) accum+=pool->chrom[i++]->rank_prob;
  if(i>0) i--;
  return i;
}
/*=================================================================================
| Selection Interface with ga_center:
=================================================================================*/
/*-----------------------------------------------------------------------------------------------------------------
| seSet()-----set user defined selection  operator into SE_Table[] and register it into  ga_center
-----------------------------------------------------------------------------------------------------------------*/
void seSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr)
{ /*--- error check  ---*/
  if(!gacValid(ga_center)) ERROR("seSet(): gacValid() check fails, exit");
```

```
 if(strlen(fn_name)==0) ERROR("seSet(): no user fun name, exit");
 if(fn_ptr==NULL) ERROR("seSet(): Null user fun ptr, exit");
 /*--- call setFun() to operate on the SE_Table[]--*/
 setFun(ga_center,SE_Table,fn_name,fn_ptr,&(ga_center->SE_fun));
}
/*-----------------------------------------------------------------------------------------------------------
| seSelect()  -set and select selection operator into ga by name
-----------------------------------------------------------------------------------------------------------*/
void seSelect(GA_CENTER_PTR ga_center,char* fn_name)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("seSelect: gacValid check fails, exit");
 if(strlen(fn_name)==0) ERROR("seSelect(): no fun name provided, exit");
 /*--- call selectFun() to operate on the SE_Table[]--*/
 selectFun(ga_center,SE_Table,fn_name,&(ga_center->SE_fun));
}
/*-----------------------------------------------------------------------------------------------------------
| seName()    -get current ga registered selection operator name
-----------------------------------------------------------------------------------------------------------*/
char* seName(GA_CENTER_PTR ga_center)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("seName(): gacValid() check fails, exit");
 /*--- call getFunName() to operate on SE_Table[]--*/
 return getFunName(ga_center,SE_Table,ga_center->SE_fun);
}
/*-----------------------------------------------------------------------------------------------------------
| seRunPool()     -setup and perform selection operator
-----------------------------------------------------------------------------------------------------------*/
CHROM_PTR seRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ int i=0;
 int index=-1;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("seRunPool: gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("seRunPool(): poolValid() check fails, exit");
 if(ga_center->SE_fun==NULL)ERROR("seRunPl:no SE_fun provid in ga,exit");
 /*---select a chromsome pool index--*/
in:index=ga_center->SE_fun(ga_center,pool); i++;
 /*--check the validation of selected chromosome index--*/
 if(index<0 || index>=pool->size)
 { WARN("invalid chromsome index selected, try again\n");
    if(i==1000)ERROR("seRunPl:fail to select valid idx in 1000 try,exit");
    goto in;
 }
 if(pool->chrom[index]==NULL)ERROR("seRunPl:select a null chrom,exit");
 /*--successfully select a valid chromosome, then return it--*/
 return pool->chrom[index];
}



/*========= end of file: selection.c =======*/
```

file: cross.c

```
/*=========================================================================
| GA crossover operations:
|
|-------------------------------Utilities--------------------------------------------------------------------------------
| xRandom1xp()--randomly generate 1 crossover point
| xRandom2xp()--randomly generate 2 sorted crossover points
| xRandom3xp()--randomly generate 3 sorted crossover points
| xRandom4xp()--randomly generate 4 sorted crossover points
| xInitKid()--------initialize children chrom structs before crossover operation is carried out.
|------------------------------Basic ga crossover operators------------------------------------------------------
| xBy1xp()------crossover 2 parents to produce 2 children by using 1 xp
| xBy2xp()------crossover 2 parents to produce 2 children by using 2 xp
| xBy3xp()------crossover 2 parents to produce 2 children by using 3 xp
| xBy4xp()------crossover 2 parents to produce 2 children by using 4 xp
| xByFlip()-----crossover 2 parents to produce 2 children by copying  each gene from
|                  a parent based on a random flip of a fair coin.
| xByAsex()-----two parents produce two children by crossing himself, no  intercross.
|                   one parent produces one child.
|--------------------------------GA crossover user interface----------------------------------------------------
| xSet()----------set user defined crossover operator into X_Table[] and register it into  ga_center
| xSelect()-------set and select crossover operator into ga by name
| xName()-------get current crossover operator name
| xRunPair()----used by gnerational and Steady_state GA
|                  called by ga_SE_X_MU_RE_2CHILD(),ga_SE_X_RE_2CHILD()
| xRunPool()----used by traditional GA, called by gaTraditional()
|---------------------------------SAGA crossover user interfaces-----------------------------------------------
| sacRunPair()----used by gnerational and Steady_state SAGA
|                    called by ga_SE_X_MU_RE_2CHILD(),ga_SE_X_RE_2CHILD()
| sacRunPool()----used by traditional SAGA, called by gaTraditional()
=========================================================================*/
#include "gah.h"
/*=========================================================================
| Data Structures
=========================================================================*/
int xBy1xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
int xBy2xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
int xBy3xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
int xBy4xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
int xByFlip(GA_CENTER_PTR ga_center,
            CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
int xByAsex(GA_CENTER_PTR ga_center,
             CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);

FN_TABLE X_Table[]=
{ {NULL,             NULL},
  {"1xp_crossover", xBy1xp},
  {"2xp_crossover", xBy2xp},
  {"3xp_crossover", xBy3xp},
  {"4xp_crossover", xBy4xp},
  {"random_flip",  xByFlip},
  {"asexual",      xByAsex},
  {NULL,           NULL},
};
/*note: FN_TABLE[] must be NULL terminated for function search to be stop*/
/*=========================================================================
| function prototype
=========================================================================*/
void xRandom1xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp);
void xRandom2xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp);
void xRandom3xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp);
void xRandom4xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp);
void xInitkid(CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);

void xSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr);
void xSelect(GA_CENTER_PTR ga_center,char* fn_name);
char* xName(GA_CENTER_PTR ga_center);
void xRunPair(GA_CENTER_PTR ga_center,
              CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
void xRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int* n);
void sacRunPair(GA_CENTER_PTR ga_center,
                CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
void sacRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int* n);

extern int chromValid(CHROM_PTR );
extern void chromReset(CHROM_PTR);
extern int chromComp(GA_CENTER_PTR, CHROM_PTR chrom1, CHROM_PTR chrom2);
extern void chromRepair(CHROM_PTR );
extern void chromCopy(CHROM_PTR src, CHROM_PTR dst);

extern int poolValid(POOL_PTR pool);

extern int gacValid(GA_CENTER_PTR ga_center);
extern float accept(GA_CENTER_PTR ga_center,CHROM_PTR p,CHROM_PTR c);

extern void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                          char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
extern void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                    char* fn_name,FN_PTR* rtn_ptr);
extern char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);

extern CHROM_PTR seRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);

/*=========================================================================
| Utilities
=========================================================================*/
/*-------------------------------------------------------------------------
| xRandom1xp()--randomly generate 1 crossover point and stored in xp[0], xp[] is an int array of 4 cells
held max upto 4 xp
-------------------------------------------------------------------------*/
void xRandom1xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp)
{ /*==error check==*/
  if(!gacValid(ga_center))ERROR("xRandom1xp:gacValid() check fails,exit");
  if(minidx<0||minidx>ga_center->chrom_dim-1)ERROR("xRandom1xp(): invalid minidx, exit");
```

```
 if(maxidx<0||maxidx>ga_center->chrom_dim-1)ERROR("xRandom1xp(): invalid maxidx, exit");
 if(minidx>maxidx)ERROR("xRandom1xp(): minidx> maxidx, exit");
 /*note:it is allowed mindx=maxidx when only ask to gen. 1 xp.*/
 if(xp==NULL)ERROR("xRandom1xp(): invalid address of xp[], exit");
 /*==flush buffer xp[]==*/
 xp[0]=-1;xp[1]=-1;xp[2]=-1;xp[3]=-1;
 /*==randomly generate a crossover point in [minidx,maxidx]==*/
 xp[0]=RANDDOM(minidx,maxidx);
}
/*------------------------------------------------------------------------------------------------------------
| xRandom2xp()--randomly generate 2 sorted crossover points and stored in xp[] orderly as xp1=xp[0],
xp2=xp[1], xp[] is an int array of 4 cells held max upto 4 xp
------------------------------------------------------------------------------------------------------------*/
void xRandom2xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp)
{ int i;
 /*==error check==*/
 if(!gacValid(ga_center))ERROR("xRandom2xp:gacValid() check fails,exit");
 if(minidx<0||minidx>ga_center->chrom_dim-1)ERROR("xRandom2xp(): invalid minidx, exit");
 if(maxidx<0||maxidx>ga_center->chrom_dim-1)ERROR("xRandom2xp(): invalid maxidx, exit");
 if((maxidx-minidx)<=0)ERROR("xRandom2xp():maxidx-minidx<=0, exit");
 /*note:it is impossible to gen. 2 differ xps when maxidx-minidx<=0*/
 if(xp==NULL)ERROR("xRandom2xp(): invalid address of xp[], exit");
 /*==flush buffer xp[]==*/
 xp[0]=-1;xp[1]=-1;xp[2]=-1;xp[3]=-1;
 /*==randomly generate 2 sorted crossover points in [minidx,maxidx]==*/
 /*--randomly generate 2 points--*/
 xp[0]=RANDDOM(minidx,maxidx);  xp[1]=RANDDOM(minidx,maxidx);
 /*--make sure they are different--*/
 i=0;
 while(xp[1]==xp[0])
 { xp[1]=RANDDOM(minidx,maxidx);i++;
   if(i==1000)ERROR("xRandom2px:fail to gen. valid xp2 in 1000 try,exit");
 }
 /*--make sure they are sorted--*/
 if(xp[0]>xp[1])SWAP(&xp[0],&xp[1]);
}
/*------------------------------------------------------------------------------------------------------------
| xRandom3xp()--randomly generate 3 sorted crossover points and stored in xp[] orderly as xp1=xp[0],
xp2=xp[1], xp3=xp[2]; xp[] is an int array of 4 cells held max upto 4 xp
------------------------------------------------------------------------------------------------------------*/
void xRandom3xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp)
{ int i,j;
 /*==error check==*/
 if(!gacValid(ga_center))ERROR("xRandom3xp:gacValid() check fails,exit");
 if(minidx<0||minidx>ga_center->chrom_dim-1)ERROR("xRandom3xp(): invalid minidx, exit");
 if(maxidx<0||maxidx>ga_center->chrom_dim-1)ERROR("xRandom3xp(): invalid maxidx, exit");
 if((maxidx-minidx)<=1)ERROR("xRandom3xp():maxidx-minidx<=1, exit");
 /*note:it is impossible to gen. 3 differ xps when maxidx-minidx<=1*/
 if(xp==NULL)ERROR("xRandom3xp(): invalid address of xp[], exit");
 /*==flush buffer xp[]==*/
 xp[0]=-1;xp[1]=-1;xp[2]=-1;xp[3]=-1;
 /*==randomly generate 3 sorted crossover points in [minidx,maxidx]==*/
 /*--randomly generate 3 points--*/
 xp[0]=RANDDOM(minidx,maxidx);  xp[1]=RANDDOM(minidx,maxidx);
```

```
  xp[2]=RANDDOM(minidx,maxidx);
  /*--make sure they are different--*/
  i=0;
  while(xp[1]==xp[0])
  { xp[1]=RANDDOM(minidx,maxidx);i++;
    if(i==1000)ERROR("xRandom3px:fail to gen. valid xp2 in 1000 try,exit");
  }
  i=0;
  while(xp[2]==xp[0]||xp[2]==xp[1])
  { xp[2]=RANDDOM(minidx,maxidx);i++;
    if(i==1000)ERROR("xRandom3px:fail to gen. valid xp3 in 1000 try,exit");
  }
  /*--make sure they are sorted--*/
  for(i=1;i<=2;i++)
  { j=i;
    while((j-1)>=0 && xp[j-1]>xp[j]){SWAP(&xp[j-1],&xp[j]);j--;}
  }
}
/*-----------------------------------------------------------------------------------------------------------------
| xRandom4xp()--randomly generate 4 sorted crossover points and stored in xp[] orderly as xp1=xp[0],
xp2=xp[1], xp3=xp[2], xp4=xp[3]; xp[] is an int array of 4 cells held max upto 4 xps
-----------------------------------------------------------------------------------------------------------------*/
void xRandom4xp(GA_CENTER_PTR ga_center,int minidx,int maxidx,int* xp)
{ int i,j;
  /*==error check==*/
  if(!gacValid(ga_center))ERROR("xRandom4xp:gacValid() check fails,exit");
  if(minidx<0||minidx>ga_center->chrom_dim-1)ERROR("xRandom4xp(): invalid minidx, exit");
  if(maxidx<0||maxidx>ga_center->chrom_dim-1)ERROR("xRandom4xp(): invalid maxidx, exit");
  if((maxidx-minidx)<=2)ERROR("xRandom4xp():maxidx-minidx<=2, exit");
  /*note:it is impossible to gen. 4 differ xps when maxidx-minidx<=2*/
  if(xp==NULL)ERROR("xRandom4xp(): invalid address of xp[], exit");
  /*==flush buffer xp[]==*/
  xp[0]=-1;xp[1]=-1;xp[2]=-1;xp[3]=-1;
  /*==randomly generate 4 sorted crossover points in [minidx,maxidx]==*/
  /*--randomly generate 4 points--*/
  xp[0]=RANDDOM(minidx,maxidx);  xp[1]=RANDDOM(minidx,maxidx);
  xp[2]=RANDDOM(minidx,maxidx);  xp[3]=RANDDOM(minidx,maxidx);
  /*--make sure they are different--*/
  i=0;
  while(xp[1]==xp[0])
  { xp[1]=RANDDOM(minidx,maxidx);i++;
    if(i==1000)ERROR("xRandom4px:fail to gen. valid xp2 in 1000 try,exit");
  }
  i=0;
  while(xp[2]==xp[0]||xp[2]==xp[1])
  { xp[2]=RANDDOM(minidx,maxidx);i++;
    if(i==1000)ERROR("xRandom4px:fail to gen. valid xp3 in 1000 try,exit");
  }
  i=0;
  while(xp[3]==xp[0]||xp[3]==xp[1]||xp[3]==xp[2])
  { xp[3]=RANDDOM(minidx,maxidx);i++;
    if(i==1000)ERROR("xRandom4px:fail to gen. valid xp3 in 1000 try,exit");
  }
  /*--make sure they are sorted--*/
```

```
 for(i=1;i<=3;i++)
 { j=i;
   while((j-1)>=0 && xp[j-1]>xp[j]){SWAP(&xp[j-1],&xp[j]);j--;}
 }
}
/*-------------------------------------------------------------------------------------------------------------
| xInitKid()----initialize children chrom structs before crossover operation is carried out
-------------------------------------------------------------------------------------------------------------*/
void xInitkid(CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ /*===check the validation===*/
 if(!chromValid(p1))ERROR("xInitKid():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xInitKid():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xInitKid():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xInitKid():chromValid(c2) check fails,exit");
 if(p1->dim<=0)ERROR("xInitKid():invalid parent1 dim <=0,exit");
 if(p2->dim<=0)ERROR("xInitKid():invalid parent2 dim <=0,exit");
 /*===Reset children chroms==*/
 chromReset(c1);/*set c1->eva=0 by chromReset()*/
 chromReset(c2);/*set c2->eva=0 by chromReset()*/
 /*==set parent index into child==*/
 c1->parent1=p1->index; c1->parent2=p2->index;
 c2->parent1=p2->index; c2->parent2=p1->index;
}
/*=============================================================================
| GA crossover operators
=============================================================================*/
/*-------------------------------------------------------------------------------------------------------------
| xBy1xp()------crossover 2 parents to produce 2 children by using 1 xp
-------------------------------------------------------------------------------------------------------------*/
int xBy1xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j, xp[4];
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xBy1xp():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xBy1xp():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xBy1xp():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xBy1xp():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xBy1xp():chromValid(c2) check fails,exit");
 /*==randomly generate 1 crossover point==*/
 xRandom1xp(ga_center, 0, p1->dim-2, xp);
 c1->xp1=xp[0]; c2->xp1=xp[0];
 /*==the first half is same as parent==*/
 for(i=0;i<=xp[0];i++)
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 /*==the second half is swapped    between parents==*/
 for(i=xp[0]+1;i<=p1->dim-1;i++)
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j]; c2->gene[i][j]=p1->gene[i][j];}
 }
 chromRepair(c1); chromRepair(c2);/*set c1->eva=0 by chromRepair()*/
 return OK;
}
/*-------------------------------------------------------------------------------------------------------------
| xBy2xp()------crossover 2 parents to produce 2 children by using 2 xps
```

```
--------------------------------------------------------------------------------------------------------------*/
int xBy2xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j, xp[4];
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xBy2xp():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xBy2xp():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xBy2xp():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xBy2xp():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xBy2xp():chromValid(c2) check fails,exit");
 /*==randomly generate 2 sorted crossover points==*/
 xRandom2xp(ga_center, 0, p1->dim-2, xp);
 c1->xp1=xp[0];c1->xp2=xp[1];
 c2->xp1=xp[0];c2->xp2=xp[1];
 /*==the genes between and 2 xps are the same as parents==*/
 for(i=xp[0]+1;i<=xp[1];i++) /*copy directly the mid part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 /*==the rest parts are swapped    between parents==*/
 for(i=0;i<=xp[0];i++)/*swap the first part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 for(i=xp[1]+1;i<=p1->dim-1;i++)/*swap the third part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 chromRepair(c1); chromRepair(c2);/*set c1->eva=0 by chromRepair()*/
 return OK;
}
/*--------------------------------------------------------------------------------------------------------------
| xBy3xp()------crossover 2 parents to produce 2 children by using 3 xps
--------------------------------------------------------------------------------------------------------------*/
int xBy3xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j, xp[4];
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xBy3xp():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xBy3xp():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xBy3xp():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xBy3xp():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xBy3xp():chromValid(c2) check fails,exit");
 /*==randomly generate 3 sorted crossover points==*/
 xRandom3xp(ga_center, 0, p1->dim-2, xp);
 c1->xp1=xp[0];c1->xp2=xp[1];
 c2->xp1=xp[0];c2->xp2=xp[1];
 /*==the genes between  2 adjacent xps are swaped between parents==*/
 for(i=0;i<=xp[0];i++)/*copy directly the first part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 for(i=xp[0]+1;i<=xp[1];i++)/*swap the second part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 for(i=xp[1]+1;i<=xp[2];i++)/*copy directly the third part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 for(i=xp[2]+1;i<=p1->dim-1;i++)/*swap the forth part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 chromRepair(c1); chromRepair(c2);/*set c1->eva=0 by chromRepair()*/
 return OK;
}
/*---------------------------------------------------------------------------------------------------------------
| xBy4xp()------crossover 2 parents to produce 2 children by using 4 xps
---------------------------------------------------------------------------------------------------------------*/
int xBy4xp(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j, xp[4];
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xBy4xp():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xBy4xp():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xBy4xp():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xBy4xp():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xBy4xp():chromValid(c2) check fails,exit");
 /*==randomly generate 4 sorted crossover points==*/
 xRandom4xp(ga_center, 0, p1->dim-2, xp);
 c1->xp1=xp[0];c1->xp2=xp[1];
 c2->xp1=xp[0];c2->xp2=xp[1];
 /*==the genes between  2 adjacent xps are swaped between parents==*/
 for(i=0;i<=xp[0];i++)/*copy directly the first part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 for(i=xp[0]+1;i<=xp[1];i++)/*swap the second part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 for(i=xp[1]+1;i<=xp[2];i++)/*copy directly the third part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 for(i=xp[2]+1;i<=xp[3];i++)/*swap the forth part between parents*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j];c2->gene[i][j]=p1->gene[i][j];}
 }
 for(i=xp[3]+1;i<=p1->dim-1;i++)/*copy directly the fifth part to each child*/
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j];c2->gene[i][j]=p2->gene[i][j];}
 }
 chromRepair(c1); chromRepair(c2);/*set c1->eva=0 by chromRepair()*/
 return OK;
}
/*---------------------------------------------------------------------------------------------------------------
| xByFlip()---crossover 2 parents to produce 2 children by copying each gene from a parent based on a
random flip of a fair coin.
---------------------------------------------------------------------------------------------------------------*/
int xByFlip(GA_CENTER_PTR ga_center,
           CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xByFlip():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xByFlip():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xByFlip():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xByFlip():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xByFlip():chromValid(c2) check fails,exit");
 /*==randomly generate 2 children based on flip of a fair coin==*/
 for(i=0;i<=p1->dim-1;i++)
 { if(RANDBIT()) /*copy directly the ith gene to each child*/
   { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p1->gene[i][j]; c2->gene[i][j]=p2->gene[i][j];}
   }
   else /*swap the ith gene between parents*/
   { for(j=0;j<=p1->dim-1;j++){ c1->gene[i][j]=p2->gene[i][j]; c2->gene[i][j]=p1->gene[i][j];}
   }
 }
 chromRepair(c1); chromRepair(c2); /*set c1->eva=0 by chromRepair()*/
 return OK;
}
/*----------------------------------------------------------------------------------------------------------------
| xByAsex()-----two parents produce two children by crossing himself, no intercrossing. one parent
produces one child.
----------------------------------------------------------------------------------------------------------------*/
int xByAsex(GA_CENTER_PTR ga_center,
            CHROM_PTR p1, CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ int i,j, xp[4];
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("xByAsex():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("xByAsex():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("xByAsex():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("xByAsex():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("xByAsex():chromValid(c2) check fails,exit");
 /*==p1 produces c1==*/
 /*--randomly generate 1 crossover point--*/
 xRandom1xp(ga_center, 0, p1->dim-2, xp);
 c1->xp1=xp[0];
 /*--exchange two blocks about xp point--*/
 /*the second half is swapped to the first half*/
 for(i=xp[0]+1;i<=p1->dim-1;i++)
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[i-(xp[0]+1)][j]=p1->gene[i][j]; }
 }/*the last row index of first block is: p1->dim-1-(xp[0]+1) */
/*the first row index of second block is: p1->dim-1-(xp[0]+1)+1 =p1->dim-1-xp[0] */
 /*the first half is swapped to the second half*/
 for(i=0;i<=xp[0];i++)
 { for(j=0;j<=p1->dim-1;j++){ c1->gene[p1->dim-1-xp[0]+i][j]=p1->gene[i][j];}
 }
 /*==p2 produces c2==*/
 /*--randomly generate 1 crossover point--*/
 xRandom1xp(ga_center, 0, p2->dim-2, xp);
 c2->xp1=xp[0];
 /*--exchange two blocks about xp point--*/
 /*the second half is swapped to the first half*/
 for(i=xp[0]+1;i<=p2->dim-1;i++)
 { for(j=0;j<=p2->dim-1;j++){ c2->gene[i-(xp[0]+1)][j]=p2->gene[i][j];}
 }/*the last row index of first block is: p2->dim-1-(xp[0]+1) */
/*the first row index of second block is: p2->dim-1-(xp[0]+1)+1=p2->dim-1-xp[0] */
 /*the first half is swapped to the second half*/
 for(i=0;i<=xp[0];i++)
 { for(j=0;j<=p2->dim-1;j++){ c2->gene[p2->dim-1-xp[0]+i][j]=p2->gene[i][j];}
 }
 chromRepair(c1);/*set c1->eva=0 by chromRepair()*/
```

```
  chromRepair(c2);/*set c1->eva=0 by chromRepair()*/
  return OK;
}/*ok 5-26-96*/
/*==========================================================================
| GA crossover user interface
==========================================================================*/
/*--------------------------------------------------------------------------
| xSet()     -set user defined crossover operator into X_Table[] and register it into  ga_center
--------------------------------------------------------------------------*/
void xSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr)
{ /*--- error check  ---*/
  if(!gacValid(ga_center)) ERROR("xSet(): gacValid() check fails, exit");
  if(strlen(fn_name)==0) ERROR("xSet(): no user fun name, exit");
  if(fn_ptr==NULL) ERROR("xSet(): Null user fun ptr, exit");
  /*--- call setFun() to operate on the X_Table[]--*/
  setFun(ga_center,X_Table,fn_name,fn_ptr,&(ga_center->X_fun));
}
/*--------------------------------------------------------------------------
| xSelect()  -set and select crossover operator into ga by name
--------------------------------------------------------------------------*/
void xSelect(GA_CENTER_PTR ga_center,char* fn_name)
{ /*--- error check  ---*/
  if(!gacValid(ga_center)) ERROR("xSelect(): gacValid() check fails, exit");
  if(strlen(fn_name)==0) ERROR("xSelect(): no fun name provided, exit");
  /*--- call selectFun() to operate on the X_Table[]--*/
  selectFun(ga_center,X_Table,fn_name,&(ga_center->X_fun));
}
/*--------------------------------------------------------------------------
| xName()    -get current crossover operator name
--------------------------------------------------------------------------*/
char* xName(GA_CENTER_PTR ga_center)
{ /*--- error check  ---*/
  if(!gacValid(ga_center)) ERROR("xName(): gacValid() check fails, exit");
  /*--- call getFunName() to operate on X_Table[]--*/
  return getFunName(ga_center,X_Table,ga_center->X_fun);
}

/*--------------------------------------------------------------------------
| xRunPair()----GA crossover user interface to call ga registered crossover operator to crossover two passed
in parents to born two children if randomly generated probability is met x_rate. Two kids are put in global
variables child1, and child2 pass out. Used by gnerational and Steady_state GA, called by
ga_SE`X_MU_RE_2CHILD(), ga_SE_X_RE_2CHILD()
--------------------------------------------------------------------------*/
void xRunPair(GA_CENTER_PTR ga_center,
              CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("xRunPair():gacValid() check fails,exit");
  if(!chromValid(p1))ERROR("xRunPair():chromValid(p1) check fails,exit");
  if(!chromValid(p2))ERROR("xRunPair():chromValid(p2) check fails,exit");
  if(!chromValid(c1))ERROR("xRunPair():chromValid(c1) check fails,exit");
  if(!chromValid(c2))ERROR("xRunPair():chromValid(c2) check fails,exit");
  if(ga_center->X_fun==NULL)ERROR("xRunPair():no X_fun provided in ga,exit");
  /*==initialize two children==*/
  xInitkid(p1, p2, c1, c2);
  /*==if random probability of these pair parents is meet x_rate==*/
  if(RANDFRAC()<=ga_center->x_rate)
  { /*==call ga registered crossover operation to crossover parents==*/
    ga_center->X_fun(ga_center,p1,p2,c1,c2);/*gen xp, do x, chromRepair*/
  }
  else/*random rate is not meet,simply duplicate*/
  { chromCopy(p1,c1); chromCopy(p2,c2);
    c1->parent1=p1->index; c1->parent2=p2->index;
    c2->parent1=p2->index; c2->parent2=p1->index;
  }/*c1 has same gene matrix & same fitness as p1, no need call EV_fun()*/
}
/*-----------------------------------------------------------------------------------------------------------------
| xRunPool()---used by traditional ga, called by gaByTraditional(). After selecting new pool according to
random or biased select policy, we select mating pool from new pool according to x_rate, all new pool
chroms have equal right to participate to compete for being parents, that is, for each chrom we exam his
random probability. If it less than x_rate, it will be put into mating pool, wait for random mating late.
After mating pool is selected (it must be even number), we do random mating and calling ga registered
crossover operator to do crossover by selecting random crossover point positions (the number of x
positions in each crossover is determined by the ga registered crossover operator) to produce an array new
children who are reside in passed-in array child. It is possible mating pool empty, so no new child, but I
choose to goto back to re-select mating pool util it is not empty. this can be changed later for different
trials
-----------------------------------------------------------------------------------------------------------------*/
void xRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int* n)
{ int i=0,j=0,k=0,l=0; /*-- *n=number of total new children in child array*/
  int* mate;             /*mating pool, parents registered by storing index */
  CHROM_PTR p1, p2;   /*working variables for current parents*/
  int idx1, idx2;          /*working variables for current parent index*/
  /*==check the validation==*/
  if(!gacValid(ga_center))ERROR("xRunPool(): gacValid check fails,exit");
  if(!poolValid(pool))ERROR("xRunPool(): poolValid() check fails, exit");
  if(child==NULL)ERROR("xRunPool(): null child array address, exit");
  if(n==NULL)ERROR("xRunPool(): null new child counter aaddress, exit");
  if(ga_center->X_fun==NULL)ERROR("xRunPool:no X_fun provid in ga,exit");
  /*==check the pool size==*/
  if(ga_center->pool_max_size!=pool->max_size)ERROR("xRPool:p max_sz,exit");
  if(pool->max_size<=0)ERROR("xRunPool(): pool max size <=0, exit");
  if(pool->size<=0)ERROR("xRunPool(): pool actual size <=0, exit");
  if(pool->size>pool->max_size)ERROR("xRunPool():pl size>max_size,exit");
  /*==initialize variables==*/
  /*--zero new child array counter--*/
  *n=0;
  /*--dynamic alloc mating pool array & initializing--*/
  if((mate=(int*)calloc(pool->size,sizeof(int)))==NULL)
     ERROR("xRunPool():alloc mating pool array fail,exit");
  for(i=0;i<=pool->size-1;i++)mate[i]=-1;/*mating pool doesn't have any parent index at beginning*/
  p1=NULL;p2=NULL;
  /*==select mating pool according to x_rate==*/
re:for(i=0;i<=pool->size-1;i++)
  { if(RANDFRAC()<=ga_center->x_rate)/*random rate is meet x_rate*/
    { if(!chromValid(pool->chrom[i]))
         ERROR("xRunPool():chrom selected as parent is invalid,exit");
      mate[j++]=i;
    }
```

```
 }
 if(j==0) goto re;/*I choose mating pool can not be empty*/
 if(j%2!=0)/*we add one more parent in order mating pool have even parents*/
 { p1=seRunPool(ga_center,pool);/*randomly add a parent to mating pool*/
   if(!chromValid(p1))ERROR("xRunPool():chrom added as parent is invalid,exit");
   mate[j++]=p1->index;
 }/*now mating pool is selected, have j parents, j even & nozero*/
 /*==random mating and crossover==*/
 while(j!=0)
 {  if(j>2)/*actually j=4,or 6, or 8,... by j even*/
    {   /*--randomly pickup two mating pool indices--*/
        k=RANDDOM(0,j-1); l=RANDDOM(0,j-1);
        /*--make sure these two indices are refer to different cell--*/
        i=0;
        while(l==k)
        { l=RANDDOM(0,j-1);i++;
          if(i==1000)ERROR("xRPL:fail to pikup differ mate idx 1000 try,exit");
        }
        /*--achieve two parents new_pool indices from these two cell--*/
        idx1=mate[k];idx2=mate[l];
        /*--reset these two cell contents as refer to empty cell*/
        mate[k]=-1;mate[l]=-1;
        /*--reduce number of mating members by 2--*/
        j-=2;
        /*--make sure all remaining j members are on left side of mate[]*/
        /*-shifting all j members in mating pool to the left of array mate[]*/
        /*-implemented as bubble any hole to the right end of array mate[]*/
        /*mate[] has j cells not -1, so no infinite loop*/
/*for(i=0;i<=j-1;i++){while(mate[i]==-1){k=i;while(k<j+1){SWAP(&mate[k],&mate[k+1]); k++;}}}*/
       /*another bubble hole routine (keep order) for reference*/
        for(i=1;i<=j+1;i++)
        {  if(mate[i]!=-1)
           {  k=i;
              while(k>0 && mate[k-1]==-1){ SWAP(&mate[k],&mate[k-1]); k--;}
           }
        }
        /*--achieve parents pointer from new_pool*/
        p1=pool->chrom[idx1]; p2=pool->chrom[idx2];
        /*--check parent index agreement--*/
        if(idx1!=p1->index || idx2!=p2->index)
           ERROR("xRunPool():2 parents index is not agree w/ pool index,exit");
        /*==initialize two children==*/
        xInitkid(p1, p2, child[*n],child[(*n)+1]);
        /*--call ga registered crossover operator to crossover p1,p2--*/
        ga_center->X_fun(ga_center,p1,p2,child[*n],child[(*n)+1]);
        /*increase the counter of child array by 2 to point to next avail*/
        *n+=2;
    }
    else if(j==2)/*j=2 since j!=0 && j<=2 && j even */
    {   /*--mating pool only remain two members--*/
       /*--achieve two parents new_pool indices from these two cell--*/
       idx1=mate[0];idx2=mate[1];
       /*--reset these two cell contents as refer to empty cell*/
       mate[0]=-1;mate[1]=-1;
```

```
        /*--reduce number of mating members by 2--*/
        j-=2;
        /*--achieve parents pointer from new_pool*/
        p1=pool->chrom[idx1]; p2=pool->chrom[idx2];
        /*--check parent index agreement--*/
        if(idx1!=p1->index || idx2!=p2->index)
        ERROR("xRunPool():2 parents index is not agree w/ pool index,exit");
        /*==initialize two children==*/
        xInitkid(p1, p2, child[*n],child[(*n)+1]);
        /*--call ga registered crossover operator to crossover p1,p2--*/
        ga_center->X_fun(ga_center,p1,p2,child[*n],child[(*n)+1]);
        /*increase the counter of child array by 2 to point to next avail*/
        *n+=2;
    }
    else ERROR("xRunPool():err on mating pool counter j when mating,exit");
  }/*now all mating pool members crossovered, all new children are in child array, the number of them is
counter n*/
  free(mate); mate=NULL;
}
/*=====================================================================
| SAGA crossover user interfaces
=====================================================================*/
/*---------------------------------------------------------------------
| sacRunPair()--SAGA (simulated annealing genetic algorithm) crossover user interface to call ga
registered crossover operator to crossover two passed in parents to born two children if randomly
generated probability is met x_rate. two kids are put in global variables child1, and child2 pass out. used
by gnerational and Steady_state SAGA,called by ga_SE_X_MU_RE_2CHILD(),ga_SE_X_RE_2CHILD()
integrate SAC portion: accepting function after xover.
---------------------------------------------------------------------*/
void sacRunPair(GA_CENTER_PTR ga_center,
                CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ float acceptkid=0.0;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("sacRunPair():gacValid() check fails,exit");
  if(!chromValid(p1))ERROR("sacRunPair():chromValid(p1) check fails,exit");
  if(!chromValid(p2))ERROR("sacRunPair():chromValid(p2) check fails,exit");
  if(!chromValid(c1))ERROR("sacRunPair():chromValid(c1) check fails,exit");
  if(!chromValid(c2))ERROR("sacRunPair():chromValid(c2) check fails,exit");
  if(ga_center->X_fun==NULL)ERROR("sacRunPair():no X_fun provided in ga.exit");
  /*==initialize two children==*/
  xInitkid(p1, p2, c1, c2);
  /*==if random probability of these pair parents is meet x_rate==*/
  if(RANDFRAC()<=ga_center->x_rate)
  { /*==call ga registered crossover operation to crossover parents==*/
    ga_center->X_fun(ga_center,p1,p2,c1,c2);/*gen xp, do x, chromRepair*/
     /*==integrate SAC portion: accepting===*/
     if(chromComp(ga_center,p1,p2)<=0)/*p1 better than p2*/
     {  /*--compare child c1 with p1 for acceptance--*/
        acceptkid=accept(ga_center,p1,c1);
       if(RANDFRAC()<=acceptkid);/*accept c1*/
       else chromCopy(p1,c1);/*accept p1*/
       /*--compare child c2 with p1 for acceptance--*/
       acceptkid=accept(ga_center,p1,c2);
       if(RANDFRAC()<=acceptkid);/*accept c2*/
```

```
        else chromCopy(p1,c2);/*accept p1*/
    }
    else/*p2 is better than p1*/
    {  /*--compare child c1 with p2 for acceptance--*/
        acceptkid=accept(ga_center,p2,c1);
        if(RANDFRAC()<=acceptkid);/*accept c1*/
        else chromCopy(p2,c1);/*accept p2*/
        /*--compare child c2 with p2 for acceptance--*/
        acceptkid=accept(ga_center,p2,c2);
        if(RANDFRAC()<=acceptkid);/*accept c2*/
        else chromCopy(p2,c2);/*accept p2*/
    }
    /*---resolve the parent index---*/
    c1->parent1=p1->index; c1->parent2=p2->index;
    c2->parent1=p2->index; c2->parent2=p1->index;
  }
  else/*random rate is not meet,simply duplicate*/
  {  chromCopy(p1,c1); chromCopy(p2,c2);
     c1->parent1=p1->index; c1->parent2=p2->index;
     c2->parent1=p2->index; c2->parent2=p1->index;
  }/*c1 has same gene matrix & same fitness as p1, no need call EV_fun()*/
}
/*-------------------------------------------------------------------------------------------------------------
| sacRunPool()-Simulated annealing genetic algorithm crossover interface, used by traditional SAGA,
called by gaTraditional(). After selecting new pool according to random or biased selection policy, we
select mating pool from new pool according to x_rate, all new pool chroms have equal right to participate
to compete for being parents, that is, for each chrom we exam his random probability. If it less than
x_rate, it will be put into mating pool, wait for random mating late. After mating pool is selected (it must
be even number), we do random mating and calling ga registered crossover operator to do crossover by
selecting random crossover point positions( the number of x positions in each crossover is determined by
the ga registered crossover operator) to produce an array new children who are resided in passed-in array
"child". It is possible that mating pool is empty, so no new child, but I choose to "goto" back to re-select
mating pool until it is not empty. This can be changed later for different trials integrate SAC portion:
accepting after each pair crossover
-------------------------------------------------------------------------------------------------------------*/
void sacRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int* n)
{ int i=0,j=0,k=0,l=0; /*-- *n=number of total new children in child array*/
  int* mate;              /*mating pool, parents registered by storing index */
  CHROM_PTR p1, p2;   /*working variables for current parents*/
  int idx1, idx2;           /*working variables for current parent index*/
  float acceptkid=0.0;     /*the accept probability of child compare to parent*/
  /*==check the validation==*/
  if(!gacValid(ga_center))ERROR("sacRunPool(): gacValid check fails,exit");
  if(!poolValid(pool))ERROR("sacRunPool(): poolValid() check fails, exit");
  if(child==NULL)ERROR("sacRunPool(): null child array address, exit");
  if(n==NULL)ERROR("sacRunPool(): null new child counter aaddress, exit");
  if(ga_center->X_fun==NULL)ERROR("xRunPool:no X_fun provid in ga,exit");
  /*==check the pool size==*/
  if(ga_center->pool_max_size!=pool->max_size)ERROR("sacRPool:p max_sz,exit");
  if(pool->max_size<=0)ERROR("sacRunPool(): pool max size <=0, exit");
  if(pool->size<=0)ERROR("sacRunPool(): pool actual size <=0, exit");
  if(pool->size>pool->max_size)ERROR("sacRunPool():pl size>max_size,exit");
  /*==initialize variables==*/
  /*--zero new child array counter--*/
```

```
 *n=0;
 /*--dynamic alloc mating pool array & initializing--*/
 if((mate=(int*)calloc(pool->size,sizeof(int)))==NULL)
    ERROR("sacRunPool():alloc mating pool array fail,exit");
 for(i=0;i<=pool->size-1;i++)mate[i]=-1;/*mating pool doesn't have any parent index at beginning*/
 p1=NULL;p2=NULL;
 /*==select mating pool according to x_rate==*/
re:for(i=0;i<=pool->size-1;i++)
 {  if(RANDFRAC()<=ga_center->x_rate)/*random rate is meet x_rate*/
    {  if(!chromValid(pool->chrom[i]))
          ERROR("sacRunPool():chrom selected as parent is invalid,exit");
       mate[j++]=i;
    }
 }
 if(j==0) goto re;/*I choose mating pool can not be empty*/
 if(j%2!=0)/*we add one more parent in order mating pool have even parents*/
 {  p1=seRunPool(ga_center,pool);/*randomly add a parent to mating pool*/
    if(!chromValid(p1))ERROR("sacRunPool():chrom added as parent is invalid,exit");
    mate[j++]=p1->index;
 }/*now mating pool is selected, have j parents, j even & nozero*/
 /*==random mating and crossover==*/
 while(j!=0)
 {  if(j>2)/*actually j=4,or 6, or 8,... by j even*/
    {  /*--randomly pickup two mating pool indices--*/
       k=RANDDOM(0,j-1);l=RANDDOM(0,j-1);
       /*--make sure these two indices are refer to different cell--*/
       i=0;
       while(l==k)
       {  l=RANDDOM(0,j-1);i++;
          if(i==1000)ERROR("sacRPL:fail to pikup differ mate idx 1000 try,exit");
       }
       /*--achieve two parents new_pool indices from these two cell--*/
       idx1=mate[k];idx2=mate[l];
       /*--reset these two cell contents as refer to empty cell*/
       mate[k]=-1;mate[l]=-1;
       /*--reduce number of mating members by 2--*/
       j-=2;
       /*--make sure all remaining j members are on left side of mate[]*/
       /*-shifting all j members in mating pool to the left of array mate[]*/
       /*-implemented as bubble any hole to the right end of array mate[]*/
       /*mate[] has j cells not -1, so no infinite loop*/
/*for(i=0;i<=j-1;i++){while(mate[i]==-1){k=i;while(k<j+1){SWAP(&mate[k],&mate[k+1]);k++;}}}*/
       /*another bubble hole routine (keep order) for reference*/
       for(i=1;i<=j+1;i++)
       {if(mate[i]!=-1){k=i;while(k>0 && mate[k-1]==-1){SWAP(&mate[k],&mate[k-1]); k--;}}}
       /*--achieve parents pointer from new_pool*/
       p1=pool->chrom[idx1]; p2=pool->chrom[idx2];
       /*--check parent index agreement--*/
       if(idx1!=p1->index || idx2!=p2->index)
          ERROR("sacRunPool():2 parents index is not agree w/ pool index,exit");
       /*==initialize two children==*/
       xInitkid(p1, p2, child[*n],child[(*n)+1]);
       /*--call ga registered crossover operator to crossover p1,p2--*/
       ga_center->X_fun(ga_center,p1,p2,child[*n],child[(*n)+1]);
    /*==integrate SAC portion: accepting===*/
    if(chromComp(ga_center,p1,p2)<=0)/*p1 better than p2*/
    {   /*--compare child[*n] with p1 for acceptance--*/
        acceptkid=accept(ga_center,p1,child[*n]);
        if(RANDFRAC()<=acceptkid);/*accept child[*n]*/
        else chromCopy(p1,child[*n]);/*accept p1*/
        /*--compare child[(*n)+1] with p1 for acceptance--*/
        acceptkid=accept(ga_center,p1,child[(*n)+1]);
        if(RANDFRAC()<=acceptkid);/*accept child[(*n)+1]*/
        else chromCopy(p1,child[(*n)+1]);/*accept p1*/
    }
    else/*p2 is better than p1*/
    {   /*--compare child[*n] with p2 for acceptance--*/
        acceptkid=accept(ga_center,p2,child[*n]);
        if(RANDFRAC()<=acceptkid);/*accept child[*n]*/
        else chromCopy(p2,child[*n]);/*accept p2*/
        /*--compare child[(*n)+1] with p2 for acceptance--*/
        acceptkid=accept(ga_center,p2,child[(*n)+1]);
        if(RANDFRAC()<=acceptkid);/*accept child[(*n)+1]*/
        else chromCopy(p2,child[(*n)+1]);/*accept p2*/
    }
    /*---resolve the parent index---*/
    child[*n]->parent1=p1->index; child[*n]->parent2=p2->index;
    child[(*n)+1]->parent1=p2->index; child[(*n)+1]->parent2=p1->index;
    /*increase the counter of child array by 2 to point to next avail*/
    *n+=2;
}
else if(j==2)/*j=2 since j!=0 && j<=2 && j even */
{   /*--mating pool only remain two members--*/
    /*--achieve two parents new_pool indices from these two cell--*/
    idx1=mate[0];idx2=mate[1];
    /*--reset these two cell contents as refer to empty cell*/
    mate[0]=-1;mate[1]=-1;
    /*--reduce number of mating members by 2--*/
    j-=2;
    /*--achieve parents pointer from new_pool*/
    p1=pool->chrom[idx1]; p2=pool->chrom[idx2];
    /*--check parent index agreement--*/
    if(idx1!=p1->index || idx2!=p2->index)
        ERROR("sacRunPool():2 parents index is not agree w/ pool index,exit");
    /*==initialize two children==*/
    xInitkid(p1, p2, child[*n],child[(*n)+1]);
    /*--call ga registered crossover operator to crossover p1,p2--*/
    ga_center->X_fun(ga_center,p1,p2,child[*n],child[(*n)+1]);
    /*==integrate SAC portion: accepting===*/
    if(chromComp(ga_center,p1,p2)<=0)/*p1 better than p2*/
    {   /*--compare child[*n] with p1 for acceptance--*/
        acceptkid=accept(ga_center,p1,child[*n]);
        if(RANDFRAC()<=acceptkid);/*accept child[*n]*/
        else chromCopy(p1,child[*n]);/*accept p1*/
        /*--compare child[(*n)+1] with p1 for acceptance--*/
        acceptkid=accept(ga_center,p1,child[(*n)+1]);
        if(RANDFRAC()<=acceptkid);/*accept child[(*n)+1]*/
        else chromCopy(p1,child[(*n)+1]);/*accept p1*/
        }
        else/*p2 is better than p1*/
        {   /*--compare child[*n] with p2 for acceptance--*/
            acceptkid=accept(ga_center,p2,child[*n]);
            if(RANDFRAC()<=acceptkid);/*accept child[*n]*/
            else chromCopy(p2,child[*n]);/*accept p2*/
            /*--compare child[(*n)+1] with p2 for acceptance--*/
            acceptkid=accept(ga_center,p2,child[(*n)+1]);
            if(RANDFRAC()<=acceptkid);/*accept child[(*n)+1]*/
            else chromCopy(p2,child[(*n)+1]);/*accept p2*/
        }
        /*---resolve the parent index---*/
        child[*n]->parent1=p1->index;        child[*n]->parent2=p2->index;
        child[(*n)+1]->parent1=p2->index;  child[(*n)+1]->parent2=p1->index;
        /*increase the counter of child array by 2 to point to next avail*/
        *n+=2;
    }
    else ERROR("sacRunPool():err on mating pool counter j when mating,exit");
  }/*now all mating pool members crossovered, all new children are in child. array, the number of them is
counter n*/
  free(mate); mate=NULL;
}




/*================= end of file: cross.c =============== */
```

file: mutation.c

```
/*=====================================================================
| Ga mutation operations:
|
|/*=================Mutation Operators =====================================
|  muByFlipBit()---------randomly choose a bit to do flip mutation for a chromos chosen by
|                        muRunChromLevel() based on mutation rate  or chrom passed in by
|                        muRunSingleChrom() and random rate met.
|  muByRandomBit()----randomly choose a bit to do random flip for a chromos chosen by
|                        muRunChromLevel() based on mutation rate or chrom passed in by
|                        muRunSingleChrom() and random rate met.
|  muBySwap()-----------randomly choose two rows to do swap mutation for a chromos matrix
|                        chosen by  muRunChromLevel() based on mutation rate, or matrix
|                         passed in by muRunSingleChrom() and random rate met.
|/*==================GA mutation Interfaces with ga_center ======================
|  muSet()------------------set user defined mutation operator into MU_Table[] and register
|                         it into  ga_center.
|  muSelect()--------------set and select mutation operator into ga by name.
|  muName()--------------get current mutation operator name.
|  muRunChromLevel()-submanager to do whole mutation on a whole pool, for each chromosome
|                        of the whole pool random to see, based on the mutation rate, if it needs
|                        mutation or not, then call ga registered mutation operator.
|  muRunBitLevel()------submanager to do whole mutation on a whole pool, for each bit of the
|                         whole pool random to see if it need mutation or not, based on mutation
|                        rate, then flip it. Not use ga registered mutation operator.
|  muRunPool()-----------manager to do whole mutation on a whole pool, call submanagers
|                         either muRunChromLevel() or muRunBitLevel() according to mu_flag:
|                         "MU_CHROM" or "MU_BIT"; note:outside can only call muRunPool()
|                         to perform these two kind mutations.
|  muRunSingleChrom()--manager to do one mutation on a single chromosome if randomly
|                           generated  probability of that chrom < mu_rate. to call muByFlipBit(),
|                          muByRandomBit() or muBySwap()to perform mutation for a single
|                          chromosome that was already selected and crossovered. It is called by
|                          ga_SE_X_MU_RE_2CHILD(), the single basic operation of one GA iter
|/*==================SAGA mutation Interface with ga_center ======================
|  samRunChromLevel()-Simulated annealing genetic algorithm mutation submanager to do
|                          whole mutation on a whole pool, for each chromosome of the whole
|                          pool random to see, based on the mutation rate, if it needs mutation,
|                          then call ga registered mutation operator, after the execution of the
|                          mutation operator, we insert the integrated SAM accepting function
|                          portion to decide if the children are accepted.
|  samRunBitLevel()------Simulated annealing genetic algorithm mutation submanager to do
|                          whole mutation on a whole pool, for each bit of the whole pool random
|                          to see if it need mutation or not, based on mutation rate, then flip it. Not
|                          use ga registered mutation operator. after the execution of the mutation
|                          operation, we insert the integrated SAM accepting function portion to
|                          decide if the children are accepted.
|  samRunPool()-----------Simulated annealing genetic algorithm mutation manager to do whole
|                          mutation on a whole pool, call submanagers either samRunChromLevel()
|                          or samRunBitLevel() according to mu_flag: "MU_CHROM" or "MU_BIT";
|                          note:outside can only call samRunPool() to perform these two kind mutations.
|  samRunSingleChrom()-Simulated annealing genetic algorithm mutation manager to do one
|                          mutation on a single chromosome if random probability of that chrom <
|                          mu_rate. to call muByFlipBit(), muByRandomBit() or muBySwap()
|                       to perform mutation for a single chromosome that was already selected and
|                       crossovered. It is called by ga_SE_X_MU_RE_2CHILD(), the single basic
|                       operation of one SAGA iteration. after the execution of the mutation
|                       operator, we insert the integrated SAM accepting function portion to
|                       decide if the children are accepted.
=====================================================================*/
#include "gah.h"
/*====================================================================
| Data Structures
=====================================================================*/
int muByFlipBit(CHROM_PTR chrom);
int muByRandomBit(CHROM_PTR chrom);
int muBySwap(CHROM_PTR chrom);
void muSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr);
void muSelect(GA_CENTER_PTR ga_center,char* fn_name);
char* muName(GA_CENTER_PTR ga_center);
void muRunChromLevel(GA_CENTER_PTR ga_center, POOL_PTR pool);
void muRunBitLevel(GA_CENTER_PTR ga_center, POOL_PTR pool);
void muRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);
void muRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom);
void samRunChromLevel(GA_CENTER_PTR ga_center, POOL_PTR pool);
void samRunBitLevel(GA_CENTER_PTR ga_center, POOL_PTR pool);
void samRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);
void samRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom);

extern int chromValid(CHROM_PTR );
extern CHROM_PTR chromAlloc(int length);
extern void chromFree(CHROM_PTR chrom);
extern void chromCopy(CHROM_PTR src, CHROM_PTR dst);
extern void chromRepair(CHROM_PTR );
extern int poolValid(POOL_PTR pool);
extern int gacValid(GA_CENTER_PTR ga_center);
extern float accept(GA_CENTER_PTR ga_center,CHROM_PTR p,CHROM_PTR c);

extern void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                        char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
extern void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                        char* fn_name,FN_PTR* rtn_ptr);
extern char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);

FN_TABLE MU_Table[]=
{       {NULL,                  NULL},
        {"simple_flip",         muByFlipBit},
        {"simple_random", muByRandomBit},
        {"swap",                muBySwap},
        {NULL,                  NULL},
};
/*note: FN_TABLE[] must be NULL terminated for function search to be stop*/

/*====================================================================
| Mutation Operators:
=====================================================================*/
/*--------------------------------------------------------------------
```

```
| muByFlipBit()--------randomly choose a bit to do flip mutation for a chromosome chosen by
muRunChromLevel() based on mutation rate, or chrom passed in by muRunSingleChrom() and random
rate meet.
-----------------------------------------------------------------------------------------------------------------*/
int muByFlipBit(CHROM_PTR chrom)/*must be FN_PTR type due to write into ga*/
{ int index=-1;
 int hi,i,j;
 /*--- check the validation of chromosome ---*/
 if(!chromValid(chrom))ERROR("muByFlipBit():chromValid() check fails,exit");
 /*--- randomly choose a bit to do flip mutation ---*/
 hi=(chrom->dim)*(chrom->dim)-1;
 index=RANDDOM(0,hi);
 /*--- spot selected bit on which row and which column--*/
 i=0;j=0;
 while(index>=chrom->dim){ index-=chrom->dim; i++;}
 j=index; /*--this bit resides on ith row and jth column--*/
 if(chrom->gene[i]==NULL)ERROR("muByFlipBit():choosed bit on null row,exit");
 /*--- flip selected bit---*/
 chrom->gene[i][j]=chrom->gene[i][j] ? 0:1;
 /*call chromRepair() to make sure this chromsome is a feasible solution*/
 chromRepair(chrom);/*set chrom->eva=0 by chromRepair()*/
 return OK;
}
/*-----------------------------------------------------------------------------------------------------------------
| muByRandomBit()-----randomly choose a bit to do random flip for a chromosome chosen by
muRunChromLevel() based on mutation rate. or chromosome passed in by muRunSingleChrom() and
random rate met.
-----------------------------------------------------------------------------------------------------------------*/
int muByRandomBit(CHROM_PTR chrom)/*must be FN_PTR type due to write into ga*/
{ int index=-1;
 int hi,i,j;
 /*--- check the validation of chromosome ---*/
 if(!chromValid(chrom))ERROR("muByRandomBit():chromValid check fails,exit");
 /*--- randomly choose a bit to do random flip mutation ---*/
 hi=(chrom->dim)*(chrom->dim)-1;
 index=RANDDOM(0,hi);
 /*--- spot selected bit on which row and which column--*/
 i=0;j=0;
 while(index>=chrom->dim){ index-=chrom->dim; i++; }
 j=index; /*--this bit resides on ith row and jth column--*/
 if(chrom->gene[i]==NULL)ERROR("muRandBit():choosed bit on null row,exit");
 /*--- random flip selected bit---*/
 chrom->gene[i][j]=RANDBIT();
 /*call chromRepair() to make sure this chromsome is a feasible solution*/
 chromRepair(chrom);/*set chrom->eva=0 by chromRepair()*/
        return OK;
}
/*-----------------------------------------------------------------------------------------------------------------
| muBySwap()-------randomly choose two rows to do swap mutation for a chromosome matrix chosen by
muRunChromLevel() based on mutation rate. or chromosome matrix passed in by muRunSingleChrom()
and random rate met.
-----------------------------------------------------------------------------------------------------------------*/
int muBySwap(CHROM_PTR chrom)   /*must be FN_PTR type due to write into ga*/
{ int i, idx1=-1, idx2=-1;
```

```
 GENE_PTR temp;
 /*--- check the validation of chromosome ---*/
 if(!chromValid(chrom))ERROR("muBySwap():chromValid check fails,exit");
 /*--- randomly choose two rows to do swap mutation( can be same) ---*/
 idx1=RANDDOM(0,chrom->dim-1);  idx2=RANDDOM(0,chrom->dim-1);
 /*--make sure they are different--*/
 i=0;
 while(idx2==idx1)
 {   idx2=RANDDOM(0,chrom->dim-1);i++;
     if(i==1000)ERROR("muBySwap():fail to gen. valid idx2 in 1000 try,exit");
 }
 /*there shouldn't be any null row in chrom matrix,so if null, must exit*/
 if(chrom->gene[idx1]==NULL || chrom->gene[idx2]==NULL)
    ERROR("muBySwap():choose at least one null row for swap, exit");
 /*--- swap selected rows --*/
 temp=chrom->gene[idx1];
 chrom->gene[idx1]=chrom->gene[idx2];
 chrom->gene[idx2]=temp;
 /*call chromRepair() to make sure this chromsome is a feasible solution*/
 chromRepair(chrom);/*set chrom->eva=0 by chromRepair()*/
 return OK;
}
/*=================================================================
| GA mutation Interface with ga_center:
=================================================================*/
/*-----------------------------------------------------------------------------------------------------------
| muSet()-----set user defined mutation operator into MU_Table[] and register it into  ga_center.
-----------------------------------------------------------------------------------------------------------*/
void muSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("muSet(): gacValid() check fails, exit");
 if(strlen(fn_name)==0) ERROR("muSet(): no user fun name, exit");
 if(fn_ptr==NULL) ERROR("muSet(): Null user fun ptr, exit");
 /*--- call setFun() to operate on the MU_Table[]--*/
 setFun(ga_center,MU_Table,fn_name,fn_ptr,&(ga_center->MU_fun));
}
/*-----------------------------------------------------------------------------------------------------------
| muSelect()-----set and select mutation operator into ga by name
-----------------------------------------------------------------------------------------------------------*/
void muSelect(GA_CENTER_PTR ga_center,char* fn_name)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("muSelect(): gacValid() check fails, exit");
 if(strlen(fn_name)==0) ERROR("muSelect(): no fun name provided, exit");
 /*--- call selectFun() to operate on the MU_Table[]--*/
 selectFun(ga_center,MU_Table,fn_name,&(ga_center->MU_fun));
}
/*-----------------------------------------------------------------------------------------------------------
| muName()    -get current mutation operator name
-----------------------------------------------------------------------------------------------------------*/
char* muName(GA_CENTER_PTR ga_center)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("muName(): gacValid() check fails, exit");
 /*--- call getFunName() to operate on MU_Table[]--*/
 return getFunName(ga_center,MU_Table,ga_center->MU_fun);
```

```
}
/*------------------------------------------------------------------------------------------------------------------
| muRunChromLevel()-submanager to do whole mutation on a whole pool, for each chromosome of the
whole pool random to see if it needs mutation or not, based on the mutation rate, then call ga registered
mutation operator
------------------------------------------------------------------------------------------------------------------*/
void muRunChromLevel(GA_CENTER_PTR ga_center. POOL_PTR pool)
{ int i=0;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("muRunChromLevel:gacValid check fails,exit");
 if(!poolValid(pool))ERROR("muRunChromLevel():poolValid check fails,exit");
 if(ga_center->MU_fun==NULL)ERROR("muRunCh:no MU_fun provided in ga,exit");
 /*for each chromsome of pool random to see if need mutation --*/
 for(i=0;i<pool->size;i++)
 {   if(RANDFRAC()<=ga_center->mu_rate)
     {   ga_center->MU_fun(pool->chrom[i]);
         ga_center->num_mut++;  ga_center->tot_mut++;
     }
 }
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*------------------------------------------------------------------------------------------------------------------
| muRunBitLevel()---submanager to do whole mutation on a whole pool, for each bit of the whole pool
random to see if it need mutation or not, based on mutation rate, then flip it. Not use ga registered
mutation operator.
------------------------------------------------------------------------------------------------------------------*/
void muRunBitLevel(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ int i=0,j=0,k=0,l=0, index=0;
 int chromdim=0; /*dim of chromosome matrix*/
 int numbit=0;     /*the number of bits of a chromosome matrix*/
 int totbit=0;     /*the total number of bits of pool*/
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("muRunBITLevel():gacValid check fails,exit");
 if(!poolValid(pool))ERROR("muRunBITLevel():poolValid check fails,exit");
 /*--initializing--*/
 chromdim=ga_center->chrom_dim;
 numbit=(ga_center->chrom_dim)*(ga_center->chrom_dim);
 totbit=numbit*pool->size;
 /*for each bit of pool random to see if need mutation --*/
 for(i=0;i<totbit;i++)
 {   if(RANDFRAC()<=ga_center->mu_rate)
     {   index=i;j=0;k=0;l=0;
        while(index>=numbit){ index-=numbit; j++; }/*--this bit is in the j^th chrom matrix--*/
        if(!chromValid(pool->chrom[j]))
           ERROR("muRunBITLevel():mutation select a bit on null chrom, exit");
        while(index>=chromdim){ index-=chromdim; k++;}
        l=index; /*--this bit resides on k^th row and L^th column--*/
        if(pool->chrom[j]->gene[k]==NULL)
           ERROR("muRunBitLevel():choosed bit on null row, exit");
        /*--- flip selected bit---*/
        pool->chrom[j]->gene[k][l]=pool->chrom[j]->gene[k][l] ? 0:1;
        /*call chromRepair() to make sure this chromsome  feasible solution*/
/*chromRepair(pool->chrom[j]);*//*for each mutation bit call it may slow down system we can improve
by only do final check before exit  function*/
        /*--update states in ga_center--*/
        ga_center->num_mut++; ga_center->tot_mut++;
     }
  }
 /*call chromRepair() to make sure every chromsome is feasible solution*/
 for(j=0;j<=pool->size-1;j++)chromRepair(pool->chrom[j]);/*set chrom->eva=0 by chromRepair()*/
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-----------------------------------------------------------------------------------------------------------------
| muRunPool()---manager to do whole mutation on a whole pool, call submanagers either
muRunChromLevel() or muRunBitLevel() according to mu_flag: "MU_CHROM" or "MU_BIT"; note:
outside can only call muRunPool() to perform two kind mutations, muRunPool() is user interface.
---------------------------------------------------------------------------*/
void muRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("muRunPool(): gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("muRunPool(): poolValid() check fails, exit");
 if(ga_center->MU_fun==NULL)ERROR("muRunPool:no MU_fun provid in ga,exit");
 /*-according to mu_flag call mutation runner at chromLevel or bitLevel-*/
 if(ga_center->mu_flag==MU_CHROM) muRunChromLevel(ga_center, pool);
 else if(ga_center->mu_flag==MU_BIT) muRunBitLevel(ga_center,  pool);
 else ERROR("muRunPool(): this mu_flag shouldn't call muRunPool, exit");
}
/*-----------------------------------------------------------------------------------------------------------------
| muRunSingleChrom()---manager to do one mutation on a single chromosome if randomly generated
probability of that chrom <= mu_rate. To call muByFlipBit(), muByRandomBit() or muBySwap() to
perform mutation for a single chromosome that was already selected and crossovered. It is called by
ga_SE_X_MU_RE(), the single basic operation of one ga iteration, and also called by gaTraditional()
when mu_flag =" MU_CHILD", mutation child immediately, in both ga.
-----------------------------------------------------------------------------------------------------------------*/
void muRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom)
{ /*==check the validation===*/
 if(!gacValid(ga_center))ERROR("muSingleChrom():gacValid check fails,exit");
 if(!chromValid(chrom))ERROR("muSingleChrom():chromValid check fails,exit");
 if(ga_center->MU_fun==NULL)ERROR("muRSCh():no MU_fun provided in ga,exit");
 /*==if random probability of this chrom is meet mu_rate==*/
 if(RANDFRAC()>ga_center->mu_rate)return;/*random rate is not meet,return*/
 /*==call ga registered mutation operation to mutate chrom==*/
 ga_center->MU_fun(chrom);
 /*==update states in ga_center==*/
 ga_center->num_mut++;  ga_center->tot_mut++;
}
/*===========================================================================
| SAGA mutation user interfaces
===========================================================================*/
/*-----------------------------------------------------------------------------------------------------------------
| samRunChromLevel()-------Simulated annealing genetic algorithm mutation submanager to do whole
mutation on a whole pool, for each chromosome of the whole pool random to see if it needs mutation or
```

not, based on the mutation rate, then call ga registered mutation operator, after the execution of the mutation operator, we insert the integrated SAM accepting function portion to decide if the children are accepted.

```
-----------------------------------------------------------------------------------------------------------*/
void samRunChromLevel(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ int i=0;
 float acceptkid=0.0;
 CHROM_PTR parent=NULL;
 /*==check the validation==*/
 if(!gacValid(ga_center))ERROR("samRunChromLevel:gacValid check fail,exit");
 if(!poolValid(pool))ERROR("samRunChromLevel():poolValid check fails,exit");
 if(ga_center->MU_fun==NULL)ERROR("samRunCh:no MU_fun provided in ga,exit");
 /*==dynamiclly alloc parent==*/
 parent=chromAlloc(ga_center->chrom_dim);
 /*==for each chromsome of pool random to see if need mutation ==*/
 for(i=0;i<pool->size;i++)
 {   if(RANDFRAC()<=ga_center->mu_rate)
     {   /*==make a copy of parent before mutation==*/
         chromCopy(pool->chrom[i],parent);
         /*==call ga registered mu op to do mutation==*/
         ga_center->MU_fun(pool->chrom[i]);/*mutate parent to child*/
         /*==integrate SAM portion: accepting===*/
         /*--compare child: pool->chrom[i] with parent for acceptance--*/
         acceptkid=accept(ga_center,parent,pool->chrom[i]);
         if(RANDFRAC()<=acceptkid);/*accept child: pool->chrom[i]*/
         else chromCopy(parent,pool->chrom[i]);/*accept parent by restoring it*/
         ga_center->num_mut++; ga_center->tot_mut++;
     }
 }/*no need for chromRepair() since either parent or child already feasible*/
 /*==free parent space==*/
 chromFree(parent);
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*----------------------------------------------------------------------------------------------------------
```

| samRunBitLevel()--Simulated annealing genetic algorithm mutation submanager to do whole mutation on a whole pool, for each bit of the whole pool random to see if it needs mutation or not, based on mutation rate, then flip it, not use registered mutation operator. after the execution of the mutation operation, we insert the integrated SAM accepting function portion to decide if the children are accepted.

```
-----------------------------------------------------------------------------------------------------------*/
void samRunBitLevel(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ int i=0,j=0,k=0,l=0, index=0;
 int chromdim=0;  /*dim of chromosome matrix*/
 int numbit=0;    /*the number of bits of a chromosome matrix*/
 int totbit=0;    /*the total number of bits of pool*/
 float acceptkid=0.0;
 CHROM_PTR parent=NULL;
 /*==check the validation==*/
 if(!gacValid(ga_center))ERROR("samRunBITLevel:gacValid check fails,exit");
 if(!poolValid(pool))ERROR("samRunBITLevel():poolValid check fails,exit");
 /*==dynamiclly alloc parent==*/
 parent=chromAlloc(ga_center->chrom_dim);
 /*==intializing==*/
 chromdim=ga_center->chrom_dim;
 numbit=(ga_center->chrom_dim)*(ga_center->chrom_dim);
 totbit=numbit*pool->size;
 /*==for each bit of pool random to see if need mutation==*/
 for(i=0;i<totbit;i++)
 {   if(RANDFRAC()<=ga_center->mu_rate)
     {   index=i;j=0;k=0;l=0;
         while(index>=numbit){ index-=numbit; j++;}/*--this bit is in the j^th chrom matrix--*/
         if(!chromValid(pool->chrom[j]))
             ERROR("samRunBITLevel():mutation select a bit on null chrom, exit");
         while(index>=chromdim){ index-=chromdim; k++;}
         l=index; /*--this bit resides on k^th row and l^th column--*/
         if(pool->chrom[j]->gene[k]==NULL)
             ERROR("samRunBitLevel():choosed bit on null row, exit");
         /*==make a copy of parent before mutation==*/
         chromCopy(pool->chrom[j],parent);
         /*==mutation by flip selected bit==*/
         pool->chrom[j]->gene[k][l]=pool->chrom[j]->gene[k][l] ? 0:1;
         /*--call chromRepair() to make sure this chromsome  feasible solution*/
         chromRepair(pool->chrom[j]);
         /*==integrate SAM portion: accepting===*/
         /*--compare child: pool->chrom[j] with parent for acceptance--*/
         acceptkid=accept(ga_center,parent,pool->chrom[j]);
         if(RANDFRAC()<=acceptkid);/*accept child: pool->chrom[j]*/
         else chromCopy(parent,pool->chrom[j]);/*accept parent by restoring it*/
         /*==update states in ga_center==*/
         ga_center->num_mut++; ga_center->tot_mut++;
     }
 }
 /*==free parent space==*/
 chromFree(parent);
 /*===set off pool->sorted flag===*//*TRUE=1, FALSE=0*/
 pool->sorted=FALSE;/*due to chromosome changes in pool*/
 /*===set off pool->updated flag===*/
 pool->updated=FALSE;/*due to chromosome changes in pool*/
}
/*-----------------------------------------------------------------------------------------------------------
| samRunPool()--Simulated annealing genetic algorithm mutation manager to do whole mutation on a
whole pool, call submanagers either samRunChromLevel() or samRunBitLevel() according to mu_flag:
"MU_CHROM" or "MU_BIT". Note: outside can only call samRunPool() to perform SAM mutation.
samRunPool() is user interface.
-----------------------------------------------------------------------------------------------------------*/
void samRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool)
{ /*==check the validation==*/
 if(!gacValid(ga_center))ERROR("samRunPool():gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("samRunPool(): poolValid() check fails, exit");
 if(ga_center->MU_fun==NULL)ERROR("samRunPool:no MU_fun provid in ga,exit");
 /*==according to mu_flag call mutation runner at chromLevel or bitLevel=*/
 if(ga_center->mu_flag==MU_CHROM) samRunChromLevel(ga_center, pool);
 else if(ga_center->mu_flag==MU_BIT) samRunBitLevel(ga_center,  pool);
 else ERROR("samRunPool(): this mu_flag shouldn't call samRunPool(), exit");
}
```

```
/*--------------------------------------------------------------------------------------------------------------
| samRunSingleChrom()--Simulated annealing genetic algorithm mutation manager to do one mutation on
a single chromosome if randomly generated probability of that chrom <= mu_rate. To call muByFlipBit(),
muByRandomBit() or muBySwap() to perform mutation for a single chromosome that was already
selected and crossovered. It is called by ga_SE_X_MU_RE(), the single basic operation of one ga
iteration, and also called by gaTraditional() when mu_flag = "MU_CHILD", mutation child immediately,
in three SAGA models. After the execution of the mutation operation, we insert the integrated SAM
accepting function portion to decide if the children are accepted.
--------------------------------------------------------------------------------------------------------------*/
void samRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom)
{ float acceptkid=0.0;
 CHROM_PTR parent=NULL;
 /*==check the validation===*/
 if(!gacValid(ga_center))ERROR("samSingleChrom:gacValid check fails,exit");
 if(!chromValid(chrom))ERROR("samSingleChrom:chromValid check fails,exit");
 if(ga_center->MU_fun==NULL)ERROR("samRSCh:no MU_fun provided in ga,exit");
 /*==if random probability of this chrom is meet mu_rate==*/
 if(RANDFRAC()>ga_center->mu_rate)return;/*random rate is not meet,return*/
 /*==dynamiclly alloc parent==*/
 parent=chromAlloc(ga_center->chrom_dim);
 /*==make a copy of parent before mutation==*/
 chromCopy(chrom,parent);
 /*==call ga registered mutation operation to mutate chrom==*/
 ga_center->MU_fun(chrom);
 /*==integrate SAM portion: accepting===*/
 /*--compare child: chrom with parent for acceptance--*/
 acceptkid=accept(ga_center,parent,chrom);
 if(RANDFRAC()<=acceptkid);/*accept child: chrom*/
 else chromCopy(parent,chrom);/*accept parent by restoring it*/
 /*==update states in ga_center==*/
 ga_center->num_mut++; ga_center->tot_mut++;
 /*==free parent space==*/
 chromFree(parent);
}
```

```
/*================== end of file: mutation.c =============*/
```

file: replace.c

```
/*=================================================================
| Ga replacement operations:
|
| --Replacement Operators--------------------------------------------------------------------------------
|  reByAppend()----------simply place two children in the tail of the pool, can only be used by
|                        generational ga to fill new pool since new pool is formed by filling in
|                        two chroms once, can not be used by traditional and steady_state ga.
|  reByParent()-----------simply replace two parents with two children in the pool, can be used
|                        by traditional and steady-state ga, can not be used by generational ga.
|  reByRank()-------------in a sorted pool replace the worest one (at tail) with one of two children
|                        once one time, then move it leftwise to suitable rank position. If weakest
|                        chromosome is better than child, no replacement when re_elitist flag on,
|                        otherwise replace the weakest anyway. Can be used by traditional and
|                        steady-state ga. Can not be used by generational ga.
|  reByFirstWeaker()----in a pool, replace the first hit weaker one with one of two children once
|                        one time, if every chromosome in pool is better than child, then no
|                        replacement. it can be used by traditional and steady_state ga, and can
|                         not be used by generational ga.
|  reByWeakest()---------in a pool, replace the weakest one with one of two children once one
|                        time, if pool is sorted it is the same as of reByRank() except ranking
|                        movement. If weakest chromosome is better than child, no replacement
|                        when re_elitist flag on, otherwise replace the weakest anyway. It can be
|                        used by traditional and steady-state ga, can not be used by generational ga.
|  reByRandom()---------replace two children into pool by random pickup victims
| --Utilities--------------------------------------------------------------------------------------------
|  rePickBest()-----------pick best 2 children from 2 parent and 2 children, it is re_elitist policy
|                        utility called by interfaces reRunPair() and reRunPool() to handle
|                        re-elitist before calling replace operator to do replacement.
|  reByRankHelper()-----insert a chromosome into sorted pool by rank. 1st: replace the
|                        weakest one(at tail). 2nd: rank it by shifting toward left. If the
|                        weakest chromosome is better than child, then no replacement when
|                        re_elitist flag on, otherwise replace the weakest anyway.
| --Replacement Interface with ga_center------------------------------------------------------------------
|  reSet()-------set user defined replacement operator into RE_Table[] and register it into  ga_center.
|  reSelect()----set and select replacement operator into ga by name.
|  reName()----get current replacement operator name.
|  reRunPair()-user interface to call ga registered replace operatorto replace two passed in  children
|             into the pool, called by ga_SE_X_MU_RE_2CHILD(), ga_SE_X_RE_2CHILD().
|  reRunPool()-user interface to replace all new children into new pool by passing twined new
|             children to ga registered replace operator to perform replacement until finished.
|             it is used only by traditional ga.
=================================================================*/
#include "gah.h"
/*=================================================================
| data structures
=================================================================*/
int reByAppend(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2);
int reByParent(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2);
int reByRank(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2);
int reByFirstWeaker(GA_CENTER_PTR ga_center, POOL_PTR pool,
                              CHROM_PTR c1, CHROM_PTR c2);
int reByWeakest(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2);
int reByRandom(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2);
```

```
FN_TABLE RE_Table[]=
{  {NULL,                       NULL},
   {"append",             reByAppend},
   {"replace_parent",        reByParent},
   {"replace_rank",            reByRank},
   {"replace_weaker", reByFirstWeaker},
   {"replace_weakest",     reByWeakest},
   {"replace_random",      reByRandom},
   {NULL,                       NULL},
};
/*note: FN_TABLE[] must be NULL terminated for function search to be stop*/
/*=====================================================================
| function prototype declarations
=====================================================================*/
void rePickBest(GA_CENTER_PTR ga_center,
                CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
void reByRankHelper(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR c);

void reSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr);
void reSelect(GA_CENTER_PTR ga_center,char* fn_name);
char* reName(GA_CENTER_PTR ga_center);
void reRunPair(GA_CENTER_PTR ga_center, POOL_PTR pool,
                CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
void reRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int n);

extern int chromValid(CHROM_PTR );
extern int chromComp(GA_CENTER_PTR, CHROM_PTR, CHROM_PTR);
extern void chromCopy(CHROM_PTR , CHROM_PTR);

extern int poolValid(POOL_PTR pool);
extern void poolInsert(POOL_PTR pool, CHROM_PTR chrom, int idx, int copy);
extern void poolAppend(POOL_PTR pool, CHROM_PTR chrom, int copy);
extern void poolSwap(POOL_PTR pool, int idx1, int idx2);
extern void poolSort(GA_CENTER_PTR ga_center,POOL_PTR pool);

extern int gacValid(GA_CENTER_PTR ga_center);
extern char* gaName(GA_CENTER_PTR ga_center);

extern void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                          char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
extern void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                                        char* fn_name,FN_PTR* rtn_ptr);
extern char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);

/*=====================================================================
| replacement operators
=====================================================================*/
/*---------------------------------------------------------------------
| reByAppend()----simply place two children in the tail of the pool,can only be used by generational ga to
fill new pool since new pool is formed by filling in two chroms once. It can not be used by traditional and
steady_state ga.
---------------------------------------------------------------------*/
```

```
int reByAppend(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2)
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reByAppend():gacValid() check fails,exit");
 if(!poolValid(pool))ERROR("reByAppend():poolValid() check fails,exit");
 if(!chromValid(c1))ERROR("reByAppend():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("reByAppend():chromValid(c2) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reByApp:p max_sz err");
 if(pool->max_size<=0)ERROR("reByAppend(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reByAppend(): pool actual size <0, exit");
 if(pool->size==pool->max_size)ERROR("reByAppend(): pool full, exit");
 if(pool->size>pool->max_size)ERROR("reByAppend():p size>max_size, exit");
 /*==check parent index==*/
 /*the purpose of check parent index is make sure c1, c2 are twins */
 if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
    ERROR("reByAppend: not agreed on parent index of twined children,exit");
 /*==make sure it is not called by traditioal and steady_state ga==*/
 if(!strcmp(gaName(ga_center),"traditional"))
    ERROR("reByAppend:reByAppend can not be used by traditional ga, exit");
 if(!strcmp(gaName(ga_center),"steady_state"))
    ERROR("reByAppend:eByAppend can not be used by steady_state ga,exit");
 /*==append two children into tail of pool by copying==*/
 poolAppend(pool,c1,TRUE); /*also set pool flags: sorted,updated FALSE*/
 poolAppend(pool,c2,TRUE);
 return OK;
}
/*-----------------------------------------------------------------------------------------------------------------
| reByParent()--simply replace two parents with two children in the pool, can be used by traditional and
steady_state ga, can not be used by generational ga.
-----------------------------------------------------------------------------------------------------------------*/
int reByParent(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2)
{ int idx1, idx2;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reByParent():gacValid() check fails,exit");
 if(!poolValid(pool))ERROR("reByParent():poolValid() check fails,exit");
 if(!chromValid(c1))ERROR("reByParent():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("reByParent():chromValid(c2) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reByPa:p max_sz,exit");
 if(pool->max_size<=0)ERROR("reByParent(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reByParent(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reByParent(): p size>max_size, exit");
 /*==check the parent index==*/
 /*the purpose of check parent index is make sure c1, c2 are twins */
 if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
    ERROR("reByPa: not agreed on parent index of twined children,exit");
 idx1=c1->parent1; idx2=c2->parent1;
 if(idx1<0||idx1>pool->size-1)ERROR("reByParent(): invalid p1 index, exit");
 if(idx2<0||idx2>pool->size-1)ERROR("reByParent(): invalid p2 index, exit");
 /*if(idx1==idx2)ERROR("reParent(): two parents are identical, exit");
 note:this may happen in generational and steady_state ga, due to they
 choose parents in a serial random selection trial, each time it select
 from the whole pool, so it is possible to choose 2 identical parents. in
 both case it will produce 2 children, in generational ga, we just append
 2 children into new pool, not replace any one in the pool; in steady_
 state ga, we use 2 children to replace 2 identicl parents, actually the
 identical parent is replaced by the 1st child, the 1st child is replaced
 by the 2nd child. only 2nd child can survive.*/
 /*==make sure parent index agree w/ child==*/
 if(idx1!=pool->chrom[idx1]->index || idx2!=pool->chrom[idx2]->index)
     ERROR("reByParent(): parent index not agree w/ pool index, exit");
 /*==make sure it is not called by generational ga==*/
 if(!strcmp(gaName(ga_center),"generational"))
     ERROR("reByParent:reByParent can not be used by generational ga,exit");
 /*==replace parents with two children into pool by copying==*/
 /*reset children pool index as parents pool index*/
 c1->index=idx1; c2->index=idx2;
 poolInsert(pool,c1,idx1,TRUE);/*1st child to replace 1st parent*/
 poolInsert(pool,c2,idx2,TRUE);/*2nd child to replace 2nd parent*/
 /*also set pool flags: sorted,updated FALSE by poolInsert()*/
 return OK;
}
/*--------------------------------------------------------------------------------------------------------------
| reByRank()----in a sorted pool replace the worest one (at tail) with one of two children once one time,
then move leftwise to a suitable rank position. If the weakest chromosome is better than child, no
replacement when elitist flag on, otherwise replace the weakest anyway. It can be used by traditional and
steady_state ga, can not be used by generational ga.
--------------------------------------------------------------------------------------------------------------*/
/*note: if we need check the pool->sorted, make sure it is sorted? yes! I have added a line poolSort() in
reByRankHelper() this line can be moved to reByRank() before call reByRkHlp().*/
int reByRank(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2)
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reByRank():gacValid() check fails,exit");
 if(!poolValid(pool))ERROR("reByRank():poolValid() check fails,exit");
 if(!chromValid(c1))ERROR("reByRank():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("reByRank():chromValid(c2) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reByRan:pl max_sz,exit");
 if(pool->max_size<=0)ERROR("reByRank(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reByRank(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reByRank(): pl size>max_size, exit");
 /*==check the parent index==*/
 /*the purpose of check parent index is make sure c1, c2 are twins */
 if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
     ERROR("reByRank: not agreed on parent index of twined children,exit");
 /*==make sure it is not called by generational ga==*/
 if(!strcmp(gaName(ga_center),"generational"))
     ERROR("reByRank():reByRank() can not be used by generational ga,exit");
 /*==rank 2 children into pool once one time by replacing the one on tail and ranking toward left*/
 reByRankHelper(ga_center,pool,c1);
 reByRankHelper(ga_center,pool,c1);
 return OK;
}
/*--------------------------------------------------------------------------------------------------------------
| reByFirstWeaker()---in a pool, replace the first hit weaker one with one of two children once one time, if
every chromosome in pool is better than child, then no replacement. It can be used by traditional and
steady_state ga, can not be used by generational ga.
--------------------------------------------------------------------------------------------------------------*/
```

```
int reByFirstWeaker(GA_CENTER_PTR ga_center, POOL_PTR pool,
                    CHROM_PTR c1, CHROM_PTR c2)
{ int i;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reByFirstW:gacValid check fails,exit");
 if(!poolValid(pool))ERROR("reByFirstWeak:poolValid() check fails,exit");
 if(!chromValid(c1))ERROR("reByFirstWeak:chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("reByFirstWeak:chromValid(c2) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reFi:pl max_sz,exit");
 if(pool->max_size<=0)ERROR("reByFirstWeaker(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reByFirstWeaker(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reByFirstWeak:pl size>max_size,exit");
 /*==check the parent index==*/
 /*the purpose of check parent index is make sure c1, c2 are twins */
 if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
    ERROR("reByWeaker: not agreed on parent index of twined children,exit");
 /*==make sure it is not called by generational ga==*/
 if(!strcmp(gaName(ga_center),"generational"))
    ERROR("reByFirstWeak:reFirstWeak can not be used by generational ga,exit");
 /*==insert 2 children into pool once one time by replacing the first meet weaker one in pool*/
 for(i=0;i<=pool->size-1;i++)
 {   if(chromComp(ga_center,c1,pool->chrom[i])<0)
     {   poolInsert(pool,c1,i,TRUE); break;
         /*1st child to replace 1st hit weaker, also set pool flags: sorted,updated FALSE*/
     }
 }/*it is possible that c1 is not inserted for c1 is weaker than any one*/
 for(i=0;i<=pool->size-1;i++)
 {   if(chromComp(ga_center,c2,pool->chrom[i])<0)
     {   poolInsert(pool,c2,i,TRUE); break;
         /*2nd child to replace 1st hit weaker, also set pool flags: sorted,updated FALSE*/
     }
 }/*it is possible that c2 is not inserted for c2 is weaker than any one*/
 return OK;
}
/*-----------------------------------------------------------------------------------------------------------
| reByWeakest()----in a pool, replace the weakest one with one of two children once one time, if pool is
sorted it is the same as of reByRank() except ranking movement. If the weakest chromosome is better than
child, no replacement when re_elitist flag on, otherwise replace the weakest anyway. It can be used by
traditional and steady_state ga, can not be used by generational ga.
-----------------------------------------------------------------------------------------------------------*/
int reByWeakest(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR c1, CHROM_PTR c2)
{ int i,idx; CHROM_PTR weakest;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reByWeakest: gacValid check fails,exit");
 if(!poolValid(pool))ERROR("reByWeakest():poolValid() check fails,exit");
 if(!chromValid(c1))ERROR("reByWeakest:chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("reByWeakest:chromValid(c2) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reWkst:pl max_sz,exit");
 if(pool->max_size<=0)ERROR("reByWeakest(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reByWeakest(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reByWeakest():pl size>max_size,exit");
 /*==check the parent index==*/
 if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
     ERROR("reByRandom: not agreed on parent index of twined children,exit");
 /*==make sure it is not called by generational ga==*/
 if(!strcmp(gaName(ga_center),"generational"))
     ERROR("reByRandom():reRandom can not be used by generational ga,exit");
 /*==insert 2 children into pool once one time by replacing the random pickup one in pool==*/
 /*--insert c1--*/
 idx=RANDDOM(0,pool->size-1);  poolInsert(pool,c1,idx,TRUE);
 /*--insert c2--*/
 idx=RANDDOM(0,pool->size-1);  poolInsert(pool,c2,idx,TRUE);
 return OK;
}
/*=====================================================================
| Utilities
=====================================================================*/
/*--------------------------------------------------------------------------------------------------------
|  rePickBest()--pick best 2 children from 2 parent and 2 children, it is re_elitist policy utility called by
interfaces reRunPair() and reRunPool() to handle re_elitist before calling replace operator to do
replacement.
--------------------------------------------------------------------------------------------------------*/
void rePickBest(GA_CENTER_PTR ga_center,
               CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1. CHROM_PTR c2)
{ /*===check the validation===*/
 if(!gacValid(ga_center))ERROR("rePickBest():gacValid() check fails,exit");
 if(!chromValid(p1))ERROR("rePickBest():chromValid(p1) check fails,exit");
 if(!chromValid(p2))ERROR("rePickBest():chromValid(p2) check fails,exit");
 if(!chromValid(c1))ERROR("rePickBest():chromValid(c1) check fails,exit");
 if(!chromValid(c2))ERROR("rePickBest():chromValid(c2) check fails,exit");
 /*===if re_elitist flag is off, return==*/
 if(!ga_center->re_elitist) return;
 /*===replace worst child with p1 if p1 better==*/
 if(chromComp(ga_center,c1,c2)>0)  /*c1 is worse than c2*/
 {   if(chromComp(ga_center,c1,p1)>0)/*c1 is also worse than p1*/
     {   chromCopy(p1,c1);          /*copy p1 to c1*/
     }
 }
 else                              /*c2 is not better than c1*/
 {   if(chromComp(ga_center,c2,p1)>0)/*c2 is also worse than p1*/
     {   chromCopy(p1,c2);          /*copy p1 to c2*/
     }
 }
 /*===replace worst child with p2 if p2 better==*/
 if(chromComp(ga_center,c1,c2)>0)  /*c1 is worse than c2*/
 {   if(chromComp(ga_center.c1,p2)>0)/*c1 is also worse than p2*/
     {   chromCopy(p2,c1);          /*copy p2 to c1*/
     }
 }
 else                     /*c2 is not better than c1*/
 {   if(chromComp(ga_center,c2,p2)>0)/*c2 is also worse than p2*/
     {   chromCopy(p2,c2);          /*copy p2 to c2*/
     }
 }
 /*==resolve indices==*/
 c1->parent1=p1->index;  c1->parent2=p2->index;
 c2->parent1=p2->index;  c2->parent2=p1->index;
}
/*--------------------------------------------------------------------------------------------------------------------
| reByRankHelper()---insert a chromosome into sorted pool by rank. 1st: replace the weakest one(at tail).
2nd: rank it by shifting toward left. If the weakest chromosome is better than child, then no replacement
when replace elitist flag on, otherwise replace the weakest anyway.
--------------------------------------------------------------------------------------------------------------------*/
/*note: if we need check the pool->sorted, make sure it is sorted? yes! I have added a line poolSort() in
reByRankHelper(). This line can be moved to reByRank() before call reByRkHlp().*/
void reByRankHelper(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR c)
{ int i;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("reBRH(): gacValid check fails,exit");
 if(!poolValid(pool))ERROR("reBRH():poolValid() check fails,exit");
 if(!chromValid(c))ERROR("reBRH:chromValid(c) check fails,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reBRH:pl max_sz,exit");
 if(pool->max_size<=0)ERROR("reBRH(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reBRH(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reBRH():pl size>max_size,exit");
 /*==sort pool==*/
 /*only using rank index, not rank_prob, so not exe poolRank*/
 if(!pool->sorted) poolSort(ga_center,pool);
 /*==replace the weakest at tail by child==*/
 /*if re_elitist on,we do not replace if child is weaker than the weakest in pool*/
 if(ga_center->re_elitist) if(chromComp(ga_center,c,pool->chrom[pool->size-1])>0) return;
 poolInsert(pool,c,pool->size-1,TRUE);/*also set sorted,updated FALSE*/
 /*==bubble new chrom to the ranked position==*/
 for(i=pool->size-1;i>0;i--)
 { if(chromComp(ga_center,pool->chrom[i-1],pool->chrom[i])<=0) break;
   poolSwap(pool,i-1,i);/*also set pool flags: sorted,updated FALSE*/
 }
 /*==update pool index, restore sorted=1 destroyed by pool insert swap*/
 /*poolIndex(pool);*//*no need*/
 return;
}
/*========================================================================
| Replacement Interface with ga_center:
========================================================================*/
/*--------------------------------------------------------------------------------------------------------------------
| reSet()------set user defined replacement operator into RE_Table[] and register it into  ga_center
--------------------------------------------------------------------------------------------------------------------*/
void reSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("reSet(): gacValid() check fails, exit");
 if(strlen(fn_name)==0) ERROR("reSet(): no user fun name, exit");
 if(fn_ptr==NULL) ERROR("reSet(): Null user fun ptr, exit");
 /*--- call setFun() to operate on the RE_Table[]--*/
 setFun(ga_center,RE_Table,fn_name,fn_ptr,&(ga_center->RE_fun));
}
/*--------------------------------------------------------------------------------------------------------------------
| reSelect()  -set and select replacement operator into ga by name
--------------------------------------------------------------------------------------------------------------------*/
void reSelect(GA_CENTER_PTR ga_center,char* fn_name)
```

```
{ /*--- error check ---*/
  if(!gacValid(ga_center)) ERROR("reSelect(): gacValid() check fails, exit");
  if(strlen(fn_name)==0) ERROR("reSelect(): no fun name provided, exit");
  /*--- call selectFun() to operate on the RE_Table[]--*/
  selectFun(ga_center,RE_Table,fn_name,&(ga_center->RE_fun));
}
/*---------------------------------------------------------------------------------------------------------------
| reName()    -get current replacement operator name
---------------------------------------------------------------------------------------------------------------*/
char* reName(GA_CENTER_PTR ga_center)
{ /*--- error check ---*/
  if(!gacValid(ga_center)) ERROR("reName(): gacValid() check fails, exit");
  /*--- call getFunName() to operate on RE_Table[]--*/
  return getFunName(ga_center,RE_Table,ga_center->RE_fun);
}
/*---------------------------------------------------------------------------------------------------------------
| reRunPair()---user interface to call ga registered replace operator to replace two passed in children into
the pool, called by ga_SE_X_MU_RE_2CHILD(), ga_SE_X_RE_2CHILD().
---------------------------------------------------------------------------------------------------------------*/
void reRunPair(GA_CENTER_PTR ga_center, POOL_PTR pool,
               CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2)
{ /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("reRunPair():gacValid() check fails,exit");
  if(!poolValid(pool))ERROR("reRunPair():poolValid() check fails,exit");
  if(!chromValid(p1))ERROR("reRunPair():chromValid(p1) check fails,exit");
  if(!chromValid(p2))ERROR("reRunPair():chromValid(p2) check fails,exit");
  if(!chromValid(c1))ERROR("reRunPair():chromValid(c1) check fails,exit");
  if(!chromValid(c2))ERROR("reRunPair():chromValid(c2) check fails,exit");
  if(ga_center->RE_fun==NULL)ERROR("reRunPair():no RE_fun provided in ga,exit");
  /*==check the pool size==*/
  if(ga_center->pool_max_size!=pool->max_size)ERROR("reRPair:pl max_sz,exit");
  if(pool->max_size<=0)ERROR("reRunpair(): pool max size <=0, exit");
  if(pool->size<0)ERROR("reRunPair(): pool actual size <0, exit");
  if(pool->size>pool->max_size)ERROR("reRunPair():pl size>max_size,exit");
  /*==check the parent index==*/
  /*the purpose of check parent index is make sure c1, c2 are twins */
  if(c1->parent1!=c2->parent2 || c1->parent2!=c2->parent1)
     ERROR("reRunPair:not agreed on parent index of twined children,exit");
  /*the purpose of check is make sure the link of child w/ parent*/
  if(c1->parent1!=p1->index || c2->parent1!=p2->index)
     ERROR("reRunPair:not match on parent index w/ twined children,exit");
  /*the purpose of check is make sure the link of parent idx w/ pool idx*/
 /* if(idx1!=pool->chrom[idx1]->index || idx2!=pool->chrom[idx2]->index)
     ERROR("reRunPair(): parent index not agree w/ pool index, exit");
 *//*we can not make that check because the pool passed in may not the host  pool of parent p1 and p2 for
the case of registered ga replace op is reByAppend, at this case the pool passed in is new_pool, and the
host  pool of parents is old_pool.*/
  /*==handling re_elitist==*/
  if(ga_center->re_elitist) rePickBest(ga_center,p1,p2,c1,c2);/*after that c1,c2 are the best*/
  /*==call ga registered replace operator to replace two children*/
  ga_center->RE_fun(ga_center,pool,c1,c2);
}
/*---------------------------------------------------------------------------------------------------------------
```

```
| reRunPool()---user interface to replace all new children into new pool by passing twined new children
to ga registered replace operator(except reByAppend()) to perform replacement until finished. It is only
used by teaditional ga
-------------------------------------------------------------------------------------------------------------*/
void reRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int n)
{ int i=0;          /*-- n=number of total new children in child array*/
 CHROM_PTR p1, p2;
 /*==check the validation==*/
 if(!gacValid(ga_center))ERROR("reRunPool(): gacValid check fails,exit");
 if(!poolValid(pool))ERROR("reRunPool(): poolValid() check fails, exit");
 if(child==NULL)ERROR("reRunPool(): null child array address, exit");
 if(n<=0)ERROR("reRunPool(): no any child in child array, n<=0, exit");
 if(ga_center->RE_fun==NULL)ERROR("reRunPool:no RE_fun provid in ga,exit");
 /*==check the pool size==*/
 if(ga_center->pool_max_size!=pool->max_size)ERROR("reRPool:p max_sz,exit");
 if(pool->max_size<=0)ERROR("reRunPool(): pool max size <=0, exit");
 if(pool->size<0)ERROR("reRunPool(): pool actual size <0, exit");
 if(pool->size>pool->max_size)ERROR("reRunPool():pl size>max_size,exit");
 /*==handling re_elitist first avoiding parents lost in pool due to replace*/
 if(ga_center->re_elitist)
 {   for(i=0;i<=n-2;i+=2)
     {   /*--parents index check--*/
        /*check the link of 2 children*/
        if(child[i]->parent1!=child[i+1]->parent2||child[i]->parent2!=child[i+1]->parent1)
            ERROR("reRunPool:not agreed on parent link of twined children,exit");
        /*achieve the parents according to record of children's parent idx*/
        p1=pool->chrom[child[i]->parent1]; p2=pool->chrom[child[i+1]->parent1];
        /*check the link of parent w/ child, the same as w/ pool idx*/
        /*this is feasible since at this stage no any move inside pool*/
        if(child[i]->parent1!=p1->index || child[i+1]->parent1!=p2->index)
            ERROR("reRunPool:not agreed on parent idx link w/ pool idx,exit");
        rePickBest(ga_center,p1,p2,child[i],child[i+1]);
        /*after that child[i],child[i+1] are the best among 4*/
     }/*after for-loop all children in child array are elitisted*/
     /*then we can do any kind of replace no fear of lost parents because we don't need parents anymore in
reByRank, reByFirstWeak, reByWeakest, we do need parent index in reByParent, in fact in this operator
only  they own children can replace them, so parents never loss but replaced by their own children*/
 }/*end of handling re_elitist*/
 /*==pass twin children to ga registered replace operator==*/
 for(i=0;i<=n-2;i+=2)
 {   /*--parents index check--*/
     if(child[i]->parent1!=child[i+1]->parent2||child[i]->parent2!=child[i+1]->parent1)
         ERROR("reRunPool:not agreed on parent index of twined children,exit");
     /*--call ga registered replace operator to replace two children*/
     ga_center->RE_fun(ga_center,pool,child[i],child[i+1]);
 }
}


/*============ end of file: replace.c =========*/
```

```
file: report.c
/*==================================================================
| Report functions
|
| /*===User interface called by ga======================================
| rpConfig()---report  ga_center config information by calling gacPrint() to print to rp_fid.
| rpReport()---user interface, manager to call rpTime(), rpShort(),...
| rpFinal()-----report final result of best chrom matrix and fitness by calling chromPrint()
|               to print to rp_fid.
| /*===Report operators & utility called by user interface========================
| rpTime()-----check to see if it is a report time.
| rpMini()-----only report current generation statistics information.
| rpShort()----short report current generation statistics information and best chrom of
|              current generation by call chromPrint().
| rpLong()-----long report of current generation statistics information the first portion
|               is same as rpShort(), the second part is print out all chroms of current
|               generation by calling chromPrint().
==================================================================*/
#include"gah.h"
/*==================================================================
| function prototypes
==================================================================*/
void rpConfig(GA_CENTER_PTR ga_center);
void rpReport(GA_CENTER_PTR ga_center,POOL_PTR pool);
void rpFinal(GA_CENTER_PTR ga_center);
int rpTime(GA_CENTER_PTR ga_center);
void rpShort(GA_CENTER_PTR ga_center,POOL_PTR pool);
void rpLong(GA_CENTER_PTR ga_center,POOL_PTR pool);
void rpMini(GA_CENTER_PTR ga_center,POOL_PTR pool);

extern int chromComp(GA_CENTER_PTR, CHROM_PTR, CHROM_PTR);
extern void chromPrint(GA_CENTER_PTR ga_center,CHROM_PTR chrom);
extern int poolValid(POOL_PTR pool);
extern int gacValid(GA_CENTER_PTR ga_center);
extern void gacPrint(GA_CENTER_PTR ga_center);

/*==================================================================
| User interface
==================================================================*/
/*------------------------------------------------------------------
| rpConfig()---report ga_center config information by calling gacPrint() to print to rp_fid.
------------------------------------------------------------------*/
void rpConfig(GA_CENTER_PTR ga_center)
{ /*--- error check  ---*/
  if(!gacValid(ga_center))ERROR("rpConfig(): gacValid() check fails,exit");
  if(ga_center->rp_fid==NULL)ERROR("rpConfig():invalid rp_fid in ga,exit");
  /*--- call gacPrint() to do report---*/
  switch(ga_center->rp_type)
  { case RP_NONE: break;
    default           : gacPrint(ga_center); break;
  }
}
/*------------------------------------------------------------------
| rpReport()---user interface, manager to call rpTime(), rpShort(),...
```

```
-----------------------------------------------------------------------------------------------------------*/
void rpReport(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("rpReport():gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("rpReport(): poolValid() check fails, exit");
 if(ga_center->rp_fid==NULL)ERROR("rpReport():invalid rp_fid in ga,exit");
 /*---check if it is the report time---*/
 if(!rpTime(ga_center))return;
 /*---report according to rp_type--*/
 switch(ga_center->rp_type)
 { case RP_NONE  : break;
   case RP_MINI   : rpMini(ga_center,pool); break;
   case RP_SHORT:  rpShort(ga_center,pool); break;
   case RP_LONG :  rpLong(ga_center,pool); break;
 }
}
/*-----------------------------------------------------------------------------------------------------------
| rpFinal()----report final result of best chrom matrix and fitness by calling chromPrint() to print to rp_fid.
-----------------------------------------------------------------------------------------------------------*/
void rpFinal(GA_CENTER_PTR ga_center)
{ int i, idx;
 FILE* fid;
 /*===check the validation==*/
 if(!gacValid(ga_center))ERROR("rpFinal():gacValid() check fails,exit");
 if(ga_center->rp_fid==NULL)ERROR("rpFinal():invalid rp_fid in ga,exit");
 /*===print info to rp_fid==*/
 fid=ga_center->rp_fid;
 /*===skip final report if RP_NONE==*/
 if(ga_center->rp_type==RP_NONE)return;
 /*===print header line===*/
 fprintf(fid,"\n");
 fprintf(fid,"*******************begin final********");
 fprintf(fid,"****************************************");
 fprintf(fid,"\n");
 /*===if ga converged==*/
 fprintf(fid,"Final ga report:\n");
 fprintf(fid,"----------------\n");
 if(ga_center->use_converge&&ga_center->converged)
     fprintf(fid,"ga has converged after %d iterations.\n",ga_center->iter-1);
 else  fprintf(fid,"reach maximum %d iterations limit.\n",ga_center->max_iter);
 /*==print the best chromosome==*/
 /*--find the most best chrom from ga_cneter best history list array--*/
 if(ga_center->max_iter<=0)/*no limit on iteration, best list has 2 cell*/
     idx=0;/*the most best chrom is at 0th cell*/
 else/*has max iter limit, best list has max_iter+2 cells to record the */
 {   idx=0;/*best of every iter from 0th to max_iter+1th cell,last cell null*/
     for(i=1;i<=ga_center->iter-1;i++)
     {   if(chromComp(ga_center,ga_center->best[idx],ga_center->best[i])>0) idx=i;
     }
 }
 fprintf(fid,"the best chrom of GA:\n");
 /*--print onto rp_fid immediately, flush fid before call chromPrint()--*/
 fflush(fid);
 /*--print most best chrom at idx cell of best history list array--*/
chromPrint(ga_center,ga_center->best[idx]);
/*===print footer line===*/
fprintf(fid,"\n");
fprintf(fid,"********************end finial*********");
fprintf(fid,"****************************************");
fprintf(fid,"\n");
/*===print onto rp_fid immediately==*/
fflush(fid);
/*==close report file==*//*rp_fid is open in gacRead(),close in rpFinal*/
fclose(fid);
}
/*=====================================================================
| Report operators & utility called by user interface
=====================================================================*/
/*-----------------------------------------------------------------------------------------------------------
| rpTime()-----check to see if it is a report time
-----------------------------------------------------------------------------------------------------------*/
int rpTime(GA_CENTER_PTR ga_center)
{ /*===check the validation==*/
 if(!gacValid(ga_center))ERROR("rpTime():gacValid() check fails,exit");
 /*===0th generation==*/
 if(ga_center->iter==0)return TRUE;
 /*===report interval reached==*/
 if(!(ga_center->iter % ga_center->rp_interval))return TRUE;
 /*===last generation==*/
 if(ga_center->iter==ga_center->max_iter)return TRUE;
 /*===ga converged===*/
 if(ga_center->use_converge&&ga_center->converged)return TRUE;
 /*=== otherwise, not time for report==*/
 return FALSE;
}
/*-----------------------------------------------------------------------------------------------------------
| rpMini()-----only report current generation statistics information
-----------------------------------------------------------------------------------------------------------*/
void rpMini(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ FILE* fid;
 /*---check the validation---*/
 if(!gacValid(ga_center))ERROR("rpMini():gacValid() check fails, exit");
 if(!poolValid(pool))ERROR("rpMini(): poolValid() check fails, exit");
 if(ga_center->rp_fid==NULL)ERROR("rpMini():invalid rp_fid in ga,exit");
 /*===print info to rp_fid==*/
 fid=ga_center->rp_fid;
 /*===print header line in 0th generation call only===*/
 if(ga_center->iter==0)
 { fprintf(fid,"\n%s\n%s\n",
          "ITER  BEST     VAR      DEV      AVE      MIN      MAX      TOT",
          "----- -------- -------- -------- -------- -------- -------- --------" );
 }
 /*---print a line of current generation--*/
 fprintf(fid,"%5d %8G %8G %8G %8G %8G %8G %8G\n",
 ga_center->iter,pool->chrom[pool->best_index]->fitness,pool->var,
 pool->dev,pool->ave,pool->min,pool->max,pool->tot_fitness);

}
```

```
/*----------------------------------------------------------------------------------------------------------------
| rpShort()----short report of current generation statistics information and best chrom of current generation
by call chromPrint().
-----------------------------------------------------------------------------------------------------------------*/
void rpShort(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ FILE* fid;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("rpShort():gacValid() check fails, exit");
  if(!poolValid(pool))ERROR("rpShort(): poolValid() check fails, exit");
  if(ga_center->rp_fid==NULL)ERROR("rpShort():invalid rp_fid in ga,exit");
  /*===print info to rp_fid==*/
  fid=ga_center->rp_fid;
  /*===print header line===*/
  fprintf(fid,"\n");
  fprintf(fid,"====================begin short========");
  fprintf(fid,"=======================================");
  fprintf(fid,"\n");
  /*===print current pool statistics===*/
  fprintf(fid,"GA generation: %d\n",ga_center->iter);
  fprintf(fid,"------------------\n");
  fprintf(fid,"size: %d  mutation: %d  (total: %d)\n",
              pool->size, ga_center->num_mut, ga_center->tot_mut);
  fprintf(fid,"tot: %g, min: %g, max: %g,\nave: %g var: %g dev: %g\n",
              pool->tot_fitness, pool->min, pool->max, pool->ave, pool->var, pool->dev);
  fprintf(fid,"best fitness: %g\n",pool->chrom[pool->best_index]->fitness);
  /*===print best chrom of current pool==*/
  fprintf(fid,"best chrom of generation:\n");
  /*---print onto rp_fid immediately, flush fid before call chromPrint()--*/
  fflush(fid);
  chromPrint(ga_center,pool->chrom[pool->best_index]);
  /*===print footer line===*/
  fprintf(fid,"\n");
  fprintf(fid,"====================end short==========");
  fprintf(fid,"=======================================");
  fprintf(fid,"\n");
  /*===print onto rp_fid immidiately==*/
  fflush(fid);
}
/*----------------------------------------------------------------------------------------------------------------
| rpLong()-----long report of current generation statistics information the first portion is same as rpShort(),
the second part is print out all chroms of current generation by calling chromPrint().
-----------------------------------------------------------------------------------------------------------------*/
void rpLong(GA_CENTER_PTR ga_center,POOL_PTR pool)
{ int i;
  FILE* fid;
  /*---check the validation---*/
  if(!gacValid(ga_center))ERROR("rpLong():gacValid() check fails, exit");
  if(!poolValid(pool))ERROR("rpLong(): poolValid() check fails, exit");
  if(ga_center->rp_fid==NULL)ERROR("rpLong():invalid rp_fid in ga,exit");
  /*===print info to rp_fid==*/
  fid=ga_center->rp_fid;
  /*===print header line===*/
  fprintf(fid,"\n");
  fprintf(fid,"====================begin long=========");
  fprintf(fid,"========================================");
  fprintf(fid,"\n");
  /*===print current pool statistics===*/
  fprintf(fid,"GA generation: %d\n",ga_center->iter);
  fprintf(fid,"-------------------\n");
  fprintf(fid,"size: %d  mutation: %d  (total: %d)\n",
              pool->size, ga_center->num_mut, ga_center->tot_mut);
  fprintf(fid,"tot: %g, min: %g, max: %g,\nave: %g var: %g dev: %g\n",
              pool->tot_fitness, pool->min, pool->max, pool->ave, pool->var, pool->dev);
  fprintf(fid,"best fitness: %g\n",pool->chrom[pool->best_index]->fitness);
  /*===print best chrom of current pool==*/
  fprintf(fid,"best chrom of generation:\n");
  /*---print onto rp_fid immediately, flush fid before call chromPrint()--*/
  fflush(fid);
  chromPrint(ga_center,pool->chrom[pool->best_index]);
  /*=addtion info of long report (the above is same of rpShort()) is print out all chroms of current pool*/
  for(i=0;i<pool->size;i++)
  {   fprintf(fid,"\nthe %dth chrom:\n",i);
      fflush(fid);
      chromPrint(ga_center,pool->chrom[i]);
  }
  /*===print footer line===*/
  fprintf(fid,"\n");
  fprintf(fid,"====================end long===========");
  fprintf(fid,"========================================");
  fprintf(fid,"\n");
  /*===print onto rp_fid immidiately==*/
  fflush(fid);
}




/*=============== end of file: report.c ==============*/
```

file: ga.c

```
/*=================================================================
| GA algorithm:
|
| /*---ga main user interface--------------------------------------------------------------------------------
| gaSet()--set user defined ga functional operator into GA_Table[] and register it into  ga_center
| gaSelect()--select ga function from GA_Table[] by name into ga_center
| gaName()--select & return ga fun name from GA_Table[] by ga_fun ptr
| gaConfig()-user interface for configuring ga algorithm
| gaReset()---reset GA, it is manager to reset ga_center, by calling gacReset() and gacRead().
| gaRun()----runner to run GA by calling gacVerify() to verify ga_center,
|             then calling ga_center->GA_fun() to setup & perform GA.
| /*---ga functional operators to generate new pool-------------------------------------------------------
| gaByTraditional()-----setup and perform traditional GA.
| gaByGenerational()---setup and perform generational GA: handling se_elitist (if provided)
|                        and  gap (if enabled), then select two parents for crossover, mutation and
|                        append to new pool until filled.
| gaBySteady_State()---setup and perform steady_state GA: only maintains one pool, during
|                        each iteration, select two parents from pool to perform crossover, mutation,
|                        if re_elitist flag on, choose best two out of four to replace two parents in
|                        pool, otherwise replace parents with new children, continues utill converge
|                        or max_iteration limit has been reached.
| /*---utilities---------------------------------------------------------------------------------------------
| gaSetup()----setup & make sure everything is ok before run ga iteration, perform final check
|              everywhere, called by ga function.
| gaIterSetup--setup & make sure everything is ok before next iteration by reset new pool, handling
|              se_elitist and gap, zero counters, only used by traditional ga and generational ga
|              but not used by steady_state ga due to it has only one pool.
| gaElitist1()---handle ga se_elitist(=1) before selection of that iteration, it would be called after
|              current pool being swaped to the old-pool and begains the next iteration, it is called
|              by gaIterSetup(), it puts 2 copies of the best of old pool to new pool.
| gaElitist2()---handle ga se_elitist(=2) before selection of that iteration, it would be called after
|              current pool being swaped to the old-pool and begains the next iteration, it is called
|              by gaIterSetup(), it put ELITIST percent number of top bests of old to new pool.
| gaGap()------handle ga generation gap before selection of that iteration.
| ga_SE_X_MU_RE_2CHILD()----basic operation of ga iteration, that is, select two parents from
|              old pool, crossover and mutation immediately to generate two children, then replace
|              into new_pool according to re_elitist flag. It is called when mu_flag: "MU_CHILD"
|              by generational and Steady-State ga within iteration.
| ga_SE_X_RE_2CHILD()-----------basic operation of ga iteration, that is, select two parents from
|              old pool, crossover but not mutation immediately to generate two children, instead
|              do replace into new_pool first, according to re_elitist flag, after finishing all crossover
|              and replace operations of that iteration, we do mutation of the whole pool. It is called
|              when mu_flag: "MU_BIT", or "MU_CHROM" by generational and Steady_State ga
|              within each iteration.
| gaStats()-----update the status data of ga_center, it is called after each new pool is formed,
|              it calls poolStats() to update new pool, then the following should be updated after
|               each iteration by gaStats(): converged, best, old_pool, new_pool of ga_center.
| /----SAGA utility------------------------------------------------------------------------------------------
| accept()-------SAGA accepting function used by SAGA's SAC and SAM to determine if accept a
|               new child or not. (1/ga_center->iter) is the scheduled annealing temperature when
|               SAGA has not max. iteration limit, (max_iter - iter) is the scheduled annealing
|               temperature when SAGA has the max.iteration limit .
=================================================================*/
```

```
#include "gah.h"
/*=====================================================================
| data structures
=====================================================================*/
/*--global variables used generational and steady_state ga: dynamical alloc space by gaSetup(), used by
ga_SE_X_MU_RE_2CHILD() and ga_SE_X_RE_2CHILD(), deallocated (free) by gaByTraditional(),
gaByGenerational(), gaBySteady_State() before exit and terminate these functions.*/

CHROM_PTR child1,child2;

int gaByTraditional(GA_CENTER_PTR ga_center);
int gaByGenerational(GA_CENTER_PTR ga_center);
int gaBySteady_State(GA_CENTER_PTR ga_center);

FN_TABLE GA_Table[]=
{ {NULL,                NULL},
  {"traditional",    gaByTraditional},
  {"generational", gaByGenerational},
  {"steady_state", gaBySteady_State},
  {NULL,                NULL},
};
/*note: FN_TABLE[] must be NULL terminated for function search to be stop*/
/*=====================================================================
| function prototype declarations
=====================================================================*/
void gaSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr);
void gaSelect(GA_CENTER_PTR ga_center,char* fn_name);
char* gaName(GA_CENTER_PTR ga_center);
GA_CENTER_PTR gaConfig(char* cfgfile, FN_PTR EV_fun);
void gaReset(GA_CENTER_PTR ga_center, char* cfgfile);
void gaRun(GA_CENTER_PTR ga_center);

void gaSetup(GA_CENTER_PTR ga_center);
void gaIterSetup(GA_CENTER_PTR ga_center);
void ga_SE_X_MU_RE_2CHILD(GA_CENTER_PTR ga_center);
void ga_SE_X_RE_2CHILD(GA_CENTER_PTR ga_center);
void gaStats(GA_CENTER_PTR ga_center);

void gaElitist1(GA_CENTER_PTR ga_center);
void gaElitist2(GA_CENTER_PTR ga_center);
void gaGap(GA_CENTER_PTR ga_center);

float accept(GA_CENTER_PTR ga_center,CHROM_PTR p,CHROM_PTR c);

extern CHROM_PTR chromAlloc(int);
extern void chromFree(CHROM_PTR );
extern void chromKill(CHROM_PTR );
extern void chromCopy(CHROM_PTR , CHROM_PTR);
extern int chromComp(GA_CENTER_PTR, CHROM_PTR , CHROM_PTR );
extern void chromVerify(GA_CENTER_PTR,CHROM_PTR );
extern int poolValid(POOL_PTR pool);
extern void poolReset(POOL_PTR pool);
extern POOL_PTR poolAlloc(int max_size);
extern void poolFree(POOL_PTR pool);
extern void poolKill(POOL_PTR pool);
extern void poolInit(GA_CENTER_PTR ga_center, POOL_PTR pool);
extern void poolAppend(POOL_PTR pool, CHROM_PTR chrom, int copy);
extern void poolAlign(POOL_PTR pool);
extern void poolRank(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolIndex(POOL_PTR pool);
extern void poolFitness(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolPtf(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void poolStats(GA_CENTER_PTR ga_center,POOL_PTR pool);

extern int gacValid(GA_CENTER_PTR ga_center);
extern void gacReset(GA_CENTER_PTR ga_center);
extern GA_CENTER_PTR gacAlloc(void);
extern void gacFree(GA_CENTER_PTR ga_center);
extern int gacRead(GA_CENTER_PTR ga_center,char* cfgfile);
extern void gacVerify(GA_CENTER_PTR ga_center);

extern char* seName(GA_CENTER_PTR ga_center);
extern CHROM_PTR seRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);

extern void xRunPair(GA_CENTER_PTR ga_center,
                     CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
extern void xRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR* child, int* n);
extern void sacRunPair(GA_CENTER_PTR ga_center,
                       CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
extern void sacRunPool(GA_CENTER_PTR ga_center,POOL_PTR pool,CHROM_PTR* child,int* n);

extern void muRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom);
extern void muRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);
extern void samRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool);
extern void samRunSingleChrom(GA_CENTER_PTR ga_center, CHROM_PTR chrom);

extern char* reName(GA_CENTER_PTR ga_center);
extern void reRunPair(GA_CENTER_PTR ga_center, POOL_PTR pool,
                      CHROM_PTR p1,CHROM_PTR p2, CHROM_PTR c1, CHROM_PTR c2);
extern void reRunPool(GA_CENTER_PTR ga_center, POOL_PTR pool,CHROM_PTR* child, int n);

extern void rpConfig(GA_CENTER_PTR ga_center);
extern void rpReport(GA_CENTER_PTR ga_center,POOL_PTR pool);
extern void rpFinal(GA_CENTER_PTR ga_center);

extern void setFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                   char* fn_name,FN_PTR fn_ptr,FN_PTR* rtn_ptr);
extern void selectFun(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,
                      char* fn_name,FN_PTR* rtn_ptr);
extern char* getFunName(GA_CENTER_PTR ga_center,FN_TABLE_PTR FN_Table,FN_PTR fn_ptr);
/*================================================================
| ga main user interface
================================================================*/
/*----------------------------------------------------------------
| gaSet()     -set user defined ga function operator into GA_Table[] and register it into ga_center.
----------------------------------------------------------------*/
void gaSet(GA_CENTER_PTR ga_center,char* fn_name,FN_PTR fn_ptr)
{ /*--- error check ---*/
```

```
 if(!gacValid(ga_center)) ERROR("gaSet(): gacValid() check fails, exit");
 if(strlen(fn_name)==0) ERROR("gaSet(): no user fun name, exit");
 if(fn_ptr==NULL) ERROR("gaSet(): Null user fun ptr, exit");
 /*--- call setFun() to operate on the GA_Table[]--*/
 setFun(ga_center,GA_Table,fn_name,fn_ptr,&(ga_center->GA_fun));
}
/*-------------------------------------------------------------------------------------------------------
| gaSelect() -set&select ga function from GA_Table[] by name into ga_center
-------------------------------------------------------------------------------------------------------*/
void gaSelect(GA_CENTER_PTR ga_center,char* fn_name)
{ /*--- error check  ---*/
 if(!gacValid(ga_center))ERROR("gaSelect(): gacValid() check fails,exit");
 if(strlen(fn_name)==0)ERROR("gaSelect(): no fun name provided, exit");
 /*--- call selectFun() to operate on the GA_Table[]--*/
 selectFun(ga_center,GA_Table,fn_name,&(ga_center->GA_fun));
}
/*-------------------------------------------------------------------------------------------------------
| gaName()    -select & return ga fun name from GA_Table[] by ga_fun ptr
-------------------------------------------------------------------------------------------------------*/
char* gaName(GA_CENTER_PTR ga_center)
{ /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("gaName(): gacValid() check fails, exit");
 /*--- call getFunName() to operate on GA_Table[]--*/
 return getFunName(ga_center,GA_Table,ga_center->GA_fun);
}
/*-------------------------------------------------------------------------------------------------------
| gaConfig() -user interface for configuring ga algorithm ga_center configuration manager: call gacAlloc()
to get memory and default reset for ga_center; register evaluation function into ga_center (only place do
so); call gacRead() to initialize ga_center by configuration file (if only default setting ga_center, pass
NULL value of cfgfile, if need user config ga_center also, pass valid cfgfile name), evaluation function
EV_fun() must be int type due to FN_PTR. no error check for cfgfile name since it is allowed to be NULL.
-------------------------------------------------------------------------------------------------------*/
GA_CENTER_PTR gaConfig(char* cfgfile, FN_PTR EV_fun)
{ GA_CENTER_PTR ga_center;
 /*--- error check  ---*/
 if(EV_fun==NULL) ERROR("gaConfig():null EV_fun ptr passed,exit");
 /*--allocate memory for ga_center--*/
 ga_center=gacAlloc();   /*alloc memory and default reset ga_center*/
 /*--registered user evaluation function --*/
 ga_center->EV_fun=EV_fun;/*this is only place to register EV_fun*/
 /*--initial config ga_center again by reading cfg file if provided*/
 if(cfgfile!=NULL && cfgfile[0]!='\0' && cfgfile[0]!='\n' ) gacRead(ga_center, cfgfile);
 return ga_center;
}
/*-------------------------------------------------------------------------------------------------------
| gaReset()    -Reset GA, it is a manager to reset ga_center, by calling gacReset(): reset GA by default reset
ga_center, then calling gacRead(): cfg file reset ga_center if cfg file is provided
-------------------------------------------------------------------------------------------------------*/
void gaReset(GA_CENTER_PTR ga_center, char* cfgfile)
{ FN_PTR temp;
 /*--- error check  ---*/
 if(!gacValid(ga_center)) ERROR("gaReset():gacValid() check fails,exit");
 /*it is allowed cfgfile may be NULL*/
 /*--save EV_fun of ga_center*/
 temp=ga_center->EV_fun;
 /*--defaut reset ga_center--*/
 gacReset(ga_center); /*actually gacReset() doesn't touch EV_fun*/
 /*--restore EV_fun*/
 ga_center->EV_fun=temp;
 /*--cfg file reset ga_center again if cfg file is provided--*/
 if(cfgfile!=NULL && cfgfile[0]!='\0' && cfgfile[0]!='\n' ) gacRead(ga_center, cfgfile);
}
/*----------------------------------------------------------------------------------------------------------
| gaRun()-runner to run GA by calling gacVerify() to verify ga_center then calling ga_center->GA_fun()
to setup & perform GA.
----------------------------------------------------------------------------------------------------------*/
void gaRun(GA_CENTER_PTR ga_center)
{ /*--- error check ---*/
 if(!gacValid(ga_center))ERROR("gaRun():gacValid() check fails,exit");
 /*---every thing in ga_center is OK--*/
 gacVerify(ga_center);
 /*--print out configuration info of ga_center--*/
 /*rpConfig(ga_center);*//*move to ga funct after gaSetup()*/
 /*---seeds random number generator--*/
 SRAND(ga_center->rand_seed);
 /*--- run the ga --*/
 ga_center->GA_fun(ga_center);
}
/*=====================================================================
| ga functional operators
=====================================================================*/
/*----------------------------------------------------------------------------------------------------------
| gaByTraditional()--setup and perform traditional GA: handling se_elitist (if provided): here elitist means
that make sure at least two copies of best chroms were put into the new pool, such that enhance the chance
for the best to win a crossover to contribute for the next generation when they participate the crossover
competition with the rest of just selected new pool (reproduce pool); handling gap(if provided): transfer
directly several chroms of old pool to the reproduce pool randomly( actually, the gap handling can be
viewed as a part of selection operation); selection: after se_elitist handler put several best chroms into new
pool, we call select operation interface to use ga registered selection operator to randomly select the rest
chroms from old pool to put into new pool untill full up to the pool size; crossover: call crossover interface
to randomly choose chroms from the whole new pool according to the x_rate for crossover to form mating
pool and randomly pair chroms, then use ga registered crossover operator randomly choose x positions to
produce children; after all children have been produced, we have 2 way to deal with bit mutations and
replacements: first way: at first mutation to children then replace, at this time ga_center mu_flag has
value: "MU_CHILD", perform muRunSingleChrom() for each new child before place him into the new
pool. that is, after crossover producing all children, we call muRunSingleChrom() for each new children,
finally call replacement user interface to replace all these new mutated children into new pool; second
way: at first replace children into new pool then do mutation for whole new pool, at this time the
ga_center mu_flag has value either "MU_CHROM" or "MU_BIT", after all new children born, do replace
first, repalce all children into new pool, then we call mutation operation interface (muRunPool()) to do
mutation on the whole new pool by using ga registered mutation operato. call: seRunPool(), xRunPool(),
muRunSingleChrom(), reRunPool, muRunPool.(for taditional ga, gap no much special since it is identical
to selection, since they all use seRunPool() to select from old to put in new).
| Traditional ga:  selection  : any
|                  crossover  : any
|                  mutation   : any
|                   replace   : any except append
----------------------------------------------------------------------------------------------------------*/
```

```
int gaByTraditional(GA_CENTER_PTR ga_center)
{ CHROM_PTR parent;          /*to hold parent selected from old to put into new pool*/
  static CHROM_PTR* child;   /*used by xRunPool(), reRunPool()*/
  static int childnum=-1;  int i;
  /*== error check ==*/
  if(!gacValid(ga_center))ERROR("gaByTraditional:gacValid check err,exit");
  /*==setup ga, make sure everything in ga_center is ok==*/
  gaSetup(ga_center);/*initializing start pool(old pool) as 0th iter pool*/
                     /*alloc new pool and best history list spaces*/
  /*--print out configuration info of ga_center--*/
  rpConfig(ga_center);
  /*--report the 0th generation--*/
  rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
  /*==dynamic allocate a children array for crossover mutation replace==*/
  /*--child array is freed before exit gaTraditional*/
  child=(CHROM_PTR*)calloc(ga_center->old_pool->size,sizeof(CHROM_PTR));
  if(child==NULL)ERROR("gaByTradit():alloc child array fail, exit");
  for(i=0;i<=ga_center->old_pool->size-1;i++) child[i]=chromAlloc(ga_center->chrom_dim);
  /*==outer loop is for each ga iteration (generation)start from iter=1==*/
  for(ga_center->iter=1; ga_center->max_iter<=0 ||
     ga_center->iter<=ga_center->max_iter; ga_center->iter++)
  {   /*=check ga converged flag to see if convergence of current generation*/
     if(ga_center->converged&&ga_center->use_converge) break;
     /*==setup for new iteration==*/
     gaIterSetup(ga_center);/*reset new_pool,and handle se_elitist & gap.*/
     /*==randomly select the chroms from old & put into new pool util full=*/
     for(;ga_center->new_pool->size<ga_center->old_pool->size;)
     {   parent=seRunPool(ga_center,ga_center->old_pool);
        chromVerify(ga_center,parent);
        poolAppend(ga_center->new_pool,parent,TRUE);
     }
     /*==before hand over to xRunPool() must treat new pool according to
          the registered select op and replace op.===*/
     /*sort new pool if se or re w/ rank*/
     if(!strcmp(seName(ga_center),"rank_biased")|| !strcmp(reName(ga_center),"replace_rank"))
     {   if(ga_center->new_pool->sorted!=2) poolRank(ga_center,ga_center->new_pool);
     }
     if(!ga_center->new_pool->sorted && !ga_center->new_pool->updated)
        poolAlign(ga_center->new_pool); /*not be  executed  if poolRank() be executed*/
     if(!ga_center->new_pool->updated)
     {   poolIndex(ga_center->new_pool); /*make sure new pool index is correct*/
        poolFitness(ga_center, ga_center->new_pool);
        poolPtf(ga_center, ga_center->new_pool);
        poolStats(ga_center, ga_center->new_pool);
     }
     /*==perform whole pool crossover use x_rate,all new chilren in child array ==*/
     if(RANDFRAC()<=ga_center->sac)/*invoke SAGA's SAC */
        sacRunPool(ga_center,ga_center->new_pool,child,&childnum);
     else xRunPool(ga_center,ga_center->new_pool,child,&childnum);
     /*if no new child due to the possibility of empty mating pool selected from new_pool*/
     if(childnum==0)goto gas;/*no child, so skip mutate and replace*/
     /*now all children hold in child array and already made feasible*/
     /*==perform mutation and replace according to mu_flag==*/
     if(ga_center->mu_flag==MU_CHILD)
```

```
    {   /*--first mutate each child, then replace into pool--*/
        /*==mutate all new children if can, still store in child array==*/
        for(i=0;i<=childnum-1;i++)/*for each child if meet mu_rate,will mut*/
        {   if(RANDFRAC()<=ga_center->sam)/*invoke SAGA's SAM */
                samRunSingleChrom(ga_center,child[i]);
            else muRunSingleChrom(ga_center,child[i]);
        }
        /*==evaluate all crossovered and mutated children in child array==*/
        for(i=0;i<=childnum-1;i++)
        {   chromVerify(ga_center,child[i]);
            if(child[i]->eva==0)ga_center->EV_fun(child[i]);/*we are sure child->eva=0 by x ahead*/
        }
        /*==replace all crossovered & mutationed children into new pool==*/
        reRunPool(ga_center,ga_center->new_pool,child,childnum);
    }/*new generation of this iter is formed in new pool*/
    else if(ga_center->mu_flag==MU_BIT || ga_center->mu_flag==MU_CHROM)
    {   /*--first do replace operator then do mutation of whole pool--*/
        /*==evaluate all crossovered and mutated children in child array==*/
        for(i=0;i<=childnum-1;i++)
        {   chromVerify(ga_center,child[i]);
            if(child[i]->eva==0)ga_center->EV_fun(child[i]);/*we are sure child->eva=0 by x ahead*/
        }
        /*==repalce all crossovered children in child array into new pool==*/
        reRunPool(ga_center,ga_center->new_pool,child,childnum);
        /*==perform whole pool mutation use mu_rate==*/
        if(RANDFRAC()<=ga_center->sam)/*invoke SAGA's SAM */
            samRunPool(ga_center,ga_center->new_pool);
        else muRunPool(ga_center,ga_center->new_pool);
        /*already made every chrom feasible*/
    }/*new generation of this iter is formed in new pool*/
    else { ERROR("gaByTradit(): invalid mu_flag, exit"); }
    /*=update stats in new pool & ga_center,swap old and new for next iter*/
gas:  gaStats(ga_center);
    /*==report if needed==*/
    rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
 }/*end of at max_iter ga iterations, old pool is the current newest generation*/
 /*==Final report==*/
 rpFinal(ga_center);
 /*==Free gene for children==*/
 chromFree(child1);/*glabal, alloc by gaSetup() */
 chromFree(child2);
 for(i=0;i<=ga_center->old_pool->size-1;i++) /*local, alloc in beginning*/
 { chromFree(child[i]); child[i]=NULL;}
 free(child);  child=NULL;
 gacFree(ga_center);
 return OK;
}
/*------------------------------------------------------------------------------------------------------------
| gaByGenerational()--setup and perform generational GA. handling se_elitist if provided, gap if enabled,
then select two parents for crossover, mutation until filled up new pool.
| Generational ga: selection  :  any
|                  crossover  :  any
|                  mutation   :  any
|                   replace   :  only append
```

```
-------------------------------------------------------------------------------------------------------------*/
int gaByGenerational(GA_CENTER_PTR ga_center)
{ /*== error check ==*/
 if(!gacValid(ga_center))ERROR("gaByGenerationl:gacValid check fails,exit");
 /*==setup ga, make sure everything in ga_center is ok==*/
 gaSetup(ga_center);/*initializing start pool(old pool) as 0th iter pool*/
                    /*alloc new pool and best history list spaces*/
 /*--print out configuration info of ga_center--*/
 rpConfig(ga_center);
 /*--report the 0th generation--*/
 rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
 /*==outer loop is for each ga iteration (generation) start from iter=1==*/
 for(ga_center->iter=1; ga_center->max_iter<=0 ||
     ga_center->iter<=ga_center->max_iter; ga_center->iter++)
 {   /*=check ga converged flag to see if convergence of current generation=*/
    if(ga_center->converged&&ga_center->use_converge) break;
    /*==setup for new iteration==*/
    gaIterSetup(ga_center);/*reset new_pool,and handle se_elitist & gap.*/
    /*==handling mu_flag, is it mutation immediately or mutation pool?==*/
    if(ga_center->mu_flag==MU_CHILD)/*mu child immediately after x */
    {   /*==inner loop for each single reproduction(basic ga op) of 2 kids==*/
       for(;ga_center->new_pool->size<ga_center->old_pool->size;)
       {   ga_SE_X_MU_RE_2CHILD(ga_center);/*SE,X,MU,RE 2 kids into new pool*/
       }/*end reproduce new pool(new generation)*/
    }/*new generation of this iter is formed in new pool when MU_CHILD */
    else if(ga_center->mu_flag==MU_CHROM || ga_center->mu_flag==MU_BIT)
    /*mu pool after SE,X,RE all kids*/
    {   /*==inner loop for each single reproduction(basic ga op) of 2 kids==*/
       for(;ga_center->new_pool->size<ga_center->old_pool->size;)
       {   ga_SE_X_RE_2CHILD(ga_center);/*SE,X,RE 2 kids into new pool*/
       }/*end reproduce new pool, but need mu pool*/
       /*==perform whole pool mutation use mu_rate==*/
       if(RANDFRAC()<=ga_center->sam)/*invoke SAGA's SAM */
          samRunPool(ga_center,ga_center->new_pool);
       else muRunPool(ga_center,ga_center->new_pool);
    }/*new generation of iter is formed in new pool when MU_CHROM,MU_BIT */
    else { ERROR("gaByGenerational(): invalid mu_flag, exit"); }
    /*=update stats in new pool & ga_center, swap old and new for next iter*/
    gaStats(ga_center);
    /*==report if needed==*/
    rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
 }/*end of at max_iter ga iterations,
 old pool is the current newest generation*/
 /*==Final report==*/
 rpFinal(ga_center);
 /*==Free gene for childrex==*/
 chromFree(child1); /*glabal, alloc by gaSetup() */
 chromFree(child2);
 gacFree(ga_center);
 return OK;
}
/*-----------------------------------------------------------------------------------------------------------
| gaBySteady_State()-setup and perform steady_state GA. only maintains one pool during each iteration,
each iteration only consists of randomly select two parents from pool to perform crossover, mutation, if
```

```
re_elitist flag on, choose best two out of four to replace two parents in pool, otherwise replace parents with
new children, continues untill convergence or max_iter limit reached. gap and se_elitist handling before
each iter is disabled for steady_state ga, since they play no roles in just one pool envolving ga like
steady_state ga.
| steady_state ga:  selection  : any
|                   crossover  : any
|                   mutation   : any
|                    replace   : any except append
---------------------------------------------------------------------------------------------------------------*/
int gaBySteady_State(GA_CENTER_PTR ga_center)
{ /*== error check ==*/
 if(!gacValid(ga_center))ERROR("gaBySteady_State:gacValid check fails,exit");
 /*==setup ga, make sure everything in ga_center is ok==*/
 gaSetup(ga_center);/*initializing start pool(old pool) as 0th iter pool*/
                    /*alloc new pool and best history list spaces*/
 /*--print out configuration info of ga_center--*/
 rpConfig(ga_center);
 /*--report the 0th generation--*/
 rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
 /*==only need one pool, free new pool, let new and old ptr to same pool*/
 poolFree(ga_center->new_pool);
 ga_center->new_pool=ga_center->old_pool;
 /*==outer loop is for each ga iteration (generation) start from iter=1==*/
 for(ga_center->iter=1; ga_center->max_iter<=0 ||
     ga_center->iter<=ga_center->max_iter; ga_center->iter++)
 {   /*=check ga converged flag to see if convergence of current generation*/
     if(ga_center->converged&&ga_center->use_converge) break;
     /*==handling mu_flag, is it mutation immidiately or mutation pool?==*/
     if(ga_center->mu_flag==MU_CHILD)/*mu child immidiately after x */
     {   /*==inner loop for each single reproduction(basic ga op) of 2 kids==*/
        /*==inner loop is a single reproduction==*/
        ga_SE_X_MU_RE_2CHILD(ga_center);/*SE,X,MU,RE 2 kids into new pool*/
     }/*new generation of this iter is formed in new pool when MU_CHILD */
     else if(ga_center->mu_flag==MU_CHROM || ga_center->mu_flag==MU_BIT)
     /*mu pool after SE,X,RE all kids*/
     {   /*==inner loop for each single reproduction(basic ga op) of 2 kids==*/
        /*==inner loop is a single reproduction==*/
        ga_SE_X_RE_2CHILD(ga_center);/*SE,X,RE 2 kids into new pool*/
        /*end reproduce new pool, but need mu pool*/
        /*==perform whole pool mutation use mu_rate==*/
        if(RANDFRAC()<=ga_center->sam)/*invoke SAGA's SAM */
           samRunPool(ga_center,ga_center->new_pool);
        else muRunPool(ga_center,ga_center->new_pool);
     }/*new generation of iter is formed in new pool when MU_CHROM,MU_BIT */
     else { ERROR("gaBySteady_State(): invalid mu_flag, exit");}
     /*=update stats in new pool & ga_center, swap old and new for next iter*/
     gaStats(ga_center);
     /*==report if needed==*/
     rpReport(ga_center,ga_center->old_pool);/*old_pool is the current pool*/
     /*acturally old_pool and new_pool is referred to the same pool, so we
       can not use gaIterSetup() to reset new_pool, no iter setup for this Steady_State GA */
 }/*end of at max_iter ga iterations,
 old pool is the current newest generation*/
 /*==Final report==*/
 rpFinal(ga_center);
 /*==Free gene for childrex==*/
 chromFree(child1);  /*glabal, alloc by gaSetup() */
 chromFree(child2);
 gacFree(ga_center);
 return OK;
}
/*==================================================================
| utilities
====================================================================*/
/*------------------------------------------------------------------------------------------------------------------
| gaSetup()-----setup and make sure everything is ok before run ga, check old and new pool memory alloc
& initi start old pool, check and alloc memory for best history list, zero counter variables:
iter=num_mut=tot_mut=0; we regard initi start pool as the result of 0th iteration. alloc memory for two
global public child pointers, perform final check everywhere.
-------------------------------------------------------------------------------------------------------------------*/
void gaSetup(GA_CENTER_PTR ga_center)
{ POOL_PTR old_pool, new_pool;
  int i;
  /*== error check ==*/
  if(!gacValid(ga_center))ERROR("gaSetup(): gacValid() check fails, exit");
  /*== make sure pool memory allocation is ok ==*/
  /*-- if needed, allocate pool memory and default reset --*/
  /*--make sure we have a valid old_pool--*/
  if(ga_center->old_pool!=NULL)
  {   if(!poolValid(ga_center->old_pool))/*old_pool is illness, kill & alloc*/
      {   poolKill(ga_center->old_pool); ga_center->old_pool=poolAlloc(ga_center->pool_max_size);
      }
  }
  else /*in gacAlloc() we set old_pool=NULL*/
  {   ga_center->old_pool=poolAlloc(ga_center->pool_max_size);
  }
  /*--make sure we have a valid new_pool--*/
  if(ga_center->new_pool!=NULL)
  {   if(!poolValid(ga_center->new_pool))/*above already sure old_pl good*/
      {   if(ga_center->new_pool!=ga_center->old_pool)/*so this if always true*/
              poolKill(ga_center->new_pool);
          ga_center->new_pool=poolAlloc(ga_center->pool_max_size);
      }
      else
      {   if(ga_center->new_pool==ga_center->old_pool)
              ga_center->new_pool=poolAlloc(ga_center->pool_max_size);
      }
  }
  else /*in gacAlloc() we set new_pool=NULL*/
  {   ga_center->new_pool=poolAlloc(ga_center->pool_max_size);
  }
  old_pool=ga_center->old_pool;
  new_pool=ga_center->new_pool;
  /*==prepare old_pool and new_pool for ga uses ==*/
  /*---Initialize starting working pool(old_pool) as 0th ga iteration--*/
  poolInit(ga_center,old_pool);/*initialize chroms and update status data*/
  /*---disable pool initial flag in ga_center to provent second initial--*/
  /*if(ga_center->ip_flag!=IP_NONE) ga_center->ip_flag=IP_NONE;
 */
 /*---pool must have even number of chromosomes--*/
 /*since in gaGen,gaStead, which are depended on ga_SE_X_MU_RE_2CHILD(),
 for each reprodue we choose two chroms as parents to reproduce two
 children to place into new according to elitist.*/
 if(old_pool->size%2!=0)
 {   /*make one more copy of best of pool*/
     poolAppend(old_pool,old_pool->chrom[old_pool->best_index],TRUE);
 }
 /*==call pool stats update routines to update old_pool==*/
 /*--Evaluate pool--*/
 /*sort old pool of 0th generation if se or re w/ rank*/
 if(!strcmp(seName(ga_center),"rank_biased")||!strcmp(reName(ga_center),"replace_rank"))
 {   if(old_pool->sorted!=2) poolRank(ga_center,old_pool);
 }
 if(!old_pool->sorted && !old_pool->updated)
     poolAlign(old_pool); /*not exe if poolRank() exe*/
 if(!old_pool->updated)
 {   poolIndex(old_pool); /*make sure new pool index is correct*/
     poolFitness(ga_center, old_pool);
     poolPtf(ga_center, old_pool);
     poolStats(ga_center, old_pool);
 }
 /*the 0th generation pool finished, ready for use to generate 1st iter*/
 /*===alloc new pool chroms space==*/
 for(i=0;i<=old_pool->size-1;i++) new_pool->chrom[i]=chromAlloc(ga_center->chrom_dim);
 /*---zero new_pool size for 1st generation--*/
 new_pool->size=0;

/*  if(old_pool->size%2!=0)
  { /*make one more copy of best of pool*/
/*      poolAppend(old_pool,old_pool->chrom[old_pool->best_index],TRUE);
 /*update stats of oth generation pool due to this poolAppend*/
 /*--Evaluate pool--*/
/*      poolIndex(old_pool);
 poolFitness(ga_center,old_pool);
 poolPtf(ga_center,old_pool);
 /*--update states of pool--*/
/*  poolStats(ga_center,old_pool);
  }/*the 0th generation pool finished, ready for use to generate 1st iter*/
  /*---zero new_pool size for 1st generation--*/
/*new_pool->size=0;
*/

 /*==make sure best chromosome history list is allocated in ga_center==*/
 /*==we alloc 2 cells if max_iter<0, the last cell used as null terminator*/
 /*==we alloc max_iter+2 cells if max_iter>0, the 0th cell is used for 0th
 generation that is initial pool, last cell used as null terminator*/
 if(ga_center->best==NULL) /*in gacAlloc() we set best=NULL*/
 {   if(ga_center->max_iter<=0)/*no max_iter limit, so alloc only 2 cells*/
     {   ga_center->best=(CHROM_PTR*)calloc(2,sizeof(CHROM_PTR));
         if(ga_center->best==NULL)ERROR("gaSetup(): alloc best 0 fails, exit");
         ga_center->best[0]=(CHROM_PTR)chromAlloc(ga_center->chrom_dim);
         if(ga_center->best[0]==NULL)ERROR("gaSetup: alloc best 1 fails, exit");
```

```
        ga_center->best[1]=NULL;/*null terminate best array of 2 cells*/
    }/*late refer best matrix by ga_center->best[0]->gene*/
    else/*has max_iter limit, so alloc max_iter+2 cells for best array*/
    {   ga_center->best=(CHROM_PTR*)calloc(ga_center->max_iter+2,sizeof(CHROM_PTR));
        if(ga_center->best==NULL)ERROR("gaSetup(): alloc best 2 fails, exit");
        for(i=0;i<=ga_center->max_iter;i++)
        {   ga_center->best[i]=chromAlloc(ga_center->chrom_dim);
            if(ga_center->best[i]==NULL)ERROR("gaSetup:alloc best 3 fails,exit");
        }
        ga_center->best[ga_center->max_iter+1]=NULL;
        /*null terminate best array of ga_center->max_iter+ 2 cells*/
    }
}
else /*ga_center has best array, so we need free old one realloc new one*/
{   i=0;
    while(ga_center->best[i]!=NULL){chromKill(ga_center->best[i]);ga_center->best[i]=NULL;i++; }
    free(ga_center->best);  ga_center->best=NULL; /*end of free old space, begain alloc new*/
    if(ga_center->max_iter<=0)/*no max_iter limit, so alloc only 2 cells*/
    {   ga_center->best=(CHROM_PTR*)calloc(2,sizeof(CHROM_PTR));
        if(ga_center->best==NULL)ERROR("gaSetup(): alloc best 4 fails, exit");
        ga_center->best[0]=(CHROM_PTR)chromAlloc(ga_center->chrom_dim);
        if(ga_center->best[0]==NULL)ERROR("gaSetup: alloc best 5 fails, exit");
        ga_center->best[1]=NULL;/*null terminate best array of 2 cells*/
    }/*late refer best matrix by ga_center->best[0]->gene*/
    else/*has max_iter limit, so alloc max_iter+2 cells for best array*/
    {   ga_center->best=(CHROM_PTR*)calloc(ga_center->max_iter+2,sizeof(CHROM_PTR));
        if(ga_center->best==NULL)ERROR("gaSetup(): alloc best 2 fails, exit");
        for(i=0;i<=ga_center->max_iter;i++)
        {   ga_center->best[i]=chromAlloc(ga_center->chrom_dim);
            if(ga_center->best[i]==NULL)ERROR("gaSetup:alloc best 3 fails,exit");
        }
        ga_center->best[ga_center->max_iter+1]=NULL;
        /*null terminate best array of ga_center->max_iter+ 2 cells*/
    }
}/*alloc best array as max_iter+2 cells for 0th cell used by initial pool
the rest max_iter cells for real iterations, and last cell as null
terminator*/

/*===save the best member of 0th iter pool into oth cell of best array==*/
chromCopy(old_pool->chrom[old_pool->best_index], ga_center->best[0]);
ga_center->best[0]->index=0;/*the best of 0th generation*/
/*===no mutations performed yet==*/
ga_center->num_mut=0;  ga_center->tot_mut=0;

/*===set initial pool as 0th generation===*/
ga_center->iter=0;/*initial pool as 0th generation. ga iter from 1 in ga*/
/*rpReport(ga_center,old_pool);*//*move to ga funct after gaSetup()*/

/*===allocate space for public child chromosomes===*/
child1=chromAlloc(ga_center->chrom_dim);
child2=chromAlloc(ga_center->chrom_dim);
}
/*----------------------------------------------------------------------------------------------------------
```

```
| gaIterSetup---setup and make sure everything is ok before next iter by zeroing new pool size and
num_mut, and handling se_elitist and gap. we have two routines to handling select elitism: gaElitist1():
put 2 copies of best of old to new pool when se_elitist=1, gaElitist2(): put ELITIST percent of top bests to
new pool when se_elitist=2. gaIterSetup() is only used by traditional ga and generational ga, it can not be
used by steady_state ga due to one pool.
-----------------------------------------------------------------------------------------------------------*/
void gaIterSetup(GA_CENTER_PTR ga_center)
{ /*==error check==*/
 if(!gacValid(ga_center))ERROR("gaIterSetup():gacValid check fails,exit");

 /*5-13-96
 /*==zero new pool size for next use==*/
 /*ga_center->new_pool->size=0;*/
 /*==zero number of mutation for next use==*/
 /*ga_center->num_mut=0;
 *//*not just size and num_mut,also need reset other pool parameters*/

 /*==reset new pool for next iteration==*/
 poolReset(ga_center->new_pool);
 /*==handling se_elitist before selection of this iter from old pool==*/
 if(ga_center->se_elitist==1) gaElitist1(ga_center);/*transfer two copies of the best*/
 else if(ga_center->se_elitist==2) gaElitist2(ga_center);/*transfer percentage of the top bests*/
 /*==handling generation gap before selection of this iter from old pool*/
 if(ga_center->gap) gaGap(ga_center);
}
/*----------------------------------------------------------------------------------------------------------
| gaElitist1()---handle ga select elitist before selection of that iter, it would be called after current pool
swaped to the old and begain the next iter, it is called by gaIterSetup(), it puts 2 copies of the best of old
pool to new pool when se_elitist=1
-----------------------------------------------------------------------------------------------------------*/
void gaElitist1(GA_CENTER_PTR ga_center)
{ /*==error check==*/
 if(!gacValid(ga_center))ERROR("gaElitist1(): gacValid check fails,exit");
 /*==no elitist, return immidiately to caller==*/
 if(ga_center->se_elitist!=1) return;
 /*==se_elitist=1, put two copies of best chromos of old into new pool==*/
 else
 { poolAppend(ga_center->new_pool, ga_center->old_pool->chrom[ga_center->old_pool->best_index],
              TRUE);
   poolAppend(ga_center->new_pool,ga_center->old_pool->chrom[ga_center->old_pool->best_index],
              TRUE);
   return;
 }
}
/*----------------------------------------------------------------------------------------------------------
| gaElitist2()--handle ga select elitist before selection of that iter it would be called after current pool
swaped to the old and begain the next iter, it is called by gaIterSetup(), it put ELITIST percent number of
the top bests of old to new pool when se_elitist=2.
-----------------------------------------------------------------------------------------------------------*/
void gaElitist2(GA_CENTER_PTR ga_center)
{ int i,k, num,size;
 int* best; /* address of array to record the best chrom index seqence*/
 POOL_PTR old_pool,new_pool;
 /*==error check==*/
```

```
  if(!gacValid(ga_center))ERROR("gaElitist2(): gacValid check fails,exit");
  /*==no elitist, return immediately to caller==*/
  if(ga_center->se_elitist!=2) return;
  /*==initializing working variables==*/
  i=0;k=0;num=0;
  size=ga_center->old_pool->size;
  old_pool=ga_center->old_pool;
  new_pool=ga_center->new_pool;
  /*---dynamic alloc space for best array--*/
  if((best=(int*)calloc(size,sizeof(int)))==NULL)ERROR("gaElitist2(): best array alloc fail, exit");
  /*==se_elitist=2,put ELITIST percent number of top bests of old into new pool*/
  /*---sort array best as a sequence from best to weakest--*/
  for(i=0;i<size;i++)
  { if(ga_center->old_pool->chrom[i]->index!=i)
      ERROR("gaElitist2(): old pool index check fail; exit");
    best[i]=i;
  }
  /*check every cell, moving that cell content, index, toward left as long as its corresponding chromosome
is better. I have a test file bubsort.c  to test the correct of this for loop*/
  for(i=1;i<size;i++)
  { k=i;
    while(k>0 && chromComp(ga_center,old_pool->chrom[best[k]], old_pool->chrom[best[k-1]])<0)
    {  SWAP(&best[k],&best[k-1]); k--;
    }
  }/*now best array stores the index list corresponding chroms from best to  weakest.*/
  /*--compute how many best we need to transfer from old to new */
  num=(int)((float)ga_center->elitist*size);
  if(num<=0)
  { free(best); WARN("gaElitist2(): elitist number <= 0, abort gaElitist2(), return!"); return;
  }
  /*--make sure num is even--*/
  if(num%2!=0) { num++; }
  /*--append  ELITIST percent of top bests of old pool to new pool--*/
  for(i=0;i<num && new_pool->size<old_pool->size;i++)
     poolAppend(new_pool,old_pool->chrom[best[i]],TRUE);
  free(best);
}
/*-----------------------------------------------------------------------------------------------------------
| gaGap()-------handle ga generation gap before selection of that iter, it would be called after current pool
swaped to the old and begain the next iter, it is called by gaIterSetup().
-----------------------------------------------------------------------------------------------------------*/
void gaGap(GA_CENTER_PTR ga_center)
{ int i, num;
  CHROM_PTR parent;
  /*==error check==*/
  if(!gacValid(ga_center))ERROR("gaGap(): gacValid check fails,exit");
  /*==disabled gap, return immidiately to caller==*/
  /*ga_center->gap=0.0: traditional ga, gap select is identical to select*/
  /*ga_center->gap=1.0: steady_state ga, keep only one pool,no need of gap*/
  if(ga_center->gap==0.0 || ga_center->gap==1.0){ return;}
  /*==handling gap, put random selected required number of chromosomes
   ===of old pool into new pool=====================================*/
  else if(ga_center->gap>0.0 && ga_center->gap<1.0)
  {   /*--how many to copy over from old to new pool--*/
```

```
    num=(int)(ga_center->gap * ga_center->old_pool->size);
    if(num<=0){ WARN("gaGap(): gap chroms number <= 0, abort gaGap(), return!");return;}
    /*--make sure num is even--*/
    if(num%2!=0) {num++;}
    for(i=0;i<num&&ga_center->new_pool->size<ga_center->old_pool->size;i++)
    {   /*--random select a chromosome from old pool--*/
        parent=seRunPool(ga_center,ga_center->old_pool);
        /*--put it into new pool--*/
        poolAppend(ga_center->new_pool,parent,TRUE);
    }
    return;
  }
  else /*ga_center->gap<0.0 or ga_center->gap>1.0*/
  {   WARN("gaGap: gap<0.0 or gap>1.0, invalid value, return to caller."); return;}
}
/*-----------------------------------------------------------------------------------------------------------
| ga_SE_X_MU_RE_2CHILD()-basic operation of ga iteration, that is, select two parents from old pool,
crossover and mutation immediately to generate two children, then replace into new_pool according to
re_elitist flag. It is called when mu_flag: "MU_CHILD" by generational and Steady-State ga within each
iteration.
-----------------------------------------------------------------------------------------------------------*/
void ga_SE_X_MU_RE_2CHILD(GA_CENTER_PTR ga_center)
{ CHROM_PTR parent1,parent2;
  /*==error check==*/
  if(!gacValid(ga_center)) ERROR("ga_SE_X_MU_RE_2CHILD(): gacValid() check fails, exit");
  /*==randomly select two parents from old pool==*/
  parent1=seRunPool(ga_center,ga_center->old_pool);
  parent2=seRunPool(ga_center,ga_center->old_pool);
  /*==verify parents==*/
  chromVerify(ga_center,parent1);
  chromVerify(ga_center,parent2);
  /*==crossover by randomly selecting the x position ==*/
  if(RANDFRAC()<=ga_center->sac)/*invoke SAGA's SAC */
       sacRunPair(ga_center,parent1,parent2,child1,child2);
  else  xRunPair(ga_center,parent1,parent2,child1,child2);
  /*==mutation two children==*/
  if(RANDFRAC()<=ga_center->sam)/*invoke SAGA's SAM */
  { samRunSingleChrom(ga_center,child1);
    samRunSingleChrom(ga_center,child2);
  }
  else
  { muRunSingleChrom(ga_center,child1);
    muRunSingleChrom(ga_center,child2);
  }
  /*==evaluate children==*/
  if(child1->eva==0) ga_center->EV_fun(child1);/*we are sure child->eva=0 by x, mu ahead*/
  if(child2->eva==0) ga_center->EV_fun(child2);
  /*==verify children==*/
  chromVerify(ga_center,child1);
  chromVerify(ga_center,child2);
  /*==replace, that is append to new pool for generational ga, any replace
        op except append op for steady_state ga, according to re_elitist flag==*/
  reRunPair(ga_center,ga_center->new_pool,parent1,parent2,child1,child2);
 }
```

```
/*-----------------------------------------------------------------------------------------------------------
| ga_SE_X_RE_2CHILD()---basic operation of ga iteration, that is, select two parents from old pool,
crossover but not mutation immediately to generate two children, then replace into new_pool according to
re_elitist flag. It is called when mu_flag: "MU_BIT", or "MU_CHROM" by generational and Steady-State
ga within iteration.
-----------------------------------------------------------------------------------------------------------*/
void ga_SE_X_RE_2CHILD(GA_CENTER_PTR ga_center)
{ CHROM_PTR parent1,parent2;
 /*==error check==*/
 if(!gacValid(ga_center))ERROR("ga_SE_X_RE_2CHILD(): gacValid() check fails, exit");
 /*==randomly select two parents from old pool==*/
 parent1=seRunPool(ga_center,ga_center->old_pool);
 parent2=seRunPool(ga_center,ga_center->old_pool);
 /*==verify parents==*/
 chromVerify(ga_center,parent1);
 chromVerify(ga_center,parent2);
 /*==crossover by randomly selecting the x position ==*/
 if(RANDFRAC()<=ga_center->sac)/*invoke SAGA's SAC */
    sacRunPair(ga_center,parent1,parent2,child1,child2);
 else  xRunPair(ga_center,parent1,parent2,child1,child2);
 /*==but not mutation two children==*/
 /*muRunSingleChrom(ga_center,child1);
   muRunSingleChrom(ga_center,child2); */
 /*==evaluate children==*/
 if(child1->eva==0) ga_center->EV_fun(child1);/*we are sure child->eva=0 by x ahead*/
 if(child2->eva==0) ga_center->EV_fun(child2);
 /*==verify children==*/
 chromVerify(ga_center,child1);
 chromVerify(ga_center,child2);
 /*==replace, that is append to new pool for generational ga, any replace
      op except append op for steady_state ga, according to re_elitist flag==*/
 reRunPair(ga_center,ga_center->new_pool,parent1,parent2,child1,child2);
}
/*-----------------------------------------------------------------------------------------------------------
| gaStats()-----update the status data of ga, the following should be updated after each iteration by
gaStats(): best, converged, old_pool, new_pool; gaStats() first update stats of new pool then update ga
variable best, converged, old_pool, new_pool;
-----------------------------------------------------------------------------------------------------------*/
void gaStats(GA_CENTER_PTR ga_center)
{ POOL_PTR temp;
 /*==error check==*/
 if(!gacValid(ga_center))ERROR("gaStats(): gacValid check fails,exit");
 /*==call pool stats update routines to update new_pool==*/
 /*--Evaluate pool--*/
 /*sort new pool if se or re w/ rank*/
 if(!strcmp(seName(ga_center),"rank_biased")||!strcmp(reName(ga_center),"replace_rank"))
 { if(ga_center->new_pool->sorted!=2) poolRank(ga_center,ga_center->new_pool);
 }
 if(!ga_center->new_pool->sorted && !ga_center->new_pool->updated)
    poolAlign(ga_center->new_pool); /*not exe if poolRank() exe*/
 if(!ga_center->new_pool->updated)
 { poolIndex(ga_center->new_pool); /*make sure new pool index is correct*/
   poolFitness(ga_center, ga_center->new_pool);
   poolPtf(ga_center, ga_center->new_pool);
```

```
    poolStats(ga_center, ga_center->new_pool);
  }
  /*==save current best chromosome into history list of best array in ga*/
  if(ga_center->max_iter>0)/*there is a valid max_iter, so best is array*/
  { chromCopy(ga_center->new_pool->chrom[ga_center->new_pool->best_index],
                  ga_center->best[ga_center->iter]);
    ga_center->best[ga_center->iter]->index=ga_center->iter;/*best of iter*/
  }
  else/*ga_center->max_iter<=0 means best only has one cell*/
  { if(chromComp(ga_center, ga_center->best[0],
                              ga_center->new_pool->chrom[ga_center->new_pool->best_index])<=0)
    {/*ga_center->best[0] is better than or at least same as the best of new
        pool, so not need to replace the current best in ga with best in new*/
    }
    else/*ga_center->best[0] is worse than the best in new pool,so replace*/
    { chromCopy(ga_center->new_pool->chrom[ga_center->new_pool->best_index],ga_center->best[0]);
      ga_center->best[0]->index=ga_center->iter;/*best of iterth generati*/;
    }
  }
  /*== is new pool converged ?===*/
  /*---check variance and deviation to see if ga converged--*/
  if((ga_center->new_pool->dev<=ga_center->critia)&& (ga_center->new_pool->dev>=0.0))
      ga_center->converged=TRUE;
  else ga_center->converged=FALSE; /*ga data had better update by gaStats*/
  /*== swap old and new pool for the next iteration==*/
  temp=ga_center->old_pool;
  ga_center->old_pool=ga_center->new_pool;
  ga_center->new_pool=temp;
}
/*=========================================================================
| SAGA utility
=========================================================================*/
/*-------------------------------------------------------------------------------------------------------------------
| accept()----SAGA accepting function used by SAGA's SAC and SAM to determine if accept a new child
or not. (1/ga_center->iter) is the scheduled annealing temperature when SAGA has not max. iteration
limit, (max_iter - iter) is the scheduled annealing temperature when SAGA has the max.iteration limit .
-------------------------------------------------------------------------------------------------------------------*/
float accept(GA_CENTER_PTR ga_center,CHROM_PTR parent,CHROM_PTR child)
{ float prob=0.0; float diff=0.0;
  float t; /*temperature*/
  /*==error check==*/
  if(!gacValid(ga_center))ERROR("accept(): gacValid() check fails, exit");
/*if(!chromValid(parent))ERROR("accept():chromValid(p) check fails,exit");
  if(!chromValid(child))ERROR("accept():chromValid(c) check fails,exit");
*/
  /*==verify parent, child==*/
  chromVerify(ga_center,parent);
  chromVerify(ga_center,child);
  /*==make sure evaluated==*/
  if(parent->eva==0) ga_center->EV_fun(parent);/*we are sure parent->eva=1 by ?*/
  if(child->eva==0) ga_center->EV_fun(child);
  /*==calculate tempurature==*//*ga_center->iter>=1*/
  if(ga_center->max_iter>0)/*there is max iter limit*/
     t=ga_center->max_iter-ga_center->iter+CRITIA;/*critia make sure t!=0*/
```

```
 else  t=(float)1/ga_center->iter;
 /*==calculate prob==*/
 if(ga_center->minimize)
 { if(child->fitness<=parent->fitness) prob=1.0;
   else { diff=(float)(child->fitness-parent->fitness); diff=(float)diff/t;  prob=exp(-diff); }
 }
 else
 { if(child->fitness>=parent->fitness) prob=1.0;
   else { diff=(float)(parent->fitness-child->fitness); diff=(float)diff/t;  prob=exp(-diff); }
 }
 return prob;
}




/*================= end of file: ga.c =================*/
```

file: main.c

```
/*=================================================================
| Genetic Algorithm driver engine
| main()  --driver
| eva_cfg()--read evaluation configure data into array and matrix.
| EV_fun() --evaluation function of chromosome
==================================================================*/
#include "gah.h"
/*=================================================================
| Data Structure used by evaluation function
==================================================================*/
/*--dynamic alloc in main() and initialized by eva_cfg() read eva.cfg--*/
static float* FC;       /*address of array of fixed cost of every terminal*/
static float* TC;       /*address of array of transfer cost of every terminal*/
static float* temp;     /*address of temp array to held the immidiate computing values*/
static int** FM;        /*address of flow matrix.  fij: number of cars need to send from i to j */
static float** LTC;     /*address of least transfer cost matrix or shortest path length matrix;
                          aij: length of shortest path from i to j */
/*=================================================================
| function prototype
-----------------------------------------------------------------=*/
void eva_cfg(const char* );
int EV_fun(CHROM_PTR chrom);

extern int chromValid(CHROM_PTR );
extern GA_CENTER_PTR gaConfig(char* cfgfile, FN_PTR EV_fun);
extern void gaRun(GA_CENTER_PTR ga_center);
extern char* getNum(FILE* fid); /*helper of poolRead()*/
/*----------------------------------------------------------------------------------------------------------------
| main() - Genetic Algorithm driver engine
----------------------------------------------------------------------------------------------------------------*/
void main(void)
{ int i,dim;
  GA_CENTER_PTR ga_center;
  /*===setup and initialize genetic algorithm===*/
  ga_center=gaConfig("e:\gaccfg.h",EV_fun); /*railroad.cfg may be null if use default settings*/
  /*===setup and initialize evaluation enviroment===*/
  /*---dynamic alloc evaluation data arrays and matrices--*/
  dim=ga_center->chrom_dim;
  FC=(float*)calloc(ga_center->chrom_dim, sizeof(float));
  if(FC==NULL)ERROR("main(): alloc FC array fail, exit");
  TC=(float*)calloc(ga_center->chrom_dim, sizeof(float));
  if(TC==NULL)ERROR("main(): alloc TC array fail, exit");
  temp=(float*)calloc(ga_center->chrom_dim, sizeof(float));
  if(temp==NULL)ERROR("main(): alloc temp array fail, exit");
  FM=(int**)calloc(ga_center->chrom_dim, sizeof(int*));
  if(FM==NULL)ERROR("main(): alloc FM row array fail, exit");
  for(i=0;i<ga_center->chrom_dim;i++)
  { FM[i]=(int*)calloc(ga_center->chrom_dim, sizeof(int));
    if(FM[i]==NULL)ERROR("main(): alloc FM ith row array fail, exit");
  }
  LTC=(float**)calloc(ga_center->chrom_dim, sizeof(float*));
  if(LTC==NULL)ERROR("main(): alloc LTC row array fail, exit");
  for(i=0;i<ga_center->chrom_dim;i++)
```

```
 { LTC[i]=(float*)calloc(ga_center->chrom_dim, sizeof(float));
   if(LTC[i]==NULL)ERROR("main(): alloc LTC ith row array fail, exit");
 }
 /*---initialize evaluation data arrays and matrices--*/
 eva_cfg("e:\eva.cfg");
 /*===run the ga===*/
 gaRun(ga_center);/*ga_center should be free before exit gaRun()*/
 /*===free alloc space==*/
 free(FC);
 free(TC);
 free(temp);
 for(i=0;i<dim;i++)
 { free(FM[i]); FM[i]=NULL;
   free(LTC[i]); LTC[i]=NULL;
 }
 free(FM);
 free(LTC);
}
/*------------------------------------------------------------------------------------------------------
| eva_cfg()--initialize evaluation data arrays matrices by read eva.cfg call getNum(), a helper of
poolRead()
------------------------------------------------------------------------------------------------------*/
void eva_cfg(const char* cfg )
{ int   i,j,dim;
  char* sptr;
  int cell;
  FILE* fid;
  /*--error check--*/
  if(strlen(cfg)==0)ERROR("eva_cfg(): cfg file is invalid, exit");
  /*==open cfg file for read==*/
  fid=fopen(cfg,"r");
  if(fid==NULL)ERROR("eva_cfg(): open input cfg file err, exit");
  /*==read in the number of node of network==*/
  sptr=getNum(fid);
  if(sptr==NULL||sscanf(sptr,"%d",&dim)!=1) ERROR("eva_cfg(): read 1st int as dim fail, exit");
  /*==read in FC array ( fixed cost array) of dim cells==*/
  for(i=0;i<dim;i++)
  { sptr=getNum(fid);
    if(sptr==NULL||sscanf(sptr,"%f",&FC[i])!=1)
       ERROR("eva_cfg(): read in FC array meet err before finish, exit");
  }
  /*==read in TC array ( transfer cost array) of dim cells==*/
  for(i=0;i<dim;i++)
  { sptr=getNum(fid);
    if(sptr==NULL||sscanf(sptr,"%f",&TC[i])!=1)
       ERROR("eva_cfg(): read in TC array meet err before finish, exit");
  }
  /*==read in FM matrix ( flow matrix ) of dim*dim cells==*/
  for(i=0;i<dim;i++)
  { for(j=0;j<dim;j++)
    { sptr=getNum(fid);
      if(sptr==NULL||sscanf(sptr,"%d",&cell)!=1)
         ERROR("eva_cfg(): premature  reading of FM, exit ");
      /*---valid FM cell value is stored ---*/
      FM[i][j]=cell;
    }
  }
}
/*--------------------------------------------------------------------------------------------------------------
| EV_fun() --evaluation function of chromosome update chrom->fitness, set chrom->eva=1;
--------------------------------------------------------------------------------------------------------------*/
int EV_fun(CHROM_PTR chrom)
{ int i,j,k;
  float least, fix;
  /*==error check==*/
  if(!chromValid(chrom))ERROR("EV_fun(): chromValid() check fails, exit");
  /*==check chrom eva flag==*/
  if(chrom->eva==1)
  { WARN("EV_fun(): the chrom has fitness, no need call EV again, return"); return OK;
  }
  /*==reset LTC matrix for computation of this chromosome==*/
  for(i=0;i<chrom->dim;i++){ for(j=0;j<chrom->dim;j++) LTC[i][j]=0;}
  /*==computing the shortest path length from i to j for every pair i<j==*/
  for(j=1;j<chrom->dim;j++)
  {   for(i=j-1;i>=0;i--)
      {  if(chrom->gene[i][j]==1) LTC[i][j]=TC[j];
         else
         {   for(k=i+1;k<=j-1;k++)  temp[k]=LTC[i][k]+LTC[k][j];
             least=temp[i+1];
             for(k=i+1;k<=j-1;k++){ if(least>temp[k])least=temp[k];}
             LTC[i][j]=least;
         }
      }
  }
  /*==computing the least transfer cost matrix by multiply corresponding
        elements between FM matrix and current LTC matrix==*/
  for(i=0;i<chrom->dim;i++)
     for(j=i+1;j<chrom->dim;j++)
         LTC[i][j]=(float)(LTC[i][j]*(float)FM[i][j]);
  /*==computing the total least transfer cost of this chromosome==*/
  least=0.0;
  for(i=0;i<chrom->dim;i++) for(j=i+1;j<chrom->dim;j++) least +=LTC[i][j];
 /*==computing the total fixed cost of this chromosome==*/
  fix=0.0;
  for(i=0;i<chrom->dim;i++)
  {   for(j=i+1;j<chrom->dim;j++){ if(chrom->gene[i][j]==1) fix += FC[i];}
  }/*it does not count the fix cost of last terminal*/
  /*==update chromosome fitness==*/
  chrom->fitness=fix+least;
  /*==optional computing fitness method, decrease the fix cost of least
        feasible set and the sure transfer cost from fitness, these two
        values are independent from the chrom matrix, they are depended on
        evaluation parameter configuration ================================*/
  /*--the fix cost of the least feasible set--*/
  fix=0.0;
  for(i=0;i<=chrom->dim-2;i++) fix += FC[i];
  /*--the sure transfer cost depends on FM (flow matrix)--*/
  least=0.0;
 for(i=0;i<chrom->dim;i++) for(j=i+1;j<chrom->dim;j++) least +=(float)((float)FM[i][j]*TC[j]);
 /*--deduction from fitness--*/
 chrom->fitness -= fix;
 chrom->fitness -= least;
 /*--set chromosome eva flag TRUE, disable repeat call EV_fun()*/
 chrom->eva=1;
 /*this is the only place to set eva=1; chromReset(), chromRepair(), any
   xover ops and mutation ops all set eva=0; */
 return OK;
}




/*=========== end of file: main.c ======*/
```

file: gaccfg.h

```
#===========================================================================
# Genetic Algorithm ga_center parameters configuration file
#===========================================================================
#-------------------------------------------------------------------------
# Seed for random number generator
#
# Usage: rand_seed my_pid
#        rand_seed number
#
#    my_pid = use system pid as random seed, in unix rand_seed=pid();
#    number = seed for random number generator, a positive integer
#
# DEFAULT: rand_seed 1
#-------------------------------------------------------------------------
# rand_seed my_pid
# rand_seed 1

#-------------------------------------------------------------------------
# How to initialize the pool
#
# Usage: initpool [random | from_file filename | interactive]
#
#    random      = generate at random based on
#                  chrom_dim, and pool_max_size
#    from_file   = read from a file
#      filename = the name of the file to read from
#    interactive = read from stdin
#
# DEFAULT: initpool random
#-------------------------------------------------------------------------
# initpool random
# initpool from_file   /z/wuh/ga/test/initpool.dat
# initpool interactive

#-------------------------------------------------------------------------
# Pool_max_size: this is only place to configure the maximum allowable
#                pool size, the actural pool size is depend on how many
#                chrom matrices are readed into pool when initializing
#                pool.
#                NOTES:
#                1)pool_max_size at least 4.
#                2)actural pool size at most pool_max_size-2.
#
# Usage: pool_max_size size
#
#    size = pool max allowed size, a positive integer, must be at least 4
#
# DEFAULT: pool_max_size 100
#-------------------------------------------------------------------------
# pool_max_size 200

#-------------------------------------------------------------------------
# Chromosome dim: this is only place to configuration chrom dim
```

```
#
# Usage: chrom_dim length
#
#   length = chromosome matrix dimension, a positive integer
#
# DEFAULT: chrom_dim 10
#------------------------------------------------------------------------
# chrom_dim 20

#------------------------------------------------------------------------
# Objective of GA:
#
# Usage: objective [minimize | maximize]
#
#   minimize = minimize evaluation function
#   maximize = maximize evaluation function
#
# DEFAULT: objective minimize
#------------------------------------------------------------------------
# objective minimize
# objective maximize

#------------------------------------------------------------------------
# When to stop the GA
#
#   Convergence means when the variance = 0, or equivalently, when
#   all the fitness values in the pool are identical, or variance is
#   within the limit of critia if provided.
#
#   Iterations means the number of generations for the ga envolving.
#   Numbers must be given as positive integers.
#
# Usage: stop_after convergence
#      stop_after number [use_convergence | ignore_convergence]
#
#   convergence          - stop when the GA converges
#   number               - stop after specified number of iterations
#     use_convergence    - will stop early if GA converges (default)
#     ignore_convergence - WILL NOT stop early even if GA converges
#
# DEFAULT: stop_after convergence
#------------------------------------------------------------------------
# stop_after convergence
# stop_after 500
# stop_after 220 use_convergence
# stop_after 500 ignore_convergence

#------------------------------------------------------------------------
#  Selection Elitism
#
#  Selection Elitism, depending on cfg, has two kind of actions when
#  initial new pool at each iteration of Traditional GA and Generational GA.
#  when se_elitist = 0, disable selection elitism handling of gaIterSetup();
#  when se_elitist = 1, makes two copies of the best performer in the old
#                  pool and places them in the new pool.
#  when se_elitist = 2, transfer ELITIST percent of top best in the old
#                  pool to the new pool.
#  thus ensuring the most fit chromosome survives.
#  (used by Traditional GA, Generational GA, not used by Steady_state GA)
#
# Usage: se_elitist [0 | 1 | 2]
#
# DEFAULT: se_elitist 2
#------------------------------------------------------------------------
# se_elitist 0
# se_elitist 1
# se_elitist 2

#------------------------------------------------------------------------
# ELITIST
#
#   ELITIST specifies how many percentage of the top bests of old pool to
#   transfer to new pool when se_elitist=2 at initializing new pool stage
#   of each iteration of Traditional GA and Generatioal GA.
#   (used by Traditional GA, Generational GA, not used by Steady_state GA)
#
# Usage: elitist  float number
#
#              float number= any foat number belongs [0.0, 1.0)
#
# DEFAULT: elitist 0.10
#------------------------------------------------------------------------
# elitist 0.10        # used by gaElitist2() when se_elitist = 2

#------------------------------------------------------------------------
#  Replacement Elitism
#
#  Replacement Elitism, if flag is on, has two kind of actions:
#  first, pick two best among two parents and two children as new children.
#  second, replace can be performed if target is worse.
#  when re_elitist = 0, disable replacement elitism handling inside reRun();
#  when re_elitist = 1, enable replacement elitism handling inside reRun();
#  NOTES:
#  can be used only when pure Traditional,Generational,and Steady_state GA,
#  if we use a variety modes of SAGA, that is, either sam>0 or sac>0,
#  we should disable replacement elitism, otherwise SAM, or SAC operators
#  have no any effect on the performance of SAGA.
#
# Usage: re_elitist [0 | 1 ]
#
# DEFAULT: re_elitist 0
#------------------------------------------------------------------------
# re_elitist 0     # we must choose re_elitist=0 if we want use SAM, or SAC
# re_elitist 1     # we can choose re_elitist=1 only when pure GA modes.

#------------------------------------------------------------------------
# Generation gap:
#
```

```
# The generation gap  represents a percentage of the population to copy
# (clone) to the new pool at initializing new pool stage of each iteration
# of Traditioanal GA an Generational GA. This only makes sense in a GA
# with two pools as in the traditional and generational model.
# NOTES:
# 1)gap plays no effect in traditional GA because gaGap() become a part of
# selection operation to select reproduce pool when traditional GA.
# 2)A gap of 0.0 is the traditional generational algorithm when generation
# GA. As the gap increases, it becomes more like a steady-state algorithm.
# 3)A gap of 1.0 essentially disables crossover since only reproduction
# occurs.
#
# Usage: gap number
#
#   number = generation gap, valid range = [0.0, 1.0]
#
# DEFAULT: gap 0.0
#------------------------------------------------------------------------
# gap 0.1

#------------------------------------------------------------------------
# CRITIA
#
# CRITIA is a float point number to specify the limit of variance of pool
# to issue the converge flag of pool.
#
# Usage: critia  float number
#
#            float number= any foat number belongs [0.0, 1.0)
#
# DEFAULT: critia 0.00000000001
#------------------------------------------------------------------------
# critia 0.00000000001        # used to issue converge flag of pool

#------------------------------------------------------------------------
# GA Type:
#
# Usage: ga [traditional | generational | steady_state]
#
#   traditional  = traditional  GA
#   generational = generational GA
#   steady_state = steady-state GA
#
# WARNING: This directive has the following side effects:
#
#      GA type        Directives set as a side effect
#    -----------    ------------------------------
#    traditional      selection       any
#                     crossover       any
#                     mutation        any
#                     replacement     any except append
#
#    generational     selection       any
#                     crossover       any
#                   mutation        any
#                   replacement     only append
#
#     steady-state  selection       any
#                   crossover       any
#                   mutation        any
#                   replacement     any except append
#
# DEFAULT: ga generational
#-----------------------------------------------------------------------
# ga traditional        #must not use replace by append
# ga generational        #must use replace by append
# ga steady_state        #must not use replace by append

#-----------------------------------------------------------------------
# Selection method:
#
# Usage: selection [fitness_biased | rank_biased | uniform_random]
#
#   fitness_biased = select according to ptf by Roulette wheel.
#   rank_biased    = select according to rank_prob by Roulette wheel.
#   uniform_random = Pick one at random
#
# DEFAULT: selection fitness_biased
#-----------------------------------------------------------------------
# selection fitness_biased
# selection rank_biased
# selection uniform_random

#-----------------------------------------------------------------------
# Rank_biased selection pressure
#
# Usage: bias number
#
#   number = rank_biased selection pressure, valid range = [0.0 .. 1.0],
#          used to compute rank_prob of each chrom, which are only used
#          for rank_biased selection
#
# DEFAULT: bias 0.8
#-----------------------------------------------------------------------
# bias 0.8

#-----------------------------------------------------------------------
# Crossover method:
#
# Usage: crossover [random_flip|asexual|1xp_crossover|2xp_crossover
#                           3xp_crossover|4xp_crossover]
#
#   random_flip   = allele assign to alternative child randomly
#   asexual       = swap two blocks w/1xp of one parent to born one kid
#   1xp_crossover = children get alternate "halves" of parents
#   2xp_crossover = children get alternate "blocks w/2 xps" of parents
#   3xp_crossover = children get alternate "blocks w/3 xps" of parents
#   4xp_crossover = children get alternate "blocks w/4 xps" of parents
```

```
#
# DEFAULT: crossover 1xp_crossover
#--------------------------------------------------------------------------
# crossover random_flip
# crossover asexual
# crossover 1xp_crossover
# crossover 2xp_crossover
# crossover 3xp_crossover
# crossover 4xp_crossover

#--------------------------------------------------------------------------
# Crossover Rate
#
# Usage: x_rate number
#
#   number = crossover rate (percentage), valid range = [0.0 .. 1.0]
#          A crossover rate of 0.0 disables crossover
#
# DEFAULT: x_rate 1.0
#--------------------------------------------------------------------------
# x_rate 0.8

#--------------------------------------------------------------------------
# Mutation method:
#
# Usage: mutation [simple_flip | simple_random | swap]
#
#   simple_flip   = flip a bit
#   simple_random = random bit value
#   swap          = swap two rows
#
# DEFAULT: mutation swap
#--------------------------------------------------------------------------
# mutation simple_flip
# mutation simple_random
# mutation swap

#--------------------------------------------------------------------------
# Mutation Rate
#
# Usage: mu_rate number
#
#   number = mutation rate (percentage), valid range = [0.0 .. 1.0]
#          A mutation rate of 0.0 disables mutation
#
# DEFAULT: mu_rate 0.0
#--------------------------------------------------------------------------
# mu_rate 0.5

#--------------------------------------------------------------------------
# Mutation flag  tell the way how to use mutation operator
#
# Usage: mu_flag  [ MU_CHROM | MU_BIT | MU_CHILD]
#
```

```
#     MU_CHROM = call mutation op to mutate a bit of the chrom which
#            is pickup from chromosome level of whole pool according
#            to mu_rate.
#     MU_BIT   = mutate a bit which is pickup from bit level of whole
#            pool according to mu_rate.
#     MU_CHILD = call mutation op to mutate a bit of the chrom immidiately
#            after it is born from crossover, also check the prob of
#            it against mu_rate to see if we can mutate it
#
#
# DEFAULT: mu_flag MU_BIT
#--------------------------------------------------------------------------
# mu_flag MU_CHROM
# mu_flag MU_BIT
# mu_flag MU_CHILD

#--------------------------------------------------------------------------
# Replacement method:
#
# Usage: replacement [append|replace_parent|replace_rank|replace_random|
#                replace_weaker|replace_weakest]
#
#   append          = append to new pool, as in generational GA
#   replace_parent  = replace parents in pool with children
#   replace_rank    = insert in sorted order, as in Genitor
#   replace_random  = replace children into pool by random pick victims
#   replace_weaker  = replace first weaker found in linear scan of pool
#   replace_weakest = replace weakest member of the pool
#
# DEFAULT: replacement append
#--------------------------------------------------------------------------
# replacement append           # used only by generational GA
# replacement replace_parent   # can't be used by generational GA
# replacement replace_rank     # can't be used by generational GA
# replacement replace_random   # can't be used by generational GA
# replacement replace_weaker   # can't be used by generational GA
# replacement replace_weakest  # can't be used by generational GA

#--------------------------------------------------------------------------
# SAM
#
# sam is a float point number to specify the probability to invoke the
# integrated simulated annealing mutation operator (SAM) to replace pure
# GA mutation operator inside mutation interface muRun(). we check
# random probability against sam value to determine if invoke SAM.
# sam=0.0: disable SAM, use standard GA mutation operator.
# sam=1.0: use SAM, disable standard GA mutation operator.
# 0.0<sam<1.0: a variety of hybrid modes.
# NOTES:
# when sam>0, we must set re_elitist=0 (disable replace elitism)
#
# Usage: sam  float number
#
#          float number= any foat number belongs [0.0, 1.0]
```

```
#
# DEFAULT: sam 0.0
#---------------------------------------------------------------------------
# sam 0.0   #when sam>0, we must set re_elitist=0 (disable replace elitism)

#---------------------------------------------------------------------------
# SAC
#
#  sac is a float point number to specify the probability to invoke the
#  integrated simulated annealing crossover operator (SAC) to replace
#  pure GA crossover operator inside crossover interface xRun(), we check
#  random probability against sac value to determine if invoke SAC.
#  sac=0.0: disable SAC, use standard GA crossover operator.
#  sac=1.0: use SAC, disable standard GA crossover operator.
#  0.0<sac<1.0: a variety of hybrid modes.
#  NOTES:
#  when sac>0, we must set re_elitist=0 (disable replace elitism)
#
# Usage: sac  float number
#
#          float number= any foat number belongs [0.0, 1.0]
#
# DEFAULT: sac 0.0
#---------------------------------------------------------------------------
# sac 0.0   #when sac>0, we must set re_elitist=0 (disable replace elitism)

#---------------------------------------------------------------------------
# Report type
#
# Usage: rp_type [none | minimal | short | long]
#
#   none     = output nothing
#   minimal  = output configuration and each iter statistics result
#   short    = output minimal + each iter best chromosome matrix only
#   long     = output short + dump pool
#
# DEFAULT: rp_type short
#---------------------------------------------------------------------------
# rp_type none
# rp_type minimal
# rp_type short
# rp_type long

#---------------------------------------------------------------------------
# Report interval
#
# Usage: rp_interval number
#
#   number = interval between reports, a positive integer
#
# DEFAULT: rp_interval 1
#---------------------------------------------------------------------------
# rp_interval 10
```

```
#------------------------------------------------------------------------
# Output report filename
#
# Usage: rp_file file_name [file_mode]
#
#    file_name = name of report file
#    file_mode = optional file mode for fopen()
#       a      = append (DEFAULT)
#       w      = overwrite
#
# DEFAULT: (write to stdout)
#------------------------------------------------------------------------
# rp_file e:\ga.out
# rp_file ga.out a
# rp_file ga.out w




/*=========== end of file: gaccfg.h =====*/
```

VITA

Hongjiang Wu

Candidate for the Degree of

Master of Science

Thesis: A GENETIC ALGORITHM APPROACH FOR CONSTRUCTING RAILROAD OPERATING PLANS

Major Field: Computer Science

Biographical:

Personal Data: Born in Urumqi City, Xinjiang Province, P. R. of China, on September 17, 1960, the son of Ding-An Wu and Xiang-Ying Liang.

Education: Received Bachelor of Science degree in Mathematics from Wuhan University, Wuhan, China, in May, 1982; Diploma of Graduate College in Applied Mathematics from Zhengzhou University, Zhengzhou, China, in July, 1987; Master of Science degree in Mathematics from Oklahoma State University, Stillwater, Oklahoma, in December, 1993; Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in July, 1996.

Experience: Mathematics lecturer for many years.

Professional Memberships: American Mathematical Society, Chinese Mathematical Society.