Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

4-19-1991

Using genetic algorithms to solve combinatorial optimization problems

Xinwei Cui
Florida International University

DOI: 10.25148/etd.FI14061560

Follow this and additional works at: https://digitalcommons.fiu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Cui, Xinwei, "Using genetic algorithms to solve combinatorial optimization problems" (1991). *FIU Electronic Theses and Dissertations*. 2684.
https://digitalcommons.fiu.edu/etd/2684

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

ABSTRACT OF THE THESIS

Using Genetic Algorithms to Solve Combinatorial Optimization Problems

by

Xinwei Cui

Florida International University, 1991

Miami, Florida

Professor Mark A. Weiss, Major Professor

Genetic algorithms are stochastic search techniques based on the mechanics of natural selection and natural genetics. Genetic algorithms differ from traditional analytical methods by using genetic operators and historic cumulative information to prune the search space and generate plausible solutions. Recent research has shown that genetic algorithms have a large range and growing number of applications.

The research presented in this thesis is that of using genetic algorithms to solve some typical combinatorial optimization problems, namely the Clique, Vertex Cover and Max Cut problems. All of these are NP-Complete problems. The empirical results show that genetic algorithms can provide efficient search heuristics for solving these combinatorial optimization problems.

Genetic algorithms are inherently parallel. The Connection Machine system makes parallel implementation of these inherently parallel algorithms possible. Both sequential genetic algorithms and parallel genetic algorithms for Clique, Vertex Cover and Max Cut problems have been developed and implemented on the SUN4 and the Connection Machine systems respectively.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

Using Genetic Algorithms to Solve Combinatorial Optimization Problems

A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Xinwei Cui

1991

To Professors Luis L. Cova, Masoud T. Milani, Mark A. Weiss:

This thesis, having been approved in respect to form and mechanical execution, is referred to you for judgment upon its substantial merit.

Dean Arthur W. Herriott
College of Arts and Sciences

The thesis of Xinwei Cui is approved.

Luis L. Cova

Masoud T. Milani

Mark A. Weiss, Major Professor

Date of Examination: April 19, 1991

Florida International University, 1991

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to all those who have been involved in the preparation of this thesis: to my professors who initiated, supervised, and contributed to the final product. A special thanks to Dr. Raimund K. Ege for his granting of a Connection Machine Account; a great appreciation to Dr. Jainendra K. Navlakha for providing the newest research publications in genetic algorithms; and many thanks to the inter-library loan department at Florida International University for their excellent services.

Xinwei Cui

Table of Contents

1 Introduction **1**

1.1 Statement of the Problems 1

1.2 Approach for Solving the Problems 5

1.3 An Outline of the Thesis 6

2 Introduction to Genetic Algorithms **8**

2.1 Historical Perspective 8

2.2 Concepts 8

2.3 Unique Features of Genetic Algorithms 13

2.4 Mathematical Foundations of Genetic Algorithms 14

2.5 Genetic Algorithms' Applications 19

3 Genetic Algorithms for Combinatorial Optimization Problems **21**

3.1 Genetic Algorithm for the Clique Problem 21

3.1.1 Clique Problem 21

3.1.2 Representation and Evaluation Function 22

3.1.3 Genetic Operators 23

3.1.4 Empirical Results 27

3.1.5 An Example 42

3.2 Genetic Algorithm for the Vertex Cover Problem 43

3.2.1 Vertex Cover Problem 43

3.2.2 Representation and Evaluation Function 48

3.2.3 Generating Input Graph at Random 49

3.2.4 Empirical Results 52

3.3 Genetic Algorithm for the Max Cut Problem 58

3.3.1 Max Cut Problem 58

3.3.2 Representation and Evaluation Function 58

3.3.3 Implementation 59

3.3.4 Comparisons of Genetic Algorithm and Greedy Algorithm . . 60

3.3.5 Comparisons of Genetic Algorithm Solutions and Optimal Solutions 61

3.4 Adapting Parameters 62

4 Parallel Genetic Algorithms **69**

4.1 Connection Machine System 69

4.2 C* Parallel Programming Language 71

4.3 Implementation of Parallel Genetic Algorithms 73

4.3.1 Parallel Data Structure 73

4.3.2 Evaluation Function 73

4.3.3 Reproduction 76

4.3.4 Crossover 80

4.3.5 Mutation 82

4.4 Analysis of the Parallel Genetic Algorithm 83

5 Advanced Techniques 87

6 Conclusions 89

7 Open Problems and Future Work 91

List of Tables

1 Clique Size = 20 in Initial Graph G 32

2 Clique size = 25 in Initial Graph G 33

3 Clique Size = 30 in Initial Graph G 34

4 Clique Size = 35 in Initial Graph G 35

5 Clique Size = 40 in Initial Graph G 36

6 Report of Initial Generation for the Example 44

7 Report of Generation 1 for the Example 45

8 Report of Generation 7 for the Example 46

9 Report of Generation 25 for the Example 47

10 Total Vertices = 50 in Initial Graph 54

11 Total Vertices = 60 in Initial Graph 55

12 Total Vertices = 70 in Initial Graph 56

13 Total Vertices = 80 in Initial Graph 57

14 Density Probability = 0.4 in Initial Graph 63

15 Density Probability = 0.5 in Initial Graph 64

16 Density Probability = 0.6 in Initial Graph 65

17 Density Probability = 0.7 in Initial Graph 66

18 Comparisons of Results for Max Cut 67

19 Increasing the Maximum Generation to Improve the Results 68

20 Total Vertices = 10 in Initial Graph G 85

21 **Total Vertices = 20 in Initial Graph G** 86

List of Figures

1 Graph with Maximum Clique Size 4 (v_1, v_4, v_6, v_9) 2

2 Graph with Minimum Vertex Cover Size 4 (v_1, v_3, v_6, v_9) . . . 3

3 Graph with Max Cut 44 4

4 Redrawn Graph with Max Cut 44 4

5 The Best Results of Generations (Clique Size = 20) 37

6 The Best Results of Generations (Clique Size = 25) 38

7 The Best Results of Generations (Clique Size = 30) 39

8 The Best Results of Generations (Clique Size = 35) 40

9 The Best Results of Generations (Clique Size = 40) 41

10 Input Graph for the Example 42

11 Parallel Algorithm for Calculating Sub-sums 77

12 Recursive Representation for Calculating Sub-sums 78

1 Introduction

1.1 Statement of the Problems

In complexity theory, NP is the class of all decision problems that can be solved by polynomial time nondeterministic algorithms. P is the class of all decision problems that can be solved by polynomial time deterministic algorithms. Informally, a decision problem L is NP-Complete if L is in NP and, for all other decision problems, L' in NP, L' can be transformed to L in polynomial time. In short, no NP-Complete problem is known to have a polynomial time deterministic algorithm, so if a problem is NP-Complete, the search for an efficient, exact algorithm for the problem should be accorded low priority [GJ79]. In practice, many problems are NP-Complete; for these we are faced with the task of finding some usable algorithms for dealing with them.

Although the theory of NP-Completeness restricts attention to decision problems, we can extend the implications of the theory to optimization problems. One approach to solve the optimization problems is that we no longer focus on finding an optimal solution, but instead try to find a near optimal solution within an acceptable amount of time.

The problems presented in this thesis are three basic NP-Complete problems which are called the *Clique*, *Vertex Cover*, and *Max Cut* problems.

A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$, where $V' \subseteq V$, $E' \subseteq E$; V, V'

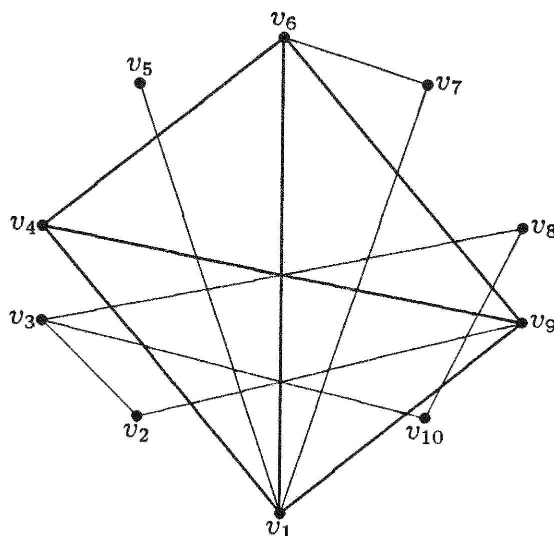

Figure 1: **Graph with Maximum Clique Size 4** (v_1, v_4, v_6, v_9)

are the sets of vertices, E, E' are the sets of edges.

A *complete subgraph* is a subgraph of given graph in which there is an edge between any two vertices.

The optimization problem for *Clique* is: In a given undirected graph, find a complete subgraph with maximum number of vertices. For example, there are 10 vertices in Figure 1. v_3, v_8, and v_{10} is a clique with size 3. v_1, v_6, and v_7 is also a clique with size 3. The maximum clique is v_1, v_4, v_6, and v_9 with size 4.

The optimization problem for *Vertex Cover* is: In a given undirected graph, find a subset of vertices such that for each edge in the graph, at least one of the endpoints belongs to the subset, and the number of vertices in the subset is minimum.

Vertex Cover is a very practical problem. For example, when one wants to monitor

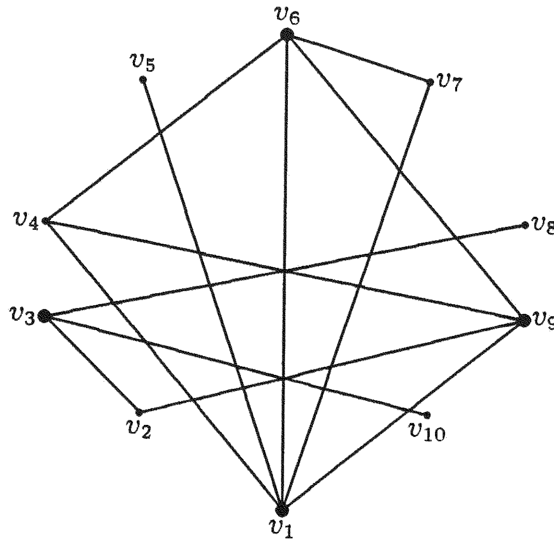

Figure 2: **Graph with Minimum Vertex Cover Size 4** (v_1, v_3, v_6, v_9)

the operation of a large network, one expects to monitor as few nodes as possible. In Figure 2, v_1, v_3, v_4, v_7, and v_9 are a vertex cover with size 5. The minimum vertex cover is v_1, v_3, v_6, and v_9 with size 4.

The optimization problem for *Max Cut* is: In a given undirected weighted graph, divide the vertices into two disjoint subsets such that the sum of the weights of the edges that have two endpoints in different subsets is maximized.

The *Max Cut* problem is also an NP-Complete problem which is very useful for the network design. In Figure 3, the max cut is 44, since clearly, Figure 3 can be redrawn as Figure 4. The vertices are divided into two disjoint subsets $V_1 = \{v_1, v_3, v_4, v_5\}$ and $V_2 = \{v_2, v_6, v_7, v_8, v_9\}$.

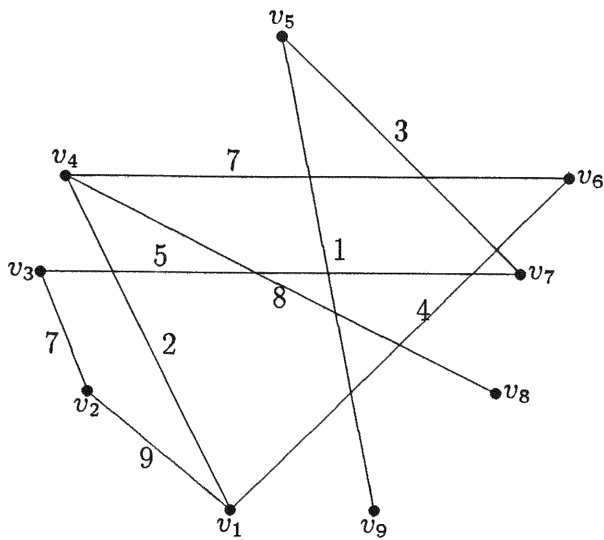

Figure 3: **Graph with Max Cut 44**

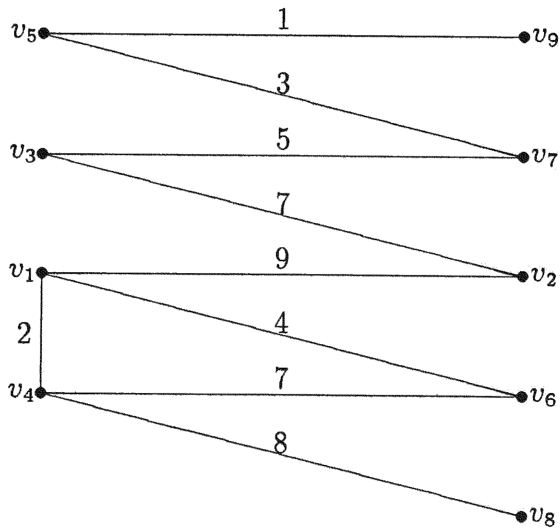

Figure 4: **Redrawn Graph with Max Cut 44**

1.2 Approach for Solving the Problems

A combinatorial optimization problem is either a *minimization problem* or a *maximization problem* and consists of the following three parts:

(1) a set of *instances*;

(2) for each instance, a finite set of *candidate solutions*; and

(3) a function that assigns to each instance and each candidate solution a positive *solution value*.

If a problem is a minimization problem, then an *optimal solution* for an instance is a candidate solution s_1 such that, s_1 is less than or equal to all candidate solutions. If a problem is a maximization problem, then an *optimal solution* for an instance is a candidate solution s_2 such that, s_2 is greater than or equal to all candidate solutions. *Clique* and *Max Cut* are *maximization problems*, while *Vertex Cover* is a *minimization problem.*

An algorithm is an *approximation algorithm* for a combinatorial optimization problem if, given any instance, it finds a candidate solution. If for all instances, the algorithm always find an optimal solution, the algorithm is called an optimization algorithm.

The *Clique, Vertex Cover* and *Max Cut* problems are NP-Complete, therefore polynomial time optimization algorithms cannot be found unless P = NP. A reasonable goal is that of finding approximation algorithms that run in low-order polynomial time and have the property that, for all instances, the algorithms can find a solution

which is close to optimal.

Genetic algorithms are stochastic search techniques based on the mechanics of natural selection and natural genetics. Genetic algorithms differ from traditional analytical methods by using genetic operators and historic cumulative information to prune the search space and generate plausible solutions. Recent research has shown that genetic algorithms can provide efficient search heuristics for solving optimization problems.

Genetic algorithms achieve both power and generality by demanding that problems be mapped into their own particular representation in order to be solved. If a fairly natural mapping exists, impressive robust performance results. *Clique, Vertex Cover* and *Max Cut* problems have very natural representations of genetic algorithms. Therefore, the genetic algorithms are useful for solving these problems.

1.3 An Outline of the Thesis

The following is a brief outline of the thesis: Section 2 introduces genetic algorithms. This section will discuss the historical perspective of genetic algorithms, concepts, unique features of genetic algorithms, mathematical foundations, and genetic algorithms' applications. Section 3 describes how to use genetic algorithms to solve *Clique, Vertex Cover* and *Max Cut* problems in detail and the effects of adaptive genetic parameters. Section 4 illustrates parallel genetic algorithms for *Clique, Vertex Cover* and *Max Cut* problems and the implementation of parallel genetic algorithms on the

Connection Machine. Section 5 presents advanced techniques of genetic algorithms. The conclusions are provided in Section 6. Then finally ended with Section 7 open problems and future work.

2 Introduction to Genetic Algorithms

2.1 Historical Perspective

John Holland is the founder of the field of genetic algorithms. His publication of the book "*Adaptation in Natural and Artificial Systems*" [Hol75] provided a summary of the work which Holland and his students had been pursuing for some time. An important theme in this wide ranging study of the properties of adaptive systems was that adaptation can be usefully modeled as a form of search through a space of structural changes which one might make to a complex system in an attempt to "improve" its behavioral characteristics. This gave rise to a methodology for abstracting and rigorously explaining the adaptive processes of natural systems and designing artificial software systems that retains the important mechanisms of natural systems. Holland described the ability of the bit string representations to encode complicated structures, and the power of simple genetic operators to improve such structures. Holland showed that even in a large domain, genetic algorithms would tend to converge on solutions that were globally optimal or nearly so.

2.2 Concepts

Genetic algorithms represent a highly idealized model of a natural process and as such can be legitimately viewed as a very high level of abstraction. Genetic algorithms are rooted in both natural genetics and computer science. Biological systems developed

successful strategies of behavior adaptation and synthesis to enhance the probability of survival and propagation during their evolution. Environmental pressures requiring these strategies have effected profound changes in biological organisms. These changes are manifested in structural and functional organization, and internal knowledge representations [Aus90]. The metaphor of a genetic algorithm is that of natural evolution and selection. In cumulative selection, each successive incremental improvement in a solution structure becomes the basis for the next generation. The principles of natural selection and population genetics are intrinsically powerful. Algorithms inspired by these principles have been successful when applied to the challenges of machine learning and function optimization [Gol89].

In a natural genetic system, the *chromosomes* consist of *genes*. Each *gene* has a value and position. The combination of *chromosomes* form the total genetic prescription for the construction and operation of some organism [Gol89]. Corresponding to the natural genetic system, in an artificial genetic system, we use *strings* and *characters* to simulate *chromosomes* and *genes*. When using a genetic algorithm to solve a problem, we need to represent the problem by a string, and to define an evaluation function. This function uses the value of the string as a parameter to evaluate the results of the problem.

In a natural genetic system, we have *populations* and *generations*. Similar to natural systems, in an artificial genetic system we also have *populations* and *generations*. We use a set of strings to represent the *populations*. We evaluate each string

through the evaluation function and form the new generation by using specific genetic operators.

The behavior of genetic algorithms can be subtle, but their basic construction and execution cycle is straightforward. A genetic algorithm is an interactive procedure maintaining a population of strings that are candidate solutions to specific problems. During each generation, the strings in the current population are rated for their effectiveness as solutions, and on the basis of these evaluations, a new population of candidate solutions is formed using genetic operators such as *reproduction, crossover,* and *mutation.*

Reproduction is a process which determines the actual number of offspring each individual string will receive in the next generation based on the value of the evaluation function, which is called the *fitness.* The strings with higher fitness value have a greater probability of contributing one or more offspring to the next generation, while the strings with lower fitness value cannot survive in the next generation. The initial population can be chosen heuristically or randomly.

When a string has been selected for the next generation, it is entered into a mating pool for the operations of *crossover* and *mutation.*

Crossover is a method for sharing information between two successful individual strings. It explores the search space by changing some of the bit values in a string. *One point crossover* can be implemented by selecting a random integer position in the string and exchanging the segments to the right of this point with another string

similarly partitioned. We illustrate the *one point crossover* operator as follows:

Suppose that A and B are two parent strings selected randomly from the mating pool, and k is the random integer position.

$$A = a_1 a_2 \dots a_k a_{k+1} a_{k+2} \dots a_n$$

$$B = b_1 b_2 \dots b_k b_{k+1} b_{k+2} \dots b_n$$

After *crossover*, the children are

$$A' = a_1 a_2 \dots a_k b_{k+1} b_{k+2} \dots b_n$$

$$B' = b_1 b_2 \dots b_k a_{k+1} a_{k+2} \dots a_n$$

Mutation is a simple operator which flips one or more bits of a string at random. In an artificial genetic system, the occasional use of the mutation operator can prevent the loss of some potentially useful genetic material.

Crossover will be affected by the *crossover probability*, which is the frequency with which the crossover operator is applied. The higher the *crossover probability*, the more quickly new strings are introduced to the population. When the crossover probability is too high, the string with higher fitness is removed faster than selection can produce improvements.

Mutation will be affected by the *mutation probability*, which is the frequency with which the mutation operator is applied. A low level of *mutation* prevents a given position from freezing at a single value.

The C code for the general outline of a genetic algorithm is the following:

```
Genetic_Algorithm ()
{
  int generation;
  encode_problem ();
  initialize_population ();
  for (generation=1; generation<=maxgeneration; generation++)
  {
    selection ();
    crossover ();
    mutation ();
    evaluation ();
    decode_string ();
    statistics ();
    report ();
  }
}
```

If we define:

L is the string length

P is the population size

G is the number of generations

then the time complexity of a genetic algorithm is $O(L \times P \times G)$.

Genetic algorithms are easy to implement. All the operator functions can be used for the different problems. When we want to solve a problem, all we need to do is to encode the problem to a string representation, input the value of the problem, define the evaluation function, and decode the string to the problem itself.

2.3 Unique Features of Genetic Algorithms

Genetic algorithms are different from traditional analytical methods. Genetic algorithms are derived from a simple model of population genetics based on the following assumptions [DS87, Gol89]:

- The problem and its inputs and outputs can be represented as fixed length strings having a finite number of possible values, at each position.
- At any time, the system maintains a population which contains a finite number of strings and represents the current set of solutions to the problem.
- Each point in the search space can be uniquely represented by a string generated from the finite number of possible values.
- Each string has a fitness, or relative ability to survive and produce offspring.

Genetic algorithms work from population points, not a single point; and use probabilistic transition rules, not deterministic rules. To use a genetic algorithm to solve a problem, we must:

- represent the problem as a finite length string over some finite alphabet.

- choose a way to create an initial population of solutions.

- define evaluation function, and use the information of the evaluation function to rate solutions in terms of their fitness.

- use genetic operators that alter the composition of children in the next generation.

- set the parameter values of the genetic algorithm.

2.4 Mathematical Foundations of Genetic Algorithms

Holland's genetic algorithm theory is called the *Schema Theorem* which describes the actual behavior of genetic algorithms. A schema is the set of all strings having certain "defining" values at designated positions in the string. For example, schema 0***1 is the set of all strings of length 5 having the value 0 in the first position and the value 1 in the last position, where * is a wild card. The Schema Theorem provides a lower bound on the expected number of representatives of a particular schema in the next generation under various genetic operators.

For any string of length l over the binary alphabet $\{0, 1\}$, there are 3^l schemata because each of the l positions may be a 0, 1, or *. A particular string contains 2^l schemata because each position may take on its actual value or a *. A population of size n contains somewhere between 2^l and $n \times 2^l$ schemata, depending upon the

population diversity.

The schema describes the similarities among the strings in the population. When we consider the strings, their fitness, and their similarities simultaneously, we can obtain a wealth of new information to help direct our search. The schemata actually processed in each generation are affected by reproduction, crossover, and mutation operators. To analyze the growth and decay of the many schemata contained in a population, we need to define two schema properties: *schema order* and *defining length*.

The order of a schema H, denoted by $o(H)$, is simply the number of fixed positions presented in the schema. For example, the order of schema 0***1 is 2.

The defining length of a schema H, denoted by $\delta(H)$, is the distance between its first and last fixed position. For example, the schema 01**1 has defining length $\delta(H) = 5 - 1 = 4$. The schema **1** has defining length 0 because the first and the last fixed positions are the same (namely the third position).

Schemata and their properties are interesting notational devices for rigorously discussing and classifying string similarities. They provide the basic means for analyzing the individual and combined effect of reproduction, crossover, and mutation on schemata contained within the population strings.

Schema theorem can be represented by the following equation:

$$m(H, t+1) \geq m(H, t) \times [\frac{f(H, t)}{\overline{f}(t)}] \times [1 - \frac{P_c \delta(H)}{l - 1} - o(H) P_m]$$

where $m(H, t)$ is the expected number of strings of a particular schema H in popu-

lation $P(t)$, $f(H,t)$ is simply the average fitness of the strings of schema H at time t, $\overline{f}(t)$ denotes the average fitness of the individuals in population $P(t)$, P_c is the crossover probability, $\delta(H)$ is the defining length of schema H , which is the distance between the first and last specific string position, l is the length of each individual string, $o(H)$ is the order of schema H , which is simply the number of fixed positions, P_m is the mutation probability.

The effect of reproduction on the expected number of schemata in the population is easy to determine. Suppose at a given time step t there are m examples of a particular schema H contained within the population $P(t)$. During reproduction, a string A_i is copied according to its fitness with probability $p_i = \frac{f_i}{\sum f_j}$. After picking a nonoverlapping population of size n with replacement from the population $P(t)$, we expect to have $m(H, t+1)$ representatives of the schema H in the population at time $t+1$ as given by the equation $m(H,t+1) = m(H,t) \times n \times \frac{f(H,t)}{\sum f_j}$, where $f(H,t)$ is the average fitness of the strings representing schema H at time t. If we recognize that the average fitness of the entire population is $\overline{f}(t) = \frac{\sum f_j}{n}$, then

$$m(H,t+1) = m(H,t)\frac{f(H,t)}{\overline{f}(t)}$$

From this equation, we know that schemata with fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples.

Reproduction alone does nothing to promote exploration of new regions of the

search space, since no new points are searched. If we only copy old strings without change, we won't ever try anything new. This is why we need crossover.

Crossover leaves a schema unscathed if it does not cut the schema, but it may disrupt a schema when it does. For example, consider a particular string of length 10 and two representative schemata within that string:

$$A = 0\,1\,1\,0\,0\,0\,1\,0\,0\,0$$

$$H_1 = 0 * * * * * * * 0 *$$

$$H_2 = * * 1 * 0 * * * * *$$

String A is represented by schemata H_1 and H_2, but the effect of crossover on the schemata are different. Suppose string A has been chosen for mating and crossover, the random crossover point is 5. We use the symbol | to mark the crossover point as follows:

$$A = 0\,1\,1\,0\,0 \mid 0\,1\,0\,0\,0$$

$$H_1 = 0 * * * * \mid * * * 0 *$$

$$H_2 = * * 1 * 0 \mid * * * * *$$

Unless string A's mate is identical to A at the fixed positions of schema, the schema H_1 will be destroyed because the fixed positions are on opposite of the crossover point, while schema H_2 will survive because the fixed positions will be carried intact to a single offspring. Schema H_1 has a defining length 8 which is bigger than defining length 2 of schema H_2. It is clear that schema H_1 is less likely to survive crossover

than schema H_2 because on average the crossover point is more likely to fall between the fixed positions. If the crossover point is chosen uniformly at random among the $l-1=9$ possible points, then schema H_1 is destroyed with probability $p_{d1} = \frac{\delta(H_1)}{l-1} = \frac{8}{9}$, schema H_2 is destroyed with probability $p_{d2} = \frac{\delta(H_2)}{l-1} = \frac{2}{9}$. Therefore schema H_1 survives with probability $p_{s1} = 1 - p_{d1} = \frac{1}{9}$, schema H_2 survives with probability $p_{s2} = 1 - p_{d2} = \frac{7}{9}$. Generally, the survival probability $p_s = 1 - \frac{\delta(H)}{l-1}$. If crossover is performed with crossover probability P_c at a particular mating, the survival probability will be

$$p_s \geq 1 - P_c \times \frac{\delta(H)}{l-1}$$

When $P_c = 1.0$, $p_s \geq 1 - \frac{\delta(H)}{l-1}$.

Suppose that that the reproduction and crossover operations are independent. Then the combined effect of crossover and reproduction is

$$m(H,t+1) \geq m(H,t) \times [\frac{f(H,t)}{\overline{f}(t)}] \times [1 - \frac{P_c\delta(H)}{l-1}]$$

The last operator is mutation. Mutation is the random change of a single position with mutation probability P_m. In order for a schema H to survive, all of the specified positions must themselves survive. Since a single character survives with probability $(1-P_m)$, and since each of mutations is statistically independent, a particular schema survives when each of the $o(H)$ fixed positions within the schema survives. Multiplying the survival probability $(1-P_m)$ by itself $o(H)$ times, the survival probability of mutation is $(1-P_m)^{o(H)}$. For small values of $P_m (P_m << 1.0)$, the survival probability of mutation is approximately $1 - o(H) \times P_m$, by the binomial theorem. Therefore the

combined effect of reproduction, crossover and mutation operations is the following equation (ignoring small cross-product terms):

$$m(H, t+1) \geq m(H, t) \times [\frac{f(H, t)}{\overline{f}(t)}] \times [1 - \frac{P_c \delta(H)}{l-1} - o(H) P_m]$$

The schema theorem tell us that schema H grows or decays depending upon a multiplicative factor. If $f(H, t) > \overline{f}(t)$, $\delta(H)$ is smaller, and $o(H)$ is smaller, then $m(H, t+1)$ will be bigger, i.e. the schemata which have higher fitness than average fitness, short defining length, and low-order will receive more strings in the next generation. [Ant89, BG87, Gol89, GB89, Hol75]

2.5 Genetic Algorithms' Applications

Genetic algorithms have a large range and growing number of applications which can be found in scientific areas such as Biology, Chemistry, Computer Science, Engineering and Operations Research, Image Processing and Pattern Recognition, Mathematics, Physical Sciences, and Social Sciences, among others. Industry has used genetic algorithms to carry out commercial optimization. The following is the brief overview of genetic algorithms' applications in different disciplines.

- **Job Shop Scheduling:** Scheduling the day-to-day workings of a job shop which is an organization composed of a number of work stations capable of performing operations on objects. Specifying which work station is to perform which operations on which objects from which contracts. [Dav85]

- **Communication Network:** Finding low-cost sets of packet switching communication network links when the network topology has been fixed. [DC87]

- **Boolean Satisfiability Problem:** Given an arbitrary boolean expression of n variables, finding an assignment to those variables such that the expression is true. [DJS89]

- **Visual Recognition:** Classifying distorted examples of different but similar classes of image patterns. [Eng85]

- **Traveling Salesman Problem:** Finding the shortest route to be taken by a salesman who must visit each of n cities exactly once. [FW90]

- **Optimization of Pipeline Systems:** Applying genetic algorithms to optimization and machine learning problems in natural gas pipeline control. [Gol89]

- **Chemometrics:** The processing of analytical information and data in chemometrics (which is a subdiscipline of analytical chemistry). [LK89]

- **Multiple Objective Optimization:** Handling multiple non-commensurable objectives such factors as cost, safety and performance. [Sch85]

- **Biological Development:** Simulating the evolution of simple multicellular systems. [Wil87]

3 Genetic Algorithms for Combinatorial Optimization Problems

3.1 Genetic Algorithm for the Clique Problem

The *Clique* problem is one of the basic NP-Complete problems. In this section, a genetic approach to generating solutions for the *Clique* problem is described. Empirical results are reported which demonstrate the effectiveness of genetic algorithms in solving combinatorial optimization problems.

In genetic algorithms, there are some important genetic parameters such as population size, maximum generation, crossover probability and mutation probability. We will discuss their effect for the *Clique* problem in section 3.4.

3.1.1 Clique Problem

The *Clique* problem can be stated as follows [GJ79]:

INSTANCE: A graph $G = (V, E)$ and a positive integer $K \leq |V|$, where V is a set of vertices, and E is a set of edges.

QUESTION: Does G contain a *clique* of size K or more, that is, a subset $V' \subseteq V$ such that $|V'| \geq K$ and every two vertices in V' are joined by an edge in E?

The *Clique* problem is NP-Complete. In complexity theory, NP is the class of all decision problems that can be solved by polynomial time nondeterministic algorithms. Although the theory of NP-Completeness restricts attention to decision problems, we

can extend the implications of the theory to optimization problems. The optimization problem for the *Clique* problem is to find the maximum K in a graph G, where K is the clique size.

3.1.2 Representation and Evaluation Function

First, the *Clique* problem is encoded as a string. In a graph $G = (V, E)$, with n vertices, the vertices are numbered from 1 to n. Solutions to the *Clique* problem can be represented by an n-bit string $c_1c_2\ldots c_n$, where $c_i = 0$ or 1, $i = 1, 2, \ldots, n$. Each bit position corresponds to the numbered vertex. $c_i = 1$ means vertex $v_i \in V'$, i.e. v_i is in the clique.

Second, the penalty method is used to define the evaluation function as follows [RPLH89]:

evaluation function = total edges which should be in the clique of size K − *penalty*

penalty = (the number of unconnected edges in V')^2

When the n-bit string $c_1c_2\ldots c_n$ has K 1 bits ($K \leq n$), the total edges contained in the clique is

$$\sum_{i=1}^{K-1} i = \frac{K(K-1)}{2}$$

When the value of the function (*fitness*) is smaller than 0, the function returns 0. If a subset $V' \subseteq V$ is a clique, then $penalty = 0$.

3.1.3 Genetic Operators

Genetic algorithms have three primary operators. They are *reproduction, crossover,* and *mutation.*

Reproduction: Genetic algorithms process populations of strings. The string length for the *Clique* problem is the number of vertices in a graph G. We use the function *flip* with the probability 0.05 to randomly generate the initial population for the *Clique* problem. Using a smaller probability to generate the initial population can obtain cliques with smaller size first, and improve the result in later generations, and can also avoid the loss of feasible solutions. The function *flip* returns a boolean value 1 (true) or 0 (false) according to the given probability parameter. The C code fragment for the function *flip* is the following:

```
# define range 2147483647.0 /* 2^31 - 1 */
flip ( probability )
float probability;
{
    return ( ( double ) random ( ) / range <= probability );
}
```

Reproduction is a process which determines the actual number of offspring each individual string will receive in the next generation based on the values of their evaluation function. The strings with higher fitness value have a greater probability of contributing one or more offspring to the next generation, while the strings with

lower fitness value cannot survive to the next generation.

There are many algorithms to select offspring. Baker's sampling algorithm is used to implement the reproduction function [Bak87]. This algorithm is more efficient than other algorithms. The time complexity of genetic algorithms' other phases is $O(L \times P)$ in each generation, where L is the string length, P is the population size. It is desirable for the reproduction phase not to increase the genetic algorithms' overall time complexity. The time complexity of Baker's algorithm is $O(P)$, while the complexity of other algorithms is $O(PlogP)$.

The basic idea of the Baker's sampling algorithm is as follows: First, compute the sum of the fitness over all strings. Second, calculate *step*, which is the sum of the fitness divided by the population size. Then generate *pointer* randomly by using a random integer uniformly distributed in [0, *step*). Finally, for each string, determine how many copies should be put into the mating pool according to the *step* and random *pointer*. The C code fragment for the *reproduction* operator is the following:

```
# define range 2147483647.0 /* 2^31 - 1 */
reproduction ( )
{
    float ptr, sum=0.0, step;
    int i, flag=0;
    step = ( float ) sumfitness / populationsize;
    ptr = random ( ) / range * step;
```

```
        for ( i = 1; i <= populationsize; i++ )
            for ( sum += ( float ) fitness [ i ]; sum > ptr; ptr += step )
                mating_pool [ ++flag ] = i;
    }
```

When a string has been selected for the next generation, it is entered into a mating pool for the operations of *crossover* and *mutation.*

Crossover: Crossover is a method for sharing information between two successful individual strings. It explores the search space by changing some of the bit values in a string. In the algorithm, a one point *crossover* operator is used. The mating schema is that two parents selected randomly from the mating pool are exchanged to produce two children to replace them. *Crossover* can be implemented by selecting a random integer position in the string and exchanging the segments to the right of this point with another string similarly partitioned. The C code fragment for the *crossover* operator is following:

```
crossover(j)
int j;
{
    int i;
    if (flip(cross_prob)==1)
        jcross=range_random(1, stringlength);
    else
```

```
jcross=stringlength;
for (i=1; i<=jcross; i++)
{
  new_string[j][i]=mutation(old_string[mate1][i]);
  new_string[j+1][i]=mutation(old_string[mate2][i]);
}
if (jcross != stringlength)
  for (i=jcross+1; i<=stringlength; i++)
  {
     new_string[j][i]=mutation(old_string[mate2][i]);
     new_string[j+1][i]=mutation(old_string[mate1][i]);
  }
}
```

Crossover will be affected by the crossover probability, which is the frequency the crossover operator is applied. When the crossover probability is too high, the string with higher fitness is removed faster than selection can produce improvements. Therefore, crossover probability 0.5 is used in the experiments.

Mutation: Mutation is a simple operator which flips one or more bits at random using some mutation probability. In an artificial genetic system, the occasional use of the mutation operator can prevent the loss of some potentially useful genetic material. Mutation probability 0.01 is used in the experiments. The C code fragment for the

mutation operator is the following:

```
mutation(bit_string)
int bit_string;
{
   int res;
   if (flip(mutation_prob)==1)
   {
     if (bit_string==1)
       res=0;
     else
       res=1;
   }
   else
     res=bit_string;
   return(res);
}
```

3.1.4 Empirical Results

For testing purposes, the input graph G is initialized by the following method:

The total number of vertices, n, in a graph G is an input parameter, and matrix M represents the graph.

$$M = \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & a_{24} & \cdots & a_{2n} \\ 0 & 0 & 0 & a_{34} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & a_{n-1n} \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

where $a_{ij} = 1$ or 0. $a_{ij} = 1$ means that there is an edge between vertex v_i and vertex v_j, for $i < j$. (Although the graph is undirected we adopt the following convention: $a_{ij} = 0$ if $i \geq j$.)

First, use function *flip* with 0.5 probability and different random seed to generate the matrix M. Second, generate m distinct random numbers between 1 and n, where $m = |V'|$ and $m \leq n$, that is, m is the number of vertices in the *clique*. Then, make the m vertices as a *clique*. When m is small, it is possible that there is a clique bigger than m. However, when m is large, it is unlikely that there is a clique bigger than m.

Running the genetic algorithm, the number of unconnected edges in V' is computed as follows:

(1) count the number of 1's in a string, put the corresponding position in a buffer;

(2) check whether the subset V' consists of a clique by using matrix M.

The C code for computing the fitness of each string is the following:

```
evaluation_function(string)
```

```
int string[maxsting];
{
    int res, count=0, i, j;
    int buffer[maxstring];
    penalty = 0;
    for (j=1; j<=stringlength; j++)
        if (string[j] == 1)
        {
          count ++;
          buffer[count] = j;
        }
    res = count * (count - 1 ) / 2;
    for (i=1, i<count; i++)
        for (j=i+1; j<=count; j++)
          if (M[buffer[i]] [buffer[j]] != 1)
            penalty ++;
    res -= penalty * penalty;
    return (max(res, 0));
}
```

In the experiment, population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01 are used to test the algorithm. Differ-

ent seed numbers and different vertex numbers in the *clique* are also used to generate the initial graph G. The tests are performed separately for the total number 50, 60, 70, 80, 90, and 100 vertices in a graph G. Empirical results are reported in Tables 1 – 5.

Let us define [GJ79]:

$$performance\ guarantee = \frac{the\ clique\ size\ in\ the\ result}{the\ clique\ size\ in\ the\ initial\ graph} \times 100$$

The results show that 77.33% tests get optimal results (the performance guarantee = 100.00), 22.67% tests get near optimal results and the average performance guarantee is 98.80% which is quite high for the heuristic algorithms or approximation algorithms. The algorithm can get "good" solutions in an acceptable amount of time. For example, the total number of vertices in a graph is 100, population size = 80, generation = 200, the running time of the algorithm is 2 or 3 minutes. All experiments run on the SUN 4 machine under the UNIX operating system.

For a specific clique problem, we can arrange two or three set of parameters to run the program and get optimal or very close to optimal results.

All the operator functions and initial population, statistics and report functions can be used for the other problems. Therefore, when solving a new problem, all we need to do is to encode the problem to a string representation, input the values of the problem and define the evaluation function properly.

Figures 5 – 9 show the best results in every generation for the problems which population size = 80, maximum generation = 200, crossover probability = 0.5, mu-

tation probability = 0.01, vertices number = 80, seed = 1, 2, 3, 5, 6, and clique size in initial graph = 20, 25, 30, 35, and 40 separately.

total vertices	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	57	1	20	20	100.00
60	62	1	20	20	100.00
70	63	1	20	20	100.00
80	99	1	20	20	100.00
90	116	1	20	20	100.00
100	111	1	20	20	100.00
50	70	2	20	20	100.00
60	60	2	20	20	100.00
70	43	2	20	20	100.00
80	89	2	20	20	100.00
90	107	2	20	20	100.00
100	67	2	20	20	100.00
50	45	3	20	20	100.00
60	70	3	20	20	100.00
70	59	3	20	20	100.00
80	129	3	20	20	100.00
90	135	3	20	20	100.00
100	130	3	20	20	100.00
50	38	4	20	20	100.00
60	51	4	20	20	100.00
70	80	4	20	20	100.00
80	102	4	20	20	100.00
90	173	4	20	19	95.00
100	110	4	20	20	100.00
50	43	5	20	20	100.00
60	47	5	20	20	100.00
70	58	5	20	20	100.00
80	92	5	20	20	100.00
90	156	5	20	20	100.00
100	93	5	20	20	100.00

Table 1: **Clique Size = 20 in Initial Graph** G

total vertices	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	67	1	25	25	100.00
60	78	1	25	25	100.00
70	96	1	25	25	100.00
80	99	1	25	25	100.00
90	56	1	25	25	100.00
100	133	1	25	25	100.00
50	68	2	25	25	100.00
60	78	2	25	25	100.00
70	69	2	25	25	100.00
80	78	2	25	25	100.00
90	85	2	25	25	100.00
100	97	2	25	25	100.00
50	64	3	25	25	100.00
60	66	3	25	25	100.00
70	120	3	25	25	100.00
80	75	3	25	25	100.00
90	176	3	25	25	100.00
50	57	4	25	25	100.00
60	60	4	25	25	100.00
70	87	4	25	25	100.00
80	73	4	25	25	100.00
90	134	4	25	25	100.00
100	135	4	25	25	100.00
50	87	5	25	25	100.00
60	67	5	25	25	100.00
70	68	5	25	25	100.00
80	83	5	25	25	100.00
90	121	5	25	25	100.00
100	65	5	25	25	100.00

Table 2: **Clique size = 25 in Initial Graph** G

total vertices	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	75	1	30	30	100.00
60	55	1	30	30	100.00
70	99	1	30	30	100.00
80	77	1	30	30	100.00
90	116	1	30	30	100.00
100	119	1	30	30	100.00
50	85	2	30	30	100.00
60	99	2	30	30	100.00
70	101	2	30	30	100.00
80	104	2	30	30	100.00
90	114	2	30	29	96.67
100	192	2	30	30	100.00
50	81	3	30	30	100.00
60	110	3	30	30	100.00
70	102	3	30	30	100.00
80	129	3	30	30	100.00
90	132	3	30	30	100.00
100	85	3	30	30	100.00
50	62	4	30	30	100.00
60	130	4	30	30	100.00
70	79	4	30	30	100.00
80	130	4	30	30	100.00
90	186	4	30	30	100.00
100	169	4	30	30	100.00
50	102	5	30	30	100.00
60	126	5	30	30	100.00
70	122	5	30	30	100.00
80	135	5	30	30	100.00
90	118	5	30	30	100.00
100	141	5	30	30	100.00

Table 3: **Clique Size = 30 in Initial Graph** G

total vertices	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	149	1	35	35	100.00
60	113	1	35	35	100.00
70	154	1	35	35	100.00
80	89	1	35	34	97.14
90	103	1	35	35	100.00
100	123	1	35	34	97.14
50	103	2	35	35	100.00
60	75	2	35	35	100.00
70	133	2	35	34	97.14
80	120	2	35	35	100.00
90	189	2	35	35	100.00
100	115	2	35	34	97.14
50	136	3	35	35	100.00
60	98	3	35	35	100.00
70	156	3	35	35	100.00
80	188	3	35	35	100.00
90	189	3	35	34	97.14
100	144	3	35	35	100.00
50	100	4	35	35	100.00
60	117	4	35	35	100.00
70	86	4	35	35	100.00
80	84	4	35	35	100.00
90	129	4	35	34	97.14
100	129	4	35	34	97.14
50	160	5	35	35	100.00
60	124	5	35	35	100.00
70	118	5	35	35	100.00
80	154	5	35	35	100.00
90	97	5	35	34	97.14
100	69	5	35	33	94.29

Table 4: **Clique Size = 35 in Initial Graph** G

total vertices	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	175	1	40	40	100.00
60	92	1	40	40	100.00
70	105	1	40	39	97.50
80	124	1	40	38	95.00
90	144	1	40	37	92.50
100	106	1	40	37	92.50
50	78	2	40	40	100.00
60	181	2	40	40	100.00
70	139	2	40	39	97.50
80	107	2	40	39	97.50
90	73	2	40	37	92.50
100	190	2	40	39	97.50
50	102	3	40	39	97.50
60	126	3	40	39	97.50
70	159	3	40	39	97.50
80	106	3	40	39	97.50
90	98	3	40	37	92.50
100	154	3	40	39	100.00
50	89	4	40	39	97.50
60	153	4	40	40	100.00
70	80	4	40	39	97.50
80	91	4	40	39	97.50
90	97	4	40	38	95.00
100	140	4	40	39	97.50
50	143	5	40	40	100.00
60	149	5	40	40	100.00
70	156	5	40	39	97.50
80	156	5	40	39	97.50
90	96	5	40	38	95.00
100	180	5	40	39	97.50

Table 5: **Clique Size = 40 in Initial Graph** G

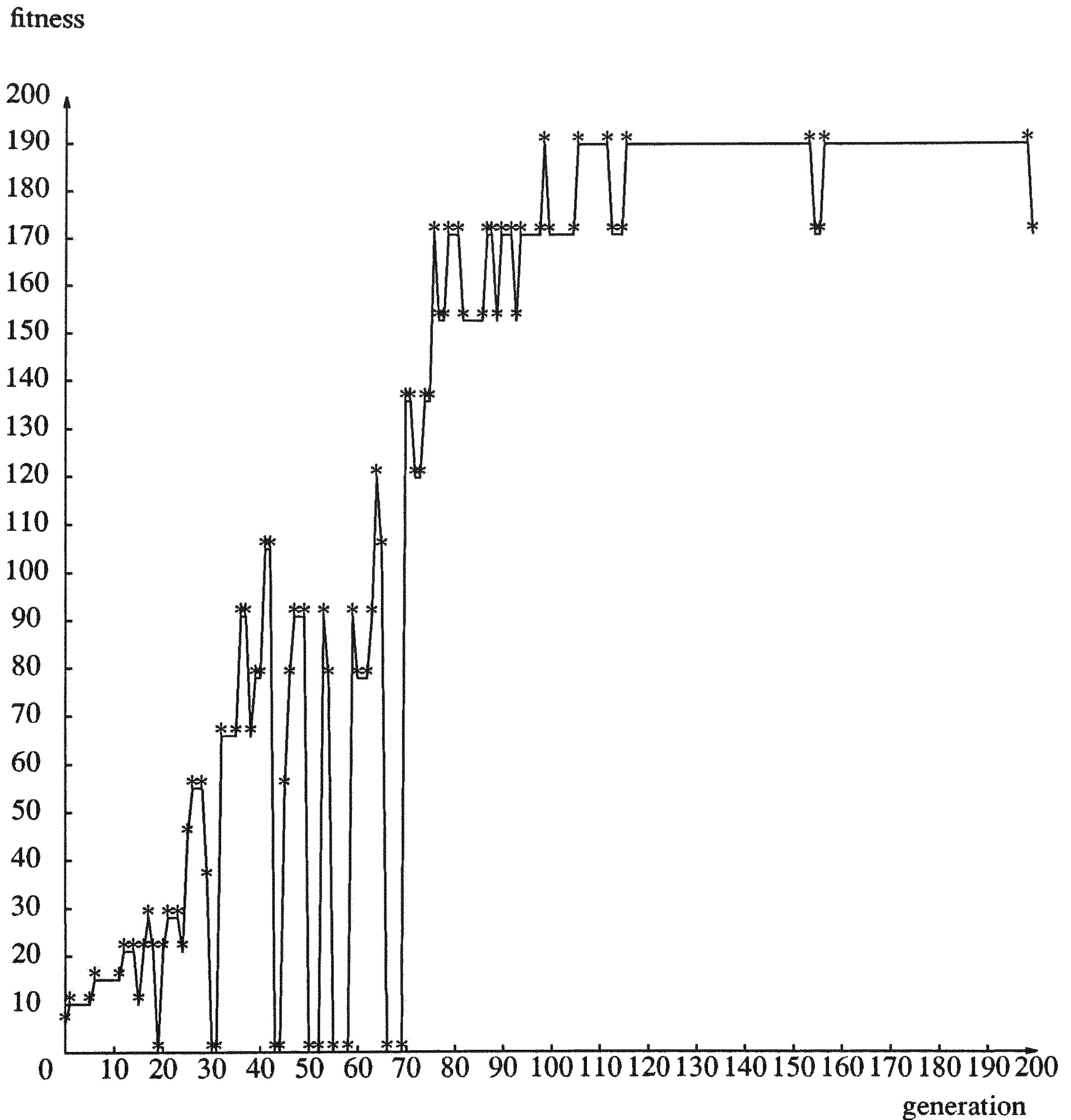

Figure 5: **The Best Results of Generations (Clique Size = 20)**

population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, seed = 1, vertices number = 80, clique size in initial graph = 20. best generation = 99. best fitness = $20 \times (20 - 1) / 2 = 190$, that is, clique size founded = 20.

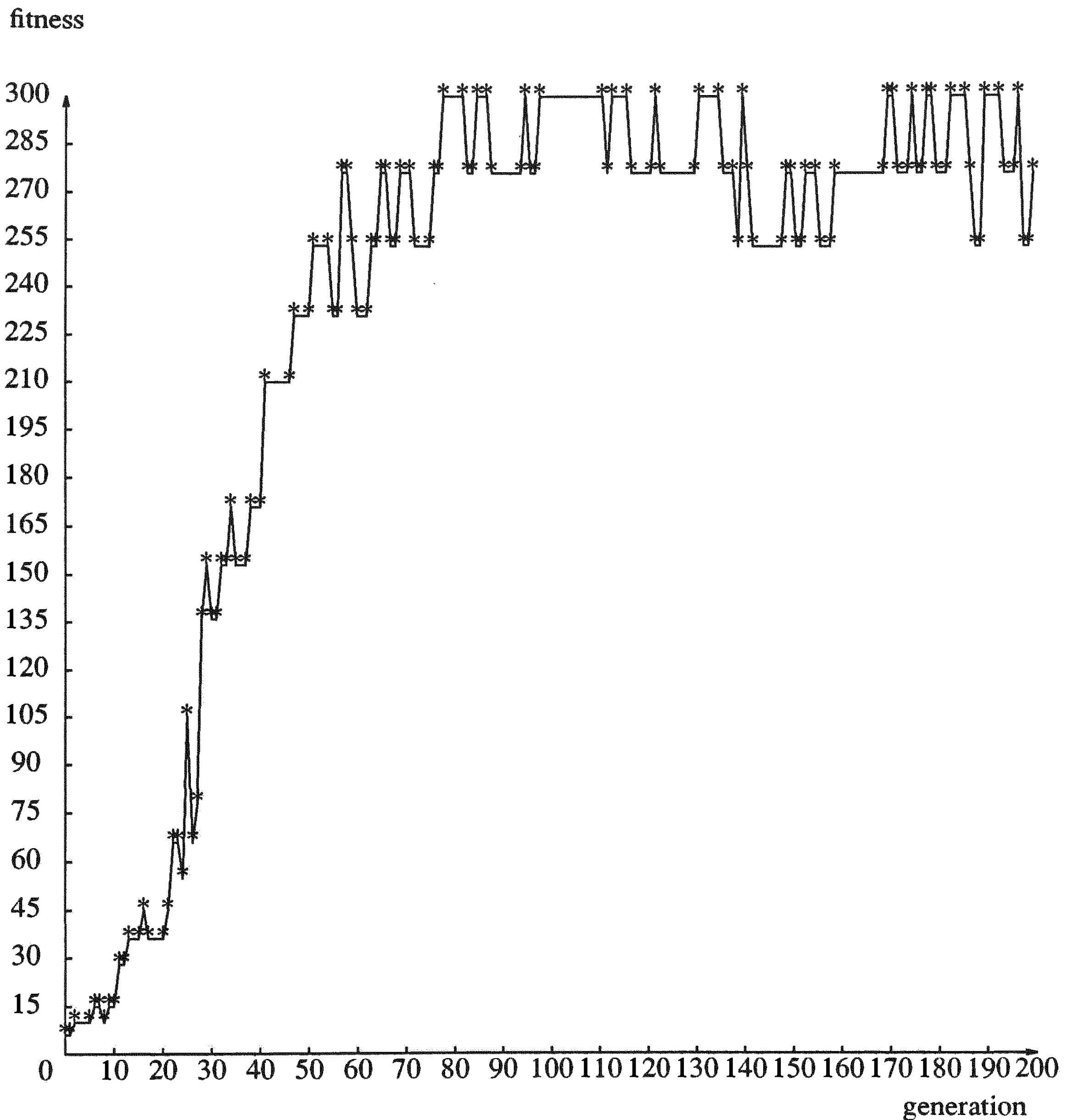

Figure 6: **The Best Results of Generations (Clique Size = 25)**

population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, seed = 2, vertices number = 80, clique size in initial graph = 25. best generation = 78. best fitness = $25 \times (25 - 1) / 2 = 300$, that is, clique size founded = 25.

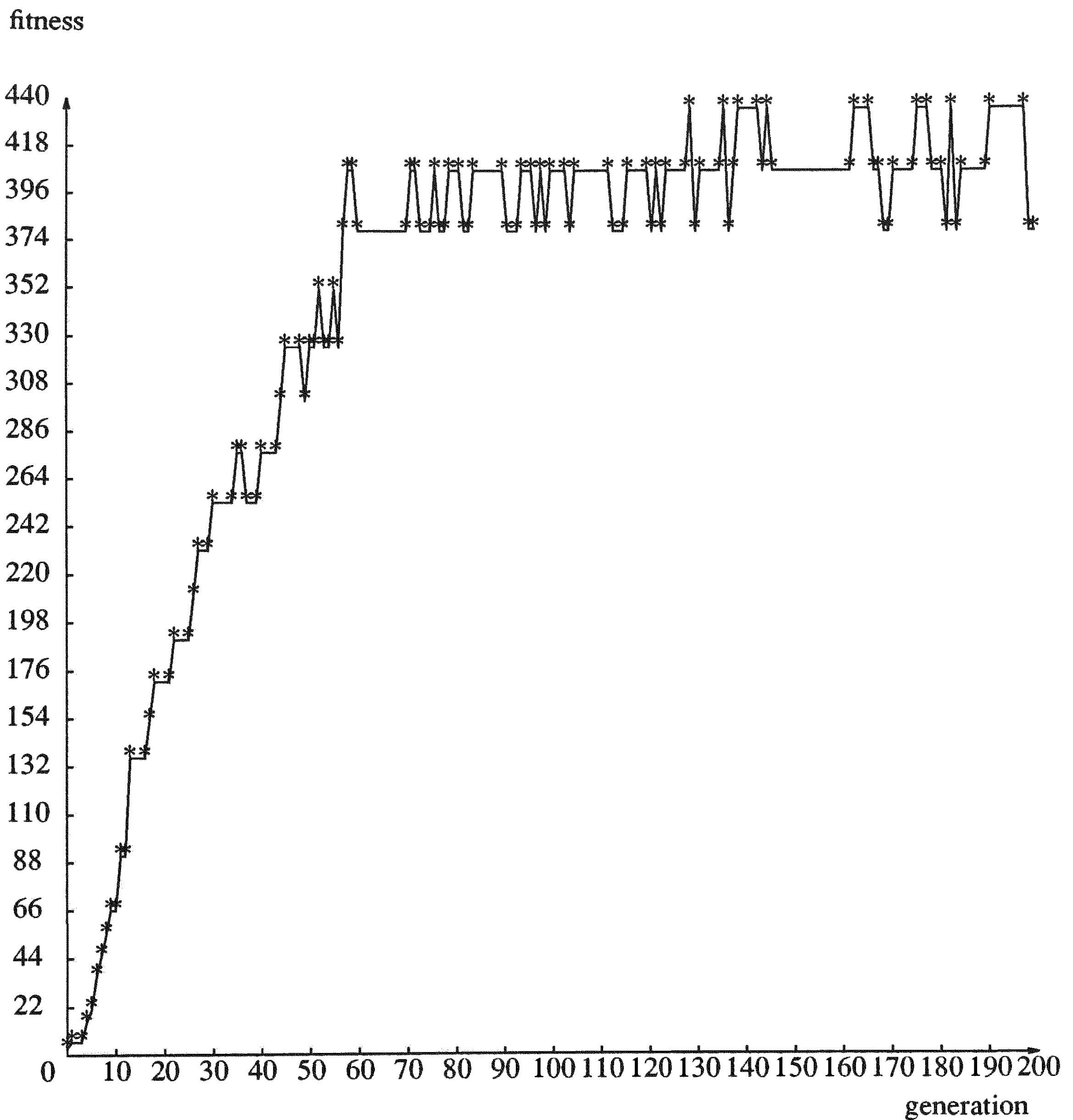

Figure 7: **The Best Results of Generations (Clique Size = 30)**

population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, seed = 3, vertices number = 80, clique size in initial graph = 30. best generation = 129. best fitness = $30 \times (30 - 1) / 2 = 435$, that is, clique size founded = 30.

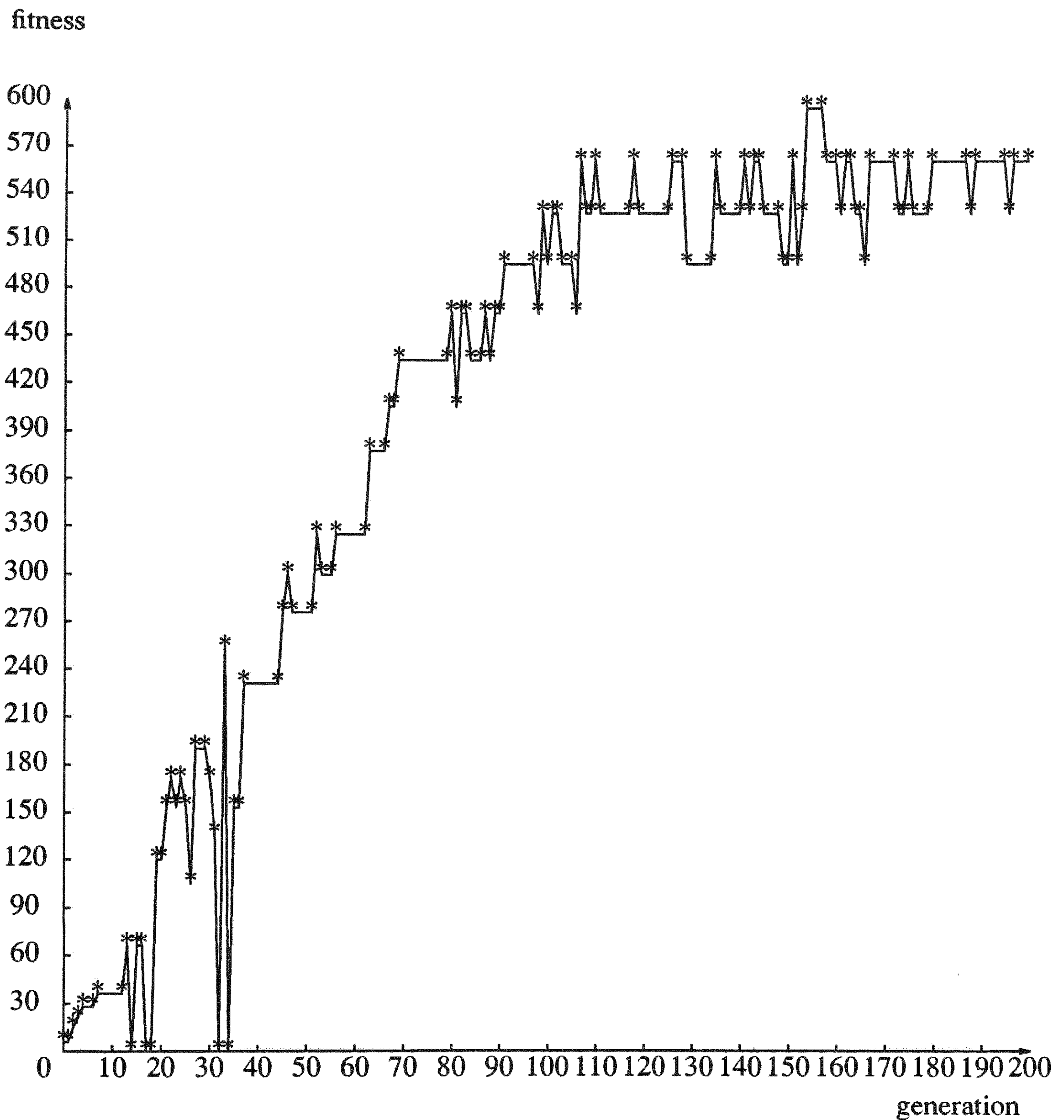

Figure 8: **The Best Results of Generations (Clique Size = 35)**

population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, seed = 5, vertices number = 80, clique size in initial graph = 35. best generation = 154. best fitness = $35 \times (35 - 1) / 2 = 595$, that is, clique size founded = 35.

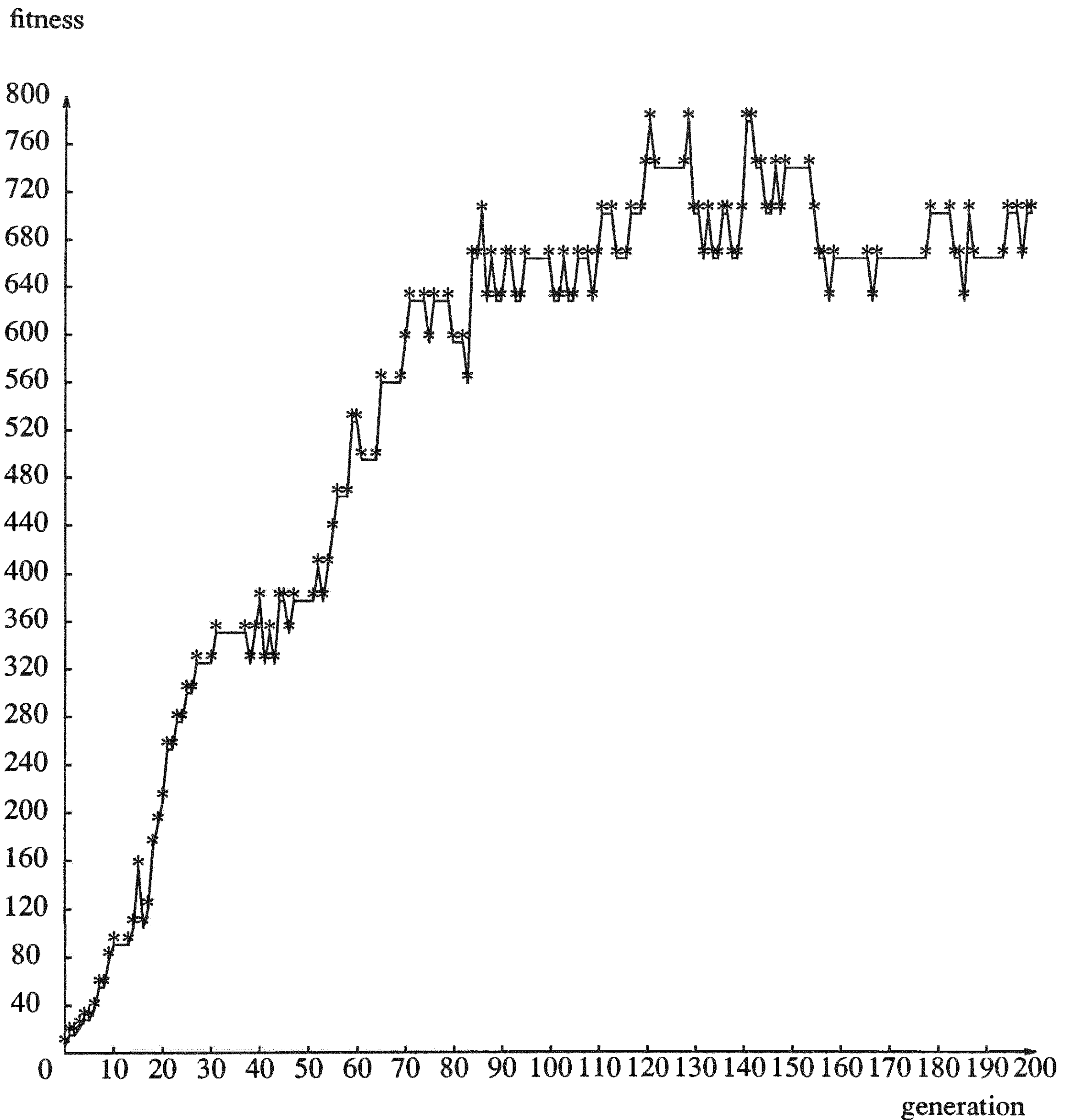

Figure 9: **The Best Results of Generations (Clique Size = 40)**

population size = 80, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, seed = 6, vertices number = 80, clique size in initial graph = 40. best generation = 121. best fitness = $40 \times (40 - 1) / 2 = 780$, that is, clique size founded = 40.

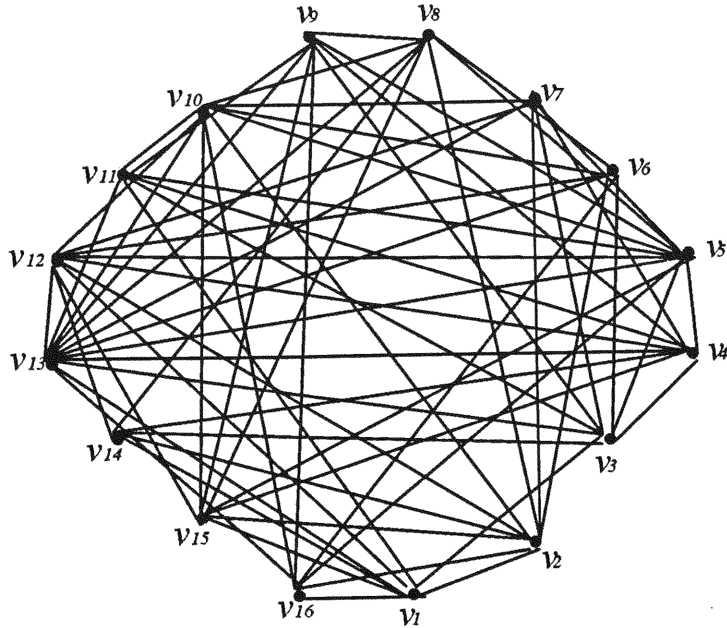

Figure 10: **Input Graph for the Example**

3.1.5 An Example

Suppose the input graph is Figure 10. The Matrix representation is

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

The output should be a clique with maximum size.

Using population size = 30, string length = 16, crossover probability = 0.5, mu-

tation probability = 0.01, random seed = 2, and clique size =6, the initial population strings, their fitnesses and reproduction strings for the next generation are shown in Table 6. The strings with higher fitness value contribute more copies in the next generation, while the strings with fitness value 0 cannot survive in the next generation.

After reproduction, the operations of crossover and mutation are performed. Tables 7, 8, 9 show the results of generation 1, 7, 25 respectively. Based on the mechanics of natural selection and natural genetics, genetic algorithms get better and better results for the problem.

The best generation for this example is 25. The best fitness is 15. The representation of the best string is 0 0 1 1 1 0 0 0 1 0 1 0 1 0 0 0, i.e. vertices $v_3, v_4, v_5, v_9, v_{11}$, and v_{13} consist of a clique with size 6.

3.2 Genetic Algorithm for the Vertex Cover Problem

3.2.1 Vertex Cover Problem

The *Vertex Cover* problem can be stated as follows [GJ79]:

INSTANCE: A graph $G = (V, E)$ and a positive integer $K \leq |V|$, where V is a set of vertices, and E is a set of edges.

QUESTION: Is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and, for each edge $\{u, v\} \in E$, at least one of u and v belongs to V'?

The *Vertex Cover* problem is a very practical problem. For example, when moni-

No.	string	fitness	R-No.	reproduction string
1	0000000000000000	0	7	0000000010011000
2	0000000100000000	0	7	0000000010011000
3	0010000100000000	0	7	0000000010011000
4	0000000000000000	0	7	0000000010011000
5	0000010000000000	0	7	0000000010011000
6	0000000000000000	0	7	0000000010011000
7	0000000010011000	2	7	0000000010011000
8	0000000000000000	0	7	0000000010011000
9	0000010000000000	0	7	0000000010011000
10	0000000000000000	0	7	0000000010011000
11	0000100000000000	0	7	0000000010011000
12	0000000000000000	0	7	0000000010011000
13	0000000000000100	0	7	0000000010011000
14	0001000000000000	0	7	0000000010011000
15	0100000000000000	0	7	0000000010011000
16	0000001000001000	1	16	0000001000001000
17	0000000000000000	0	16	0000001000001000
18	0000000000000000	0	16	0000001000001000
19	0000000000000000	0	16	0000001000001000
20	0000010000000000	0	16	0000001000001000
21	0000000000000000	0	16	0000001000001000
22	0000000000001000	0	16	0000001000001000
23	0000000000000000	0	16	0000001000001000
24	0000000000000000	0	26	0000100000010000
25	0000000000000000	0	26	0000100000010000
26	0000100000010000	1	26	0000100000010000
27	0000001000000000	0	26	0000100000010000
28	0000000000000000	0	26	0000100000010000
29	0000000000001000	0	26	0000100000010000
30	0000000000000000	0	26	0000100000010000
maximum fitness = 1			average fitness = 0.13	

Table 6: **Report of Initial Generation for the Example**

No.	parents	pts	string	fitness	R-No.	reproduction string
1	(3, 15)	16	0000000011011000	2	1	0000000011011000
2	(3, 15)	16	0000000010011000	2	2	0000000010011000
3	(16, 10)	16	0000001000001000	1	3	0000001000001000
4	(16, 10)	16	0000000010011000	2	4	0000000010011000
5	(19, 24)	5	0000000000010000	0	6	0000101000001000
6	(19, 24)	5	0000101000001000	3	6	0000101000001000
7	(17, 6)	16	0001001000001000	3	7	0001001000001000
8	(17, 6)	16	0000010010011000	2	7	0001001000001000
9	(27, 29)	16	0000100000010000	1	8	0000010010011000
10	(27, 29)	16	0000100000010000	1	9	0000100000010000
11	(26, 25)	16	0000100000010000	1	11	0000100000010000
12	(26, 25)	16	0000100000010000	1	12	0000100000010000
13	(22, 28)	16	0000001000001000	1	14	0000100000010000
14	(22, 28)	16	0000100000010000	1	16	0000011000001000
15	(18, 20)	10	0000001000001000	1	16	0000011000001000
16	(18, 20)	10	0000011000001000	2	17	0000000010011000
17	(7, 21)	13	0000000010011000	2	18	0000001000001000
18	(7, 21)	13	0000001000001000	1	21	0000000010011000
19	(23, 30)	16	0000001000001000	1	22	0000000010011000
20	(23, 30)	16	0000100000000000	0	22	0000000010011000
21	(4, 1)	16	0000000010011000	2	23	0000000110011000
22	(4, 1)	16	0000000010011000	2	24	0000000010011000
23	(2, 5)	4	0000000110011000	2	25	0000000010011000
24	(2, 5)	4	0000000010011000	2	26	0000000010011000
25	(8, 9)	16	0000000010011000	2	26	0000000010011000
26	(8, 9)	16	0000000010011000	2	27	0000000010011000
27	(12, 11)	16	0000000010011000	2	28	0000000010011000
28	(12, 11)	16	0000000010011000	2	29	0000000010011000
29	(13, 14)	15	0000000010011000	2	30	0000000010011000
30	(13, 14)	15	0000000010011000	2	30	0000000010011000
			max fitness = 3			avg fitness = 1.60

Table 7: **Report of Generation 1 for the Example**

No.	parents	pts	string	fitness	R-No.	reproduction string
1	(11, 28)	8	0000100010011000	5	1	0000100010011000
2	(11, 28)	8	0000000010001000	1	3	0001001010001000
3	(1, 15)	1	0001001010001000	5	4	0000101010011000
4	(1, 15)	1	0000101010011000	6	5	0001001000101000
5	(20, 29)	16	0001001000101000	5	6	0000100010011000
6	(20, 29)	16	0000100010011000	5	7	0001001000011000
7	(14, 8)	11	0001001000011000	6	8	0001000010001000
8	(14, 8)	11	0001000010001000	3	9	0010100010011000
9	(25, 17)	5	0010100010011000	6	10	0010100010011000
10	(25, 17)	5	0010100010011000	6	11	0000101010011000
11	(18, 19)	16	0000101010011000	6	12	0000000010011000
12	(18, 19)	16	0000000010011000	2	13	0000100010011000
13	(21, 5)	4	0000100010011000	5	14	0010101010001000
14	(21, 5)	4	0010101010001000	9	15	0010101000101000
15	(9, 2)	16	0010101000101000	9	15	0010101000101000
16	(9, 2)	16	0000101010011000	6	16	0000101010011000
17	(10, 16)	12	0010101000101000	9	17	0010101000101000
18	(10, 16)	12	0010100010011000	6	17	0010101000101000
19	(12, 6)	16	0001001000101000	5	18	0010100010011000
20	(12, 6)	16	0000100010001000	3	19	0001001000101000
21	(3, 13)	9	0000101010001000	5	20	0000100010001000
22	(3, 13)	9	0001001010001000	5	21	0000101010001000
23	(27, 22)	6	0000101010011000	6	23	0000101010011000
24	(27, 22)	6	0000101010001000	5	23	0000101010011000
25	(23, 24)	4	0000100010011000	5	24	0000101010001000
26	(23, 24)	4	0010101010011000	6	25	0000100010011000
27	(26, 4)	16	0001001010011000	6	26	0010101010011000
28	(26, 4)	16	0000101000001000	3	27	0001001010011000
29	(30, 7)	14	0000100010011000	5	29	0000100010011000
30	(30, 7)	14	0001000010011000	5	30	0001000010011000
			max fitness = 6			avg fitness = 5.30

Table 8: **Report of Generation 7 for the Example**

No.	parents	pts	string	fitness
1	(19, 13)	16	0011101010101000	17
2	(19, 13)	16	0011101010101000	17
3	(8, 9)	8	0011101010101000	17
4	(8, 9)	8	0011101010101000	17
5	(3, 11)	16	0011001010101001	0
6	(3, 11)	16	0010101000101000	9
7	(4, 6)	6	0010101010101000	11
8	(4, 6)	6	0011101010101000	17
9	(27, 25)	16	0010101001101000	11
10	(27, 25)	16	0011101010001000	14
11	(12, 30)	11	0010101010101000	11
12	(12, 30)	11	0011101010101000	17
13	(26, 21)	16	0011100010101000	15
14	(26, 21)	16	0011101010101000	17
15	(7, 17)	16	0011101010101000	17
16	(7, 17)	16	1011101010101000	0
17	(10, 14)	14	0011101010101000	17
18	(10, 14)	14	0010101000101000	9
19	(1, 5)	16	0011101010101000	17
20	(1, 5)	16	0011101010101000	17
21	(15, 18)	10	0010101001101000	11
22	(15, 18)	10	0011101010101000	17
23	(28, 2)	16	0010101010100001	0
24	(28, 2)	16	0011101010101000	17
25	(29, 20)	16	0011101010101000	17
26	(29, 20)	16	0011101010101000	17
27	(16, 22)	16	0011101010101000	17
28	(16, 22)	16	0011101010101000	17
29	(24, 23)	16	0011101010101000	17
30	(24, 23)	16	0011101010101000	17
	max fitness = 15		avg fitness = 13.80	

Table 9: **Report of Generation 25 for the Example**

toring the operation of a large network, one wants to monitor as few nodes as possible.

The *Vertex Cover* problem is also one of the basic NP-Complete problems. The optimization problem for the *Vertex Cover* problem is to find the minimum K in a graph G that satisfies all the constraints.

3.2.2 Representation and Evaluation Function

First, the *Vertex Cover* problem is encoded as a string. In a graph $G = (V, E)$, the vertices are numbered from 1 to n, where n is the total number of the vertices in a graph G. The *Vertex Cover* problem can be represented by an n-bit string $c_1c_2 \ldots c_n$, where $c_i = 0$ or 1, $i = 1, 2, \ldots, n$. Each bit position corresponds to the numbered vertex. $c_i = 1$ means vertex $v_i \in V'$, i.e. v_i is in the *Vertex Cover*.

Second, the penalty method is used to define the evaluation function as follows [RPLH89]:

evaluation function = the total number of vertices − the number of 1's in a string − *penalty*

penalty = (the number of uncovered edges by V')^2

When the number of uncovered edges by V' is small, the penalty is small. The string corresponding to the evaluation function still has the opportunity to survive in the next generation. When the number of uncovered edges by V' is bigger, the penalty increases dramatically due to the square function. The *fitness* of this string will be much smaller. The strings with higher *fitness* are expected to contribute more copies to the next generation.

When the value of the evaluation function is smaller than 0, the function returns 0.

If a subset $V' \subseteq V$ is a vertex cover, then $penalty = 0$.

3.2.3 Generating Input Graph at Random

For testing purposes, the input graph G is initialized by the following method:

Given the total number of vertices, n, in a graph G as input parameter and use matrix M to represent the graph.

$$M = \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & a_{24} & \cdots & a_{2n} \\ 0 & 0 & 0 & a_{34} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & a_{n-1n} \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

where $a_{ij} = 1$ or 0. $a_{ij} = 1$ means that there is an edge between vertex v_i and vertex v_j, with the same convention as used in representing a graph for the clique problem.

First, use function *flip* with 0.5 probability and different random seed to generate the matrix M. The function *flip* returns a value 1 or 0 according to the given probability parameter.

Second, generate k different random numbers between 1 and n, where $k \leq n$, and k is a input parameter. Then make the k vertices a complete graph (which is called a *clique*) with size k.

Third, compute the complement G_c of G as the input graph, where $G_c = (V, E_c)$ with $E_c = \{\{u,v\} : u, v \in V \text{ and } \{u,v\} \notin E\}$.

By using this method to generate the input graph randomly, we know what the optimal result should be. The best size of a vertex cover is $|V| - k$ due to the following relationship between *Clique* and *Vertex Cover* problems:

For any graph $G = (V, E)$ and subset $V' \subseteq V$:

V' is a *Clique* with maximum size K for G *if and only if* $V - V'$ is a *Vertex Cover* with minimum size $K' = |V| - K$ in the complement G_c of G.

Running the genetic algorithm, the number of uncovered edges by V' is computed as follows by using the matrix M:

(1) count the number of 1's in a string, put the corresponding position in a buffer;

(2) check whether every edge in E has at least one endpoint in V' by using matrix M. If $a_{ij} = 1$ and at least one of i and j belongs to V', set $a_{ij} = 0$.

(3) count the number of $a_{ij} = 1$ elements in matrix M. This is the number of uncovered edges by V'.

The C code for computing the fitness of each string is the following:

```
evaluation_function (string)
int string[maxstring];
{
    int i, j, res, count = 0;
    int buffer[maxstring], a[maxstring][maxstring];
```

```
penalty = 0;
for (i=1; i<=stringlength; i++)
    if (string[i] == 1)
    {
        count++;
        buffer[count] = i;
    }
for (i=1; i<=stringlength; i++)
    for (j=1; j<=stringlength; j++)
        a[i][j] = m[i][j];
for (i=1; i<=count; i++)
    for (j=buffer[i]+1; j<=stringlength; j++)
        if (a[buffer[i]][j] == 1)
            a[buffer[i]][j] = 0;
for (i=1; i<=stringlength; i++)
    for (j=1; j<=count; j++)
        if (a[i][buffer[j]] == 1)
            a[i][buffer[j]] = 0;
for (i=1; i<stringlength; i++)
    for (j=i+1; j<=stringlength; j++)
        if (a[i][j] == 1)
```

```
            penalty++;
    res = stringlength − count − penalty * penalty;
    if (res >= 0)
        return (res);
    else
        return (0);
}
```

3.2.4 Empirical Results

In the experiment, the function *flip* with the probability 0.95 is used to generate the initial population for the *Vertex Cover* problem randomly. Using larger probabilities to generate the initial population obtains vertex covers with bigger size first, improves the result in later generations, and also avoids the loss of feasible solutions.

For the *Clique* problem, the function *flip* with the probability 0.05 is used to generate the initial population. The relationship between *Vertex Cover* and *Clique* is that *Vertex Cover* is a minimization problem, and *Clique* is a maximization problem. A larger probability for the *Vertex Cover* and a smaller probability for the *Clique* can avoid the loss of feasible solutions for both problems.

The *reproduction* function is also implemented by using Baker's sampling algorithm [Bak87].

In the experiment, population size = 80, maximum generation = 200, crossover

probability = 0.5, mutation probability = 0.01 are used, and graphs drawn at random with 50, 60, 70, and 80 vertices are considered to test the algorithm.

Let us define the performance guarantee for the *Vertex Cover* problem as follows:

$$performance\ guarantee = \frac{the\ vertex\ cover\ size\ in\ the\ result}{the\ optimal\ vertex\ cover\ size} - 1.00$$

The results show that 61% of the tests get optimal results (the performance guarantee = 0.00), 39% of the tests get near optimal results and the average performance guarantee is 2.65%. These percentages come from the testings. Since the evaluation function of vertex cover differs from the evaluation function of clique, and the input data are also different for both problems, the results are different.

Some of the empirical results are reported in Tables 10 – 13, where

best generation = the generation for which we get the best result

seed = the input parameter for generating graphs randomly

clique size = the input parameter for generating the graph

best vertex cover size = $|V|-$ clique size

vertex cover size in result = obtained vertex cover size by running the genetic algorithm

performance guarantee = defined performance guarantee

best generation	seed	clique size	best vertex cover size	vertex cover size in result	performance guarantee
84	1	30	20	21	0.050
115	1	25	25	25	0.000
82	1	20	30	30	0.000
68	1	15	35	35	0.000
21	1	10	40	40	0.000
156	2	30	20	20	0.000
99	2	25	25	25	0.000
52	2	20	30	30	0.000
48	2	15	35	35	0.000
86	2	10	40	42	0.050
148	3	30	20	21	0.050
102	3	25	25	25	0.000
74	3	20	30	30	0.000
70	3	15	35	35	0.000
176	3	10	40	40	0.000
148	4	30	20	20	0.000
104	4	25	25	25	0.000
74	4	20	30	30	0.000
82	4	15	35	35	0.000
62	4	10	40	42	0.050
117	5	30	20	20	0.000
116	5	25	25	25	0.000
77	5	20	30	30	0.000
51	5	15	35	35	0.000
41	5	10	40	40	0.000
145	6	30	20	20	0.000
121	6	25	25	25	0.000
86	6	20	30	30	0.000
49	6	15	35	35	0.000
26	6	10	40	41	0.025

Table 10: **Total Vertices = 50 in Initial Graph**

best generation	seed	clique size	best vertex cover size	vertex cover size in result	performance guarantee
156	1	40	20	23	0.150
109	1	35	25	25	0.000
144	1	30	30	30	0.000
96	1	25	35	35	0.000
86	1	20	40	40	0.000
84	2	40	20	23	0.150
139	2	35	25	26	0.040
130	2	30	30	30	0.000
95	2	25	35	35	0.000
87	2	20	40	40	0.000
164	3	40	20	21	0.050
196	3	35	25	25	0.000
110	3	30	30	30	0.000
118	3	25	35	35	0.000
62	3	20	40	40	0.000
188	4	40	20	23	0.150
124	4	35	25	27	0.080
129	4	30	30	31	0.033
117	4	25	35	35	0.000
88	4	20	40	40	0.000
123	5	40	20	22	0.100
142	5	35	25	26	0.040
111	5	30	30	30	0.000
122	5	25	35	35	0.000
105	5	20	40	40	0.000
114	6	40	20	24	0.200
171	6	35	25	27	0.080
149	6	30	30	30	0.000
190	6	25	35	35	0.000
46	6	20	40	40	0.000

Table 11: **Total Vertices = 60 in Initial Graph**

best generation	seed	clique size	best vertex cover size	vertex cover size in result	performance guarantee
158	1	40	30	32	0.067
158	1	35	35	36	0.029
111	1	30	40	42	0.050
162	1	25	45	45	0.000
111	1	20	50	50	0.000
97	2	40	30	33	0.100
161	2	35	35	35	0.000
136	2	30	40	41	0.025
163	2	25	45	45	0.000
93	2	20	50	50	0.000
122	3	40	30	32	0.067
171	3	35	35	37	0.057
193	3	30	40	40	0.000
113	3	25	45	45	0.000
85	3	20	50	50	0.000
151	4	40	30	32	0.067
168	4	35	35	36	0.029
108	4	30	40	41	0.025
101	4	25	45	45	0.000
86	4	20	50	50	0.000
139	5	40	30	31	0.333
139	5	35	35	35	0.000
162	5	30	40	40	0.000
177	5	25	45	45	0.000
92	5	20	50	50	0.000
141	6	40	30	33	0.100
187	6	35	35	37	0.057
123	6	30	40	40	0.000
126	6	25	45	45	0.000
69	6	20	50	50	0.000

Table 12: **Total Vertices = 70 in Initial Graph**

best generation	seed	clique size	best vertex cover size	vertex cover size in result	performance guarantee
171	1	40	40	43	0.075
73	1	35	45	47	0.044
147	1	30	50	51	0.020
112	1	25	55	56	0.018
89	1	20	60	60	0.000
107	2	40	40	43	0.075
119	2	35	45	47	0.044
141	2	30	50	50	0.000
126	2	25	55	55	0.000
126	2	20	60	60	0.000
177	3	40	40	43	0.075
177	3	35	45	47	0.044
78	3	30	50	51	0.020
198	3	25	55	55	0.000
112	3	20	60	60	0.000
145	4	40	40	43	0.075
106	4	35	45	48	0.067
175	4	30	50	50	0.000
141	4	25	55	55	0.000
83	4	20	60	60	0.000
168	5	40	40	43	0.075
190	5	35	45	46	0.022
105	5	30	50	51	0.020
154	5	25	55	55	0.000
92	5	20	60	60	0.000
145	6	40	40	44	0.100
190	6	35	45	45	0.000
155	6	30	50	51	0.020
168	6	25	55	56	0.082
141	6	20	60	60	0.000

Table 13: **Total Vertices = 80 in Initial Graph**

3.3 Genetic Algorithm for the Max Cut Problem

3.3.1 Max Cut Problem

The *Max Cut* problem is also an NP-Complete problem which is very useful for network design.

The *Max Cut* Problem can be stated as follows [GJ79]:

INSTANCE: A graph $G = (V, E)$, weight $w(e) \in Z^+$ for each $e \in E$, and a positive integer K, where V is the set of vertices, and E is the set of edges, and Z^+ is the set of all positive integers.

QUESTION: Is there a partition of V into disjoint sets V_1 and V_2 such that the sum of the weights of the edges from E that have one endpoint in V_1 and one endpoint in V_2 is at least K?

The optimization problem for *Max Cut* problem is to find the maximum K that satisfies all the constraints.

3.3.2 Representation and Evaluation Function

The *Max Cut* problem can be represented by a string with length $|V|$, where $|V|$ is the total number of vertices in the graph G. The value of each bit in the string is 0 or 1. The vertices in a graph G are numbered from 1 to n, where $|V| = n$, and the bit position of the string corresponds to the numbered vertex. All the vertices which correspond to the bit value 1's in a string belong to the set V_1. The other vertices belong to the disjoint set V_2. For example, we have a graph with 10 vertices, and a

string representation for this graph is 1 0 0 1 1 0 1 1 0 1. This means that the vertices v_1, v_4, v_5, v_7, v_8, and v_{10} belong to set V_1, and vertices v_2, v_3, v_6, and v_9 belong to set V_2.

The evaluation function is defined as follows:

evaluation function = the sum of the weights of the edges from E that have one endpoint in V_1 and one endpoint in V_2.

The expected result is to get the maximum value of the evaluation function.

3.3.3 Implementation

A matrix is also used to represent the graph G. A *seed* is chosen to generate random numbers in range [1, 10] as the weights of the edges in the experiment. We provide a *density probability* which is the probability that the non-zero elements appear in the matrix as an input parameter. The *density probability* is a real number in (0, 1]. When the *density probability* is very small, the matrix will be sparse.

The function *flip* with probability 0.4 is used to generate the initial population for the *Max Cut* problem randomly. Baker's sampling algorithm is used to implement the *reproduction* function [Bak87].

In the experiment, population size = 500, maximum generation = 200, crossover probability = 0.5, mutation probability = 0.01, and hundreds of graphs drawn at random are considered to test the algorithm.

3.3.4 Comparisons of Genetic Algorithm and Greedy Algorithm

The greedy algorithm for the *Max Cut* problem works as follows:

- Initially, V_1 and V_2 are empty, and the variable $maxcut = 0$.
- Sort the edges by their weights in descending order.
- For each edge:

  (1) If two endpoints of the edge are not in V_1 and V_2, put one endpoint in V_1 and another endpoint in V_2. Add the weight of the edge to *maxcut*.

  (2) If one endpoint of the edge is in V_1 (or V_2), another endpoint is not in V_2 (or V_1), put the another endpoint in V_2 (or V_1). Add the weight of the edge to *maxcut*.

  (3) If one endpoint of the edge is in V_1, another endpoint is in V_2 , add the weight of the edge to *maxcut*.

- The result is in *maxcut*.

The same input graphs are used to test the genetic algorithm and greedy algorithm for the *Max Cut* problem. The empirical results show that the genetic algorithm provides better solutions on average than the greedy algorithm. When the matrix of the graph is very sparse (*density probability* ≤ 0.2), the greedy algorithm works well in most of the cases. When the *density probability* is higher, the genetic algorithm works better than greedy algorithm.

The Tables 14 – 17 show the comparisons of genetic algorithm and greedy algorithm for the *Max Cut* problem, where

best generation = the generation for which we get the best result by the genetic algorithm

seed = the input parameter for generating graphs randomly

density probability = the input parameter which is the probability that the non-zero elements should appear in the matrix

genetic algorithm = the results by running genetic algorithm

greedy algorithm = the results by running greedy algorithm

performance ratio = $\frac{genetic\ algorithm}{greedy\ algorithm}$

If the performance ratio is bigger than 1, the genetic algorithm is better than the greedy algorithm.

3.3.5 Comparisons of Genetic Algorithm Solutions and Optimal Solutions

For any weighted graph $G = (V, E)$, there is no polynomial deterministic algorithm for *Max Cut* problems. In order to compare optimal solutions with genetic algorithm solutions we have to use smaller input size, and calculate the optimal solutions by hand. With the same input to genetic algorithm programs, Table 18 shows the comparisons of genetic algorithm solutions and optimal solutions for the graph $G = (V, E)$ where $|V| = 10$. From table 18, we can see that the genetic algorithm solutions are identical to the optimal solutions calculated by hand for the random generated testing data.

3.4 Adapting Parameters

The research of adapting parameter settings is a branch of genetic algorithm theory. The parameter settings of a genetic algorithm have a significant impact on the performance of the genetic algorithm. Finding good parameter settings for a problem is a very hard task. Parameter settings involve population size, maximum generation, crossover probability, and mutation probability. We need to consider the limited resources for a problem when we set these parameters [Dav89, SCED89].

Two useful strategies have been employed by researchers in the genetic algorithm field to find good parameter settings. One was contained in Kenneth De Jong's thesis work [DJ75]; the other was described by John Grefenstette [Gre86]. Some new techniques for setting operator probabilities are also developed [Dav89, SCED89].

A large number of experiments are performed to test the effects of the different parameter settings for the *Clique* problem. In general, increasing the maximum generation can improve the result, but will require more computing time. For a set of parameters, if the best generation is close to the maximum generation, we can try to increase the maximum generation to get a better result. Otherwise, we can increase the population size to reduce the stochastic effects and improve long-term performance at a larger cost per generation. A high crossover probability and a low mutation probability can yield good performance for the *Clique* problem. The experimental results of adapting parameters are reported in Table 19, where population size = 80. crossover probability = 0.5. mutation probability = 0.01.

total vertices	best generation	seed	genetic algorithm	greedy algorithm	performance ratio
80	43	6	3839	3569	1.076
80	121	7	4042	3961	1.020
80	156	8	3920	3734	1.050
80	106	9	3944	3735	1.056
80	33	10	3791	3715	1.020
80	195	11	3905	3644	1.072
80	131	12	3838	3688	1.041
80	134	13	3850	3666	1.050
80	77	14	3750	3664	1.023
80	137	15	3710	3636	1.020
60	46	6	2374	2277	1.043
60	109	7	2284	2110	1.082
60	155	8	2312	2163	1.069
60	178	9	2419	2276	1.063
60	43	10	2243	2040	1.100
60	72	11	2340	2216	1.056
60	64	12	2326	2261	1.029
60	53	13	2193	2110	1.039
60	158	14	2122	1955	1.085
60	55	15	2172	2036	1.067
50	48	6	1576	1431	1.101
50	37	7	1666	1638	1.017
50	141	8	1617	1600	1.011
50	137	9	1610	1460	1.103
50	17	10	1607	1425	1.128
50	56	11	1682	1668	1.008
50	112	12	1667	1568	1.063
50	51	13	1578	1514	1.042
50	173	14	1639	1551	1.057
50	120	15	1601	1453	1.102

Table 14: **Density Probability = 0.4 in Initial Graph**

total vertices	best generation	seed	genetic algorithm	greedy algorithm	performance ratio
80	184	6	4791	4647	1.031
80	117	7	4998	4798	1.042
80	55	8	4856	4435	1.095
80	189	9	4790	4639	1.033
80	114	10	4790	4656	1.029
80	125	11	4875	4692	1.039
80	47	12	4671	4504	1.037
80	138	13	4842	4536	1.067
80	28	14	4549	4424	1.028
80	67	15	4505	4295	1.049
60	62	6	2956	2714	1.089
60	74	7	2798	2637	1.061
60	15	8	2782	2610	1.066
60	135	9	2880	2707	1.064
60	65	10	2749	2387	1.152
60	10	11	2763	2659	1.039
60	38	12	2816	2655	1.061
60	81	13	2720	2611	1.042
60	52	14	2626	2469	1.064
60	140	15	2727	2557	1.066
50	41	6	1968	1967	1.001
50	86	7	2054	1868	1.100
50	165	8	1944	1825	1.065
50	166	9	1945	1788	1.088
50	75	10	1980	1966	1.007
50	94	11	2095	1938	1.081
50	0	12	2013	1834	1.098
50	57	13	1897	1827	1.038
50	40	14	1958	1821	1.075
50	135	15	1987	1830	1.086

Table 15: **Density Probability = 0.5 in Initial Graph**

total vertices	best generation	seed	genetic algorithm	greedy algorithm	performance ratio
80	181	6	5694	5280	1.078
80	198	7	5821	5710	1.019
80	109	8	5718	5333	1.072
80	149	9	5691	5469	1.041
80	7	10	5519	5264	1.048
80	109	11	5784	5350	1.081
80	104	12	5643	5448	1.036
80	128	13	5735	5488	1.045
80	145	14	5476	5285	1.036
80	86	15	5408	5316	1.017
60	35	6	3487	3255	1.071
60	46	7	3366	3247	1.037
60	107	8	3235	3119	1.037
60	118	9	3427	3047	1.125
60	181	10	3224	3042	1.060
60	160	11	3181	3068	1.037
60	118	12	3313	3167	1.046
60	109	13	3170	3021	1.049
60	3	14	3133	2986	1.049
60	50	15	3206	3078	1.042
50	36	6	2258	2271	0.994
50	146	7	2376	2257	1.053
50	55	8	2316	2169	1.068
50	196	9	2324	2231	1.042
50	155	10	2276	2147	1.060
50	117	11	2409	2317	1.040
50	155	12	2330	2287	1.019
50	62	13	2274	2139	1.063
50	87	14	2345	2148	1.092
50	18	15	2313	2181	1.061

Table 16: **Density Probability = 0.6 in Initial Graph**

total vertices	best generation	seed	genetic algorithm	greedy algorithm	performance ratio
80	183	6	6595	6345	1.039
80	24	7	6734	6546	1.029
80	196	8	6702	6344	1.056
80	175	9	6558	6178	1.062
80	169	10	6463	6257	1.033
80	58	11	6618	6413	1.032
80	77	12	6610	6288	1.051
80	139	13	6600	6366	1.037
80	22	14	6373	5945	1.072
80	125	15	6313	6048	1.044
60	138	6	3890	3600	1.081
60	170	7	3843	3678	1.045
60	118	8	3776	3562	1.060
60	29	9	3919	3676	1.066
60	23	10	3784	3580	1.057
60	187	11	3741	3339	1.133
60	92	12	3780	3604	1.049
60	19	13	3659	3605	1.015
60	182	14	3663	3362	1.090
60	39	15	3724	3570	1.043
50	133	6	2604	2449	1.063
50	106	7	2721	2399	1.134
50	95	8	2662	2464	1.080
50	157	9	2635	2345	1.124
50	74	10	2642	2618	1.009
50	159	11	2727	2606	1.046
50	44	12	2643	2558	1.033
50	164	13	2619	2535	1.033
50	132	14	2650	2342	1.132
50	58	15	2674	2588	1.033

Table 17: **Density Probability = 0.7 in Initial Graph**

seed	density probability	optimal solutions	genetic algorithm	greedy algorithm
1	0.4	75	75	68
2	0.4	66	66	64
3	0.4	72	72	70
4	0.4	65	65	53
5	0.4	92	92	70
6	0.3	76	76	57
7	0.3	94	94	89
8	0.3	73	73	57
9	0.3	36	36	36
10	0.3	57	57	46
11	0.2	35	35	35
12	0.2	38	38	38
13	0.2	40	40	38
14	0.2	30	30	28
15	0.2	48	48	48

Table 18: **Comparisons of Results for Max Cut**

total vertices	maximum generation	best generation	seed	clique size in initial G	clique size in result	performance guarantee
50	100	59	1	35	33	94.29
50	200	149	1	35	35	100
60	100	65	1	35	33	94.29
60	200	113	1	35	35	100
70	100	92	1	35	34	97.14
70	200	154	1	35	35	100
90	100	96	1	20	18	90.00
90	200	116	1	20	20	100.00
100	100	70	1	20	19	95.00
100	200	111	1	20	20	100.00
50	100	79	3	35	34	97.14
50	200	136	3	35	35	100
60	100	95	3	30	29	96.67
60	200	110	3	30	30	100
70	100	75	3	30	29	96.67
70	200	102	3	30	30	100
80	100	58	3	30	29	96.67
80	200	129	3	30	30	100
90	100	79	3	30	29	96.67
90	200	132	3	30	30	100
50	100	84	5	35	34	97.14
50	200	160	5	35	35	100
60	100	96	5	35	34	97.14
60	200	124	5	35	35	100
70	100	64	5	35	34	97.14
70	200	118	5	35	35	100
80	100	99	5	35	33	94.29
80	200	154	5	35	35	100
90	100	96	5	30	29	96.67
90	200	118	5	30	30	100

Table 19: **Increasing the Maximum Generation to Improve the Results**

4 Parallel Genetic Algorithms

An important open question in the study of genetic algorithms is the optimal size of a population. The problem centers around the tradeoff between the amount of genetic search that can be done and the amount of real time available. If the population size is too small, then the genetic algorithm will have a, possibly improperly, constrained search space because of an insufficient number of schemata in the population. If the population size is too large however, an inordinate amount of time will be required to perform all the evaluations [PLG87].

One method of overcoming the genetic search vs. real time problem is to develop parallel genetic algorithms.

Genetic algorithms are inherently parallel. The Connection Machine system [Hil85] makes parallel implementations of these inherently parallel algorithms possible. This section describes the Connection Machine system, the C* parallel programming language, and the implementation of a genetic algorithm on the Connection Machine.

4.1 Connection Machine System

The Connection Machine system is a data parallel computing system which associates one processor with each data element (SIMD machine). This computing style exploits the natural computational parallelism inherent in many data-intensive problems.

The Connection Machine is an integrated system of hardware and software.

The hardware elements of the system include front-end computers that provide

the development and execution environments for the system software, a parallel processing unit of 64K processors that execute the data parallel operations, and a high-performance data parallel I/O system.

The front-end serves three primary functions in the Connection Machine system:

- It provides an applications development and debugging environment.
- It runs applications, transmitting instructions and data to the Connection Machine parallel processing unit.
- It provides maintenance and operations utilities for controlling the Connection Machine and diagnosing problems.

Each data processor has its own memory and an arithmetic-logic unit that can operate on variable-length operands. All processors have the same amount of memory, either 64K bits or 256K bits apiece. These processors not only can process the data stored in their memory, but also can be logically interconnected so that information can be exchanged among the processors. All these operations happen in parallel on all processors. Thus, the Connection Machine hardware directly supports the data parallel problem model.

The Connection Machine system implements data parallel programming constructs directly in hardware and microcode. Interprocessor communication is implemented by a special-purpose high speed network. Processors that hold interrelated data elements store pointers to one another. When data is needed, it is passed over

the network to the appropriate processors. The network supports completely general patterns of communication, but additional special hardware supports certain commonly used regular patterns of communication. Nearest-neighbor communication in a multidimensional rectangular grid is particularly efficient.

The system software is based upon the operating system or environment of the front-end computer. The visible software extensions are minimal. Users can program using familiar languages and programming constructs, with all the development tools provided by the front-end computer. Programs have normal sequential control flow; new synchronization structures are not needed. Thus, users can easily develop programs that exploit the power of the Connection Machine hardware.

4.2 C* Parallel Programming Language

The C* language, developed by Thinking Machines is a data parallel extension of the C programming language. C* programs are similar in style to C programs; the extensions are easy to learn. Parallel code looks like serial code, but is executed in all parallel processors simultaneously. C* features a single new data type, a synchronous execution model, and a minimal number of extensions to C statement and expression syntax. C* relies on existing C operators, applied to parallel data, to express such notations as broadcasting, reduction, and interprocessors communication in both regular and irregular patterns.

In C* a new keyword is introduced to specify parallel data which is called a

domain. All data in C* is divided into two kinds, scalar and multiple, which are described by using two new keywords, **mono** and **poly**. Also C* introduces a *member function* of a domain to specify the parallel codes. A data declaration that appears within a domain declaration or within a member function of a domain will by default be **poly**. All other declarations are by default **mono**. In terms of the computational model, **poly** quantities are precisely those residing in the memories of the processor array, and **mono** quantities are precisely those residing in the memory of the front-end computer.

All code in C* is also divided into two kinds: serial and parallel. Code that belongs to a domain is parallel, and may be executed by many data processors at once. All other code is serial, and is executed by the front-end as if it were ordinary sequential C code. The two types of code are distinguished by syntactic context: code may belong to a domain (and therefore be parallel) only as the body of a member function of the domain or as the substatement of a selection statement that selects the domain. The two types of code are written using the same syntax; all standard C expression operators and all standard C statement types may be used in parallel code in exactly the manner in which they are used in serial code.

4.3 Implementation of Parallel Genetic Algorithms

4.3.1 Parallel Data Structure

The primary parallel data structure in genetic algorithm is population. In a population, there are a lot of strings. Each string associates with a fitness and a penalty, and is stored in a processor. In the selection procedure, we need to generate the same number of string copies of the current generation in the next generation, and put the position of the string in a mating pool. In the crossover procedure, we need to generate the random crossover points. All of these operations can be done in parallel. The parallel data structure is described as the following:

```
domain population
{
  int x;
  int old_string[maxstring], new_string[maxstring];
  int old_fitness, new_fitness;
  int penalty, pool, jcross;
  int sum, tempix;
} pop_size[maxpopulation];
```

4.3.2 Evaluation Function

In each generation, the genetic algorithm uses strings as parameters to calculate the evaluation function. In serial code, if there are n strings in a population, the

evaluation function will need to be calculated n times. In a parallel algorithm, n processors can do this work at the same time. For example, a member function for *Clique* problem is described as the following:

```
population :: evaluation_function(string)
int string[maxstring];
{
  mono int i, j;
  int res = 0, count = 0;
  int buffer[maxstring];
  penalty = 0;
  for (j=1; j<=stringlength; j++)
      if (string[j] == 1)
      {
         count++;
         buffer[count] = j;
      }
  res = count * ( count - 1 ) / 2;
  for (i=1; i<count; i++)
      for (j=i+1; j<=count; j++)
         if (m[buffer[i]][buffer[j]] != 1)
            penalty++;
```

```
    res − = penalty * penalty;
    if (res >= 0)
        return (res);
    else
        return (0);
}
```

After calculating n fitnesses parallelly, we can use another member function to select the largest among all the fitnesses. The parallel code to select the largest fitness is the following:

```
population :: max()
{
    mono int j, max1, position;
    int ix, buffer, tempix;
    buffer = 0;
    tempix = 0;
    if (penalty == 0)
        buffer = old_fitness;
    max1 = (>?=buffer);
    ix = this - pop_size;
    if (pop_size[ix].buffer == max1)
        tempix = ix;
```

```
    position = (>?=tempix);
    sumfitness = (+= old_fitness);
}
```

4.3.3 Reproduction

The reproduction procedure is implemented by using a biased roulette wheel method [Gol89]. First calculate n sub-sums which are the sums of the fitness from the first string to the current string. Then generate n random numbers in range between 1 and the sum of fitness over all strings parallelly. If the random number is less than or equal to a sub-sum which is the smallest that can be found, the string corresponding to the sub-sum is copied to the next generation. The strings with a higher fitness value have a higher probability of contributing one or more offspring in the next generation.

For example, the n ($n = 8$) fitnesses are 5, 7, 2, 6, 1, 0, 4, 3. The sub-sums are 5, 12, 14, 20, 21, 21, 25, 28. The algorithm is represented in the Figure 11.

The Figure 11 can be recursively represented in Figure 12. Therefore the parallel time of this algorithm is:

$$T(n) = T(\frac{n}{2}) + 1$$

$$T(n) \cong logn$$

We use bitwise operations to implement this algorithm. The parallel calculation of n sub-sums is described as following member function.

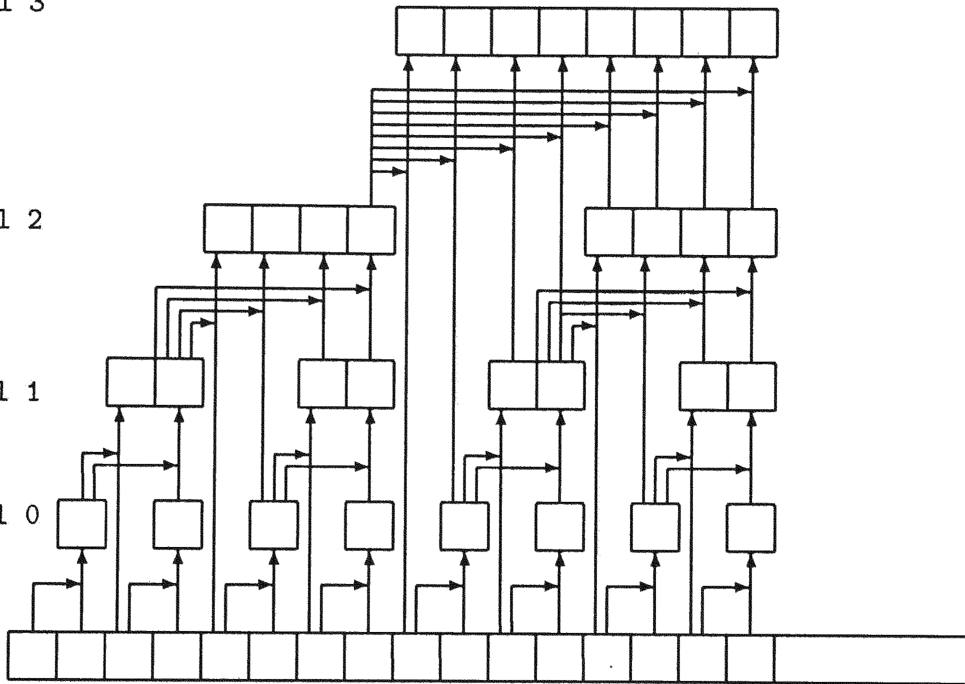

Figure 11: **Parallel Algorithm for Calculating Sub-sums**

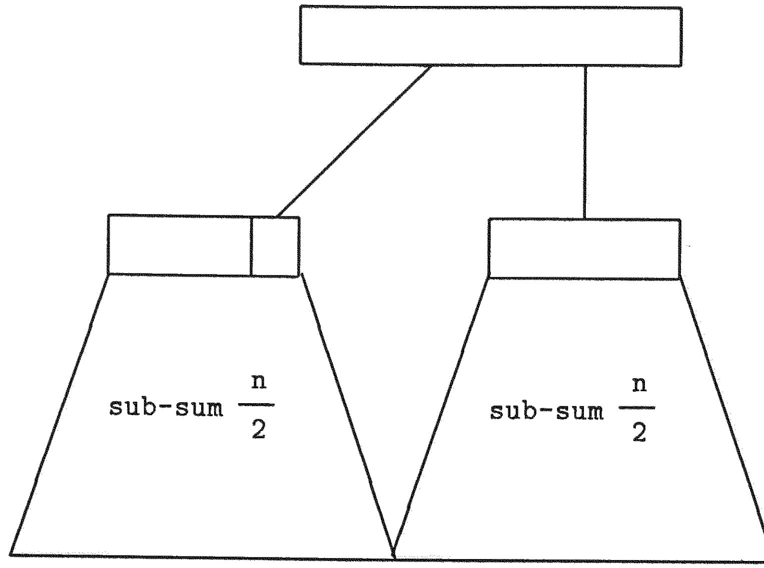

Figure 12: **Recursive Representation for Calculating Sub-sums**

```
population :: sum_sum(i)
int i;
{
  int idx2, ix;
  ix = this - pop_size;
  if ((ix >> i) & 1)
     {
      idx2 = (ix & ~ ((1 << i) - 1 )) - 1;
      pop_size[ix].sum = pop_size[ix].sum + pop_size[idx2].sum;
     }
```

```
}
```

The above member function is invoked log n times, for $i = 0, 1, 2, \ldots, \log n$, where n is the population size.

The parallel selection procedure uses binary search to find the positions:

```
population :: select()
{
  int first, last, ind, found;
  x = int ((double) rand() / range * sumfitness);
  first = 0;
  last = populationsize - 1;
  found = 0;
  while (first <= last && found == 0)
  {
    ind = (int) ((first + last) / 2 );
    if (ind == 0)
    {
      pool = 0;
      found = 1;
    }
    else
      if (pop_size[ind].sum >= x && pop_size[ind - 1].sum < x)
```

```
        {
            pool = ind;
            found = 1;
        }
        else
            if (pop_size[ind].sum >= x)
                last = ind - 1;
            else
                first = ind + 1;
    }
    if (found == 0)
        pool = 0;
}
```

4.3.4 Crossover

Parallel implementation of the crossover operator requires communication between processors. The even numbered processors are selected to communicate with the odd numbered processors (which is the even number minus 1). Within the code of a member function, the variable **this** is a parallel pointer to the currently executing domain instances. The member function for crossover is the following:

```
population :: crossover ( )
```

```
{
  int j, ix;
  x = (double) rand () / range;
  if (x <= cross_prob)
      jcross = (int) ((double) rand () / range * stringlength + 1);
  else
      jcross = stringlength;
  ix = this - pop_size;
  if (ix % 2)
  {
      for (j=1; j<=pop_size[ix].jcross; j++)
      {
        pop_size[ix-1].new_string[j] = pop_size[ix-1].old_string[j];
        pop_size[ix].new_string[j] = pop_size[ix].old_string[j];
      }
      if (pop_size[ix].jcross < stringlength)
        for (j=pop_size[ix].jcross+1; j <= stringlength; j++)
        {
          pop_size[ix-1].new_string[j] = pop_size[ix].old_string[j];
          pop_size[ix].new_string[j] = pop_size[ix-1].old_string[j];
        }
```

```
    }
  }
```

4.3.5 Mutation

The mutation operator can also be implemented parallelly. Each string is processed by one processor. The member function for mutation is the following:

```
population :: mutation()
{
  int j, ix;
  ix = this - pop_size;
  for (j=1; j<=stringlength; j++)
      if ((double) rand () /range <= mutation_prob)
      {
          if (pop_size[ix].new_string[j] == 1)
              pop_size[ix].new_string[j] = 0;
          else
              pop_size[ix].new_string[j] = 1;
      }
}
```

4.4 Analysis of the Parallel Genetic Algorithm

Suppose that each processor represents one string. The number of total processors used in the algorithm is the population size. We use n to represent the number of processors. The analysis of parallel genetic algorithm for each phase is:

- The evaluation phase: n processors calculate the fitness of the evaluation function at the same time by using n strings as parameters. There is no communication between processors. Therefore the time complexity of this phase is not affected by the number of processors.

- Selecting the largest fitness phase: Each processor has one fitness value. We want to select the largest fitness value in all processors and to know which processor occupies this value. C* programming language provides the operation of finding the maximum number in all processors by using the operator >?=. This operator requires $O(\log n)$ parallel time.

- Reproduction phase: This phase consists of two parts. One is to calculate the sub-sum which is the sum from the fitness value of the first processor to the fitness value of the current processor. A parallel algorithm was developed to do this work in $O(\log n)$ time. Another is the selection phase which is performed by generating the n random numbers at the same time, and then using binary search in n processors to get n positions parallelly. The time complexity of this phase is $O(\log n)$.

- Crossover phase: This phase needs communication between processors. The first processor communicates with the second; the third with the fourth; etc. All the communications can be done at the same time. It is not affected by the number of processors.

- Mutation phase: By using a member function, this phase can be done parallelly. It is also not affected by the number of processors.

When increasing the number of processors, the running time of the parallel genetic algorithm only has a little increase, and we can get better solutions. Some experimental results are shown in Tables 20 – 21. maximum generation = 20, crossover probability = 0.5, mutation probability = 0.01 are used in the experiments. The time is counted in seconds.

seed	clique size in initial G	clique size in results	best generation	number of processors	CM time (seconds)	front end virtual time
1	5	4	16	100	14.53	19.70
2	5	4	10	100	14.59	19.92
3	5	4	3	100	14.63	19.67
4	5	4	13	100	14.85	19.71
5	5	4	18	100	14.60	19.54
1	5	5	15	1000	15.97	21.00
2	5	4	5	1000	16.07	21.03
3	5	5	14	1000	16.13	21.07
4	5	4	8	1000	16.04	21.08
5	5	4	0	1000	15.19	20.11
1	5	5	15	4000	18.89	23.05
2	5	4	3	4000	19.13	22.22
3	5	5	8	4000	19.78	22.30
4	5	5	4	4000	22.68	23.56
5	5	5	14	4000	19.16	23.61
1	6	4	16	100	16.73	20.26
2	6	5	13	100	14.77	19.84
3	6	5	13	100	19.49	20.81
4	6	5	18	100	22.00	21.11
5	6	4	3	100	18.66	21.08
1	6	6	19	1000	16.11	21.13
2	6	5	7	1000	16.09	20.89
3	6	5	7	1000	15.94	20.92
4	6	5	16	1000	16.17	21.23
5	6	5	8	1000	16.02	21.07
1	6	6	19	4000	16.88	22.08
2	6	5	7	4000	16.77	21.88
3	6	5	4	4000	16.76	21.83
4	6	6	10	4000	16.79	21.92
5	6	5	8	4000	16.75	21.95

Table 20: **Total Vertices = 10 in Initial Graph G**

seed	clique size in initial G	clique size in results	best generation	number of processors	CM time (seconds)	front end virtual time
1	6	5	10	100	54.79	60.14
2	6	4	4	100	53.47	59.64
3	6	4	4	100	59.16	60.52
4	6	5	5	100	49.48	58.74
5	6	4	8	100	58.90	61.50
1	6	7	17	1000	50.50	60.69
2	6	5	10	1000	61.19	62.79
3	6	5	5	1000	55.57	61.78
4	6	6	19	1000	49.41	60.81
5	6	5	8	1000	52.32	61.42
1	6	7	17	4000	49.60	60.96
2	6	5	0	4000	50.32	61.20
3	6	5	1	4000	53.81	61.73
4	6	6	15	4000	65.02	63.26
5	6	5	3	4000	49.76	61.03
1	8	5	3	100	52.47	62.05
2	8	5	13	100	51.71	60.32
3	8	5	11	100	54.48	58.69
4	8	5	5	100	49.76	61.11
5	8	5	13	100	47.84	58.90
1	8	7	17	1000	66.82	68.92
2	8	5	6	1000	63.45	69.28
3	8	7	12	1000	51.89	66.14
4	8	6	13	1000	53.79	68.23
5	8	6	14	1000	48.64	60.10
1	8	7	17	4000	55.94	70.10
2	8	6	14	4000	56.85	67.39
3	8	7	12	4000	52.53	66.95
4	8	6	12	4000	60.94	69.12
5	8	6	4	4000	51.53	63.07

Table 21: **Total Vertices = 20 in Initial Graph G**

5 Advanced Techniques

Based on mechanisms of natural selection and genetics, by simulating some features of biological evolution, genetic algorithms can be used to solve problems where traditional search and optimization methods are less effective [Dav91].

There are some encoding techniques for representing the problems. Bit string representation is the most common encoding technique used by genetic algorithm researchers. Bit string representation is theoretically tractable, and its simplicity makes it easy to prove theorems. Other encoding techniques are numerical representation and order-based representation. Numerical representation uses real numbers in a string. Order-based representation uses a random permutation to represent the problem.

There are also several algorithms for the selection procedure [Bak87].

The crossover techniques often used are one point crossover and two point crossover. Two point crossover is also a method for sharing information between two successful individual strings. Two point crossover can be implemented by selecting two random integer positions in the string and exchanging the segments from the first position to the second position with another string similarly partitioned.

Suppose that A and B are two parent strings selected randomly from the mating pool, and i, j are two random integer positions.

$$A = a_1 a_2 \ldots a_{i-1} a_i a_{i+1} \ldots a_{j-1} a_j a_{j+1} \ldots a_n$$

$$B = b_1 b_2 \ldots b_{i-1} b_i b_{i+1} \ldots b_{j-1} b_j b_{j+1} \ldots b_n$$

After *crossover*, the children are

$$A = a_1 a_2 \ldots a_{i-1} b_i b_{i+1} \ldots b_{j-1} b_j a_{j+1} \ldots a_n$$

$$B = b_1 b_2 \ldots b_{i-1} a_i a_{i+1} \ldots a_{j-1} a_j b_{j+1} \ldots b_n$$

6 Conclusions

In this thesis a genetic approach to generating solutions for the *Clique, Vertex Cover*, and *Max Cut* problems is presented. All of these problems are NP-Complete problems. The techniques used are Baker's sampling algorithm, the biased roulette wheel method for the selection procedure, and one point crossover operator.

A series of experimental results are reported in this thesis. The experimental results show that 77.33% of tests get optimal solutions, the average performance guarantee is 98.80% for the clique problems; 61% of tests get optimal solutions, the average performance guarantee is 2.65% for the vertex cover problems; and when the *density probability* is higher, the genetic algorithm works better than the greedy algorithm for the max cut problems. For the smaller size random generated testing data, the genetic algorithm solutions are exactly identical to the optimal solutions for the max cut problems. Genetic algorithms can provide efficient search heuristics for solving combinatorial optimization problems.

Genetic algorithms are easy to code. All the operator functions and initial population, statistics and report functions can be used for other problems. Therefore, when solving a new problem, what we need to do is to encode the problem to a string representation, input the values of the problem and define the evaluation function properly.

The evaluation function must have a nonnegative value. In many problems, the objective is more naturally stated as the minimization problems, such as the *Vertex*

Cover problem. We need to map the natural objective function to an evaluation function. The following transformation is commonly used in genetic algorithms:

$$
\begin{aligned}
f'(x) &= C_{max} - f(x) \quad when\ f(x) < C_{max} \\
&= 0 \qquad\qquad\qquad otherwise
\end{aligned}
$$

where $f(x)$ is the natural objective function, $f'(x)$ is the evaluation function, C_{max} is a coefficient. For the *Vertex Cover* problem, we use the total number of vertices as the coefficient which is the largest value of the objective function.

In a problem with one or more constraints, a cost or penalty with all constraint violations should be included in the evaluation function. The penalty method can be used to solve this kind of problem. We usually square the violations of the constraints.

Genetic algorithms are inherently parallel. Multi-processors make it possible to study genetic algorithms on larger domains. Because of the availability of larger population sizes, we can improve the results considerably by implementing the parallel genetic algorithms. In this thesis, a parallel genetic algorithm is implemented on the Connection Machine system.

7 Open Problems and Future Work

The field of genetic algorithms is growing rapidly. Some applications of genetic algorithms demonstrate commercial potential. We can divide genetic algorithms into three major categories:

- Genetic algorithm theory
- Optimization
- Machine learning

In those three major categories we can list the most interesting open problems and future work as follows:

- finding new representation techniques;
- defining evaluation function with vector fitness;
- developing new genetic operators;
- discovering more efficient parallel selection algorithms;
- changing genetic algorithm parameters dynamically;
- applying genetic algorithms to more application areas that traditional search lacks efficient algorithms;
- applying genetic algorithm techniques to the neural network computing architectures.

We believe that the scientists will have more understandings on the processes of biological evolutions and the behaviors of natural intelligence. Therefore the simulation of biological evolution processes and natural intelligence behaviors will be implemented by computer scientists more naturally and practically. We have no doubt that genetic algorithms will play very important role in the fields of AI (Artificial Intelligence) and NI (Natural Intelligence) for the next decade.

References

[Ant89] Jim Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[Aus90] Scott Austin. An introduction to genetic algorithms. *AI Expert*, March 1990.

[Bak87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.

[BG87] Clayton L. Bridgs and David E. Goldberg. An analysis of reproduction and crossover in a binary-coded genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.

[Dav85] Lawrence Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

[Dav89] Lawrence Davis. Adaptive operator probabilities in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[Dav91] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[DC87] Lawrence Davis and Susan Coombs. Genetic algorithms and communication link speed. In *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.

[DJ75] Kenneth A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigen, 1975.

[DJS89] Kenneth A. De Jong and Willian M. Spears. Using genetic algorithms to solve np-complete problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[DS87] Lawrence Davis and Martha Steenstrup. Genetic algorithms and simulated annealing: An overview. In *Genetic Algorithms and Simulated Annealing*, 1987.

[Eng85] Amold C. Englander. Machine learning of visual recognition using genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

[FW90] D'Ann Fuquay and Darrell Whitley. Genetic algorithm solutions for the traveling salesman problem. In *ACM 28th Annual Southeast Regional Conference*, 1990.

[GB89] John J. Grefenaatette and James E. Baker. How genetic algorithms works: A critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, 1979.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[Gre86] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-16, No. 1, January/February 1986.

[Hil85] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[Hol75] John H. Holland. *Adaption in Natural and Artificial Systems.* University of Michigen Press, 1975.

[LK89] C. B. Lucasins and G. Kateman. Application of genetic algorithms in chemometric. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[PLG87] Chrisila B. Pettey, Michael R. Leuze, and John J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.

[RPLH89] Jon T. Richardson, Mark R. Palmer, Gunar E. Liepins, and Mike Hilliard. Some guidelines for genetic algorithms with penalty functions. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[SCED89] J. David Schaffer, Richart A. Caruana, Larry J. Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithm for function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[Sch85] J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

[Wil87] Stewart W. Wilson. A genetic algorithm and biological development. In *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.