

ALGORITHMS FOR THE MINIMUM-
COST SET-COVERING PROBLEM

By

CHIN-CHIEH CHIANG

Master of Science

Oklahoma State University

Stillwater, Oklahoma

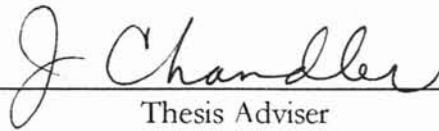
2002

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
August, 2002

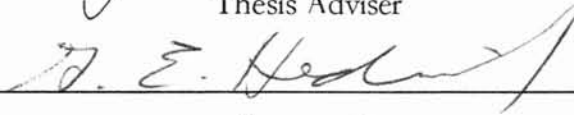
Oklahoma State University Library

ALGORITHMS FOR THE MINIMUM-COST
SET-COVERING PROBLEM

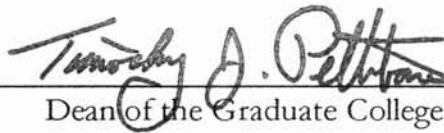
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

PREFACE

The set-covering problem is an optimization problem that models many resource-selection problems. For a smaller scale set covering problem, a greedy solution is a good approximation compared to the optimal solution. However, as the input set size gets larger, the ratio bound of the greedy solution may grow and be bounded by a logarithm function. In this paper, we explore some other algorithms and generalize the set-covering problem so that each set in the given cover has an associated cost, which is an arbitrary positive integer instead of one. Since the set-covering problem is known to be NP-complete, it is almost impossible to determine the optimal solution for any large sized problem. Indeed, solving the larger scale set covering problem would imply an enormous computational effort, in both time and computer memory. In addition, this paper investigates the performance of other algorithms, namely genetic algorithms and simulated annealing algorithms, on the minimal set-covering problem.

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. John Chandler for his intelligent supervision, constructive guidance, inspiration and friendship. My sincere appreciation extends to my other committee members Dr. Hedrick and Dr. Dai, whose guidance, assistance, encouragement, and friendship are also invaluable.

TABLE OF CONTENTS

Chapter 1 . Introduction.....	1
Chapter 2 . Problem Formulation	2
Chapter 3 . Greedy Algorithm.....	3
Chapter 4 . Optimal Algorithm	6
Chapter 5 . Genetic Algorithm.....	10
5.1 Introduction	10
5.2 Population size and initial population	11
5.3 Fitness of individuals.....	14
5.4 Parent selection technique.....	15
5.5 Crossover operator.....	15
5.6 Mutation operator	16
5.7 Population replacement	18
5.8 Overview	19
Chapter 6 . Simulated Annealing Algorithm.....	20
Chapter 7 . Computational results.....	23
Chapter 8 . Conclusions.....	28
Appendix	31

Chapter 1. Introduction

Let X be a set of m elements, and let F be a collection of subsets $S_j \subset X$, $1 \leq j \leq n$ with $X = \bigcup_{j=1}^n S_j$. Set covering is the problem of finding the smallest number of subsets in F that cover all elements.

Now, we generalize the set-covering problem so that each set S_j in the family F has an associated cost and the cost of a cover C is the sum of the costs associated with the chosen subset $S_j \in C$. The minimal cost set covering is the problem of finding the cover C with the smallest cost.

Because the problem of selecting a minimum number of subsets that cover all elements in X is NP-complete, adding the constraint of minimal cost make the problem more complicated. In the rest of the paper, we will focus on the general case and explore four algorithms, a greedy approximation, an optimization algorithm using the method of stacks and splits, a genetic algorithm and a simulated annealing algorithm.

Chapter 2 . Problem Formulation

We first introduce some notations. Let $X = \{1, 2, 3, \dots, m\}$ and let $F = \{S_1, S_2, S_3, \dots, S_n\}$ with $X = \bigcup_{j=1}^n S_j$. For each S_j , there is an associated cost $c_j = \text{cost}(S_j)$, $j = 1, \dots, n$. We will assume that for each $i \in X$, there is at least one $S_j \in F$ covering i . Let $\mathbf{M} = [a_{ij}]$ be the $m \times n$ matrix whose j -th column is the characteristic vector of S_j . In other words, $a_{ij} = 1$, if $i \in S_j$, and $a_{ij} = 0$, otherwise. Furthermore, let the column vector $\mathbf{c} = (c_j)_{n \times 1}$ denote the cost vector. Finally, we have the following set-covering problem:

$$\text{Minimize } \mathbf{c}^T \mathbf{y}$$

Subject to

$$\mathbf{M} \mathbf{y} \geq \mathbf{1}$$

$$y_j \in \{0, 1\}, \quad j = 1, \dots, n$$

Note that the problem is similar to an integer programming problem. The column vector $\mathbf{y} = (y_j)_{n \times 1}$ represents an instance in the solution space, where $y_j = 1$ if S_j is selected, and $y_j = 0$ otherwise.

Chapter 3 . Greedy Algorithm

The technique of a greedy algorithm for the set-covering problem is as follows: iteratively, select the set that covers the most yet uncovered elements, until all elements are covered. As for the minimal cost set-covering problem, each set has a cost, and one needs to find a collection of sets of smallest total cost that covers all elements. We use a greedy algorithm for this problem, but instead of picking a set that covers the most yet uncovered elements, we pick the set with the smallest average cost, which is the cost of the set divided by the number of yet uncovered elements that it contains.

Greedy Algorithm (X, F)

1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. $\text{cost} \leftarrow 0$
4. **while** $U \neq \phi$
5. **do** select an $S \in F$ that minimizes $\frac{\text{cost}(S)}{|S \cap U|}$
6. $U \leftarrow U \setminus S$
7. $C \leftarrow C \cup \{S\}$
8. $\text{cost} \leftarrow \text{cost} + \text{cost}(S)$ //end while loop
9. **return** C, cost

In the rest of this section, we will discuss the ratio bound of the above greedy algorithm. Let $H(d) = \sum_{i=1}^d \frac{1}{i}$ be the d -th harmonic number. Let C be the set cover obtained by the above algorithm and let C^* be the minimal cost (optimal) set cover. Then the ratio bound ρ would be $\text{cost}(C)/\text{cost}(C^*)$.

Let $S_1, S_2, S_3, \dots, S_k$ be the sequence of sets $\in F$ chosen by greedy algorithm such that $\bigcup_{j=1}^k S_j = X$. Therefore, $\text{cost}(C) = \sum_{j=1}^k \text{cost}(S_j)$. Using the above sequence of sets, a new partition of X can be defined inductively as follows:

$$S'_1 = S_1$$

$$S'_j = S_j \setminus \bigcup_{l=1}^{j-1} S_l \quad (1 < j \leq k)$$

Let $\text{cost}(x) = \frac{\text{cost}(S_j)}{|S'_j|}$ for $x \in S'_j$, $1 \leq j \leq k$. Then $\text{cost}(S_j) = \text{cost}(S'_j)$, $1 \leq j \leq k$. Let

$x_1, x_2, x_3, \dots, x_m$ be a sequence of all elements of X in the ascending cost order. To see the upper bound of $\text{cost}(x)$: Given x_j , there is a unique S'_b for some b , $1 \leq b \leq k$, such that $x_j \in S'_b$. Thus, $x_j, x_{j+1}, \dots, x_m \in S_b \cup S_{b+1} \cup \dots \cup S_k$, and hence

$$|S_b \cup S_{b+1} \cup \dots \cup S_k| \geq m - j + 1.$$

Observe that $\text{cost}(x_j) = \frac{\text{cost}(S_b)}{|S'_b|} = \min \left\{ \frac{\text{cost}(S)}{|S \setminus (S_1 \cup \dots \cup S_{b-1})|} : S \in F \setminus \{S_1, \dots, S_{b-1}\} \right\}$

$$\leq \min \left\{ \frac{\text{cost}(S)}{|S \setminus (S_1 \cup \dots \cup S_{b-1})|} : S \in C^* \setminus \{S_1 \cup \dots \cup S_{b-1}\} \equiv \Delta \right\}$$

$$\leq \frac{\sum_{S \in \Delta} \text{cost}(S)}{\sum_{S \in \Delta} |S \setminus (S_1 \cup \dots \cup S_{b-1})|} \leq \frac{\text{cost}(C^*)}{m - j + 1}.$$

Therefore,

$$\text{cost}(C) = \sum_{j=1}^m \text{cost}(x_j) \leq \sum_{j=1}^m \frac{\text{cost}(C^*)}{m-j+1} = \text{cost}(C^*) \cdot H(m) = \text{cost}(C^*) \cdot H(|X|)$$

So, the ratio bound $\rho = H(|X|)$.

Moreover, $\rho = H(|X|) \leq 1 + \int_1^m \frac{1}{x} dx = 1 + \ln|X|$.

The ratio bound given in [6] is $H(\max|S| : S \in F)$ with the assumption that the cost of each set is one. Thus, the gap between the approximation solution and the optimal solution gets larger if the constraint of costs is considered.

Chapter 4 . Optimal Algorithm

When searching for the optimal solution to a problem that has many feasible solutions, the search may progress in one of several ways, depending on the structure of the problem. The approaches are exhaustive search, dynamic programming, a greedy algorithm, and branch and bound. The algorithm we present here uses the method of splits and stacks to reach the optimal solution with minimal cost that covers the given set.

Recall in chapter 2 that the set-covering problem can be formulated as covering at a minimal cost all the rows of an $m \times n$ matrix of ones and zeros by a subset of columns. That is,

$$\text{Minimize } \mathbf{c}^T \mathbf{y}$$

Subject to

$$\mathbf{M}\mathbf{y} \geq \mathbf{1}$$

$$y_j \in \{0, 1\}, \quad j=1, \dots, n$$

For convenience in developing the optimal algorithm, we may assume that the entries in the cost vector \mathbf{c} are non-decreasing with $c_1 \leq c_2 \leq \dots \leq c_n$. In practice, for implementing this algorithm, we can find a permutation p on $\{1, 2, 3, \dots, n\}$ such that $c_{p(1)} \leq c_{p(2)} \leq \dots \leq c_{p(n)}$ taking time $O(n \ln n)$ with any of the sorting methods introduced in [6].

Let $\mathbf{Y} = \{0, 1\}^n$, where $n = |F|$, then $|\mathbf{Y}| = 2^n$. An instance $\mathbf{y} = (y_j) \in \mathbf{Y}$ represents a selection of $\{S_1, S_2, \dots, S_n\}$ such that $y_j = 1$ if S_j is selected, and $y_j = 0$ otherwise.

We say \mathbf{y} is a feasible solution if the union of corresponding subsets is a cover of X . Also that \mathbf{y} has a corresponding cost, namely $\text{cost}(\mathbf{y}) = \sum_{j=1}^n y_j \cdot \text{cost}(S_j)$.

The search space is \mathbf{Y} with 2^n elements. The trivial solution is $\mathbf{1} = (1,1,1,\dots,1)$ and the infeasible solution is $\mathbf{0} = (0,0,0,\dots,0)$ in any set covering problem. Our tracking for getting the optimal solution begins with $\mathbf{1} = (1,1,1,\dots,1)$. For any given parent $\mathbf{y} = (y_1, y_2, y_3, \dots, y_n)$, flip each entry (bit) of \mathbf{y} to generate a child and continue this process to generate all children of \mathbf{y} until flipping meets the first 0 or reaches the end.

For instance, if the parent is (11011), then the first child is (01011) obtained by flipping the first entry of the parent, the second child is (10011) by flipping the second entry of the parent, and then stop since the third entry of the parent is 0. If the parent is (11111), then there are five children obtained by flipping each entry of the parent. Note that the Hamming distance of the parent and any of the children is one.

Split Algorithm (*parent*)

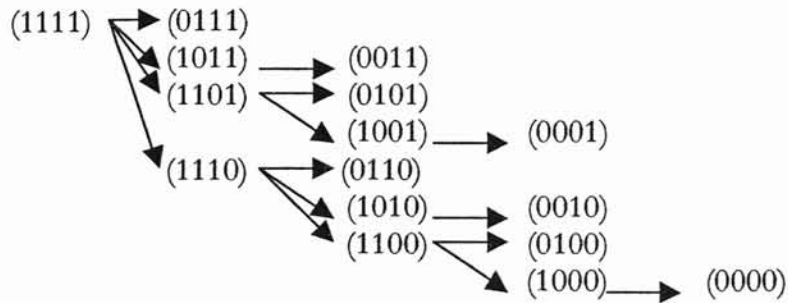
```

1   $i \leftarrow 1$ 
2  while ( $parent[i] \neq 0$  and  $i \leq n$ )
3      do for  $j \leftarrow 1$  to  $n$ 
4          do  $child[j] \leftarrow parent[j]$  //end for loop
5           $child[i] \leftarrow 0$ 
6           $i \leftarrow i + 1$  //end while loop
```

In the process of generating children, the set covering cost corresponding to each child is non-increasing, and the set covering cost of each child is less than

that of the parent. Besides, the union of subsets corresponding to each child is contained in that of the parent. Below example shows the implementation of the Split Algorithm:

Example: $n = 4$



From above example, it is easy to see that the tree generated by the Split Algorithm has the following properties:

1. The set covering cost among the sibling is non-increasing in the order of their ages.
2. If the parent is infeasible, then all its children are infeasible.

Property 2 removes infeasible solutions from the search space and hence improve the convergence of optimal solution. By virtue of property 1, children among the sibling are pushed onto a stack in the order of their ages to maintain the top one in the stack is optimal at the run time. Note that all items in the stack are feasible solutions and the youngest child (minimal cost) is always the top one at that moment.

Optimal Algorithm

```
1. optimal ← 1
2. Push(Stack, 1)
3. while stack ≠  $\phi$ 
4.   do y ← Pop(Stack)
5.     if y is not reducible
6.       then update optimal
7.     else while y is reducible
8.       do Push(Stack, y.child) // end inner while loop
9.       //end outer while loop
10. return optimal
```

In line 5, we define that \mathbf{y} is reducible if it has at least one child which is a feasible solution. Thus, if the child is \mathbf{z} and $\mathbf{Mz} \geq \mathbf{1}$, then \mathbf{y} is reducible. The computation of $\mathbf{Mz} \geq \mathbf{1}$ takes time $\Theta(mn)$, but it can be improved to $\Theta(m)$. To see this: if we already have the result of \mathbf{My} , we can get the result of \mathbf{Mz} by subtracting some column vector of \mathbf{M} from \mathbf{My} since the Hamming distance between \mathbf{y} and \mathbf{z} is one.

Chapter 5 . Genetic Algorithm

5.1 Introduction

The theoretical foundations of genetic algorithms (GAs) were originally developed by Holland. He began his work on genetic algorithms in the early 60s. A first achievement was the publication of *Adaptation in Natural and Artificial Systems* [14] in 1975. Later, David Goldberg, a student of John Holland, is one of the preeminent researchers in the field. David Goldberg's *Genetic Algorithms in Search, Optimization and Machine Learning* [12] is by far the best introduction to genetic algorithms.

The idea of GAs is based on the evolutionary process of biological organisms in nature. During the process of evolution, natural populations evolve according to the principals of natural selection and “survival of the fittest”. Individuals which are more successful in adapting to their environment will have a better chance of surviving and reproducing, while individuals which are less successful will be eliminated. This means that the genes from the highly fit individuals will spread to an increasing number of individuals in each successive generation. The combination of good characteristics from highly adapted ancestors may produce even more fit offspring.

A GA simulates these processes by taking an initial population of individuals and then applying genetic operators in each reproduction. Each individual in this population is encoded into a string or chromosome which represents a feasible solution to a given problem. The fitness of an individual is evaluated with respect to a given objective function. Highly fit individuals or solutions are given opportunities to reproduce with other highly fit individuals by exchanging pieces

of their genetic information. This crossover procedure produces new offspring solutions, which share some characteristics taken from both parents. Mutation is often applied after crossover by altering some genes in the chromosomes. The offspring can either replace the whole population or replace less fit individuals. This evaluation-selection-reproduction cycle is repeated until a satisfactory solution is found. Below are the basic steps of a simple Genetic Algorithm. The more detailed of each step will be discussed later.

Genetic Algorithm

1. Generate an initial population
2. Evaluate fitness of individuals in the population
3. Repeat
 4. Select parents from the population
 5. Recombine parents to produce children
 6. Evaluate fitness of the children
 7. Replace some or all of the population by the children
8. Until a satisfactory solution has been found

5.2 Population size and initial population

The size of the population and the initial population are chosen such that the solution domain is suitably covered by the population. To determine the size of population, we need to consider the density of a set-covering problem which is the ratio of ones in the matrix. If the density of a randomly generated SCP is ρ , then the average number of ones in each solution is $\frac{1}{\rho}$. In order for the union of individuals in the population to cover the entire solution domain, we need to

have the population size at least N such that $N \times \frac{1}{\rho} > n$, where n is the column size of the matrix m by n . Therefore, the population size we choose here is $cn\rho$ for some positive integer $c \geq 1$.

In the process of initializing the population, we expect each individual is generated randomly, but the individual may be not a feasible solution. To conquer this, extra variables to record the information for each row and column are declared first in order to modify each randomly generated individual and make it a feasible solution. We define those variables as:

Let I = the set of all rows
 J = the set of all columns
 $r_i = \{j \in J \mid \text{column } j \text{ covers row } i\}$, for each $i \in I$
 $c_j = \{i \in I \mid \text{row } i \text{ covers column } j\}$, for each $j \in J$
 $S = \{j \in J \mid \text{column } j = 1\}$ with respect to an individual
 $U = \{i \in I \mid \text{row } i \text{ is not covered yet}\}$ with respect to S
 $w_i = |\{j \in S \mid \text{column } j \text{ covers row } i\}|$, for each $i \in I$

Once the population size is determined, the way of generating each individual is by stepping through each row of the matrix and selecting a column randomly which covers that row. This would guarantee that each individual is a feasible solution. We have the following algorithm:

Algorithm5.1.1

1. $S \leftarrow \phi$
 $w_i \leftarrow 0$ for each $i \in I$
2. **for** each row $i \in I$ **do**
 - a. randomly select a column $j \in r_i$
 - b. **if** $j \notin S$ **then do**
 - $S \leftarrow S \cup \{j\}$
 - $w_i \leftarrow w_i + 1$ for each $i \in c_j$

By the above algorithm, it is easy to see that each row would be covered by at least one column in the resultant S . On the other hand, the resultant S may contain redundant columns, so we used extra variables w_i in the algorithm to update the row weight (number of columns in S that cover row i) whenever a new column is added to S .

It is straightforward to remove redundant columns in the resultant S by stepping through each column in S and checking the rows that it covers. If the row weights w_i for those rows that it covers are all greater than or equal to 2, then the column is redundant and safe to be removed. So, we have the following algorithm:

Algorithm 5.1.2 (feasible S)

1. **for** each $j \in S$ in increasing order
2. **if** $w_i \geq 2$ **for** each $i \in c_j$
3. **then**
 - a. $S \leftarrow S \setminus \{j\}$
 - b. $w_i \leftarrow w_i - 1$ for each $i \in c_j$

Without loss of generality, recall that we assume the cost of each column is in non-decreasing order. Therefore, if we modified step 1 in above algorithm by selecting $j \in S$ in decreasing order, then the resultant S would be a cover with smaller cost. However, when we initial the population, we prefer each individual is randomly generated without redundant columns, and thus the population can cover the solution domain. The cost of each individual is not a big issue at this moment, because other operators, like crossover and mutation, will be applied later to produce offspring and thus for those unfit individuals with heavy cost will be replaced by their offspring. A slight modification for the above algorithm is given as below:

Algorithm5.1.3 (feasible S)

1. $tmpS \leftarrow S$
2. **while** $tmpS \neq \emptyset$ **do**
3. select $j \in tmpS$ randomly
4. $tmpS \leftarrow tmpS \setminus \{j\}$
5. **if** $w_i \geq 2$ **for** each $i \in c_j$ **then**
 - a. $S \leftarrow S \setminus \{j\}$
 - b. $w_i \leftarrow w_i - 1$ for each $i \in c_j$

Finally, we have the following algorithm to generate the initial population:

Initial Population Algorithm

1. **for** $i \leftarrow 1$ to $popuSizeN$
 - a. Algorithm5.1.1
 - b. Algorithm5.1.3

5.3 Fitness of individuals

Once the population is initialized, we might consider this population as a matrix P with size $N \times n$, where N the population size, and n the number of columns in the set covering problem. The binary representation of an individual's chromosome is a n -bit string. A value of 1 for the j -th bit implies the column j is in the individual S (solution). The fitness of an individual is directly related to its objective function value. With the binary representation, the fitness f_i of an individual i (the i -th row of matrix P) is calculated simply by

$f_i = \sum_{j=1}^n \text{cost}(j) * P[i][j]$ where $\text{cost}(j)$ the cost of column j and $P[i][j]$ the ij -entry in matrix P . A lower fitness individual means less cost.

5.4 Parent selection technique

Parent selection is the task of assigning reproductive opportunities to each individual in the population. The method used widely is proportionate selection. An individual with lower fitness (cost) will have higher chance of being selected. The probability of an individual i being selected is based on the probability distribution, that is

$$\text{Pr}(i) = \frac{1/f_i}{\sum_{i=1}^N 1/f_i}, \text{ where } f_i \text{ is the fitness of the } i\text{-th individual and } N \text{ is the}$$

population size. For each mating, two different individuals will be selected in the population for reproduction via crossover and mutation.

5.5 Crossover operator

The crossover operator works by randomly generating one crossover point and then swapping segments of two parent strings to produce two child strings. For example,

Let $p_1 = a_1 a_2 \dots a_n$, $p_2 = b_1 b_2 \dots b_n$ be two parent strings. Generate a crossover point $k, 1 \leq k < n$. Then the two child strings are

$c_1 = a_1 \dots a_k b_{k+1} \dots b_n$, $c_2 = b_1 \dots b_k a_{k+1} \dots a_n$. The purpose of applying the crossover operator to the parent strings is to generate two child strings and to see if the child strings have better fitness. Since the crossover point k is generated randomly in $[1, n)$, either one of the child strings could be identical to one or other of the parent strings. To avoid this, the range of choosing k must be restricted as below:

$$\min\{j \mid a_j \neq b_j\} \leq k < \max\{j \mid a_j \neq b_j\}$$

Note that the child strings might not be feasible solutions. The mutation operator would be applied first before we make them into feasible solutions.

5.6 Mutation operator

The mutation operator is applied to each offspring after crossover. It is another important operator that randomly changes the genes present in the chromosome. The expected number of changes, called the mutation rate, is either fixed or varies according to a certain criterion. Mutation is considered as a background operator that provides the ability to allow for a small amount of random search. It also helps to guard against the loss of important genetic information that is either not present at all in the initial population, or has been discarded during earlier evolutionary process. Thus, it reintroduces valuable

genetic information into the current population and allows the genetic algorithm to explore the new search space.

Notice that the offspring generated by the crossover and mutation operators may violate the problem constraints that some rows may not be covered. We use a repair algorithm presented by Beasley [4] to maintain feasible solutions. This repair algorithm is responsible for covering uncovered rows. To make an offspring feasible, additional operators are needed. The notations will be the same as we used before:

- Let I = the set of all rows
 J = the set of all columns
 $r_i = \{j \in J \mid \text{column } j \text{ covers row } i\}$, for each $i \in I$
 $c_j = \{i \in I \mid \text{row } i \text{ covers column } j\}$, for each $j \in J$
 $S = \{j \in J \mid \text{column } j = 1\}$ with respect to an individual
 $U = \{i \in I \mid \text{row } i \text{ is not covered yet}\}$ with respect to S
 $w_i = |\{j \in S \mid \text{column } j \text{ covers row } i\}|$, for each $i \in I$

The algorithm below searches for additional columns based on the ratio between the cost of a column and the number of uncovered rows which it covers, and hence it makes an offspring S into a feasible solution:

Algorithm 5.5.1 (offspring S)

1. $w_i \leftarrow |S \cap r_i|$, for all $i \in I$
2. $U = \{i \in I \mid w_i = 0\}$
3. **for** each $i \in U$
 - a. find the column $j \in r_i$ such that $\text{cost}(j) / |U \cap c_j|$ is minimum
 - b. $S \leftarrow S \cup \{j\}$
 - c. $w_i \leftarrow w_i + 1$ for each $i \in c_j$
 - d. $U \leftarrow U \setminus c_j$

It is interesting to note that once the repair process is completed and an infeasible individual is converted into a feasible one, then a local optimization step can be performed by the following algorithm which removes the redundant columns from a feasible solution:

Algorithm 5.5.2 (feasible S)

1. **for** each $j \in S$ in decreasing order
2. **if** $w_i \geq 2$ **for** each $i \in c_j$
3. **then**
 - a. $S \leftarrow S \setminus \{j\}$
 - b. $w_i \leftarrow w_i - 1$ **for** each $i \in c_j$

In line 1 of algorithm 5.5.2, columns with higher costs are dropped first.

5.7 Population replacement

The replacement strategy for a genetic algorithm not only decides which individuals that shall be replaced by the new offspring, but it also establishes the types of replacement. The types of commonly used method are steady-state replacement and generational replacement. The method of the former one is that once a new feasible child solution is generated, the child will replace a randomly chosen member in the population. In our implementation, we use the latter method, in which the whole parent generation is replaced by a new population of children.

5.8 Overview

The following steps are used in our GA for the SCP:

1. Generate an initial population of N random feasible solutions.
2. Select two solution p_1 and p_2 randomly from the population.
3. Combine p_1 and p_2 to form two new solutions c_1 and c_2 using the crossover operator.
4. Mutate one randomly selected column in each c_i , and then make them feasible and remove redundant columns.
5. If c_i is identical to any one of the solutions in the population or more cost than p_i , place p_i back into the population; otherwise c_i replace p_i and insert into the population.
6. Repeat steps 2-5 until a satisfactory solution has been found.

Chapter 6 . Simulated Annealing Algorithm

Annealing is a technique that was adopted from the thermodynamic process. Simulated annealing is a problem solving technique based on the way in which a metal is annealed in order to increase its strength. When a heated metal is cooled very slowly, it freezes into a regular crystalline structure.

The objective here is that the energy of the crystalline solid should be a minimum as in a perfect solid. Once the temperature is increased to a very high value, a cooling schedule is adopted which is applied to the molten solid. That is, the temperature is decreased in a regular fashion or decremented by a certain value that results in a new state. As the temperature is decreased, if the energy of the new state is less than the energy of the previous state, the new state is accepted. If the decrease in temperature increases the energy of the new state, the temperature is increased to its previous value. This process is repeated until a crystalline solid is formed that is in a minimum energy state.

This idea has been applied to a number of optimization problems. The basic idea of this approach is to start with an initial solution to the problem and to improve the solution iteratively. The more iterations, the better the solution produced.

A simulated annealing algorithm searches for the optimum solution to a given problem in an analogous way. Specifically, it moves randomly in the solution space looking for a solution that minimizes the value of some objective function. Because it is generated randomly, a given move may cause the object function to increase, to decrease, or to remain unchanged.

A simulated annealing algorithm always accepts moves that decrease the value of the object function. Moves that increase the value of the objective function are accepted with the probability

$$p = e^{-\Delta/T}$$

where Δ is the change in the value of the object function and T is a control parameter called the temperature. A random number generator that generates numbers distributed uniformly on the interval $[0,1]$ is sampled, and if the sample is less than p , the move is accepted.

By analogy with the physical process, the temperature T is initial by high. Therefore, the probability of accepting a move that increases the objective function is initial by high. The temperature is gradually decreased as the search processes. In the end, the probability of accepting a move that increases the objective function becomes very small. In general, the temperature is lowered in accordance with an annealing schedule.

The most commonly used annealing schedule is called exponential cooling. Exponential cooling begins at some initial temperature, T_0 , and decreases the temperature in steps according to $T_{k+1} = \alpha T_k$, where $0 < \alpha < 1$. Typically, a fixed number of moves must be accepted at each temperature before proceeding to the next temperature. The algorithm terminates either when the temperature reaches some final value T_f , or when some other stopping criterion has been met.

The choice of suitable values for α , T_0 , and T_f is highly problem-dependent. However, empirical evidence suggests that a good value for α is 0.95 and that T_0 should be chosen so that the initial acceptance probability is 0.8.

Finally, there is the question of selecting the initial solution from which to begin the search. A key requirement is that it be generated quickly. Therefore, the initial solution we select here is the result of a greedy algorithm.

Simulated Annealing Algorithm

1. Initialize T_0 , T_f , and α
2. $S_{curr} \leftarrow$ greedy solution
3. **while** $T_0 > T_f$ **do**
 - a. Select next solution S_{next} in the neighborhood of S_{curr}
 - b. **if** $\text{cost}(S_{next}) < \text{cost}(S_{curr})$
then $S_{curr} \leftarrow S_{next}$
else
if $\text{random}[0, 1] < \exp\{\text{cost}(S_{curr}) - \text{cost}(S_{next})/T_0\}$
then $S_{curr} \leftarrow S_{next}$
 - c. $T_0 \leftarrow \alpha T_0$

In step 3c, the method of generating uniformly random real numbers between 0 and 1 was introduced by Dr. Chandler [5]. D.H. Lehmer's multiplicative congruent method is used to generate one long cycle that contains every integer value of seed from 1 to 2147483646 exactly once. From that method, we can produce real numbers distributed uniformly on the desired interval.

Chapter 7 . Computational results

The algorithms presented in this paper were implemented in C++ and tested on a PC with 128 MB of RAM and a CPU speed of 700 MHz. The test problem sets were obtained electronically from OR-Library posted by J.E. Beasley. The program code was tested on various sizes and densities. The set-covering problem was solved by the approximation of each algorithm with up to 400 rows and 4,000 columns.

Below table shows the SCP test files that we used in our implementation. The density is determined from the Genetic Algorithm.

No.	File Name	Size(Row by column)	Density	Unicost
1	Scp41.txt	200 * 1000	0.020035	N
2	Scp410.txt	200 * 1000	0.01951	N
3	Scp42.txt	200 * 1000	0.019895	N
4	Scp43.txt	200 * 1000	0.019915	N
5	Scp44.txt	200 * 1000	0.020035	N
6	Scp45.txt	200 * 1000	0.019685	N
7	Scp46.txt	200 * 1000	0.0204	N
8	Scp47.txt	200 * 1000	0.019585	N
9	Scp48.txt	200 * 1000	0.020075	N
10	Scp49.txt	200 * 1000	0.019765	N
11	Scp51.txt	200 * 2000	0.0199825	N
12	Scp510.txt	200 * 2000	0.0199975	N
13	Scp52.txt	200 * 2000	0.0199875	N
14	Scp53.txt	200 * 2000	0.020035	N
15	Scp54.txt	200 * 2000	0.019835	N
16	Scp55.txt	200 * 2000	0.019635	N
17	Scp56.txt	200 * 2000	0.0199825	N
18	Scp57.txt	200 * 2000	0.0201425	N
19	Scp58.txt	200 * 2000	0.0198	N
20	Scp59.txt	200 * 2000	0.019675	N
21	Scp61.txt	200 * 1000	0.017025	N
22	Scp62.txt	200 * 1000	0.0178075	N
23	Scp63.txt	200 * 1000	0.01814	N
24	Scp64.txt	200 * 1000	0.017315	N

25	Scp65.txt	200 * 1000	0.017175	N
26	Scpa1.txt	300 * 3000	0.0201	N
27	Scpa2.txt	300 * 3000	0.0200789	N
28	Scpa3.txt	300 * 3000	0.0200833	N
29	Scpa4.txt	300 * 3000	0.02009	N
30	Scpa5.txt	300 * 3000	0.0200767	N
31	Scpb1.txt	300 * 3000	0.0499022	N
32	Scpb2.txt	300 * 3000	0.0498456	N
33	Scpb3.txt	300 * 3000	0.0498844	N
34	Scpb4.txt	300 * 3000	0.0498722	N
35	Scpb5.txt	300 * 3000	0.04986	N
36	Scpc1.txt	400 * 4000	0.020025	N
37	Scpc2.txt	400 * 4000	0.0199694	N
38	Scpc3.txt	400 * 4000	0.0199794	N
39	Scpc4.txt	400 * 4000	0.01998	N
40	Scpc5.txt	400 * 4000	0.0199706	N
41	Scpd1.txt	400 * 4000	0.0500819	N
42	Scpd2.txt	400 * 4000	0.0500581	N
43	Scpd2.txt	400 * 4000	0.050095	N
44	Scpd4.txt	400 * 4000	0.0500169	N
45	Scpd5.txt	400 * 4000	0.0500381	N
46	Scpe1.txt	50 * 500	0.19648	Y
47	Scpe2.txt	50 * 500	0.2004	Y
48	Scpe3.txt	50 * 500	0.20156	Y
49	Scpe4.txt	50 * 500	0.19796	Y
50	Scpe5.txt	50 * 500	0.20052	Y

The density of a SCP is the ratio of ones in the matrix.

The table below shows the result of cost and execution time for each algorithm on five SCP's with size 50*500, density near 0.2, and unicast. The initial population size of the Genetic Algorithm was based on the density of SCP and the number of iterations is set to 500.

File No.	G cost	G time	SA cost	SA time	GA cost	GA time	GA pop
46	5	0	5	0	6	0	98
47	5	0	5	0	5	0	100
48	5	0	5	0	5	0	100
49	6	0	5	0	5	0	98
50	5	0	5	0	5	0	100

Notations: G (Greedy), SA (Simulated Annealing), GA (Genetic),
 Pop (initial population size).
 0 means the execution is less than one second.

Similarly, we have the results tested on different set sizes various from 200*1000 to 400*4000. Also, we set the number of iterations for the Genetic Algorithm to 500.

File No.	G cost	G time	SA cost	SA time	GA cost	GA time	GA pop
1	467	1	437	0	383	0	20
2	559	1	531	0	469	0	19
3	585	2	530	0	449	0	19
4	595	2	525	0	446	0	19
5	552	2	509	0	451	0	20
6	581	1	512	0	470	0	19
7	609	1	588	0	502	1	20
8	484	1	446	0	408	0	19
9	538	1	523	1	458	0	20
10	745	1	669	0	612	0	19
11	290	2	270	0	224	0	39
12	294	3	276	0	249	0	39
13	349	2	331	0	289	1	39
14	249	2	235	0	208	1	40
15	267	2	252	0	225	0	39
16	237	3	210	0	205	0	39

17	251	3	224	0	195	0	39
18	327	3	301	0	275	0	40
19	324	3	312	0	258	0	39
20	308	2	289	0	267	0	39
21	101	0	101	0	100	0	34
22	120	0	118	0	109	1	35
23	106	0	106	0	101	1	36
24	104	0	103	0	103	0	34
25	108	0	107	0	103	0	34
26	288	6	261	0	233	2	60
27	285	6	271	0	260	1	60
28	270	6	245	0	232	1	60
29	279	5	242	1	228	1	60
30	270	6	247	0	227	1	60
31	77	3	73	0	139	2	149
32	86	4	78	0	138	2	149
33	90	3	88	1	134	2	149
34	91	3	82	0	116	2	149
35	79	3	76	0	142	2	149
36	258	12	237	0	237	3	80
37	249	12	220	0	232	2	79
38	277	11	258	0	250	2	79
39	257	11	237	0	245	3	79
40	234	11	220	0	223	2	79
41	71	5	68	0	271	4	200
42	73	5	70	0	212	4	200
43	83	6	81	0	181	4	200
44	69	6	66	1	190	4	200
45	69	5	66	0	266	4	200

From the above table, we can see the costs and execution times for Genetic Algorithm were performed better than the other two algorithms on test files (No. 1-30 and 36-40).

The densities of those test files (No. 1-30, 36-40) are less than or around 0.02. For those test files (No. 31-35, 41-45), the densities are around 0.05, and the performance of the Genetic Algorithm was weaker. To improve this, the results are in below table as we set the number of iterations to 1000, 1500, respectively.

File No.	Greedy		T = 500		T = 1000		T = 1500	
	cost	time	cost	time	cost	time	cost	time
31	77	3	139	2	71	4	68	4
32	86	4	138	2	81	4	73	5
33	90	3	134	2	78	3	75	5
34	91	3	116	2	75	3	75	5
35	79	3	142	2	77	4	68	5
41	71	5	271	4	65	6	60	9
42	73	5	212	4	77	7	65	9
43	83	6	181	4	77	7	72	10
44	69	6	190	4	72	7	61	10
45	69	5	266	4	67	7	62	9

Greedy vs. GA for each T, where T the number of iterations in GA.

Chapter 8 . Conclusions

The approach was implemented and tested on various set sizes and densities to compare the costs and execution times for each algorithm. We have presented the results from our experiments with a greedy algorithm, a simulated annealing algorithm, and genetic algorithm. The simulated annealing algorithm takes the greedy solution generated by the greedy algorithm as the initial solution and then randomly search the solution space to improve the cost within a very short time. The genetic algorithm is a very powerful method which can obtain very good results in a reasonable time in comparison with the other two algorithms.

References

- [1] A.O.L. Atkin, B.J. Birch, *Computers in Number Theory*, Academic Press Inc. (London) Ltd., 1971
- [2] T. Back, *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, Inc., 1996
- [3] J-P. Barthelemy, G. Cohen, A. Lobstein, *Algorithmic Complexity*, pp. 79-104, UCL Press Limited, 1996
- [4] J.E. Beasley and P.C. Chu, A genetic algorithm for the set covering problem. the European Journal of Operational Research, June 1995.
- [5] John P. Chandler, class notes, unpublished.
- [6] T. H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, pp. 974-985, The MIT Press, 1997
- [7] W.I. Cook, W.H. Cunningham, W.R. Pulleyblank, A.Schrijver, *Combinatorial Optimization*, John Wiley & Sons, Inc., 1998
- [8] A. Dolan, J. Aldous, *Networks and Algorithms*, pp. 403-411, John Wiley & Sons, Ltd., 1993
- [9] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, 1979
- [10] K.O. Geddes, S.R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1993
- [11] M. Gen, R. Cheng, *Genetic Algorithms and Engineering Design*, John Wiley & Sons, Inc., 1996
- [12] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub Co, 1989
- [13] R.L Haupt, S.E. Haupt, *Practical Genetic Algorithms*, John Wiley & Sons, Inc., 1998
- [14] J.H. Holland, *Adaptation in Natural and Artificial Systems*, Bradford Books, 1975
- [15] J.H. Holland, *Emergence from Chaos to Order*, Perseus Press, 1998
- [16] J.H. Holland, Genetic Algorithm, Scientific American, July 1992
- [17] R. Horst, P.M. Pardalos, *Handbook of Global Optimization*, Kluwer Academic Publishers, 1995

- [18] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs* , Springer Verlag, 1996
- [19] M.W. Padberg, M.P. Rijal, *Location, Scheduling, Design and Integer Programming*, Kluwer Academic Publishers, 1996
- [20] R. Sedgewick, *An Introduction to Analysis of Algorithms*, Addison-Wesley Pub. Co, 1996
- [21] R. Sedgewick, *Algorithms*, Addison-Wesley Pub Co, 1998
- [22] W.T. Trotter, *Combinatorics and Partially Ordered Sets*, The Johns Hopkins University Press, 1992
- [23] A. Tucker, *Applied Combinatorics*, John Wiley & Sons, 1980
- [24] D.J.A. Welsh, *Combinatorial Mathematics and its Applications* , Academic Press Inc. (London) Ltd., 1969
- [25] S.G. Williamson, *Combinatorics for Computer Science*, Computer Science Press, Inc, 1985

Appendix

C++ code for the set-covering problem:

1. data.h
2. genetic.h
3. greedy.h
4. simuAnnealing.h
5. split.h
6. stack
7. genetic.cpp
8. greedy.cpp
9. main.cpp
10. simuAnnealing.cpp
11. split.cpp
12. stack.cpp

Note: Source code (40 KB) and test files (5.13MB) are available upon request at chiangc@a.cs.okstate.edu.

Example: Sample output of scpe1.txt.....pp. 32

Vita 3

CHIN-CHIEH CHIANG

Candidate for the Degree of

Master of Science

Thesis: ALGORITHMS FOR THE MINIMUM-COST SET-COVERING PROBLEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Kaohsiung city, Taiwan.

Education: Received Bachelor of Science degree in Applied Mathematics from Feng-Chia University, Taiwan in June 1987; receive Master of Science degree in Mathematics from University of Connecticut, Storrs, Connecticut in May 1994; studied in the Ph. D. program of Mathematics at Michigan State University from 1994 to 1998.
Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in August 2002.

Experience: Taught at Kaohsiung Sr. High School, Taiwan as a Mathematics teacher from 1989 to 1992.
Employed by Upperspace Corp. Oklahoma as a software developer, 2000 to 2001.

Professional Memberships: American Mathematical Society; passed Actuarial Exam 100.