A GENETIC ALGORITHM FOR SOLVING A NONLINEAR COMBINATORIAL OPTIMIZATION PROBLEM

By

YUNLONG CHEN

Bachelor of Science

Chengdu University of Science and Technology

Chengdu, People's Republic of China

1982

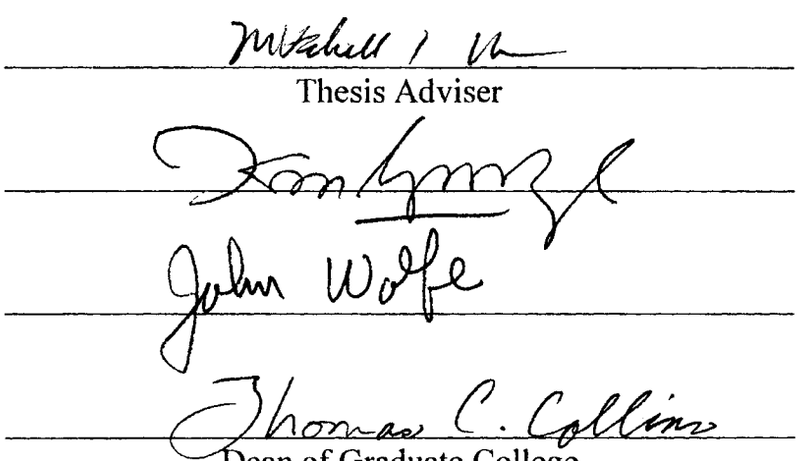
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in Partial Fulfillment of
the Requirements
the Degree of
MASTER OF SCIENCE
December, 1994

A GENETIC ALGORITHM FOR SOLVING

A NONLINEAR COMBINATORIAL

OPTIMIZATION PROBLEM

Thesis Approved:



Thesis Adviser

Dean of Graduate College

PREFACE

A practical combinatorial optimization problem of nonlinear discrete functions emerges in a personal computer software package, NPK$PLUS. An algorithm in NPK$PLUS involves so much time to find an optimal solution of the problem such that the efficiency of the software is reduced and the scope of application is limited. The problem is NP-completeness problem. In most cases, an optimal solution to the problem can not be obtained in a reasonable amount of time[Aarts 89].

Based on the characteristics of the problem, an algorithm composed of the ideas from nonlinear programming, genetic algorithms and simulated annealing algorithms is developed. The algorithm can be used to obtain an optimal or near optimal solution, which satisfies the combinatorial constraint, in a reasonably short amount of time. For the latter, two reference optimal solutions are given by the algorithm. The algorithm has been programmed in the C programming Language and has been tested on an IBM 386 compatible personal computer. Near optimal solutions are obtained within 5 seconds.

ACKNOWLEDGMENTS

I take this opportunity to express my most sincere appreciation to Dr. Mitchell Neilsen, my principal advisor, for his most kind suggestion of the problem, and his invaluable guidance during this investigation and preparation of this thesis.

I wish to express my sincere gratitude to Dr. K. M. George, and Dr. John Wolfe, the other members of my graduate committee, for their guidance and counseling. I also wish to express my thanks to Dr. David Nofziger for giving me opportunity to solve the problem. I wish to express my thanks to Dr. Wayne Powell for much help during my graduate study here, and I also would like to express my thanks to everyone who has taught me or given me help in this department.

The completion of this work on my part is in reality the result of effects made by literally hundreds of people who have been a part of my life, beginning with my grand parents Dr. William Wuxiang Chen and Mrs.Yuqing Liu Chen, my parents Mr. Boqiang Chen and Mrs. Zheren Lu Chen, my wife's parents Mr. Monggui Zhou and Mrs. Gourong Chang Zhou, and My Uncle Mingqiang Chen and Qingqiang Chen. I never forget Uncle Mr. Juntao Pi and Aunt Mrs. Guoyao Chang Pi, Brother Sherwin Jin and Sister Mrs. Aili Pi Jin, Aunt Zhiqiang Liu Mei and Uncle Mr. Zhuyan Mei. I feel this is one place that appreciation for them should be expressed. Without their constant guidance and support, it is impossible for me to have the success today.

Last, my appreciation goes to my wife Baiming Zhou for her many sacrifices, infinite patience and constant understanding. I also wish this thesis is a gift to my daughter, Shelly Chen.

TABLE OF CONTENTS

LIST OF TABLES

Table page

CHAPTER I

INTRODUCTION

1.1 A Real Problem

NPK$PLUS is a computer software package that provides decision making support for managing fertilizer use. It has three main options:

A. Interpret soil test results for a specific field, display fertilizer requirements for long range productivity and display the 10 best fertilizer use options for maximizing return on fertilizer investments for one year.

B. Estimate yield, cost and return for a specific fertilizer application;

C. Find the best fertilizer application rates for each field to maximize return subject to a limited fertilizer investment [Johnson 91].

For Option B, fifty fertilizer options are generated for each field. Each fertilizer option has its own cost rate, return rate and identification number.

For Option C, the software determines optimum fertilizer practices for a collection of fields and crops. Thus, if a farmer chooses to limit the total amount of money invested in fertilizer, NPK$PLUS will find the best way to spend the money to obtain the maximum return. The best combination of fertilizer options must be selected from all combinations of fertilizer options. A combination of fertilizer options is a fertilization plan for all fields, which is made by taking one fertilizer option per field. A fertilizer option with identification number 0, which has zero cost and zero return, specifies the case of no fertilizer for a field.

NPK$PLUS Version 4 solves the combinatorial optimization problem by traversing all combinations of fertilizing options of all field (one fertilizer option per field). If it works in the case that each field has 50 fertilizer options, the traversal algorithm will involve about 50^n comparisons for n fields. For this reason, NPK$PLUS Version 4 is designed to work for no more than 5 fields in practice. For Option C of NPK$PLUS Version 4, only the top 10 fertilizer options out of 50 fertilizer options generated in option B for a field are used in searching for an optimal solution.

1.2 The Combinatorial Optimization Problem

The combinatorial optimization problem can be described as follows:

Find Maximum Total Return;

Total Return = $r_1(x_1) + \ldots + r_n(x_n)$;

$r_i(x_i)$ = return/acre of option x_i × number of acres in field i,

x_i is an option number.

Bound Constraint:

Total Cost ≤ Capital;

Total Cost = $c_1(c_1) + \ldots + c_n(c_n)$;

$c_i(x_i)$ = cost/acre of option x_i × number of acres in field i,

x_i is an option number.

Combinatorial Constraint:

A whole field must be fertilized with only one fertilizer option.

Fertilizer options of a field are given in a fertilizer option table. So, if there are n fields, there will be n fertilizer option tables. In a fertilizer option table, there are several

items for a fertilizer option: option identification number, field identification number, unit cost, unit return and unit return rate.

1.3 A Traversal Algorithm

For Option C of NPK$PLUS Version 4, a traversal algorithm shown below, is used to find a solution to the above combinatoral optimization problem.

Algorithm 0 A Traversal Algorithm

Input: Capital (limited money), n field(s) ($0 < n < 6$), n fertilizer option tables (each fertilizer option table has the top 10 fertilizer options as shown below in Table 1.)

BEGIN

```
1. Total_Return=Total_Cost=temp_return=temp_cost=0;
   c1=c2=c3=c4=c5=0;
2. FOR (i1 = 1; i1 ≤ 10; i1 ++)
       FOR (i2 = 1; i2 ≤ 10; i2 ++)
           ...
           FOR (in = 1; in ≤ 10; in ++) {
              temp_cost =f1[i1].cost +f2[i2].cost+... +fn[in].cost;
              temp_return =f1[i1].return + f2[i2].return+... +fn[in].return;
              IF (( temp_cost ≤ Capital) && (Total_Return < temp_return )) {
                Total_Return = temp_return;
                Total_Cost = temp_cost;
                c1=i1;
                c2=i2;
                c3=i3;
```

```
            c4=i4;
            c5=i5;
        }
        ELSE IF ((temp_cost < Total_Cost) && (temp_return = Total_Return)) {
            Total_Return = temp_return;
            Total_Cost = temp_cost;
            c1=i1;
            c2=i2;
            c3=i3;
            c4=i4;
            c5=i5;
        }
    }
```

3. Output: Total_Return, Total_Cost, c1, c2, c3, c4, c5.

END

In the above algorithm, ci, $0 < i < 6$, is a fertilizer option number of the i_th field; fj[ij].cost is the cost of a fertilizer option of the j_th field, fj[ij].return is the return of a fertilizer option of the j_th field, where ij is a variable with respect to a fertilizer option id. There are 4×10^n comparisons, about 10×10^n arithmetic operations and 7×10^n other operations in the worst case for n fields. The computing complexity is about 20×10^n. If each field has 50 fertilizer options in consideration, the computing complexity will rise to 20×50^n.

In fact, many farmers have more than five fields. They need a software program to help them manage the fertilization. For satisfying their practical need, we find a new alogorithm to improve Option C of NPK$PLUS.

Table 1 Top 10 Fertilizer Options With Projected Yield And Return [Johnson 92]

Option #	Yield	N ---	P pound /acre	K ---	Lime ton /acre	Cost $	Return $	Return Rate %
1	39.8	50	20	20	0.0	17.60	31.50	179
2	39.4	50	20	10	0.0	16.50	31.12	189
3	38.9	50	10	20	0.0	15.40	30.74	200
4	38.4	50	10	10	0.0	14.30	30.40	213
5	38.7	40	20	20	0.0	15.40	30.30	197
6	38.3	40	20	10	0.0	14.30	30.03	210
7	40.0	60	20	20	0.0	19.80	29.77	150
8	37.8	40	10	20	0.0	13.20	29.76	225
9	37.4	40	10	10	0.0	12.10	29.52	244
10	39.5	60	20	10	0.0	18.70	29.37	157

The new algorithm combines the ideas of nonlinear programming, genetic algorithms and simulated annealing algorithms, to solve the combinatorial optimization problem. A program, written in C, has been developed. It returns approximately optimal solutions to the combinatorial optimization problem for up to 20 fields. If each field has less than 11 options, the solutions are found within 5 seconds on an IBM 386 compatible.

In Chapter 2, genetic algorithms, simulated annealing and nonlinear programming are introduced. In Chapter 3, the problem is analyzed. In Chapter 4, an outline of the algorithm is given. In Chapter 5, details of the new algorithm are given. In Chapter 6, some numerical results are given.

CHAPTER II

RELATIVE ALGORITHMS

2.1 Introduction

The following are some general concepts of combinatorial optimization problems given by Emile Aarts [Aarts 89].

Definition 1 A combinatorial optimization problem is either a *minimization problem* or a *maximization problem* and is specified by a set of problem instances.

Definition 2 An *instance* of a combinatorial optimization problem can be formalized as a pair (S, f), where the *solution space* S denotes the finite set of all possible solutions and the *cost function* f is a mapping defined as

$$f: S \rightarrow \boldsymbol{R} \tag{2.1}$$

In the case of minimization, the problem is to find a solution $i_{opt} \in S$ which satisfies

$$f(i_{opt}) \leq f(i) \text{ for all } i \in S. \tag{2.2}$$

In the case of maximization, i_{opt} satisfies

$$f(i_{opt}) \geq f(i) \text{ for all } i \in S. \tag{2.3}$$

Such a solution i_{opt} is called a *globally-optimal solution,* either *minimal* or *maximal,* or simply an *optimum,* either a *minimum* or a *maximum;* $f_{opt} = f(i_{opt})$ denotes the optimal cost, and S_{opt} the set of optimal solutions.

Definition 3 Let (S, f) be an instance of a combinatorial optimization problem. Then a *neighbourhood structure* is a mapping

$$N : S \rightarrow 2^S, \tag{2.4}$$

which defines for each solution $i \in S$ a set $S_i \subseteq S$ of solutions that are 'close' to i in some sense. The set S_i is called the *neighbourhood* of solution i, and each $j \in S_i$ is called a *neighbouring solution* or *neighbour* of i. Furthermore, we assume that $j \in S_i \Leftrightarrow i \in S_j$.

Definition 4 Let (S, f) be an instance of a combinatorial optimization problem and let N be a neighbourhood structure, then $i^* \in S$ is called a *locally optimal solution* or simply a *local optimum* with respect to N if i^* is better than, or equal to, all its neighbouring solutions with respect to their cost. More specifically, in the case of minimization, i^* is called a *local minimal solution* or simply a *local minimum* if

$$f(i^*) \leq f(j), \text{ for all } j \in S_i*, \tag{2.5}$$

an in the case of maximization, i^* is called a *locally maximal solution* or simply a *local maximum* if

$$f(i^*) \geq f(j), \text{ for all } j \in S_i*., \tag{2.6}$$

Definition 5 Let (S, f) be an instance of a combinatorial optimization problem and let N be a neighbourhood structure. Then N is called *exact* if, for each $i^* \in S$ that is locally optimal with respect to N, i^* is also globally optimal.

A class of combinatorial optimization problems is said to be in the class of NP-complete problems, if any algorithm to find an optimal solution to such a problem, requires a computational effort that grows superpolynomially with the size of the

problem. Because of the property of NP-completeness, optimal solutions can not be obtained in reasonable amounts of computation time[Aarts 89].

Over the past few decades, a wide variety of such problems emerged from such diverse areas as management science, computer science and engineering and many efforts have been devoted to developing methods for solving the problems. In consideration of reducing computation time, a class of algorithms to find approximate optimal solutions are considered, including genetic algorithms and simulated annealing algorithms.

2.2 Genetic Algorithm

Genetic algorithms (GA) were first introduced by John Holland in his 1975 book *Adaptation in Natural and Artifical Systems* as a parameter optimization method. Genetic algorithms imitate the principles of natural evolution in the real world. Since then, genetic algorithms have been used to solve a diverse range of engineering problems.

Over many years, biologists have identified the theory of natural selection which governs the evolution of the biological world. Natural selection works on organisms through its performance in producing offspring. Species create specific genetic material for offspring. The parent(s) influence the inheritable structure and function of the offspring through the specific genetic material. There are several ways to create the genetic material such as sexual recombination (crossover), mutation (random modification). Only in the case of mutation, the genetic endowment of offspring is different from that of the parent. An individual is evaluated through its reproductive performance (called fitness).

Genetic algorithms transform a population of individual objects into a new generation of the population using the theory of natural selection. That is, using the

principle of survival of the fittest and natural genetic operations such as crossover (sexual recombination) and mutation. Each individual in the population not only has a fitness value, but also is a possible solution to the given problem. The genetic algorithm keeps breeding the population of individuals to find an approximate optimal solution or a very good solution to the problem. Sometimes it may find the best solution.

There are four steps in a genetic algorithm [Soucek 92]:

1. Initialization - generate an initial population of the search nodes. Let $S_0 = \{ s_i \mid i = 1,..., n\}$ be the initial population and s_i be the i_th search node;

2. Fitness evaluation - calculate the fitness function value for each nodes. Let $f(s_i)$ be the fitness function value of s_i.

3. Genetic operation - generate new search nodes randomly by fitness value and genetic operator;

4. Repeat 2 and 3 until a stopping criterion is met.

In practice, there are four main tasks that must be done carefully for constructing a genetic algorithm [Soucek 92]:

1. Choice of population of search nodes and representation of parameters,

2. Determination of the fitness function,

3. Design of an offspring genetic operator,

4. Determination of the probabilities which control the genetic operator.

That is, to design a coding scheme to encode the parameters of the problem into a string. The coding scheme must be simple and be easily manipulated by a genetic operator. A fitness function must be relevant to the objective function of the problem such that a good search node (string) is one that will have a high fitness value.

There are some basic genetic operators: crossover, inversion and reorder. The crossover genetic operator allows production of new strings through a combination of

parts of strings. In the case of a crossover genetic operator, pairs of strings are selected randomly and part of a string to which swapping takes place is selected randomly, too. The following examples show the operations of a crossover genetic operator, an inverse genetic operator and a reordering genetic operator [Soucek 92].

Example 1 Consider two strings A1 and A2:

A1: 1 2 3 4 5 6 7 8 A2: 2 1 4 3 5 7 8 6

Using crossover, generates two new strings

A1': 1 2 3 4 | 5 7 8 6 A2': 2 1 4 3 | 5 6 7 8

Example 2 Consider two strings A1 and A2:

A1: 1 2 3 4 5 6 7 8 A2: 2 1 4 3 5 7 8 6

Using inversion in two parts, generates two new strings

A1': 4 3 2 1 | 8 7 6 5 A2': 3 4 1 2 | 6 8 7 5

Example 3 Consider two strings A1 and A2:

A1: 1 2 3 4 5 6 7 8 A2: 2 1 4 3 5 7 8 6

Using reordering, generates two new strings

A1': 1 2 | 6 7 8 | 3 4 5 A2': 2 1 | 7 8 6 | 4 3 5

2.3 Simulated Annealing Algorithm

The simulated annealing algorithm was introduced in combinatorial optimization by Kirkpatrick, Gelatt, Vecchi and Cerny in the early 1980's. The algorithm is based on a simulation of the physical annealing process of solids. It is an another approximation algorithm for solving large combinatorial optimization problems.

Annealing is a thermal process to reduce the energy state of a solid through a heat bath. There are two steps in the process [Barker & Henderson 1976; Kirkpatrick, Gelatt & Vecchi, 1982; 1983]:

1. Increase the temperature of the heat bath to a maximum value at which the solid melts.

2. Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

The arrangement of particles of the solid in the liquid phase are set randomly. In the ground state, the particles are set in a structured lattice and the energy of state of the solid is minimal.

Algorithm 1 (Simulated Annealing Algorithm) [Aarts 89]

Initialize($i_{start,}$, c_0, L_0,);

k=0;

i=i_{start};

while (c_k != stop criterion) {

for l:=1 to L_k; do {

generate (j from S_i);

if f(j) ≤ f(i) then i=j

elseif $\exp((f(i)-f(j))/c_k) > \text{random}[0,1)$ then i=j;

}

k=k+1;

calculate L_k;

calculate c_k;

}

Halt.

In Algoritm 1, c is the control parameter, L is the searching length, S is the searching set and f is an evaluation function.

CHAPTER III

ANALYSIS OF THE PROBLEM

3.1 Characteristics of The Problem

The combinatorial optimization problem has some characteristics:

1. A fertilizer option table contains two basic functions $c(x)$ and $r(x)$, where x is the option identification number. $c(x)$ and $r(x)$ are nonlinear, non-negative and discrete.

2. Because the scales of N, P, K are fixed to 10 pounds, there is not an obvious relation between any two fertilizer options of one field in the sense of fertilizer composition. On the other hand, there is a relation between fertilizer options for one field in the sense of cost, return, return rate and increasing return rate.

3. For different fields, the i_th field and the j_th field, there are no relations between option identification number variables x_i and x_j for functions $r_i(x_i)$, $r_j(x_j)$, $c_i(x_i)$ and $c_j(x_j)$. The condition is different from a common nonlinear programming problem, in which the objective function $f(x)$ and condition functions $g_i(x)$, $1 \leq i \leq n$, have a common variable vector $x = \{x_1, \ldots, x_n\}$.

4. The bound constraint and the combinatorial constraint show that the problem is related to nonlinear programming and combinatorial optimization.

3.2 Assumption And Simplification

Based on the above analysis, the following assumption and simplification are made.

1. Because the focus of the problem is on cost and return, N, P, K and Lime items of a fertilizer option will not be considered in the process of solving the problem.

2. Because the problem is to minimize cost and maximize return, fertilizer options of each field can be reduced by sorting the fertilizer option table on both cost and return items into monotone decreasing order with respect to option identification number x and discard those fertilizer options in which cost item or return item does not fit a monotone decreasing order.

3. After sorting cost and return items into monotone decreasing order, a new table is created for a field. In a new table, new functions $c'(x)$ and $r'(x)$ are nonlinear, nonnegative, discrete and monotone. Function values of $c'(x)$ are dependent; i.e., the function value of upper fertilizer option can be obtained by adding some value to the lower fertilizer option, i.e. $c'(i) = c'(i+1) + a,\ a > 0$. Function values of $r'(x)$ are dependent, too; i.e., $r'(i) = r'(i+1) + b,\ b > 0$.

CHAPTER IV

OUTLINE OF THE ALGORITHM

4.1 Outline Of The Algorithm

Because the problem is special and is relative to nonlinear programming and combinatorial optimization, any single existent algorithm such as any algorithm for nonlinear programming, does not fit. The strategy of "divide and conquer" is taken to design an approximation algorithm for the problem.

There are two approaches in the algorithm. If the first approach fails, the second approach will continue to work until solutions are found. The first approach is to use the increasing return/cost rate in the second normal tables as a heuristic to search globally. In the first approach, the main goal is to maximize the return and minimize the cost under the bound of limited capital, and the global searching is among the second normal tables. The searching uses a large granulity, which means it may cross many options in an input table within one searching step. This results in global and effective searching. Some ideas of nonlinear programming are adopted here.

The second approach is to combine a genetic algorithm and a simulated annealing algorithm to search globally. In the second approach, the main goal is to satisfy the combinatorial constraint while using a genetic algorithm to search using a small step size (option by option in the increasing rate table) and using the simulated annealing algorithm recursively to search globally. Both the genetic algorithm and the simulated annealing algorithm, attempt to maximize the return and minimize the cost with the help

of increasing return/cost rate on the increasing rate tables and the second normal form tables.

There are three main parts in the algorithm :

1. Global heuristic searching,

2. Recursive simulated annealing algorithm searching,

3. Local genetic algorithm searching.

The detailed algorithm is presented in the following chapter.

4.2 Tables

Since the functions, $c'(x)$ and $r'(x)$ are given in the form of a table, some operations of the algorithm work on the functions and return results in the form of a table. Definitions of some tables, which are used in the algorithm, are given in order to deal with the problem easily. They are the input table, the output table, the first normal form table, the increase in return/cost rate table and the second normal form table. An input table is a fertilizer option table which contains two functions, $r(x)$ and $c(x)$ for a field. One field has only one input table. An output table lists component option of an optimal solution or a near optimal solution for each field. The first normal form table is an option table, which is the starting point of the algorithm. The increase in return/cost rate table and the second normal form table are two option tables, which are two parts of the first approach in the algorithm.

Definition 7 (First Normal Form Table)

An option table is said to be the *first normal form table* if and only if the return and cost terms in the table are monotone decreasing with respect to the option identification number x.

Definition 8 (Input Table)

An *input table* of an option table is its first normal form table. It has field size and field identification number items and does not have Yield, N, P, K, and Lime items.

Table 2 The First Normal Form Table For Table 1

Option #	Yield	N ---------	P pound /acre	K ---------	Lime ton /acre	Cost $	Return $	Return Rate %
1	39.8	50	20	20	0.0	17.60	31.50	179
2	39.4	50	20	10	0.0	16.50	31.12	189
3	38.9	50	10	20	0.0	15.40	30.74	200
4	38.4	50	10	10	0.0	14.30	30.40	213
5	37.8	40	10	20	0.0	13.20	29.76	225
6	37.4	40	10	10	0.0	12.10	29.52	244

The option numbers in Table 2 are updated.

Definition 9 (The Increase In Return/Cost Rate Table)

An input table is said to be the *increase in return/cost rate table* if and only if each item includes the percentage of increase in return/cost rate denoted by rc, which is

calculated on the base of cost level and return level of the lower option with respect to the option number i by the following formula:

Increase in Return/Cost Rate of Option i = { [Return (i) - Return ($i+1$)] / [Cost (i) - Cost ($i+1$)] }× 100.

Definition 10 (The Second Normal Form Table)

An increase in return/cost rate table is said to be the *second normal form table* if and only if the increase in return/cost rate terms in the table are monotone increasing with respect to the option identification number x.

Table 3 Input Table Of Table 2.

Field id	Option id	Field Size (acre)	Cost $	Return $	Return Rate %
1	1	50	17.60	31.50	179
1	2	50	16.50	31.12	189
1	3	50	15.40	30.74	200
1	4	50	14.30	30.40	213
1	5	50	13.20	29.76	225
1	6	50	12.10	29.52	244

Table 4 The Increase In Return/Cost Rate Table of Table 3.

Field id	Option id	Field Size (acre)	Cost $	Return $	Increase In Return/Cost Rate %
1	1	50	17.60	31.50	34.5
1	2	50	16.50	31.12	34.5
1	3	50	15.40	30.74	30.9
1	4	50	14.30	30.40	58.2
1	5	50	13.20	29.76	21.8
1	6	50	12.10	29.52	244

Table 5 The Second Normal Form Table of Table 4.

Field id	Option id	Field Size (acre)	Cost $	Return $	Increase In Return/Cost Rate %
1	1	50	17.60	31.50	33.33
1	4	50	14.30	30.40	40
1	6	50	12.10	29.52	244

Table 6 The Output Table.

Field	1	2	...	n
Option	1	3	...	2

4.3 Design Choices

For adopting the concepts of the genetic algorithm and simulated annealing algorithm in the new algorithm, several choices must be made.

1. Population of Search Nodes

The initial population of search nodes of simulated annealing algorithm is fertilizer options in the second normal form table of all fields.

The population of search nodes of genetic algorithm is fertilizer options in the increase in return/cost rate table of all fields.

2. Fitness Function/Cost Function

The fitness function value of an option for genetic algorithm is the increase in return/cost rate of the option in the increase in return/cost rate table.

The cost function value of an option for simulated annealing algorithm is the increase in return/cost rate of the option in the second normal form table.

3. Genetic Operator

The fertilizer options in the second normal form table of all fields are sorted into a list L in a monotone decreasing order. Let L = { op_1, op_2, ... , op_n }, rc of $op_i \geq$ rc of

op_{i+1} ; op_1 + op_2 is offspring of op_1, i.e. (i+1)_th offspring = i_th offspring + op_{i+2} for the simulated annealing algorithm.

For a field, any option in the second normal form table can be a head of a group of options in the increase in return/cost rate table. In the genetic algorithm, an i_th offspring may be generated by adding an option A in the increase in return/cost rate table to the (i-1)_th offspring. Option A is in a group headed by a option in list L.

4. Probabilty

Values 0 and 1 are generated randomly. If 1 is obtained, the genetic algorithm will be executed within a group options headed by the op_{i+2} of list L during the simulated annealing searching at the i_th step. If 0 is obtained, it will be executed within a group options headed by the op_{i+1} of list L during the simulated annealing searching at the i_th step.

5. Stop Criterion

The capital and the length of fertilizer option list L are used as stopping criteria.

CHAPTER V

THE ALGORITHM

5.1 Global Heuristic Searching

There are several steps to realize global heuristic searching:

1. Generate the increase in return/cost rate table for each field,
2. Generate the second normal form table for each field,
3. Sort options on the increase in return/cost rate among the second normal form tables into a decreasing order list L,
4. Pick options from L, one by one, and calculate the accumulative cost C,
5. Obtain an optimal solution or obtain a solution with maximum return and minimum cost that does not satisfy the combinatorial constraint and go to the second approach (simulated annealing).

Algorithm 2 Generate The Increase In Return/Cost Rate Table

Input: An input table.

1. Calculate option number n.
2. For (i = n – 1; i > 0; i – 1)

   rc(i) = (r(i)– r(i+1)) / (c(i) - c(i+1)).
3. Output: An increase in return/cost rate table.
4. Halt.

Algorithm 3 Generate The Second Normal Form Table

Input: An increase in return/cost rate table.

BEGIN

1. Calculate option number n.

2. Initialize rc(n+1)=0.

3. FOR (i=n-1; i < 0; i-1)

    IF (rc(i+1) < rc(i)) {

        calculate a new rc(i) based on the option form row i+2;

        drop the option on row i+1;

        break;

    }

  IF (i=0)

    Goto 4;

  ELSE

    Goto 1;

4. Output: A second normal form table.

END

Algorithm 4 Sorting

Input: Field number n, n second normal form tables.

BEGIN

0. L[m], m=0.

1. FOR (i=1; i<n+1; i+1)

   Count option number N_i for field i;

2. Temp = rc(N_1) ;

   FOR (i=2; i<n+1; i+1) {

      IF (Temp < rc(N_i))

         Temp = rc(N_i);

      ELSE IF (Temp = rc(N_i)) {

         IF (Temp.fieldsize < i-th field size) {

            Temp = rc(N_i);

         }

         ELSE IF (Temp.fieldsize = i-th field size) {

            IF (rc(N_i-1) = 0 or upper rc of Temp < rc(N_i-1))

               Temp = rc(N_i);

         }

      }

   }

3. L[m] = Temp; m+1; option number of Temp's field –1.

4. IF all N_i = 0, i = 1, ... , n, Goto 5; ELSE Goto 2.

5. Output: A list L[m].

END

Algorithm 5 Global Heuristic Searching

Input: List L[m], field number, Capital, error.

BEGIN

1. Cost =0;

2. FOR (i=0; i<m;i+1) {

Cost =Cost + L[i]' Cost;

IF (Cost > Capital + error)

Goto 3.

ELSE IF (Capital – error ≤ Cost ≤ Capital + error) {

Put L[i] into output table

Goto 4.

}

Put L[i] into output table

}

3. Output: The output table (the optimal solution), Goto 5.

4. Output: The output table (the solution with maximum return and minimum cost, but not satisfy the combinatorial constraint), Goto 6.

5. Halt (end searching).

6. Halt (continue on for the second approach).

END

After the global heuristic searching, an output table is generated. The increase in return/cost rate of each option in the output table is no less than the return rate of any one of the other options in L, which are not obtained from the simulated annealing searching and the genetic searching of the algorithm until now. In other words, the options in the output table have the maximum return and minimum cost because they have the maximum increase in return/cost rate among all options of all fields.

5.2 Local Genetic Searching

The local genetic searching involves searching the increase in return/cost rate tables, the second normal form tables and L[m]. From L[m] and the second normal form tables, a range in the increase in return/cost rate tables, is searched; selected items are placed in L[m] in sorted order.

Algorithm 6 Local Genetic Searching

Input: L[Step], L[Step+1], option_id = option number of L[Step]

BEGIN

1. FOR (i=1,i<group size; i+1)

   IF (optoin_id-i.rc > L[Step+1].rc)

      option B = option_id-i and Goto END;

2. IF L[Step].rc < L[Step+1].rc or L[Step].size ≠ L[Step+1].size

      option B = L[Step+1], Goto END;

   ELSE

      Step+1, Goto 1;

END

5.3 Simulate Annealing Searching

The algorithm involves algorithm 6 and executes algorithm 6 recursively.

Algorithm 7 Simulated Annealing Searching

Input: Last_Remain, list L, output table P, Step;

BEGIN

IF L[Step+1] is not the last item in L {

1. IF random() = 1, Step+1, Algorithm 7 (Simulated Annealing Searching)

   ELSE IF random() = 0, Goto 2;

2. Algorithm 6 (Local Genetic Searching) among the group options represented by the last option A put in P and the next option in L to generate an option B;

  IF there is an option B

    Goto 3;

  ELSE

    Step+1, Algorithm 7 (Simulated Annealing Searching) ;

3. IF exp((B.rc-A.rc)/ Last_Remain) > random[0,1)

    put in P and Algorithm 7 (Simulated Annealing Searching);

  ELSE

    Goto 2;

}

ELSE (L[Step+1] is the last item in L.)

Algorithm 6 (Local Genetic Searching) among the group options represented by the last option A put in P and the next option in L to generate an option B;

IF there is an option B

put in P and Goto END;

ELSE

Goto END;

END

CHAPTER VI

NUMERICAL RESULTS

6.1 Introduction

A program written in C has been implemented on an IBM 386 compatible personal computer. It works on an example with 20 discrete functions, obtains solution(s) within 5 seconds.

The output of the program has three phases:

1. The optimal solution or a partial result obtained by global heuristic searching,

2. An optimal solution, that does not satisfy the combinatorial constraint obtained by global heuristic searching,

3. A near optimal solution satisfying the combinatorial constraint obtained by genetic and simulated annealing searching.

When the first approach of the algorithm obtains an optimal solution, the output has only phase 1. Otherwise, the output has the three phases.

In the following sections, the numerical results of an example not only shows the differences among the three phases, but also shows the quality of solution obtained from phase 3.

6.2 Result of Global Heuristic Searching

Table 7 Optimal Solution Or A Partial Result

Field	1	2	3	4	5	6	7	8	9	10
Option	1	4	4	4	6	6	1	4	4	4
Field	11	12	13	14	15	16	17	18	19	20
Option	6	6	4	4	4	4	6	6	4	6

Fund $3200.00, Cost $3186.70, Return $8218.84, Return 257.91%.

6.3 Result Of Phase 2

Let L[i] be an item of L[m], which is the last one checked in the global heuristic searching routine, and results in C > Capital + error. Based on the results of global heuristic searching, to take the option given by L[i] and to use the rest Capital after L[i-1] to fertilize partial of the field which is identified by L[i] to get the solution with maximum return and minimum cost.

RESULT (An optimal solution not satisfying the combinatorial constraint):

Extra option 1 for field 13, fertilize 4.030318 acre(s) only!

The other acre(s), use the option on the above list (phase 1).

Fund $3200.00, Cost $3200.00, Return $8284.29, Return 258.88%.

6.4 Result Of Simulated Annealing

Table 8 A Near Optimal Solution

Field	1	2	3	4	5	6	7	8	9	10
Option	1	4	4	4	6	6	1	4	4	4
Field	11	12	13	14	15	16	17	18	19	20
Option	6	6	2	4	4	4	6	6	4	6

Fund $3200.00, Cost $3197.70, Return $8227.29, Return 257.29%

6.5 Comparison

Two interesting things are obtained by comparing the above three output phases:

1. The solution in phase 2 has the highest return,

2. Although the solution in phase 3 has $8.45 more return than the return of the solution in phase 1, it has $11 more cost than the cost of the solution in phase 1.

According to the structure of the algorithm, the return of the solution in phase 2 is never less than the return of the solution in phase 1. The solution in phase 3 may have more return and less cost than the solution in phase 1. It is also possible that the other cases between the solutions of phase 1 and phase 3 may occur. So, in some case, three phase outputs are provided for user to choose from.

The computing complexity of the algorithm is about 13×10^2 for 20 field, 10 options per field, 11×50^2 for 20 fields, 50 options per field. Comparing with the trversal

algorithm, the algorithm reduces the computing complexity from $O(m^n)$ to $O(m^2)$, where n is field number, m is option number per field.

REFERENCES

[Aarts 89] Aarts, E. h. L. and Korst, J., Simulated Annealing and Boltzmann Machines: a Stochastic Approach to Combinatorial Optimization and Neural Computing, John Wiley & Sons, New York 1989.

[Barker 76] Barker, J.A., and Henderson, D., What is "liquid"? Understanding the states of matter, Reviews of Modern Physics 48, 587-671.

[Bazaraa 79] Bazaraa, M. S., Nonlinear Programming: theory and algorithms, Wiley, New York 1979.

[Goldberg 89] Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley Pub. Co., 1989.

[Himmelblau 72] Himmelblau, D. M., Applied Nonlinear Programming, MCGRAW_HILL Book Co., 1972.

[Johnson 92] Johnson, G. V. and Nofziger, D. L., NPK$PLUS: A Computer Program to Examine Agronomic and Economic Value of Alternative Fertilizer Rates, J. Prod. Agric. 5:415-420 1992.

[Johnson 91] Johnson, G. V., Nofziger, D. L., and Hunter, T. D., NPK$PLUS: An Interactive Microcomputer Program to Interpret Soil Test Results and To Evaluate The Economics of Alternative Fertilizer Application Rates, version 4, OSU 1991.

[Kinnear 94] Kinnear, Kneneth E., Advances in Genetic Programming, The MIT Press, Cambridge, Massachusetts 1994.

[Kirkpatrick 82] Kirkpatrick, S., C.D. Gelatt JR., and M. P. Vecchi, Optimization by simulated annealing, IBM Research Report RC 9355.

[Kirkpatrick 83] Kirkpatrick, S., C.D. Gelatt JR., and M. P. Vecchi, Optimization by simulated annealing, Science 220, 671-680.

[Laarhoven 87] Laarhoven, Peter J. M., Simulated Annealing: theory and applications, Kluwer Academic Publishers, Boston 1987.

[Lee 91] Lee, Bang W., Sheu, Bing J., Hardware Annealing in Analog VLSI Neurocomputing, Kluwer Academic Publishers, Boston 1991.

[Martos 75] Martos, B., Nonlinear Programming Theory and Methods, American Elsevier Pub. Co., New York 1975.

[Michalewicz 92] Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag Berlin Heidelberg, New York 1992.

[Pfaffenberger 76] Pfaffenberger, Roger C. and Walker, David A., Mathematical Programming for Economics and Business, The Iowa State University Press, Ames, Iowa 1976.

[Soucek 92] Soucek, B., Dynamic, Genetic, and Chaotic Programming, John Wiley & Sons, New York 1992.

[Sposito 75] Sposito, V. A., Linear and Nonlinear Programming, The Iowa State University Press, Ames, Iowa 1975.

APPENDIX A

A COMBINATORIAL OPTIMIZATION PROGRAM OF FINDING THE BEST RETURN WITH LIMITED CAPITAL AMONG 1-20 FIELD(S) FOR Computer Software Series CSS-47 NPK$PLUS: AN INTERACTIVE MICROCOMPUTER PROGRAM TO INTERPRET SOIL TEST RESULTS AND TO EVALUATE THE ECONOMICS OF ALTERNATE FERTILIZER APPLICATION RATES

/**********

A COMBINATORIAL OPTIMIZATION PROGRAM OF FINDING

THE BEST RETURN WITH LIMITED CAPITAL

AMONG 1-20 FIELD(S)

AUTHOR

YUNLONG CHEN

MAY 12, 1994

FOR

Computer Software Series CSS-47

==========

NPK$PLUS:

AN INTERACTIVE MICROCOMPUTER PROGRAM TO INTERPRET SOIL TEST

RESULTS AND TO EVALUATE THE ECONOMICS OF ALTERNATE

FERTILIZER APPLICATION RATES

==========

DESCRIPTION

PURPOSE

The program reads cost and return data of many fields (1-20), a capital and a field number, finds the best return with the limited capital among the options of all fields.

INPUT

1. A data file of field cost and return information. There aresix items:

int fieldid;	field id number
int optionid;	option id number
float fieldsize;	field size
float cost;	cost per acre
float rtn;	return per acre
int rc;	return rate (%) per acre

2. Capital (input from keyboard)

float F;	capital.

3. Field number (input from keyboard)

int N;	working on N fields.

OUTPUT

A list of option number for each field. Limited Capital, total cost of the decision. Total return, total return rate (%) of the decision.

METHOD

Combination of ideas of nonlinear programming method, genetic algorithm and simulated annealing.

VARIABLE TYPE
GLOBAL VARIABLE

```
struct record {     keep one record from input data file
    int fieldid;
    int optionid;
    float fieldsize;
    float cost;
    float rtn;
    int rc;
};

typedef struct record rec;    record variable type: rec

struct field {              keep records for each field
    rec option[10];
};

typedef struct field fld;     field variable type: fld

int N                 field number
int L[21]             option number record for 20 fields
int delta             allowable error for capital
int rdnum             random number
int p_start           decision step counter
float F               capital
fld f1[21]            field records with original options
                            and return/cost calculated based on
                            the lower option
fld tf1[21]           field records with merged options
                            and return/cost calculated based on
                            the lower merged option

rec DP[200]           sorting results of merged options of
                            all fields

rec po[200]           decision package of merged options of
                            all fields

rec choice[21]        keep final results
```

FUNCTION

```
1.  rec init()                initial a record

2.  int LC(f,l)               table length counter (option number counter)
    fld f;
    int l;

3.  fld MD(f,l)               merge options and calculate return/cost
    fld f;                    on the lower merged option base in order
    int l;                    to get a new talbe with monotone decreas-
                              ing return/cost.

4.  fld ROC(f,l)              calculate return/cost on the lower
    fld f;                    option base
    int l;

5.  SRCH1(B,s_start,s_end)    recursive search the next option and put
    float B;                  it in decision package
    int s_start,s_end;

6.  rec trans(r2)             transfer a record
    rec r2;

*************************************************************************/
/* 05/12/94 version */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <fcntl.h>
#include <float.h>
#include <math.h>
#include <string.h>

struct record {       /* keep a record from input data file */
  int fieldid;
  int optionid;
  float fieldsize;
  float cost;
  float rtn;
  int rc;
};

typedef struct record rec;   /* record variable type: rec */

struct field {               /* keep records of a field   */
  rec option[10];
```

```
};

typedef struct field fld;       /* field variable type: fld  */

int N,L[21],delta,rdnum,p_start;
float F;
fld f1[21],tf1[21];
rec DP[200],po[200],choice[21];

void main(void)
{

  char  Fund[6], fieldnum[3], filename[21];
  char  a1[3],a2[2],a3[4],a4[5],a5[5],a6[5];
  int   i,j,l,i1,j1,L1[21],L2[21],temp1,Bup_step,DP_counter;
  int   s_start,s_end,info,fid,op1,op2,argu,t_rc,t_rc1;
  float Last_T2, T1_cost, T2_cost, TCost, DPcost, DPrtn;
  float f_portion, dif_cost, MIN, SMC, SSC, Sts;
  fld   f[21],f2[21],tf2[21], ROC(), MD();
  rec   temp, rd0, rd1, rd2, init(), trans();
  FILE  *fp;

  /*
   * initializing variables
   */

  T1_cost=T2_cost=Last_T2=TCost=DPcost=DPrtn=0.000000;

  for(i=0;i<21;i++) {
    L[i]=0;
    L1[i]=0;
    L2[i]=0;
  }

  for(i=0;i<21;i++) {

    for(j=0;j<10;j++) {
      f[i].option[j]=init();
      f1[i].option[j]=init();
      f2[i].option[j]=init();
      tf1[i].option[j]=init();
      tf2[i].option[j]=init();
    }
    choice[i]=init();
  }
```

```
for(i=0;i<200;i++) {
 po[i]=init();
 DP[i]=init();
}

temp=init();

/*
 * Input data
 */

printf("\nEnter the name of the data file: \n");
gets(filename);

printf("Enter the Fund: \n");
gets(Fund);
F=atoi(Fund);

printf("Fund is $%.2f;\n", F);
printf("\nEnter the field number: \n");

gets(fieldnum);
N=atoi(fieldnum);

printf("There are (is) %d field(s).\n", N);

if ((fp = fopen(filename, "r")) == NULL) {
  printf("cannot open file!\n");
  exit(1);
}

/*
 * reading data file
 */

while(!feof(fp)) {

  fscanf(fp,"%s %s %s %s %s %s",a1,a2,a3,a4,a5,a6);
  temp.fieldid=atoi(a1);
  temp.optionid=atoi(a2);
  temp.fieldsize=atof(a3);
  temp.cost=atof(a4);
  temp.rtn=atof(a5);
  temp.rc=atoi(a6);
```

```
  for(i=0; i<21; i++)

   if (i==temp.fieldid-1) {

       for(j=0; j<10; j++)

        if (j==temp.optionid-1) {
          f[i].option[j]=trans(temp);   /* get record        */
          L[i]=j;                       /* keep option number */
          break;
        }
       break;
   }
   temp=init();
}

fclose(fp);

/*
 * calculate the sum of max cost of all fields and min cost
 * among all fields
 */

SMC=SSC=Sts=0.000000;
delta=5;
MIN=f[0].option[L[0]-1].cost*f[0].option[L[0]-1].fieldsize;

for(i=0;i<N;i++) {
 Sts=f[i].option[0].cost*f[i].option[0].fieldsize;

 if((0<(Sts/2)) && ((Sts/2)<delta))
  delta=Sts/2;
 SMC=SMC+Sts;
 SSC=SSC+f[i].option[L[i]-1].cost*f[i].option[L[i]-1].fieldsize;

 if(MIN>f[i].option[L[i]-1].cost*f[i].option[L[i]-1].fieldsize)
  MIN=f[i].option[L[i]-1].cost*f[i].option[L[i]-1].fieldsize;
}

printf("delta = %d, SMC = %.2f, SSC = %.2f, MIN = %.2f\n", delta, SMC, SSC, MIN);

if(F>SMC) {
 printf("Take option 1 for every field!!!\n");
 exit(2);
```

```
}

if(F<MIN) {
 printf("Take option 0 for every field, or raise fund !!!\n");
 exit(3);
}

/*
 * create field tables with monotone decreasing return/cost
 */

for(i=0; i<N+1; i++) {
 f1[i]=ROC(f[i],L[i]);     /* f1[i] table with r/c     */
 tf1[i]=MD(f1[i],L[i]);
}

for(i=0; i<N+1; i++)

  for(j=0;j<L[i];j++) {
   tf2[i]=MD(tf1[i],L[i]);
   tf1[i]=MD(tf2[i],L[i]);  /* tf1[i] table with md r/c  */
  }

s_end=0;

for(i=0;i<N;i++) {
 L1[i]=LC(tf1[i],L[i]);
 L2[i]=L1[i];            /* option number in MD table */
 s_end=s_end+L1[i];
}

temp=init();
T1_cost=T2_cost=TCost=DPcost=DPrtn=0.000000;
temp1=Bup_step=info=0;

/*
 * Finding the best return with the limited capital
 */

while(TCost<SMC) {

  /*
   * sorting option with respect to return/cost
   */
```

```
for(i=0;i<N;i++) {

 if((temp.rc<tf1[i].option[L1[i]-1].rc) && (0<=L1[i]-1) ) {
     temp=trans(tf1[i].option[L1[i]-1]);
     temp.cost=tf1[i].option[L1[i]-1].cost-tf1[i].option[L1[i]].cost;

     if(0<L1[i]-1)                  /* more than one option ? */
      temp1=tf1[i].option[L1[i]-2].rc;  /* get return/cost      */
     else
      temp1=0;
 }
 else if(temp.rc==tf1[i].option[L1[i]-1].rc) {  /* if r/c's are equal */

     if(temp.fieldsize<tf1[i].option[L1[i]-1].fieldsize) {/* compare size */
      temp=trans(tf1[i].option[L1[i]-1]);
      temp.cost=tf1[i].option[L1[i]-1].cost-tf1[i].option[L1[i]].cost;

      if(0<L1[i]-1)                 /* more than one option ? */
       temp1=tf1[i].option[L1[i]-2].rc; /* get return/cost of higher option */
      else
       temp1=0;
     }
     else if(temp.fieldsize==tf1[i].option[L1[i]-1].fieldsize) {
                                    /* If sizes are equal    */
                                    /* compare higher option */

      if(0<L1[i]-1) {               /* more than one option ?  */

       if((0<temp1) && (temp1<tf1[i].option[L1[i]-2].rc)) { /* compare r/c */
        temp=trans(tf1[i].option[L1[i]-1]);
        temp.cost=tf1[i].option[L1[i]-1].cost-tf1[i].option[L1[i]].cost;

        if(0<L1[i]-1)               /* Is there an option ? */
            temp1=tf1[i].option[L1[i]-2].rc; /* get r/c of higher option */
        else
            temp1=0;
       }
      }
      else if(0<temp1) {
       temp=trans(tf1[i].option[L1[i]-1]);
       temp.cost=tf1[i].option[L1[i]-1].cost-tf1[i].option[L1[i]].cost;

       if(0<L1[i]-1)
        temp1=tf1[i].option[L1[i]-2].rc;
       else
```

```
          temp1=0;
        }
       }
  }
}

T1_cost=temp.cost*temp.fieldsize;

/*
 * Heuristic Searching
 */

if(info==0) {
  T2_cost=F-TCost;

  if(T1_cost<T2_cost-delta) { /* within capital bound */
      DP[Bup_step]=trans(temp);
      Last_T2=T2_cost;
      Bup_step++;
      TCost=TCost+T1_cost;
      T1_cost=0.000000;

      for(j=0;j<N;j++)        /* update option counter */

        if(temp.fieldid==tf1[j].option[L1[j]-1].fieldid) {
          temp=init();

          if(0<=L1[j]-1) {
            L2[j]=L1[j];
            L1[j]--;
          }
        }

  }
  else if((T2_cost-delta<=T1_cost) && (T1_cost<=T2_cost+delta)) {
                                  /* within capital error */
      DP[Bup_step]=trans(temp);
      Bup_step++;
      TCost=TCost+T1_cost;
      T1_cost=0.000000;

      for(j=0;j<N;j++)        /* update option counter */

        if(temp.fieldid==tf1[j].option[L1[j]-1].fieldid) {
          temp=init();
```

```
        if(0<=L1[j]-1) {
          L2[j]=L1[j];
          L1[j]--;
        }
      }

    printf("-------------------The Initial Solution-------------------\n");
    printf("Field    1   2   3   4   5   6   7   8   9   10\n");
    printf("Option");

    if(N<=10) {

      for(i=0;i<N;i++) {
        printf("    %d", tf1[i].option[L1[i]].optionid);
        DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
        DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
      }
      printf("\n-------------------------------------------------------\n");
    }
    else if(10<N) {

      for(i=0;i<10;i++) {
        printf("    %d", tf1[i].option[L1[i]].optionid);
        DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
        DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
      }
      printf("\n-------------------------------------------------------\n");
      printf("Field    11   12   13   14   15   16   17   18   19   20\n");
      printf("Option");

      for(i=10;i<N;i++) {
        printf("    %d", tf1[i].option[L1[i]].optionid);
        DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
        DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
      }
      printf("\n-------------------------------------------------------\n");
    }
    printf("Fund: $%.2f, Cost: $%.2f,\n", F, DPcost);
    printf("Return: $%.2f, Return Rate: %.2f%%.\n", DPrtn, DPrtn*100/DPcost);
    printf("End of the program!\n");
    exit(4);
}
else { /* T1_cost > T2_cost+delta */
    DP[Bup_step]=trans(temp);
```

```
f_portion=T2_cost/temp.cost;
printf("------------------The Initial Solution------------------\n");
printf("Field   1   2   3   4   5   6   7   8   9   10\n");
printf("Option");

if(N<=10) {
 for(i=0;i<N;i++) {
  printf("   %d", tf1[i].option[L1[i]].optionid);
  DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
  DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
 }
 printf("\n----------------------------------------------------------\n");
}
else if(10<N) {

 for(i=0;i<10;i++) {
  printf("   %d", tf1[i].option[L1[i]].optionid);
  DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
  DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
 }
 printf("\n----------------------------------------------------------\n");
 printf("Field   11  12  13  14  15  16  17  18  19  20\n");
 printf("Option");

 for(i=10;i<N;i++) {
  printf("   %d", tf1[i].option[L1[i]].optionid);
  DPcost=DPcost+tf1[i].option[L1[i]].cost*tf1[i].option[L1[i]].fieldsize;
  DPrtn=DPrtn+tf1[i].option[L1[i]].rtn*tf1[i].option[L1[i]].fieldsize;
 }
 printf("\n----------------------------------------------------------\n");
}
printf("Fund: $%.2f, Cost: $%.2f, ", F, DPcost);
printf("Return: $%.2f, Return Rate: %.2f%%.\n\n", DPrtn, DPrtn*100/DPcost);

for(i=0;i<N;i++)

 if(temp.fieldid==tf1[i].option[L1[i]-1].fieldid) {

  if(tf1[i].option[L1[i]].cost > 0)
   DPcost=DPcost+(temp.cost-tf1[i].option[L1[i-1]].cost)*f_portion;
  else
   DPcost=DPcost+temp.cost*f_portion;
  DPrtn=DPrtn+(temp.rtn-tf1[i].option[L1[i]].rtn)*f_portion;
  break;
 }
```

```
      printf("----------The Maximum Return Solution----------\n");
      printf("Extra option %d for field %d, fertilize %.2f acre(s) only!\n",
       temp.optionid,i+1, f_portion);
      printf("the rest acre(s) of field %d, take the option on the above table!\n",i+1);
      printf("Fund: $%.2f, Cost: $%.2f, ", F, F);
      printf("Return: $%.2f, Return Rate: %.2f%%.\n\n", DPrtn, DPrtn*100/DPcost);
      DP[Bup_step]=trans(temp);

      for(i1=0;i1<N+1;i1++)

       for(j1=0;j1<L[i1]+1;j1++) {

        if(f1[i1].option[j1].optionid==tf1[i1].option[L1[i1]].optionid) {
         choice[i1]=init();
         choice[i1]=trans(f1[i1].option[j1]);
         break;
        }
       }

      for(i=0;i<N;i++)

       if(temp.fieldid==tf1[i].option[L1[i]-1].fieldid) {
        temp=init();

        if(0<=L1[i]-1)
         L1[i]--;
       }

      temp=init();
      s_start=Bup_step;  /* the next item (not in DP) */
      Bup_step++;
      info=1;
   }
}
else if(info==1) { /* store sorting results */
  DP[Bup_step]=trans(temp);
  TCost=TCost+T1_cost;
  T1_cost=0.000000;

  for(i=0;i<N;i++) {

      if(temp.fieldid==tf1[i].option[L1[i]-1].fieldid) {
       temp=init();
```

```
            if(0<=L1[i]-1)
              L1[i]--;
            }
        }

        if(Bup_step==s_end)
            break;
        else {
            Bup_step++;
            temp=init();
        }
      }
    }

    for(i=0;i<s_start;i++)
      po[i]=trans(DP[i]);

    p_start=s_start;

    /*
     * Simulated annealing searching
     */

    rd0=init();
    rd1=init();
    rd2=init();
    rdnum=random(2);

    if(rdnum==0)
      SRCH1(T2_cost,s_start,s_end,po);
    else {
      rd2=trans(DP[s_start]);     /* next item    */
      rd1=trans(po[s_start-1]);   /* current item */

      for(i=p_start-2;i>-1;i--) {
        rd0=trans(po[i]);         /* lower value of current item columne */

        if(rd0.fieldid==rd1.fieldid)
            break;
      }
      if(i<0)
        rd0=init();

      fid=rd1.fieldid-1;
      op1=rd1.optionid;
```

```
op2=rd0.optionid;

if((op1+1<op2-1) && (0<rd0.rc)) {
 t_rc=0;

 for(j=op1+1; j<op2-1; j++) {
     t_rc=(f1[fid].option[j-1].rtn-rd0.rtn)/
        (f1[fid].option[j-1].cost-rd0.cost);

     if(rd2.rc<=t_rc) {
      t_rc1=(f1[fid].option[op1-1].rtn-rd0.rtn)/
            (f1[fid].option[j-1].cost-rd0.cost);
      argu=(t_rc-t_rc1)/Last_T2;
      rdnum=random(100);
      argu=exp(argu)*100;

      if( rdnum < argu ) {
       dif_cost=choice[fid].cost-(f1[fid].option[j].cost-rd0.cost);
       T2_cost=T2_cost+dif_cost;
       choice[fid]=trans(f1[fid].option[j]);
       SRCH1(T2_cost,s_start,s_end);
       break;
      }
     }
 }
}
else if (rd2.cost!=0 && rd1.cost!=0) {
 argu=(rd2.rtn/rd2.cost-rd1.rtn/rd1.cost)/Last_T2;
 rdnum=random(100);
 argu=exp(argu)*100;

 if( rdnum < argu ) {
     dif_cost=f1[fid].option[op1-1].cost*f1[fid].option[op1-1].fieldsize;
     T2_cost=T2_cost+dif_cost;

     if(0<rd0.rc)
      choice[fid]=trans(rd0.optionid);
     else
      choice[fid]=init();
     SRCH1(T2_cost,s_start,s_end);
 }
 else {
     SRCH1(T2_cost,s_start,s_end);
 }
}
```

```
 }

} /* END OF MAIN */


/*****************************************
                FUNCTION init
    Initialize a record.
*****************************************/
rec init()
{

 rec r1;

 r1.fieldid=r1.optionid=r1.rc=0;
 r1.fieldsize=r1.cost=r1.rtn=0.000000;

 return(r1);

} /* end of init */


/*****************************************
                FUNCTION trans
    Transfer data of a record to another record.
    r2  record
*****************************************/
rec trans(r2)
rec r2;
{

 rec r1;

 r1.fieldid=r2.fieldid;
 r1.optionid=r2.optionid;
 r1.fieldsize=r2.fieldsize;
 r1.cost=r2.cost;
 r1.rtn=r2.rtn;
 r1.rc=r2.rc;

 return(r1);

} /* end of trans */
```

```
/*************************************************
                FUNCTION ROC
   Calculate return/cost based on the lower option level.

   current option return/cost
                = (current option return - lower option return)
           /(current option cost - lower option cost).
   f    field
   l    option number
*************************************************/
fld ROC(f,l)
fld f;
int l;
{

  int i,j,k;
  float m1,m2,m3,m4;

  m1=m2=m3=m4=0.000000;

  for(i=0;i<l;i++)

   if((0.000000<f.option[i+1].cost) && (1<=f.option[i].fieldid)
    && (f.option[i].fieldid<=N) ) {
    m1=f.option[i].rtn*100;
    m2=f.option[i+1].rtn*100;
    m3=f.option[i].cost-f.option[i+1].cost;

    if(m3==0.000000)
        break;
    m4=m1-m2;
    f.option[i].rc=m4/m3;
    m1=m2=m3=m4=0.000000;
   }

   return(f);

} /* end of ROC */


/*****************************************************
                FUNCTION MD
   Combine some adjacent options into one option to make a new
   field talbe with less options such that the return/cost in
   the new field table is monoton decreasing while option number
```

```
    is reducing.
    f   field
    l   original option number
*****************************************************/
fld MD(f,l)
fld f;
int l;
{

  int i,j,k,m,id,tmp1,tmp2;
  float m1,m2,m3,m4;
  fld ff1;

  tmp1=tmp2=k=m=0;
  m1=m2=m3=m4=0.000000;

  for(j=0;j<10;j++) {
   ff1.option[j]=init();
  }

  for(i=0;i<=l;i++) {
   tmp1=f.option[i].rc;
   id=f.option[i].fieldid;

   if((f.option[i+1].rc<=tmp1) && (id==f.option[i+1].fieldid))  {
     m1=f.option[i].rtn*100;
     m2=f.option[i+2].rtn*100;
     m3=f.option[i].cost-f.option[i+2].cost;
     if(m3==0.000000)
         break;
     m4=m1-m2;
     tmp1=m4/m3;
     m1=m2=m3=m4=0.000000;

     for(j=i+2;j<=l;j++) {
         k=0;

         if((f.option[j].rc<=tmp1) && (id==f.option[j].fieldid)
           && (f.option[j+1].rc!=0) )  {
           m1=f.option[i].rtn*100;
           m2=f.option[j].rtn*100;
           m3=f.option[i].cost-f.option[j].cost;

           if(m3==0.000000)
            break;
```

```
            m4=m1-m2;
            tmp1=m4/m3;
            k=j;
           }
      }
      ff1.option[m]=trans(f.option[i]);
      ff1.option[m].rc=tmp1;
      tmp1=0;
      m1=m2=m3=m4=0.000000;
      m++;

      if(0<k)
         i=k;
      else
         i++;
    }
    else if((f.option[i].rc>0) /* && (f.option[i+1].rc==0) */){
      ff1.option[m]=trans(f.option[i]);
      m++;

      if(f.option[i+1].rc==0)
          i=l+1;
    }
  }

  return(ff1);

} /* end of MD */


/*****************************************************
                FUNCTION LC
   Count option number of a field after grouping the options into
   monoton decreasing return/cost order.
   f   field
   l   option number before regrouping.
*****************************************************/
int LC(f,l)
fld f;
int l;
{

  int i;

  for(i=0; i<l; i++)
```

```
  if(f.option[i].rc==0)
    break;

 return(i);

} /* end of LC */


/*****************************************************
                FUNCTION SRCH1
   Recursively search next item and put it into decision package.
   B      rest Fund
   s_start start position on sorted array of options
   s_end   end position on the sorted array of options
*****************************************************/
SRCH1(B,s_start,s_end)
float B;
int s_start,s_end;
{

  int step,i,i2,j,k1,fid,fid1,t_rc1,t_rc10,argu1;
  int temp1,l,l0,l1,l2,op1,op2,p1,p2,flag,TR1,TR2,TRP;
  float ss,rc,T1_cost,T2_cost,T3_cost,TCost;
  float dp1cost, dp1rtn, Las_T2, dif_cost1;
  rec temp_rec,next,T3,rd10,rd11,rd12;

  /* initializing */

  TCost=T1_cost=T2_cost=T3_cost=0;
  dp1cost=dp1rtn=0.000000;

  temp_rec=trans(DP[s_start]);    /* get the current item */
  next=trans(DP[s_start+1]);      /* get the next item    */

  if(s_start+1==s_end)
    flag=1;
  else
    flag=0;

  while(temp_rec.fieldid==next.fieldid) {  /* the last item checking */
    s_start++;

    if(s_start==s_end) {
      flag=1;
```

```
    break;
   }
   else
    next=trans(DP[s_start+1]);
}

if(flag==1) { /* the last item */
  T2_cost=B-TCost;                    /* get current fund */
  fid=temp_rec.fieldid;
  fid--;

  if(0<choice[fid].rc) {
    op1=temp_rec.optionid;
    op2=choice[fid].optionid;

    if(op1+1<op2)

        for(j=op1; j<op2-1; j++) {
          ss=(f1[fid].option[j].cost-choice[fid].cost)*
            temp_rec.fieldsize;

          if(ss<=T2_cost+delta) {
            choice[fid]=trans(f1[fid].option[j]);
            break;
          }
        }
}
        printf("------------------ Near Optimal Solution ----------------\n");
        printf("Field    1   2   3   4   5   6   7   8   9   10\n");
        printf("Option");

        if(N<=10) {

         for(i=0;i<N;i++) {
          printf("    %d", choice[i].optionid);
          dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
          dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
         }
        printf("\n----------------------------------------------------------\n");
        }
        else {

         for(i=0;i<10;i++) {
          printf("    %d", choice[i].optionid);
          dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
```

```
        dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
        }
       printf("\n-----------------------------------------------------\n");
       printf("Field   11  12  13  14  15  16  17  18  19  20\n");
       printf("Option");

        for(i=10;i<N;i++) {
         printf("   %d", choice[i].optionid);
         dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
         dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
        }
       printf("\n-----------------------------------------------------\n");
    }
  printf("Fund: $%.2f, Cost: $%.2f, ", F, dp1cost);
  printf("Return: $%.2f, Return Rate: %.2f%%.\n", dp1rtn, dp1rtn*100/dp1cost);
}

if(flag==0) {   /* not the last item */
  l=next.fieldid-1;
  l0=next.optionid;

  if(0<l0)
    l0--;
  else
    l0=0;

  T1_cost=f1[l].option[l0].cost*next.fieldsize-choice[l].cost*next.fieldsize;

  Las_T2=T2_cost;
  T2_cost=B-TCost;
  fid=temp_rec.fieldid-1;

  if(0<choice[fid].rc) {
    op1=temp_rec.optionid;
    op2=choice[fid].optionid;

    if(op1+1<op2) {

        for(j=op1; j<op2-1; j++) {
              /* decompose the current item, one option by one option */
          ss=(f1[fid].option[j].cost-choice[fid].cost)*
            temp_rec.fieldsize;

          if(ss<=T2_cost+delta) {
            rc=(f1[fid].option[j].rtn-choice[fid].rtn)/
```

```
          (f1[fid].option[j].cost-choice[fid].cost);
         p1=rc*100;
         p2=next.rc;

         if(p2 <= p1) {
          T3_cost=(f1[fid].option[j].cost-choice[fid].cost)
              *temp_rec.fieldsize;
          T3=trans(f1[fid].option[j]);
          T3.rc=p1;
          break;
         }
         else {
          T3=trans(f1[fid].option[j]);
          T3.rc=p1;

          for(i2=0;i2<s_end;i2++)

              if(DP[i2].rc<=T3.rc)
               break;

          for(k1=i2+1;k1<s_end+1;k1++)
              DP[k1]=trans(DP[k1-1]);
          DP[i2]=trans(T3);
          s_end++;
          T3_cost=0;
          break;
         }
        }
       }
  }
 }
 else {
  T3_cost=0;
 }

while(TCost<B+delta) {

 if(0<T3_cost) {

  if(T3_cost<T2_cost-delta) {
      choice[fid]=trans(f1[fid].option[j]);
      po[p_start]=trans(f1[fid].option[j]);
      p_start++;
      TCost=TCost+T3_cost;
      T3_cost=0.000000;
```

```
Las_T2=T2_cost;
T2_cost=B-TCost;

if(T1_cost<T2_cost-delta) {
 Las_T2=T2_cost;
 l1=next.fieldid-1;
 l2=next.optionid-1;

 if(0<l2)
  l2--;
 else
  l2=0;

 TCost=TCost+T1_cost;
 choice[l1]=trans(f1[l1].option[l2]);
 po[p_start]=trans(f1[l1].option[l2]);
 p_start++;
 T1_cost=0.000000;
 next=init();
}
else if((T2_cost-delta<=T1_cost) && (T1_cost<=T2_cost+delta)) {
 l1=next.fieldid-1;
 l2=next.optionid-1;
 choice[l1]=trans(f1[l1].option[l2]);
 po[p_start]=trans(f1[l1].option[l2]);
 p_start++;
 TCost=TCost+T1_cost;
 T1_cost=0.000000;
 next=init();

printf("------------------ Near Optimal Solution ----------------\n");
printf("Field    1   2   3   4   5   6   7   8   9   10\n");
printf("Option");

if(N<=10) {

 for(i=0;i<N;i++) {
  printf("    %d", choice[i].optionid);
  dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
  dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
 }
printf("\n----------------------------------------------------------\n");
}
else {
```

```
    for(i=0;i<10;i++) {
     printf("    %d", choice[i].optionid);
     dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
     dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n-------------------------------------------------------\n");
    printf("Field    11   12   13   14   15   16   17   18   19   20\n");
    printf("Option");

    for(i=10;i<N;i++) {
     printf("    %d", choice[i].optionid);
     dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
     dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n-------------------------------------------------------\n");
}
     printf("Fund: $%.2f, Cost: $%.2f, ", F, dp1cost);
     printf("Return: $%.2f, Return Rate: %.2f%%.\n", dp1rtn, dp1rtn*100/dp1cost);
     break;
    }
    else {
     rd10=init();
     rd11=init();
     rd12=init();
     rdnum=random(2);

     if(rdnum==0) {
      SRCH1(T2_cost,s_start+1,s_end);
      break;
     }
     else {
      rd12=trans(DP[s_start+1]);
      rd11=trans(po[p_start-1]);

      for(i=p_start-2;i>-1;i--) {
       rd10=trans(po[i]);

       if(rd10.fieldid==rd11.fieldid)
       break;
      }

      if(i<0)
       rd10=init();
      fid=rd11.fieldid-1;
      op1=rd11.optionid;
```

```
op2=rd10.optionid;

if((op1+1<op2-1) && (0<=rd10.rc)) {
 t_rc1=0;

 for(j=op1+1; j<op2-1; j++) {
     t_rc1=(f1[fid].option[j-1].rtn-rd10.rtn)/
        (f1[fid].option[j-1].cost-rd10.cost);
     t_rc10=(f1[fid].option[j-2].rtn-rd10.rtn)/
        (f1[fid].option[j-2].cost-rd10.cost);

     if(rd12.rc<=t_rc1) {
      argu1=(t_rc1-t_rc10)/Las_T2;
      rdnum=random(100);
      argu1=exp(argu1)*100;

      if( rdnum < argu1 ) {
       dif_cost1=choice[fid].cost-(f1[fid].option[j].cost-rd10.cost);
       T2_cost=T2_cost+dif_cost1;
       choice[fid]=trans(f1[fid].option[j]);
       SRCH1(T2_cost,s_start+1,s_end);
       break;
      }
     }
 }
}
else if (rd12.cost!=0 && rd11.cost!=0) {
 argu1=(rd12.rtn/rd12.cost-rd11.rtn/rd11.cost)/Las_T2;
 rdnum=random(100);
 argu1=exp(argu1)*100;

 if( rdnum < argu1 ) {
     dif_cost1=f1[fid].option[op1-1].cost*f1[fid].option[op1-1].fieldsize;
     T2_cost=T2_cost+dif_cost1;

     if(0<rd10.rc)
      choice[fid]=trans(rd10.optionid);
     else
      choice[fid]=init();
     SRCH1(T2_cost,s_start+1,s_end);
     break;
 }
 else {
     SRCH1(T2_cost,s_start+1,s_end);
     break;
```

```
        }
       }
      }
     }
}
else if((T2_cost-delta<=T3_cost) && (T3_cost<=T2_cost+delta)) {
    choice[fid]=trans(T3);
    po[p_start]=trans(T3);
    p_start++;
    TCost=TCost+T3_cost;
    T3_cost=0.000000;
    printf("----------------- Near Optimal Solution ----------------\n");
    printf("Field    1   2   3   4   5   6   7   8   9   10\n");
    printf("Option");

    if(N<=10) {

     for(i=0;i<N;i++) {
      printf("    %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n----------------------------------------------------------\n");
    }
    else {

     for(i=0;i<10;i++) {
      printf("    %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n----------------------------------------------------------\n");
    printf("Field    11  12  13  14  15  16  17  18  19  20\n");
    printf("Option");

     for(i=10;i<N;i++) {
      printf("    %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n----------------------------------------------------------\n");
 }
    printf("Fund: $%.2f, Cost: $%.2f, ", F, dp1cost);
    printf("Return: $%.2f, Return Rate: %.2f%%.\n", dp1rtn, dp1rtn*100/dp1cost);
    break;
```

```
}
else {
    rd10=init();
    rd11=init();
    rd12=init();
    rdnum=random(2);

    if( rdnum==0 ) {
     SRCH1(T2_cost,s_start,s_end);
     break;
    }
    else {
     rd12=trans(DP[s_start]);
     rd11=trans(po[p_start-1]);

     for(i=p_start-2;i>-1;i--) {
      rd10=trans(po[i]);

      if(rd10.fieldid==rd11.fieldid)
      break;
     }

     if(i<0)
      rd10=init();
     fid=rd11.fieldid-1;
     op1=rd11.optionid;
     op2=rd10.optionid;

     if((op1+1<op2-1) && (0<=rd10.rc)) {
      t_rc1=0;

      for(j=op1+1; j<op2-1; j++) {
       t_rc1=(f1[fid].option[j-1].rtn-rd10.rtn)/
             (f1[fid].option[j-1].cost-rd10.cost);
       t_rc10=(f1[fid].option[j-2].rtn-rd10.rtn)/
             (f1[fid].option[j-2].cost-rd10.cost);

       if(rd12.rc<=t_rc1) {
          argu1=(t_rc1-t_rc10)/Las_T2;
          rdnum=random(100);
          argu1=exp(argu1)*100;

          if( rdnum < argu1 ) {
           dif_cost1=choice[fid].cost-(f1[fid].option[j].cost-rd10.cost);
           T2_cost=T2_cost+dif_cost1;
```

```
                choice[fid]=trans(f1[fid].option[j]);
                SRCH1(T2_cost,s_start,s_end,po);
                break;
               }
             }
            }
           }
           else if (rd12.cost!=0 && rd11.cost!=0) {
            argu1=(rd12.rtn/rd12.cost-rd11.rtn/rd11.cost)/Las_T2;
            rdnum=random(100);
            argu1=exp(argu1)*100;

            if( rdnum < argu1 ) {
             dif_cost1=f1[fid].option[op1-1].cost*f1[fid].option[op1-1].fieldsize;
             T2_cost=T2_cost+dif_cost1;

             if(0<rd10.rc)
                 choice[fid]=trans(rd10.optionid);
             else
                 choice[fid]=init();
             SRCH1(T2_cost,s_start,s_end);
             break;
            }
            else {
             SRCH1(T2_cost,s_start,s_end);
             break;
            }
           }
          }
        }
      }

      if(T3_cost==0) {
       T2_cost=B-TCost;

       if(T1_cost<T2_cost-delta) {
          Las_T2=T2_cost;
          l1=next.fieldid-1;
          l2=next.optionid-1;

          if(0<l2)
           l2--;
          else
           l2=0;
          TCost=TCost+T1_cost;
```

```
    choice[l1]=trans(f1[l1].option[l2]);
    po[p_start]=trans(f1[l1].option[l2]);
    p_start++;
    T1_cost=0.000000;
    next=init();
}
else if((T2_cost-delta<=T1_cost) && (T1_cost<=T2_cost+delta)) {
    l1=next.fieldid-1;
    l2=next.optionid-1;
    choice[l1]=trans(f1[l1].option[l2]);
    po[p_start]=trans(f1[l1].option[l2]);
    p_start++;
    TCost=TCost+T1_cost;
    T1_cost=0.000000;
    next=init();
    printf("------------------ Near Optimal Solution ----------------\n");
    printf("Field   1   2   3   4   5   6   7   8   9   10\n");
    printf("Option");

    if(N<=10) {

     for(i=0;i<N;i++) {
      printf("   %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n-------------------------------------------------------\n");
    }
    else {

     for(i=0;i<10;i++) {
      printf("   %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n-------------------------------------------------------\n");
    printf("Field   11  12  13  14  15  16  17  18  19  20\n");
    printf("Option");

     for(i=10;i<N;i++) {
      printf("   %d", choice[i].optionid);
      dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
      dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
     }
    printf("\n-------------------------------------------------------\n");
```

```
 }
    printf("Fund: $%.2f, Cost: $%.2f, ", F, dp1cost);
    printf("Return: $%.2f, Return Rate: %.2f%%.\n", dp1rtn, dp1rtn*100/dp1cost);
    break;
}
else {
    rd10=init();
    rd11=init();
    rd12=init();
    rdnum=random(2);

    if(rdnum==0) {
     SRCH1(T2_cost,s_start+1,s_end);
     break;
    }
    else {
     rd12=trans(DP[s_start+1]);
     rd11=trans(po[p_start]);

     for(i=p_start-1;i>-1;i--) {
      rd10=trans(po[i]);

      if(rd10.fieldid==rd11.fieldid)
      break;
     }

     if(i<0)
      rd10=init();
     fid=rd11.fieldid-1;
     op1=rd11.optionid;
     op2=rd10.optionid;

     if((op1+1<op2-1) && (0<=rd10.rc)) {
      t_rc1=0;

      for(j=op1+1; j<op2-1; j++) {
       t_rc1=(f1[fid].option[j-1].rtn-rd10.rtn)/
            (f1[fid].option[j-1].cost-rd10.cost);
       t_rc10=(f1[fid].option[j-2].rtn-rd10.rtn)/
            (f1[fid].option[j-2].cost-rd10.cost);

       if(rd12.rc<=t_rc1) {
          argu1=(t_rc1-t_rc10)/Las_T2;
          rdnum=random(100);
          argu1=exp(argu1)*100;
```

```
                if( rdnum < argu1 ) {
                  dif_cost1=choice[fid].cost-(f1[fid].option[j].cost-rd10.cost);
                  T2_cost=T2_cost+dif_cost1;
                  choice[fid]=trans(f1[fid].option[j]);
                  SRCH1(T2_cost,s_start+1,s_end);
                  break;
                }
              }
             }
            }
            else if (rd12.cost!=0 && rd11.cost!=0) {
             argu1=(rd12.rtn/rd12.cost-rd11.rtn/rd11.cost)/Las_T2;
             printf("argu1 1 %d\n",argu1);
             rdnum=random(100);
             argu1=exp(argu1)*100;

             if( rdnum < argu1 ) {
              dif_cost1=f1[fid].option[op1-1].cost*f1[fid].option[op1-1].fieldsize;
              T2_cost=T2_cost+dif_cost1;

              if(0<rd10.rc)
                  choice[fid]=trans(rd10.optionid);
              else
                  choice[fid]=init();
              SRCH1(T2_cost,s_start+1,s_end);
              break;
             }
             else {
              SRCH1(T2_cost,s_start+1,s_end);
              break;
             }
            }
           }
  }
}

s_start++;

if(s_start==s_end) {
      printf("---------- Near Optimal Solution ----------\n");
      printf("Field   1   2   3   4   5   6   7   8   9   10\n");
      printf("Option");

      if(N<=10) {
```

```
        for(i=0;i<N;i++) {
         printf("    %d", choice[i].optionid);
         dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
         dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
        }
       printf("\n-----------------------------------------------------------\n");
       }
       else {

        for(i=0;i<10;i++) {
         printf("    %d", choice[i].optionid);
         dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
         dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
        }
       printf("\n-----------------------------------------------------------\n");
       printf("Field    11   12   13   14   15   16   17   18   19   20\n");
       printf("Option");

        for(i=10;i<N;i++) {
         printf("    %d", choice[i].optionid);
         dp1cost=dp1cost+choice[i].cost*choice[i].fieldsize;
         dp1rtn=dp1rtn+choice[i].rtn*choice[i].fieldsize;
        }
       printf("\n-----------------------------------------------------------\n");
    }
    printf("Fund: $%.2f, Cost: $%.2f, ", F, dp1cost);
    printf("Return: $%.2f, Return Rate: %.2f%%.\n", dp1rtn, dp1rtn*100/dp1cost);
    break;
   }
   next=trans(DP[s_start+1]);
   l=next.fieldid-1;
   l0=next.optionid-1;
   T1_cost=(f1[l].option[l0].cost-choice[l].cost)*next.fieldsize;
   T2_cost=B-TCost;
  }
 }

} /* end of SRCH1 */

/*  ------ end of the functions ------  */
```

APPENDIX B

INPUT TABLES OF EXAMPLE

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
1	1	10	6.6	16.24	246
1	2	10	5.5	15.37	279
1	3	10	4.4	14.50	330
1	4	10	3.3	13.68	415

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
2	1	10	6.6	9.64	146
2	2	10	5.5	9.34	170
2	3	10	4.4	9.04	205
2	4	10	3.3	8.78	266
2	5	10	2.2	5.78	263

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
3	1	8	6.6	13.95	211
3	2	8	5.5	13.28	241
3	3	8	4.4	12.61	287
3	4	8	3.3	11.98	363

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
4	1	20	6.6	10.53	160
4	2	20	5.5	10.15	185
4	3	20	4.4	9.78	222
4	4	20	3.3	9.44	286

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
5	1	40	17.6	31.5	179
5	2	40	16.5	31.12	189
5	3	40	15.4	30.74	200
5	4	40	14.3	30.4	213
5	5	40	13.2	29.76	225
5	6	40	12.1	29.52	244

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
6	1	30	15.40	29.78	193
6	2	30	14.30	29.11	204
6	3	30	13.20	28.58	216
6	4	30	12.10	28.03	232
6	5	30	11.00	27.48	250
6	6	30	9.9	26.98	273

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
7	1	15	6.6	16.24	246
7	2	15	5.5	15.37	279
7	3	15	4.4	14.50	330
7	4	15	3.3	13.68	415

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
8	1	10	6.6	9.64	146
8	2	10	5.5	9.34	170
8	3	10	4.4	9.04	205
8	4	10	3.3	8.78	266
8	5	10	2.2	5.78	263

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
9	1	8	6.6	13.95	211
9	2	8	5.5	13.28	241
9	3	8	4.4	12.61	287
9	4	8	3.3	11.98	363

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
10	1	20	6.6	10.53	160
10	2	20	5.5	10.15	185
10	3	20	4.4	9.78	222
10	4	20	3.3	9.44	286

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
11	1	40	17.6	31.5	179
11	2	40	16.5	31.12	189
11	3	40	15.4	30.74	200
11	4	40	14.3	30.4	213
11	5	40	13.2	29.76	225
11	6	40	12.1	29.52	244

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
12	1	30	15.40	29.78	193
12	2	30	14.30	29.11	204
12	3	30	13.20	28.58	216
12	4	30	12.10	28.03	232
12	5	30	11.00	27.48	250
12	6	30	9.9	26.98	273

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
13	1	5	6.6	16.24	246
13	2	5	5.5	15.37	279
13	3	5	4.4	14.50	330
13	4	5	3.3	13.68	415

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
14	1	1	6.6	9.64	146
14	2	1	5.5	9.34	170
14	3	1	4.4	9.04	205
14	4	1	3.3	8.78	266
14	5	1	2.2	5.78	263

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
15	1	8	6.6	13.95	211
15	2	8	5.5	13.28	241
15	3	8	4.4	12.61	287
15	4	8	3.3	11.98	363

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
16	1	25	6.6	10.53	160
16	2	25	5.5	10.15	185
16	3	25	4.4	9.78	222
16	4	25	3.3	9.44	286

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
17	1	40	17.6	31.5	179
17	2	40	16.5	31.12	189
17	3	40	15.4	30.74	200
17	4	40	14.3	30.4	213
17	5	40	13.2	29.76	225
17	6	40	12.1	29.52	244

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
18	1	3	15.4	29.78	193
18	2	3	14.3	29.11	204
18	3	3	13.20	28.58	216
18	4	3	12.10	28.03	232
18	5	3	11.00	27.48	250
18	6	3	9.9	26.98	273

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
19	1	25	6.6	10.53	160
19	2	25	5.5	10.15	185
19	3	25	4.4	9.78	222
19	4	25	3.3	9.44	286

Field #	Option #	Field Size (acre)	Cost $	Return $	Return Rate %
20	1	40	17.6	31.5	179
20	2	40	16.5	31.12	189
20	3	40	15.4	30.74	200
20	4	40	14.3	30.4	213
20	5	40	13.2	29.76	225
20	6	40	12.1	29.52	244

VITA

Yunlong Chen

Candidate for the Degree of

Master of Science

Thesis: A GENETIC ALGORITHM FOR SOLVING A NONLINEAR COMBINATORIAL OPTIMIZATION PROBLEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Chongqing, Sichuan, P.R. China, On January 23, 1954, the 2nd son of Boqiang and Zheren Chen.

Education: Received Bachelor of Science degree in Applied Math from Chengdu University of Science and Technology, Chengdu, P. R. China in July 1982; received Master of Science degree in Applied Math from Oklahoma State University, Stillwater, Oklahoma in December 1993, respectively. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1994.

Experience: Raised in Chongqing, Sichuan, P. R. China; employed as assistant software engineer by Jialin Machine Factory, Chongqing 1982-1985; employed as software engineer by Yuke MIS Technologies, Inc. , Chongqing 1986-1990; employed by Oklahoma State University, Department of Math and Department of Agronomy as graduate teaching assistant and as programmer, 1991-1994.