1997

A Comparison of genetic and other algorithms for the traveling salesman problem

Martin F. Schlapfer
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Schlapfer, Martin F., "A Comparison of genetic and other algorithms for the traveling salesman problem" (1997). *Master's Theses*. 1528.
DOI: <https://doi.org/10.31979/etd.g3ye-jv5r>
https://scholarworks.sjsu.edu/etd_theses/1528

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A Comparison of Genetic and other Algorithms for the Traveling Salesman Problem

A Thesis

Presented to

**The Faculty of the Department of Mathematics and Computer Science
San Jose State University**

In Partial Fulfillment

**of the Requirements for the Degree
Master of Science**

by

Martin F. Schlapfer

August 1997

UMI Number: 1386229

UMI Microform 1386229
Copyright 1997, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© 1997
Martin F. Schlapfer
ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF MATHEMATICS

Dr. John Mitchem

Dr. Edward Schmeichel

Dr. Sam Khuri

APPROVED FOR THE UNIVERSITY

M. Lou Lewandowski

ABSTRACT

A Comparison of Genetic and other Algorithms for the Traveling Salesman Problem

by Martin F. Schlapfer

In this paper I give an introduction to the Traveling Salesman Problem, then I discuss the role of Genetic Algorithms and how they work. In Chapter Three I discuss the application of the Genetic Algorithm to the Traveling Salesman Problem. In this chapter I discuss the resources that we drew from, TSPLIB and LibGA, and I also discuss two crossover operators for the TSP, (Bui-Moon and Whitley's Edge Recombination). In Chapter Four I provide results for experiments applied using both operators combined with the two most common GA models and various local optimization procedures. In Chapter Five I present other approximation heuristics for the Traveling Salesman Problem, specifically the Simulating Annealing approach. Also discussed are traditional methods such as the Christofides Algorithm. In this chapter I provided some results of experiments using these other heuristics and comparisons to the Genetic Algorithm. Then in Chapter Six I finish with a discussion of what research is being presented in Johnson et al. and give my own conclusions about the promises of GA with respect to the Traveling Salesman Problem.

ACKNOWLEDGMENTS

I would like to express my appreciation to Dr. John Mitchem for providing an excellent learning experience for me at San Jose State University. Dr. Mitchem, thank you very much for all the hours you have spent with me on this Thesis. Your insights have proven invaluable in many instances. Thank you Dr. Schmeichel and Dr. Khuri for your useful comments and willingness to sit on the committee. Also, I would like to extend a big thank you to my wife, Sheila, who put up with my late nights in which I often studied tangents to this Thesis that would nowhere make it in here.

TABLE OF CONTENTS

CHAPTER ONE

The Traveling Salesman Problem

Introduction	1
Applications	1
NP-Completeness	2
Assumptions: Triangle Inequality and Euclidean Distances	2
Bounds in approximating the Traveling Salesman Problem	3
Ratio Bounds and Triangle Inequality	3
Lower Bounds	3
Held-Karp Lower Bound	4
Example of Subgradient Optimization Method	7
Upper Bounds	9
Expected Optimal Tour Weight for Random Instances in the Euclidean Plane	10

CHAPTER TWO

Genetic Algorithms

Introduction	11
Encoding Scheme - Representing the Population	12
Binary Encoding	13
Order Based Encodings	13
Locus Based Encodings	13
Selecting the Initial Population	14
Stopping Criteria	14
Selection	15
Objective Function and the Fitness Function	15
Fitness Proportionate Selection	15
Rank Based Selection	17
Reproduction	23
Mutation	24
Simple Example	24

Theory	26
Schema Theory	27
Implicit Parallelism	28
Discussion	29

CHAPTER THREE

Genetic Algorithms Applied to the Traveling Salesman Problem

Resources	30
TSPLIB	30
Random Instances	31
LibGA Software Package	32
Population Representation and Initialization	32
Reproduction	32
Genetic Edge Recombination Operator (GERO)	33
Bui-Moon Recombination Operator	36
Preprocessing	42
Mutation	43
Local Optimization	43
Martin's Local Optimization Algorithm	45
Discussion	46

CHAPTER FOUR

Comparison of Various Genetic Operators

Steady State Model and the Generational Model	47
Elitism	48
Diversification or "Dropping the Bomb"	48
Stopping Criteria	49
Effects of Local Optimization	49
Effect of Changing from the Steady State Model to the Generational Model	54

CHAPTER FIVE

Discussion of Other Approximation Algorithms

Nearest Neighbor Approximation Heuristic	60
--	----

Minimum Spanning Tree Approximation Heuristic	62
Christofides Approximation Heuristic	65
Simulated Annealing	66
Summary	68

CHAPTER SIX

Summary and Other Interesting Works

Discussion of the Johnson et al. Paper	70
Conclusions	79

APPENDICES

Appendix A: TSPLIB	82
Appendix B: LibGA	85

REFERENCES

CHAPTER ONE

The Traveling Salesman Problem

Introduction

The Traveling Salesman Problem (TSP) is a classic example of a combinatorial optimization problem. A combinatorial optimization problem is a problem in which there exists a combination of elements of the problem that optimizes that instance. In the TSP, a traveling salesman would like to visit each city in his territory while minimizing the cost of traveling among all the cities. He only visits a city once and then goes on to the next city, returning to where he began after visiting all the cities in his territory. The elements to consider for optimization in the TSP are the ordering of the cities in the salesman's territory. If each pair of cities has a cost associated with traveling between them, then there exists an ordering of cities which minimizes the cost of the salesman's tour of the cities in his territory.

The TSP can be stated in various ways depending on the context of the discipline in which it is used. In the field of graph theory, the salesman's territory is represented by a complete graph. Each vertex in the graph is a city in the salesman's territory. The weight of an edge between two vertices in the graph is the cost associated with traveling between the respective cities. Thus, the TSP can be stated as follows:

Given a complete graph G and a weighting function, w , on the edges of G , find a permutation, p , of the vertices in G that minimizes:

$$\left[\sum_{i=1}^{N-1} w(p_i, p_{i+1}) \right] + w(p_N, p_1) \quad (1.1)$$

where $w(p_i, p_j)$ is the weight of the edge in G between the i^{th} and j^{th} vertices in the permutation given by p , and N is the size of the vertex set of G .

Applications

The TSP has applications in many industries. The Printed Circuit Board (PCB) problem

is possibly the simplest example. In the PCB problem, a manufacturer of printed circuit boards would like to determine the most economical procedure to drill holes on a circuit board. In this problem the manufacturer wants to minimize the distance that the tool must travel between the positions of the holes on the circuit board, thus increasing productivity. Other examples for which the TSP is used include scheduling and vehicle routing problems [Reinelt 1994].

NP-Completeness

The TSP belongs to a class of problems known as NP-Complete problems. NP-Complete problems are a special class of NP problems. NP problems are those decision problems for which a proposed solution can be *verified* in polynomial time. NP-Complete problems carry the further distinction that if any one NP-Complete problem can be decided in polynomial time, then every problem in NP can be decided in polynomial time [Cormen et al. 1990]. This follows from the fact that all problems in NP can be reformulated in polynomial time to any particular NP-Complete problem. Since the TSP belongs to the class of NP-Complete problems, most experts believe that it is unlikely that a polynomial algorithm to solve it will be found. Thus, much research has emphasized finding good approximations to optimal solutions.

Assumptions: Triangle Inequality and Euclidean Distances

The majority of research involving these approximation algorithms considers instances of the TSP where the triangle inequality holds. The triangle inequality means that given a weighting function on the edges in a graph, the following holds for any three vertices, A, B, and C, in the graph:

$$w(A,C) \leq w(A,B) + w(B,C) \quad (1.2)$$

That is, the weight assigned in traveling directly from vertex A to vertex C, is no more than the weight of first traveling to vertex B and then traveling from vertex B to vertex C.

Algorithms discussed in this work involve graphs whose vertices correspond to points in the plane and the weight of the edge joining two vertices is the Euclidean distance between

the two vertices. For two vertices, A and B, the Euclidean distance between them, denoted $d(A,B)$, is defined as:

$$d(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2} \quad (1.3)$$

where, $A = (A_x, A_y)$ is the location of vertex A in Euclidean two space. Our weighting function for edges $w(A,B)$ now becomes the Euclidean distance metric $d(A,B)$. The Euclidean distance metric clearly satisfies the triangle inequality.

The set of Traveling Salesman Problems that satisfy the Euclidean distance metric are known as the set of Euclidean TSP's. Throughout this work, unless otherwise noted, we will assume that we are working with complete graphs G whose vertices are given by (x,y) coordinates on the two dimensional Euclidean plane and the weighting function for edges will be the Euclidean distance function as given in equation 1.3. The vertices of G , denoted $V(G)$ is given by the set $\{v_1, v_2, \dots, v_N\}$ where $N = |V(G)|$.

Bounds in approximating the Traveling Salesman Problem

Ratio Bounds and Triangle Inequality

For some approximation algorithms there exist well known constant ratio bounds. That is, the approximation algorithm is guaranteed to do no worse than some constant ratio to the optimal. For these approximation algorithms, the existence of the constant ratio bound depends on the assumption that the triangle inequality holds. If the triangle inequality does not hold, it can be shown that no approximation algorithm with a constant ratio bound can exist to solve the TSP unless $P = NP$ [Cormen et al. 1990]. Thus, the triangle inequality is a very important assumption, and the TSP seems to become a harder problem without it.

Lower Bounds

The ratio bounds for a significant number of the traditional TSP problems that will be presented are based on a well known lower bound which we describe below. We first describe a somewhat crude lower bound, namely the weight of a **minimum spanning tree**,

for the optimal solution to the TSP. That is, the weight of the optimal tour $w(\tau^*)$ in G is at least as much as the weight of a minimum spanning tree T of G . The following algorithm based on Prim's method finds a minimum spanning tree in G [Cormen et al. 1990].

Algorithm **MST**(G)

- (1) Let $A = V(G)$ and $T = \{\}$.
- (2) For each vertex v_i in A let $\text{weight}(v_i) = \infty$ and $p(v_i) = 0$.
- (3) While A is not empty do
 - (3.1) Let v_i be a vertex of minimum weight in A .
 - (3.2) Let $A = A - \{v_i\}$ and $T = T \cup \{v_i\}$.
 - (3.3) For each vertex $v_j \in A$, if $w(v_i, v_j) < \text{weight}(v_j)$ then let $\text{weight}(v_j) = w(v_i, v_j)$ and $p(v_j) = i$.
- (4) Return $p(v_i)$ for all i .

Using the Fibonacci Heap data structure, algorithm **MST** finds a minimum spanning tree in $O(N^2)$ time [Cormen et al. 1990].

Held-Karp Lower Bound

Tighter lower bounds for an instance of the TSP are obtained by using an iterative method involving minimum one-trees [Held-Karp 1970, Lawler 1985]. A **one-tree** of G is a spanning tree of the graph $G - v_1$ together with two edges incident to vertex v_1 . A **minimum one-tree** of G is a minimum spanning tree on $G - v_1$ together with two edges of least weight incident to vertex v_1 . We label the one-trees of G with $H_1, H_2, \dots, H_m$ and denote their associated weights, $w(H_1), w(H_2), \dots, w(H_m)$.

Following these definitions, consider the weight $w(\tau^*)$ of an optimal tour in G . τ^* is certainly a one-tree, but is not necessarily a minimum one-tree in G . Thus $w(\tau^*)$ has weight greater than or equal to the weight of a minimum one-tree of G .

The iterative method that Held and Karp introduced finds a vector $\lambda = (\lambda_1, \lambda_2, \lambda_3, \dots, \lambda_N)$ that represents an increased weight associated with visiting each vertex. For example, if $w(v_i, v_j)$ represents the weight of traveling between vertices v_i and v_j then the new weight becomes $w(v_i, v_j) + \lambda_i + \lambda_j$. We denote G_λ as the graph G with the modified weights induced by λ . Since in any tour of G every vertex is entered exactly once and exited exactly once, the weight on any tour in G is increased by exactly twice the sum of the weights of λ .

In particular, the weight of an optimal tour becomes:

$$w(\tau^*) + 2 \sum_{i=1}^N \lambda_i \quad (1.4)$$

where τ^* is an optimal tour. (Note: τ^* is an optimal tour in G_λ as well, since every tour in G increases by exactly the same amount).

However, for a given λ , the weight increase for one-trees will vary. Letting d_{ik} denote the degree of vertex v_i in the one-tree H_k of G , then for any one-tree H_k in G the increased weight becomes:

$$w(H_k) + \sum_{i=1}^N \lambda_i d_{ik} \quad (1.5)$$

and represents the weight of the one-tree H_k in G_λ .

The method of Held and Karp seeks to find a vector λ that minimizes the difference between equation 1.4 and 1.5 over the minimum one-trees of G_λ . That is, we want to minimize:

$$w(\tau^*) + 2 \sum_{i=1}^N \lambda_i - w(H_k) - \sum_{i=1}^N \lambda_i d_{ik} \quad (1.6)$$

Since $w(\tau^*)$ is constant we can neglect it and therefore we obtain:

$$2 \sum_{i=1}^N \lambda_i - w(H_k) - \sum_{i=1}^N \lambda_i d_{ik} \quad (1.7)$$

Minimizing equation 1.7 is the same as maximizing:

$$w(H_k) + \sum_{i=1}^N \lambda_i d_{ik} - 2 \sum_{i=1}^N \lambda_i \quad (1.8)$$

Thus we want to find a vector λ that maximizes equation 1.8 over the minimum one-trees of G_λ . Hence we want to maximize:

$$L(\lambda) = \min_k \left[w(H_k) + \sum_{i=1}^N \lambda_i d_{ik} \right] - 2 \sum_{i=1}^N \lambda_i \quad (1.9)$$

If we find a λ that yields a minimum one-tree of G_λ for which every vertex is of degree 2, then the minimum one-tree is an optimal tour of G_λ . It follows easily that it is also an optimal tour of G . The difference between the result of maximizing equation 1.9 over the minimum one-trees of G and the optimal tour weight $w(\tau^*)$ is what is known as the Held-Karp gap. That is, if maximizing equation 1.9 does not result in a one-tree that is a tour, then equation 1.9 is a lower bound (namely the Held-Karp lower bound) on the optimal tour weight.

One method used for attempting to maximize equation 1.9 is known as **subgradient optimization**. The method begins with λ as the zero vector and at each iteration j updates the vector λ by:

$$\lambda_i^{j+1} = \lambda_i^j + t^j(d_i^j - 2), \quad i \in V \quad (1.10)$$

where λ_i^j is the i^{th} coordinate of λ after iteration j and,

$$t^j = \alpha (U - L(\lambda^j)) / \sum_{i \in V} (d_i^j - 2)^2 \quad (1.11)$$

Here U is an upper bound which we specify below, $\alpha = 2$, $L(\lambda^j)$ is the result of equation 1.9 applied to λ^j obtained in the j^{th} iteration, and d_i^j is the degree of vertex v_i at iteration j of the minimum one-tree found in equation 1.9. The constant α can be periodically reduced by some factor. Also, the upper bound U can decrease from iteration to iteration, but never increases, and is never bigger than the U given in our version. See Lawler 1985 for further details.

The result of algorithm **SubgradientOptimization** is either: 1) an optimal tour (by the fact that a minimum one-tree in G_λ of which all vertices are of degree two was found), or 2) an approximation of the lower bound (the Held-Karp lower bound) on the optimal tour weight as described earlier.

Algorithm **SubgradientOptimization**(G)

- (1) Let $j = 0$.
- (2) Let U be three-halves the weight of a MST of G .
- (3) Let λ^j be the zero vector.
- (4) Construct G_{λ^j} by modifying the edge weights of G such that if $w(v_k, v_i)$ is the weight of the edge between vertices v_k and v_i in G , then the weight of the edge between vertices v_k and v_i in G_{λ^j} becomes $w(v_k, v_i) + \lambda^j_k + \lambda^j_i$.
- (5) Find a minimum one-tree H^j in G_{λ^j} .
- (6) While H^j is not a tour and $w(H^j) < U$ do
 - (6.1) Let d_i^j be the degree of vertex v_i in H^j .
 - (6.2) Use equation (1.7) to find λ^{j+1} .
 - (6.3) Let $j = j+1$.
 - (6.4) Perform steps (4) and (5).

Example of Subgradient Optimization Method

Using equations 1.10 and 1.11 we give the following example from Lawler et. al 1985. Consider the graph in Figure 1.1. Initially, $\lambda_i^0 = 0$ and $U = 25$ (any MST has weight 17). Note that in this example the triangle inequality does not hold (see the edges between vertices 3, 4, and 5) and U is chosen slightly differently from the algorithm above. Also for convenience, the graph is not complete. The minimum one-tree has weight 21 and is shown by the thick lines in Figure 1.1. Note $L(\lambda^0) = 21 + 0 - 0 = 21$. With the first iteration through the while loop we obtain:

Figure 1.1

$$d_i^0 = (2, 2, 4, 1, 1, 3, 2, 1),$$

$$t^0 = 2(25-21)/8 = 1, \text{ and}$$

$$\lambda_i^1 = (0, 0, 2, -1, -1, 1, 0, -1).$$

Using the vector λ^1 the graph G_{λ^1} is obtained by modifying the edge weights of G such that if $w(v_i, v_j)$ represents the weight of the edge between vertices v_i and v_j , then the new weight becomes $w(v_i, v_j) + \lambda_i^1 + \lambda_j^1$. The graph G_{λ^1} is shown in Figure 1.2 and the minimum one-tree is shown by the thick lines. The minimum one-tree has weight 23 and $L(\lambda^1) = 24 + (-1) - 0 = 23$. Moving on to the next iteration we obtain:

$$d_i^1 = (2, 3, 2, 2, 2, 1, 2, 2),$$

$$t^1 = 2(25-23)/2 = 2, \text{ and}$$

$$\lambda_i^2 = \lambda_i^1 + t^1(d_i^1 - 2) = (0, 2, 2, -1, -1, -1, 0, -1).$$

In step 6.4 of the algorithm the graph G_{λ^2} is obtained by augmenting the edge weights of G using the vector λ^2 . The graph G_{λ^2} is shown in Figure 1.3 and the minimum one-tree of G_{λ^2} is shown by the thick lines.

Since the vertices of the minimum one-tree found all are of degree two, we have in fact

Figure 1.2

Figure 1.3

found an optimal tour of G_{λ^2} (which is therefore an optimal tour of G) and the algorithm halts. Although this is a simple example, it encapsulates the idea of the subgradient optimization method.

An interesting aside as noted by Lawler, et al. is that by Cayley's formula, there are $N^{(N-2)}$ distinct spanning trees in the complete graph K_N . Therefore, the number of distinct one-trees of K_N is,

$$\binom{N-1}{2}(N-1)^{(N-3)} = \frac{1}{2}(N-2)(N-1)^{(N-2)}.$$

That is, there are $N-1$ choose two ways of picking two edges incident on vertex v_1 and $(N-1)^{(N-3)}$ distinct spanning trees in $K_N - \{v_1\}$. Since K_N has $\frac{1}{2}(N-1)!$ tours starting at v_1 , then on average, one out of every $(N-1)^{(N-3)}/(N-3)!$ one-trees is a tour [Lawler 1985].

Upper Bounds

A well known algorithm for an upper bound of the TSP is Christofides algorithm [Papadimitriou 1982]. It has a ratio bound of 1.5. That is, it finds a tour that has a weight

no worse than 1.5 times the weight of an optimal tour in G . The method finds a minimum spanning tree T in G and then using the odd degree vertices of T (denoted $\text{odd}(T)$) finds a minimum matching M . The subgraph of $M+T$ in G is then used to find a tour. It can be shown that this tour will have weight no more than 1.5 times the optimal tour in G . We will discuss Christofides algorithm in further detail in a Chapter Five.

Expected Optimal Tour Weight for Random Instances in the Euclidean Plane

The expected weight of the optimal tour when vertices are placed randomly in the two dimensional Euclidean plane has been shown to be approximated by the following formula:

$$E(w(\tau^*)) = K\sqrt{NR} \quad (1.12)$$

where, K is an empirical constant measured to be approximately equal to 0.7124 ± 0.0002 , N is the number of vertices, and R is the square area [Johnson et al. 1995].

CHAPTER TWO

Genetic Algorithms

Introduction

Genetic Algorithms are based on the idea of Darwin's evolutionary theory of survival of the fittest. In a Genetic Algorithm (GA) operations are performed on a set of potential solutions, called a **population**. A single potential solution in the population is said to be an **individual** in the population. Each individual in the population is in general an approximation to the optimal solution of the problem. Thus, the GA is an approximation algorithm. As the GA iterates, the individuals generally become better approximations to the optimal solution. The GA is a subset of a class of algorithms known as evolutionary algorithms [Heitkoetter 1994]. In its simplest form, the GA consists of three operators. They are: selection, reproduction, and mutation. At each iteration the three operators are performed on the population.

Figure 2.1 illustrates the basic heuristic during one iteration of the GA. At the beginning of an iteration the GA has a population A. One iteration through the process of selection, reproduction and mutation the GA obtains the population B. Population B is then used as input to the next iteration of the GA.

Each individual in a population is given a **fitness value** derived from a **fitness function**.

Figure 2.1

The fitness value of an individual is used in the selection processes to decide which individuals in the population will be used in the reproduction process. The reproduction process recombines individuals selected during the selection process. The reproduction process typically uses two individuals, called **parents**, given in the selection process and produces two new individuals, called **children**. The mutation process operates on the children produced during the reproduction process. Generally, mutation operates by making a slight modification to the children and is usually performed with only low probability. The children are then placed in the next population. The algorithm **SimpleGA** outlines the heuristic of the simple GA just described.

Algorithm **SimpleGA**

- (1) Select initial population P_0 .
- (2) Let $i = 0$.
- (3) While not stopping criteria met for P_i
 - (3.2) Let $i = i+1$.
 - (3.3) Let $P_i = \{\}$.
 - (3.3) For $j = 1$ to $|P_{i-1}|/2$
 - (3.3.1) Select two parents a and b from P_{i-1} .
 - (3.3.2) Recombine a and b to obtain children c_1 and c_2 .
 - (3.3.3) Mutate c_1 and c_2 .
 - (3.3.4) Let $P_i = P_i \cup \{c_1, c_2\}$.
- (4) Return best individual in P_i .

Encoding Scheme - Representing the Population

The three operations in the **SimpleGA** operate on the underlying representation of the individuals in the population. A common encoding scheme for representing individuals uses the binary alphabet. That is, every individual is represented by a sequence consisting of zeros and ones. The length of the sequence used for the encoding is taken to be constant over the population so that each individual has equal length. We usually let l represent the length of the encoding. The number of individuals in the population is the size of the population and is usually represented by n . The encoding scheme used in the GA is dependent on the problem for which the GA is being used. We discuss two encodings below, binary encodings and order based encodings. Also an extension to the encoding scheme, a locus based encoding, is briefly discussed.

Binary Encoding

The binary encoding scheme was the first of the encoding schemes. In fact, Schema Theory (discussed later in the chapter), which is the mathematical analysis of why GA's should work the way they do, is based upon GA's utilizing a population that has been encoded using the binary encoding scheme. The binary encoding scheme is very simple. Each individual in the population is represented by a sequence of ones and zeros. The following set illustrates a population **A** of individuals with a binary encoding:

$$\mathbf{A} = \{ 01100, 01001, 00110, 10010, 11111, 11000 \}$$

The size of population **A** is 6. The length of the encoding of each individual in **A** is constant and equal to 5. So $n = 6$ and $l = 5$.

Order Based Encodings

The binary encoding above is difficult to apply to many problems. For example, the binary encoding scheme does not lend itself naturally to the TSP. In the TSP we would like an order based encoding where each individual is a permutation of l numbers. For example, here

$$\mathbf{B} = \{ 53421, 13245, 43521, 15423, 23154, 42153 \}$$

is an order based encoding of length five. Each individual in population **B** is a permutation of 5 digits. This representation lends itself naturally to the TSP since each individual in **B** represents a tour in a graph with five vertices.

Locus Based Encodings

In the locus based encoding scheme we encode both ordering and value [Goldberg 1989]. Thus, an individual is represented by two sequences. Populations **A** and **B** above could both be written as a locus based encoding. For example, we could represent population **A** as a locus based encoding as follows:

$$A' = \left\{ \begin{pmatrix} 01100 \\ 12345 \end{pmatrix}, \begin{pmatrix} 00011 \\ 14325 \end{pmatrix}, \begin{pmatrix} 00110 \\ 12345 \end{pmatrix}, \begin{pmatrix} 10010 \\ 12345 \end{pmatrix}, \begin{pmatrix} 11111 \\ 12345 \end{pmatrix}, \begin{pmatrix} 11000 \\ 12345 \end{pmatrix} \right\}.$$

The second individual of A' has the first, fourth and third positions equal to zero and the second and fifth equal to one. Each individual in population A' is an encoding that consists of ordering and value (the lower sequence is the ordering and the upper sequence is the value). In particular, notice the second individual in which we have swapped locations two and four. The fitness of an individual (which will be used in the selection step) may depend on ordering, value, or a combination of the two depending on how the fitness operation is implemented. Thus locus based encodings carry along ordering and value in the encoding scheme. A locus based encoding is used in one of the TSP reproduction operators that will be presented in the next chapter. We will discuss the idea (and possible advantage) of locus based encodings when we discuss the Schema Theory later in this section.

Selecting the Initial Population

Generally, the initial population is chosen completely at random. For binary encodings this is accomplished by flipping a fair coin. If l is the size of the encoding of individuals in the population, then for each individual a fair coin is flipped l times. The encoding corresponds to the sequence obtained by the l coin flips where, perhaps, tail corresponds to a zero and heads corresponds to one, or vice versa. The case for selecting an order based encoding (for the TSP) will be discussed in the next chapter. It has been shown for the TSP [Davis 1987] that using more elaborate methods for the selection of the initial population does not help the GA reach its stopping criteria faster than it would have by simply selecting the population at random.

Stopping Criteria

Usually the stopping criterion is such that we say we have converged if each individual in the population attains the same fitness value. In most GA's there are a number of statistics computed at every iteration. In particular, when each individual in the population has the same fitness value, the variance of the fitness values in the population will be zero. In the TSP this stopping criterion is often achieved fairly quickly but without getting particularly close to an optimal solution. In the case of our analysis of the TSP using GA with

comparisons to other algorithms we will generally allow the GA to run for some specified amount of time or some specified number of iterations. The stopping criterion discussed in the literature is quite vague and does not seem to have much relevant value.

Selection

In this step of the algorithm, two individuals are chosen from the current population to move onto the reproduction step. We call the two individuals selected the **parents** of the reproduction step. The selection is based on the fitness value of the individual. The more highly fit individuals are generally given a higher probability of being selected in this step. The idea is to mate highly fit individuals and recombine them to produce better fit children, i.e. individuals in the subsequent population. Before discussing the two commonly used methods for selection, we discuss briefly the relationship between the fitness function and the objective function of a problem.

Objective Function and the Fitness Function

The fitness function is directly related to the objective function of the problem. The objective function represents the objective of the optimization problem at hand. For example, the objective function of the TSP is the function given by equation 1.1. The fitness function takes any potential solution to the objective function and assigns it a fitness value in such a way that solutions which are closer to the optimal have higher fitness values. We will write the fitness function as $f(x)$, and the objective function as $o(x)$, where x is an individual in the population.

Fitness Proportionate Selection

In a GA that uses the fitness proportionate method for selecting parents, the probability $p(x)$, of selecting an individual x in the population as a parent in the reproduction step is given by:

$$p(x) = f(x) / \sum_{i=1}^n f(i) \quad (2.1)$$

That is, the probability of selecting individual x is based on the proportion of that individual's fitness contribution to the entire population.

Figure 2.2

The roulette wheel example in Figure 2.2 above summarizes the fitness proportionate approach [Goldberg 1989]. In this example there are six individuals in the current population. Individual numbered 1 has a fitness value of 24. Individual numbered 2 has a fitness value of 18, and so on. The sum of the fitness values of the six individuals is 100. Thus, individual 1 has a 24% chance of being chosen for selection. A uniform random variable, r , between 0 and 1 is chosen. The individual is chosen by moving around the wheel and stopping at the value of the random variable. In the roulette wheel example any random number chosen that is between .24 and .42 will yield individual number 2. Individual number 2 has an 18% chance of being chosen. Clearly, higher fit individuals have a higher chance of being selected for reproduction.

For fitness proportionate selection, since the TSP is a minimization in the objective function, we give a tour t_j a fitness that is proportional to the sum over the entire population of $o(t_j)$. That is,

$$f(t_j) = (o(t_j) + c)^{-1} \sum_{i=1}^n (o(t_i) + c) \quad (2.2)$$

The constant c is chosen such that for any tour t in the population, $o(t) + c$ is greater than or equal to one [Corcoran et al. 1993].

Example

Figure 2.3

Let t_j be the tour ABCDEF shown in Figure 2.3. The edge weights have been labeled. The tour weight of t_j is 15 (i.e. $\alpha(t_j) = 15$). Suppose this tour belongs to a population of size $n = 10$ and that the sum of the weights of the tours in the population is 150. That is,

$$\sum_{i=1}^{10} (\alpha(t_i) + c) = 150, \text{ where } c = 0, \text{ because } \alpha(t_i) \text{ is bigger than one for each } i.$$

Then the fitness of t_j is,

$$f(t_j) = \frac{150}{15} = 10.$$

Rank Based Selection

In the rank based selection scheme, during the evaluation of a population, the individuals are ranked according to their fitness value. In the TSP, the tour (individual) with the least weight is ranked number one, the tour with the second least weight is ranked second, and so on. The individual ranked number one is said to be the highest ranked. Thus in ranked based selection, the selection is based on the ranking of the tours. Rank based selection has a number of advantages. First, it creates a probability distribution function that remains the same from generation to generation. That is, the highest ranked individual (having rank one), will always have the same probability of being picked at each generation. Notice that the

fitness proportionate method does not. Second, the rank based selection is much more generic than fitness based selection over various problem instances. And third, it can be used to speed or slow the genetic search process by varying the selection pressure, a variable that will be discussed below [Whitley 1989]. Figure 2.4 encapsulates the relationship between rank and fitness.

Figure 2.4

During the reproduction step in a Genetic Algorithm two parents are selected at random for reproduction. A function that selects higher ranked individuals with greater probability is necessary. We obtained the following function for rank based selection from a paper by Darrell Whitley [Whitley 1989]:

$$i = \left\lceil n \frac{b - \sqrt{b^2 - 4(b-1)r}}{2(b-1)} \right\rceil \quad (2.3)$$

Here n is the size of the population, b is the bias term (which is also known as selection pressure), and r is a uniform random variable strictly between zero and one. The bias term, b , can be chosen from 1.0 to 2.0 exclusive. For fixed values of n and b and the randomly chosen value of r , the resulting i given by equation 2.3 specifies the individual selected for the reproduction step. Thus if $i = 3$, the third ranked individual has been selected. We will show that equation 2.3 selects higher ranked individuals with a greater probability. In fact the bias term b indicates how much more probable the highest ranked individual is to occur over the

median ranked individual. A bias value of 1.5 indicates that the highest ranked individual will occur 1.5 times more often than the median ranked individual and 3 times more often than the worst ranked individual. Thus the fitness of an individual is based on its rank which is in turn based on the objective function. Equation 2.3 implicitly has the fitness of each individual built into it. Instead of assigning this fitness to each individual (as shown in Figure 2.4) and then using fitness proportionate selection we randomly select a ranked individual in the population using equation 2.3. Equation 2.3 is derived from the following linear function:

$$y = 2(1-b)x + b \quad (2.4)$$

The integral of equation 2.4 with respect to x from zero to one is the area of the trapezoid of height 1.0 and bases b and $2-b$, and that area is equal to 1.0. As shown in Figure 2.5, the idea is to solve for x_0 if we are given the area r under the linear function.

Figure 2.5

Solving for x_0 yields the appropriately ranked individual. Higher ranked individuals will have a higher probability of being chosen. Given r , a random number chosen from a uniform distribution strictly between 0 and 1, the following integral is used to solve for x_0 :

$$\int_0^{x_0} (2(1-b)x + b) dx = r \quad (2.5)$$

Clearly from Figure 2.5 the x_0 given in equation 2.5 exists between 0 and 1 and is unique. Solving the integral yields:

$$\begin{aligned}
 (1-b)x_0^2 + bx_0 &= r, \\
 (1-b)x_0^2 + bx_0 - r &= 0
 \end{aligned}
 \tag{2.6}$$

Now let

$$y = (1-b)x_0^2 + bx_0 - r \tag{2.7}$$

Since $b > 1$, the graph of equation 2.7 is a parabola which opens downward with the y intercept equal to $-r$. Thus, the graph of equation 2.7 is either:

Figure 2.6

And since the required root of equation 2.7 is positive, the right hand parabola in Figure 2.6 is the correct one. Using the quadratic equation to find the roots of equation 2.7 we obtain:

$$\begin{aligned}
 x_0 &= \frac{(-b) \pm \sqrt{b^2 - 4(1-b)(-r)}}{2(1-b)} \\
 &= \frac{b \mp \sqrt{b^2 - 4(b-1)r}}{2(b-1)}
 \end{aligned}
 \tag{2.8}$$

Thus in the right hand graph of Figure 2.6:

$$x_0' = \frac{b - \sqrt{b^2 - 4(b-1)r}}{2(b-1)} \quad \text{and} \quad x_0'' = \frac{b + \sqrt{b^2 - 4(b-1)r}}{2(b-1)}.$$

The x_0 in Figure 2.5 is unique and between 0 and 1. If $x_0' = x_0''$ we use the x_0' formula. If $x_0' \neq x_0''$, then x_0'' cannot be between 0 and 1. For if it were then equation 2.7 would have two roots between 0 and 1. Thus the unique root between 0 and 1 is the equation given by x_0' .

Now that we have the correct equation for x_0 , there exists an i such that:

$$\frac{(i-1)}{n} < x_0 \leq \frac{i}{n}, \quad \text{or equivalently,} \quad i-1 < nx_0 \leq i \quad (2.9)$$

Therefore,

$$nx_0 = n \frac{b - \sqrt{b^2 - 4(b-1)r}}{2(b-1)} \quad (2.10)$$

and since $\lceil nx_0 \rceil = i$, we obtain:

$$i = \left\lceil n \frac{b - \sqrt{b^2 - 4(b-1)r}}{2(b-1)} \right\rceil \quad (2.11)$$

which is equation 2.3.

As we mentioned earlier, equation 2.3 selects higher ranked individuals with greater probability. That is:

$$p(i) \geq p(i+1), \quad \text{for } 1 \leq i \leq n-1.$$

By our derivation of i , and by examining Figure 2.5, we obtain:

$$p(i) = \frac{\int_{(i-1)/n}^{i/n} (2(1-b)x+b) dx}{\int_0^1 (2(1-b)x+b) dx} = \int_{(i-1)/n}^{i/n} (2(1-b)x+b) dx$$

Figure 2.7

It is clear from Figure 2.5 and Figure 2.7 that in fact $p(i) \geq p(i+1)$, for $1 \leq i \leq n-1$.

If we let $o(t)$ represent the result of applying equation 1.1 to tour t , then rank based selection ranks the tours in the population using $o(t)$. The tours are then given a fitness value based on this ranking. We could use equation 2.4 to get the fitness value and use fitness proportionate selection, but instead we simply use equation 2.3 to randomly select an individual. The tour with the highest rank (ranked number one), is essentially given the highest fitness. The fitness value assigned by equation 2.4 to the highest ranked individual is b . The lowest ranked individual is thus given a fitness value of $2-b$ by equation 2.4. The bias value b is also known as the selection pressure. The uniform probability distribution is

obtained as b approaches one.

I find rank based selection a very natural and elegant method. The problem with fitness proportionate is that fitness scaling needs to be performed. When fitness values begin to get closer together the probability distribution of the fitness proportionate selection function becomes closer to uniform. Rank based selection avoids this and maintains a consistent probability distribution throughout. It is also entirely non problem specific. That is, one does not need to “figure out” what the scaling parameters need to be. Also, the search rate can be slowed very easily by decreasing the selection pressure parameter b in equation 2.3. By slowing the search rate we mean that as b approaches 1.0 all individuals have approximately equal chances of being selected and thus we do not necessarily move closer to an optimal solution in the objective function. In contrast, as b approaches 2.0 we would expect the GA to converge much more rapidly, and to a solution that is not necessarily close to optimal.

Reproduction

The reproduction step uses two individuals obtained by the selection step. In the reproduction step the goal is to recombine information obtained in the parents to obtain two children. The resulting children have information from both parents and we hope that the recombination yields children that have better fitness values than the parents.

The simplest form of reproduction uses what is known as **crossover**. The crossover operator is best described on binary encodings of the population. Consider two parents P_1 and P_2 selected from a population over a binary encoding of length 5 as shown in Figure 2.8.

Figure 2.8

The crossover operator randomly selects a crossover point (in this case 3). The first child

C_1 is created by using the first half of the information contained in parent one concatenated to the second half of the information contained in parent two. The process is reversed for the second child C_2 . This crossover is not restricted to just one crossover point as shown in Figure 2.8. Often more than one crossover point is chosen in which case the child is constructed by copying the odd segments from one parent and the even segments from the other parent.

In the next chapter we will discuss two crossover methods from the literature that have been used specifically for recombining tours of a TSP. One of them is an order based encoding operator, and the other is a locus based encoding operator.

Mutation

As the population evolves, the mutation operator plays the important role of constantly injecting diversity into the population. In this sense, mutation does not allow the entire population to converge to a single point in the solution space. In binary encodings mutation occurs by randomly “flipping” a bit. In order based encodings mutation occurs by randomly swapping two positions. The **mutation rate** is the rate at which this random “flipping” of bits occurs. The mutation rate is generally very low so that a random bit flip (our swap) occurs with a small probability each time a child is generated. At times it may be necessary to increase the mutation rate in order to rapidly produce change in the population. This is explained further in Chapter Four.

Simple Example

Here we give a simple example of how one might set up a simple GA and we go through one round of selection and reproduction (see Goldberg 1989 for a more detailed example). Suppose we wish to optimize the function $g(x) = x^2$ over the integer interval between 0 and 31. We first have to choose an encoding scheme and a fitness function $f(x)$. We will use a binary encoding scheme where the length of each individual is represented by a sequence of 0's and 1's of length 5. Thus the binary encoding can be evaluated by the fitness function as a value between 0 and 31. We will let the fitness function be equal to the objective function. That is, $f(x) = x^2$, where x is an individual in the population whose encoding evaluates to an

Table 2.1
Population P_0

Individual x in P_0	Fitness of x , $f(x)=x^2$.	Probability of selection, $p(x)$.
01011	121	0.085
10101	441	0.311
01110	196	0.138
11001	625	0.440
00110	036	0.025
00001	001	0.001
Total Fitness of Population P_0 :		1420
Average fitness:		237

integer value between 0 and 31. Therefore, the fitness value of any particular individual in the population will be between 0 and 31^2 .

Next we have to decide on a population size and initialize our population. We will use a population size of 6 and initialize the population randomly. Suppose we have the initial population P_0 which has been generated by simply flipping a fair coin as shown in Table 2.1.

If we use fitness proportionate selection method, then by equation 2.1 we derive the probability $p(x)$ that an individual will be selected for reproduction as shown in the rightmost column of Table 2.1.

Table 2.2
Population P_1

Individual x in P_1 (parents in P_0 , crossover point)	Fitness of x , $f(x)=x^2$.	Probability of selection, $p(x)$.
10100 (2,3,4)	400	0.136
01111 (2,3,4)	225	0.077
10101 (2,4,1)	441	0.151
11001 (2,4,1)	625	0.212
11001 (4,4,3)	625	0.212
11001 (4,4,3)	625	0.212
Total Fitness of Population P_0 :		2941
Average fitness:		490

Table 2.2 shows a subsequent population after selection and reproduction. The first two numbers in the parenthesis of the individuals in column one indicate which parents were used in creating that individual. The third number in the parenthesis indicates the crossover point that was used in the reproduction step. Thus, the first individual in P_1 was obtained from the selection of individuals 2 and 3 in the population P_0 and by crossing over at location 4. The second individual in P_1 is the other child of this reproduction (since we obtain two children from the crossover operator). Notice that the population average fitness increased from 237 to 490.

Theory

The theory behind the effectiveness of Genetic Algorithms is based on the use of the binary encoding scheme. First we develop some definitions.

The average fitness value in the population is given by:

$$\bar{f} = \frac{\sum_{i=1}^n f(i)}{n} \quad (2.12)$$

When selection is done with replacement, the expected number times a solution, x in the population, is selected for reproduction, from a population of size n is:

$$E_n(x) = n p(x) = n \frac{f(x)}{\sum_{i=1}^n f(i)} = \frac{f(x)}{\bar{f}} \quad (2.13)$$

The left hand side of equation 2.13 is also known as the **target sampling rate of x at iteration t** , denoted $tsr(x, t)$ instead of $E_n(x)$. Iteration t (or time t) refers to which population is being sampled. We begin with $t = 0$ and at each iteration of the GA we let t increment by one. Thus, t is simply the iteration of the GA. Notice that if x were represented twice in the population then the expected number of times x is drawn during the selection process is twice that of equation 2.13.

Schema Theory

In schema theory we are interested in the similarities between the solutions in the population. Schemata are **similarity templates** between different solutions. We add a symbol, #, to the finite alphabet to represent a “don’t care” symbol. In the language of mathematics a schema is the set of all individuals which can be formed by replacing each # by 0 or 1. In a binary alphabet the following are example schemata:

$$\begin{aligned} H_1 &= 0\#100 = \{ 01100, 00100 \}, \\ H_2 &= 0\#00\# = \{ 00000, 01000, 00001, 01001 \} \end{aligned}$$

The selection operator in the GA induces a selection rate on the schemata represented by the current population at time t .

Let $m(H,t)$ be the number of individuals in the population at time t that represent schema H . We can now define the target sampling rate in terms of schemata. The target sampling rate of schemata H at time t is given defined as:

$$tsr(H,t) = \frac{1}{m(H,t)} \sum_{i=1}^{m(H,t)} tsr(x_i,t) , x_i \in H \quad (2.14)$$

Substituting equation 2.13 for $tsr(x_i,t)$ yields,

$$\begin{aligned} tsr(H,t) &= \frac{1}{m(H,t)} \sum_{i=1}^{m(H,t)} \frac{f(x_i)}{\bar{f}} , x_i \in H \\ &= \frac{f(H)}{\bar{f}} , \end{aligned} \quad (2.15)$$

where $f(H)$ represents the average fitness of the solutions in the population that represent schema H . The expected number of times schema H will be represented at iteration $t+1$ is given by:

$$\begin{aligned}
\text{Expected } m(H, t+1) &= m(H, t) \text{ } tsr(H, t) \\
&= m(H, t) \frac{f(H)}{\bar{f}} \quad (> m(H, t) \text{ when } f(H) > \bar{f})
\end{aligned} \tag{2.16}$$

Equation 2.16 indicates that above average schema survive the selection process and below average schema die off.

The *defining length* of schema H , denoted $\delta(H)$, is defined as the distance between the first and last fixed position in schema H . The *order* of schema H , denoted $o(H)$, is the number of fixed positions in schema H . For example, the order of schema $H_1 = 0\#100$ is 4 and the order of schema $H_2 = 0\#00\#$ is 3. The defining length of schema H_1 is 4, whereas the defining length of schema H_2 is 3.

During the reproduction step (using the crossover operator), schemata of short defining length are more likely to survive. That is, if we look at a particular schema H and its defining length, equation 2.14 can be rewritten so that the rate at which better fit schemata grow (or decline) in the population depends also on the defining length. If we were to rewrite this we would see that schemata of short defining length are more adept to survive the reproduction process, whereas schemata of long defining length would grow (decline) less (more) rapidly. (See Goldberg 1989 for further details on this particular aspect of schema theory).

Implicit Parallelism

The implicit processing of this many schemata is what is known as the implicit parallelism in the GA. The total number of possible schemata of length l in a population using the binary encoding scheme is 3^l . And, in general, for alphabets of cardinality k , there exist $(1+k)^l$ schemata.

Each solution in the population represents 2^l schemata, since each position can take on its original symbol or the $\#$ symbol. If n represents the size of the population, then the total number of schemata represented by the population is at least 2^l and at most $n2^l$.

The GA implicitly processes between 2^l and $n2^l$ schema at each iteration. This is what is known as the implicit parallelism of the GA.

Discussion

It should be clear that a Genetic Algorithm is clearly suitable for implementation as a parallel algorithm. For example, if we have K processors available working on a population of size N , each processor could independently perform $(N/2)/K$ selection and reproduction steps on the current population. Using this model, it suggests that a factor of $O(K)$ speed up could be obtained from the use of a parallel Genetic Algorithm.

CHAPTER THREE

Genetic Algorithms Applied to the Traveling Salesman Problem

Resources

TSPLIB

We have adopted instances from the TSPLIB library [Reinelt 1994]. This problem set consists of instances of TSP graphs used in the research of many branches of problems that involve the TSP. We have adopted a subset of the graphs from Reinelt that use the Euclidean distance metric and whose coordinates are integers. In this problem set, the distance function used is an integer function and is the same as equation 1.3 with the additional caveat that rounding is performed on the result. Thus we obtain the following function which is used as the distance metric for our problem instances:

$$d'(A,B) = \left\lfloor d(A,B) + 0.5 \right\rfloor = \left\lfloor \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2} + 0.5 \right\rfloor \quad (3.1)$$

Rounding is performed by adding 0.5 and taking the floor of the result (recall, floor(x) returns the greatest integer less than or equal to x).

This raises the following question as to the validity of the triangle inequality: Do there exist vertices A,B,C with coordinates on the two dimensional Euclidean integer plane such that

$$\left\lfloor d(A,B) + 0.5 \right\rfloor > \left\lfloor d(A,C) + 0.5 \right\rfloor + \left\lfloor d(C,B) + 0.5 \right\rfloor \quad (3.2)$$

given,

$$d(A,B) \leq d(A,C) + d(C,B) \quad ?$$

The answer to this question is yes and invalidates our assumption of the triangle inequality. Consider the vertices A, B, and C given by the integer coordinates (0,0), (24,22), and (8,8). Substituting these values into equation (3.2) yields:

$$\left\lfloor 32.56 + 0.5 \right\rfloor \stackrel{?}{>} \left\lfloor 11.31 + 0.5 \right\rfloor + \left\lfloor 21.26 + 0.5 \right\rfloor$$

$$33 > 11 + 21$$

$$33 > 32$$

We must assume that the graphs we are working with from TSPLIB have been checked for this potential problem and instead of investigating this further, we take on faith that the instances below are in fact optimal. Table 3.1 lists some of the instances from the TSPLIB that are TSP instances in the two dimensional integer Euclidean plane. Reinelt gives these instances the type name EUC_2D.

Table 3.1

Instance Name	Number of Vertices	Optimal Tour Weight
eil51	51	426
st70	70	675
eil76	76	538
pr76	76	108159
KroA100	100	21282
KroC100	100	20749
KroD100	100	21294
rd100	100	7910
eil101	101	629
lin105	105	14379
d198	198	15780
pr299	299	48191
lin318	318	42090
pcb442	442	50778
pr1002	1002	259045

Random Instances

Another popular research instance of Euclidean TSP are those instances selected randomly in the Euclidian plane. For these problem instances Johnson, et al. have calculated

an expected optimal tour length given by equation 1.12 mentioned at the end of Chapter One. Although these are attractive instances because they are easy to generate, it has been argued that the random instances are not indicative of typical “real world” applications [Johnson et al. 1995].

LibGA Software Package

The LibGA software package is a library of GA routines written in the C programming language by Arthur L. Corcoran at University of Tulsa [Corcoran 1993]. This package is easily adopted to the problem instance, in our case the TSP. We have added two reproduction operators, functions to read TSPLIB instances and corresponding fitness and objective function operators, and also functions that perform timing.

Population Representation and Initialization

The representation of an individual in the population is simply a permutation of the vertex set of G . Clearly, this representation of an individual also represents a tour in G , since we are considering complete graphs. The initial population for GA's applied to the TSP will be chosen randomly. A random tour is obtained by randomly picking a start vertex and then randomly picking vertices not yet picked (this can be also viewed as generating a random permutation of size $|V(G)|$). As noted in Chapter Two, in a study by Davis 1987 this seems to be very adequate. Any other methods used do not tend to make the GA converge any faster.

Reproduction

In the previous chapter we discussed “simple crossover” on binary strings. Since individuals in the population of the TSP are represented by permutations, we need a method of reproduction that works on permutations. There are many problems in using crossover on permutations. Specifically in the TSP, since after a crossover, each vertex in the graph must be represented in order for the tour to be a valid TSP tour. Notice that simple crossover as shown in Figure 2.7 (Chapter Two) does not insure that every vertex will be represented in the child. There have been many attempts at producing good crossover operators for the

TSP. We present two operators from the literature: the Genetic Edge Recombination Operator [Whitley et al. 1989] and the Bui-Moon Crossover operator [Bui and Moon 1994].

Genetic Edge Recombination Operator (GERO)

Whereas many crossover operators that work on tours operate on the vertex set of G , the GERO operates on the edges represented in the parent tours (the edge set from the result of combining the two parents). The GERO operator works nicely in generating valid tours without the post processing that is often necessary in other operators resulting in the need to fix the resulting child to be a valid tour. The operator is classified as an order based operator since it creates a child that yields a new ordering of the vertices from the parent orderings.

This operator concentrates on transferring edges from parents to the child while producing valid tours in the process. Whitley et al. state that "... it is clear that optimizing a Traveling Salesman Problem involves finding the best combined set of edges that constitute a legal tour. What is important is not that a particular city (vertex) occurs in a particular position, but rather that the genetic operators preserve and exploit critical links (edges) that contribute to the minimization of the overall tour."

In their paper Whitley et al. cite earlier operators that are vertex ordering dependent. They state that in all instances of these operators the overall effectiveness of the GA stems from how effective the crossover operator is in preserving edges from the parents. Therein they hypothesized, "to improve the performance of the GA at solving the TSP one must develop a recombination operator that transfers edges from parents to offspring as effectively as possible".

The edge recombination operator relies on the adjacency list data structure obtained when the parent tours are combined to form a Hamiltonian subgraph of the original graph. When two parent tours are combined, a vertex has at least two adjacent vertices and at most four in the adjacency list. Consider the two tours shown in Table 3.2. The resulting adjacency list is shown.

Table 3.2

Tour	A	B	C	D	E	F
Tour	B	D	C	A	E	F

Vertex	Adjacency List				
A	B	F	C	E	
B	A	C	D	F	
C	B	D	A		
D	C	E	B		
E	D	F	A		
F	A	E	B		

From this adjacency list the authors describe an algorithm for producing a child tour, algorithm **EdgeRecombination** shown below. Shortly, we will go through an example of the algorithm.

Algorithm **EdgeRecombination**(P1, P2)

- (1) Initialize M to the set of all vertices.
- (2) Construct an adjacency list for each vertex from the result of combining P1 and P2.
- (3) Randomly select a vertex, v_i from M of highest degree.
- (4) Let $M = M - \{v_i\}$ and $i=1$.
- (5) while M is not empty do
 - (5.1) For each vertex v in M that contains v_i in its adjacency list, remove v_i from v 's adjacency list.
 - (5.2) From the adjacency list of v_i , let W be the set of vertices in v_i 's adjacency list.
 - (5.3) If W is empty, randomly pick a v in M, otherwise pick a v in W that has the smallest adjacency list attached to it. If there are ties, then randomly select a v among the ties.
 - (5.4) Let $i = i + 1$, $v_i = v$, and $M = M - \{v_i\}$.

Algorithm **EdgeRecombination** creates a new tour $v_1, v_2, \dots, v_n$ in which most of the edges were obtained from a parent. Another offspring can be obtained by repeating the algorithm.

To show how this algorithm works, consider the tours shown in Table 3.2. The combination of the two tours is shown by the thin lines of the graph shown in Figure 3.1. The

bold edges in the graph shown in Figure 3.1 is the resulting tour after running algorithm **EdgeRecombination**.

Figure 3.1

The algorithm begins by randomly selecting a vertex of highest degree. The possibilities are vertices A and B (both have degree 4). Suppose B is chosen. Line 5.1 removes B from the adjacency list of vertices A, C, D and F. At line 5.3 the algorithm randomly selects among the vertices adjacent to B of smallest degree which are C, D, and F (all of which now have degree 2 and vertex A has degree 3 by the removal of B from their adjacency lists). Suppose C is chosen. The algorithm goes on to create the resulting child tour BCDEAF as shown. The edge connecting the first vertex v_1 to the last vertex v_n is never chosen by the algorithm and is simply inherent in the construction of the tour.

There are two ways in which edges that are in neither of the two parent tours can be introduced to the child from this algorithm. The first way is at line 5.3. If W is empty, a vertex must randomly be selected from the set M of unvisited vertices. This introduces an edge to the child tour that is in neither of the parent tours. The second way that an edge that is in neither of the parent tours can be introduced to the child tour is after the completion of the algorithm when the edge that connects vertex v_1 to v_n is tacked on.

Mutation is inherent in the algorithm. In line 5.3 of the algorithm, if the set W is empty

then a vertex must be selected randomly from the set M of unvisited vertices. This results in selecting an edge that is contained in neither of the parents. In this respect the algorithm inherently has mutation embedded within it. The authors claim the algorithm is 95-99% effective in transferring edges from parents to their offspring. In our implementation of this algorithm we have also found this to be the case.

An interesting idea that the authors mention is the idea that the edge combinations in a tour relate to a binary representation of the tour for use in schema theory. If we let a bit represent the inclusion of an edge in the tour then the binary representation of a tour has length $n(n-1)/2$ where 0 means the edge is not in the tour, and 1 means the edge is in the tour. In the above example the tour ABCDEF can be represented by the following binary string:

	ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
ABCDEF	1	0	0	0	1	1	0	0	0	1	0	0	1	0	1

Using the example of combining the tours in Table 3.2 we obtain the following binary representation of the tours:

	ab	ac	ad	ae	af	bc	bd	be	bf	cd	ce	cf	de	df	ef
ABCDEF	1	0	0	0	1	1	0	0	0	1	0	0	1	0	1
BDCAEF	0	1	0	1	0	0	1	0	1	1	0	0	0	0	1
BCDEAF	0	0	0	1	1	1	0	0	1	1	0	0	1	0	0

The authors of the GERO claim that at any position where the parents have the same bit, any child tour will have that bit. (Our example shows that this claim is not true). Notice the resulting child tour BCDEAF shown in the last row of the table above. Edge EF is in both of the parent tours but does not occur in the child tour. Furthermore, when an edge belonging to neither parent is inserted into a child the two zero bits representing the edge in the parents becomes a one in the child.

Bui-Moon Recombination Operator

The Bui-Moon operator was conceived in order to operate as a traditional crossover

operator and have the ability to effectively extract adjacency information [Bui and Moon 1994]. This operator uses a locus based encoding scheme so that the order of the vertices can be reassigned in a preprocessing step to preserve good schemata [Bui and Moon 1994]. We will discuss this preprocessing step in the next section and a possible change to the algorithm that we introduce to avoid the preprocessing altogether. First we give a description of the recombination operator. This operator uses a locus based encoding scheme. Recall, in the locus based encoding scheme each individual is encoded by two sequences. One sequence that represents the ordering and the other that represents the value. For example, A is a population of four tours in a graph of five vertices:

$$A = \left\{ \begin{pmatrix} ADBEC \\ BECAD \end{pmatrix}, \begin{pmatrix} EBCDA \\ BCDAE \end{pmatrix}, \begin{pmatrix} EBCDA \\ DABCE \end{pmatrix}, \begin{pmatrix} AEBCD \\ EDABC \end{pmatrix} \right\}$$

In the notation shown above we will let the lower sequence of each individual denote its ordering and the upper sequence denote the value. Thus, in the first individual of population A , the value of B is A , the value of E is D , the value of C is B , the value of A is E , and the value of D is C . We can view any individual in A as a tour of a five vertex graph where the value of a vertex given in the ordering sequence is the successor of that vertex in the tour. For example, the first individual of population A represents the tour $BAEDC$ since A is the successor of B , E is the successor of A , D is the successor of E , C is the successor of D , and B is the successor of C . Thus, we can easily derive the tour represented by an individual in the population. Likewise, we can easily derive the value sequence if we are given a tour and an ordering sequence. That is, given an individual represented by a tour t and an ordering sequence o we can derive the locus based encoding of that individual by finding the value sequence.

The preprocessing step in the Bui-Moon crossover operator seeks to find a “good” permutation of the vertices to use as an ordering for the entire population (the importance of which we will discuss in the next section). This ordering is then held fixed for the rest of the GA. We will let o denote the ordering that is found during the preprocessing step. As noted above, if we are given a tour t and an ordering o for an individual we can easily derive the value sequence for the individual.

The Bui-Moon operator uses a **successor** and **predecessor** sequence for each parent. The

successor sequence given by an ordering o of tour t , denoted $Succ(t, o)$, is a sequence given by the ordering o that indicates the successor vertex of every vertex in the tour t . Similarly the predecessor sequence given by an ordering o of tour t , denoted $Pred(t, o)$, is a sequence given by the ordering o that indicates the predecessor of every vertex in the tour t . For example, the value sequence ADBEC of the first individual of population A can be derived from the tour EDCBA and the ordering BECAD by:

$$Succ \left(EDCBA, BECAD \right) = ADBEC$$

The predecessor operation is similar. For the tour EDCBA and the ordering BECAD we obtain:

$$Pred \left(EDCBA, BECAD \right) = CADBE$$

Bui-Moon perform a preprocessing step that seeks to find a “good” permutation of the vertices in G to obtain the ordering sequence o that remains fixed for each individual in the population. In fact, if one wishes to leave out the preprocessing step, a default ordering can be used. In our implementation of the Bui-Moon crossover the input is two individuals which are permutations that represent tours (just as in GERO). Since the order sequence is fixed during the GA it is not carried along in the encoding of an individual, and instead, it is held as a global constant. Assuming that we have an ordering o that is the same for each individual the algorithm **BuiMoon** is as follows:

Algorithm **BuiMoon**(T_1, T_2)

Assumption: We have an ordering sequence o .

- (1) Let $S_1 = Succ(T_1, o)$ be the successor values of T_1 .
 Let $P_1 = Pred(T_1, o)$ be the predecessor values of T_1 .
 Let $S_2 = Succ(T_2, o)$ be the successor values of T_2 .
 Let $P_2 = Pred(T_2, o)$ be the predecessor values of T_2 .
- (2) Select k random crossover points of the ordering o . This results in $k+1$ intervals created by the k crossover points. Label the crossover intervals $0, 1, 2, \dots, k$.
- (3) Copy the values from the odd intervals of S_1 to the child.
- (4) Copy the values from the even intervals of S_2 to the child. If a duplicated appearance with a value copied from S_1 in (3) occurs, do not copy it.
- (5) Fill each unfilled position a in step (4) with one of three remaining adjacent vertices of a with the following priorities: 1) the predecessor of a in T_2 , the successor of a in T_1 , the predecessor of a in T_1 . If a duplicated appearance with a value already copied is detected, do not copy it.

- (6) Randomly fill the yet unfilled positions with unused vertices.
- (7) The vertices copied now represents the successor sequence of the child, $S_c = \text{Succ}(C, o)$. If this does not represent a valid tour, use **ResolveCycles** to obtain a valid tour.

Although the child that is produced through step 6 of the algorithm contains all vertices of G , it does not necessarily produce a valid tour of G . So in step 7 we have to remove any cycles that resulted from the crossover to obtain a valid tour. This is done by the algorithm **ResolveCycles** which is explained in the following example.

Example

Consider the following two parent tours T_1 and T_2 for the Bui-Moon crossover operator:

$$T_1 = EDCBA$$

$$T_2 = BECDA$$

where $o = BECAD$.

In line 1 of the algorithm we determine the successor and predecessor sequence for each tour given by the ordering o as shown in the following table:

o	B	E	C	A	D
S_1	A	D	B	E	C
P_1	C	A	D	B	E
S_2	E	C	D	B	A
P_2	A	B	E	D	C
Interval	0	1	2		

Next we randomly select k crossover points out of the 4 possible crossover points. This results in $k + 1$ crossover intervals. For this example suppose that k is two and the randomly chosen crossover points are 2 and 3. The randomly chosen intervals are labeled and separated by the heavy lines in the table above. In step three of the algorithm we copy the values in the odd intervals of S_1 to the successor values S_c of the child. So at this point we have the following child:

o	B	E	C	A	D
S_1	A	D		E	C
P_1	C	A	D	B	E
S_2	E	C	D	B	A
P_2	A	B	E	D	C
S_c	-	-	B	-	-
Interval	0	1	2		

In step four, we copy from the even intervals of the S_2 to the child. We do not copy the vertices that we previously copied in step three. Thus we now have the following child:

o	B	E	C	A	D
S_1	A	D		E	C
P_1	C	A	D	B	E
S_2			D	B	
P_2	A	B	E	D	C
S_c	E	C	B	-	A
Interval	0	1	2		

The grey cells distinguish which vertices we copied to the child. Notice that vertex B in the second even interval of S_2 was not copied since it had been copied in a previous step. In step five, vertex D drops down to the child and we never have to execute step six of the algorithm. Thus we obtain the following child:

o	B	E	C	A	D
S_1	A	D		E	C
P_1	C	A	D	B	E
S_2			D	B	
P_2	A	B	E		C
S_c	E	C	B	D	A
Interval	0	1	2		

In step seven of the algorithm we examine the successor sequence of the child S_c with respect to the ordering o and notice that it does not represent a valid tour. In fact the child has two disjoint cycles. The heavy lines in Figure 3.2 represents the child that we have

obtained thus far. The thin lines are the result of combining the two parent tours.

Figure 3.2

In step seven algorithm **ResolveCycles** is called to obtain a valid tour. The input to **ResolveCycles** is a successor sequence S_T given by the ordering o . As noted earlier S_T may not represent a valid tour. The output is a modified sequence S_T given by the ordering o that represents a valid tour. This algorithm works by finding an initial cycle c_0 . While all vertices are not represented in c_0 we find another cycle c_1 and merge it with c_0 such that c_0 contains vertices of c_0 and c_1 . It should be clear that finding any cycle can easily be done by picking a yet unrepresented vertex in the ordering o and following its successor value in sequence S_T until we reach the picked vertex. Merging two cycles can easily be done by swapping successor values in sequence S_T of the two vertices that were used to find each cycle. For example, using the sequence of the child $S_c = ECBDA$ given by the order $o = BECAD$, suppose we are at the point where we have two cycles, $c_0 = AD$ and $c_1 = CBE$ as shown in Figure 3.3. Assume that the picked vertex that was used to find c_0 was A and likewise, the picked vertex used to find c_1 was C. Thus merging cycles c_0 and c_1 is accomplished by swapping the successor values of C and A in S_T to obtain the merged cycle c_0

Figure 3.3

= ABECD as shown in Figure 3.3.

Preprocessing

Bui and Moon discovered that finding a good ordering (or “indexing”) of the vertices significantly improved the performance of the GA [Bui and Moon 1994]. Recall, in the section on schema theory at the end of Chapter Two, we mentioned that schema of short defining length are more adept at surviving the reproduction process [Goldberg 1989]. Bui and Moon state, “The key idea of reindexing is using clusters in cities’ geographical distribution, i.e., a clustered subset of cities has a higher probability of constructing high quality schemata than a set of arbitrary cities.” As shown in the Figure 3.4 the upper sequence is a schema where *s* represents specific positions and the *·* represents the “don’t care” symbol (see Chapter Two). The lower sequence is the same schema after preprocessing has been performed to obtain a “good ordering”. The clusters in the lower schema have a better chance of surviving the crossover operation because of the short defining length. Bui and Moon claim that “on average, the GA with the preprocessing step, produces slightly better results in **visibly** less time.”

Figure 3.4

I found the preprocessing step a very interesting idea of testing schema theory and I have a couple of ideas that might be interesting to test the operator. First, test the difference in the operator using the optimal tour as the ordering of the vertices vs. a (possibly) random ordering. Second, the ordering that is used could come from one of the two parents. That is we could use the ordering of one parent to produce one child and then switch to the ordering of the other parent to produce the second child. This removes the preprocessing step and makes it dynamic (i.e. part of the operator). As the population evolves, individuals in the population become better approximations, and hence their ordering becomes closer to optimal. I think this type of dynamic ordering would be beneficial. The ordering could evolve just as the individuals (which are approximations to the optimal solution) evolve. In the case of the TSP, I would expect the ordering and the optimal solution to converge to identical solutions.

Mutation

We have taken a very simple approach to mutation since there does not seem to be very much literature on the subject of mutation as it is applied to the TSP. Our mutation randomly selects two cities in the permutation and swaps their location.

Local Optimization

Local optimization (or local improvement) is the process of considering simple tour modifications that can be used to find “neighbor” tours of lesser weight. The neighborhood of a tour are those tours that can be obtained by a single move in a local optimization step. We have implemented from the literature two well known local optimization heuristics. They are 2-Opt and 3-Opt. As shown in Figures 3.5 and 3.6 the 2-Opt and 3-Opt heuristics are simply edge exchanges. If 2-Opt or 3-Opt reduces the tour weight it is accepted otherwise the maneuver is not performed. The local optimization heuristic is performed on child tours produced in the crossover step. For the Genetic Algorithm to remain competitive to other approximation heuristics, it is imperative that local optimization is performed [Johnson et al. 1995].

Algorithm 2-Opt(T)

- (1) Randomly select two edges from T, say $v_i v_j$ and $v_k v_l$.
- (2) If $w(v_i, v_j) + w(v_k, v_l) \leq w(v_i, v_k) + w(v_j, v_l)$ return.
- (3) Otherwise replace $v_i v_j$ and $v_k v_l$ by $v_i v_k$ and $v_j v_l$.

Algorithm 3-Opt(T)

- (1) Randomly select three edges from T, say $v_i v_j$, $v_k v_l$, and $v_m v_n$.
- (2) If $w(v_i, v_j) + w(v_k, v_l) + w(v_m, v_n) \leq w(v_i, v_l) + w(v_j, v_m) + w(v_k, v_n)$ return.
- (3) Otherwise replace $v_i v_j$, $v_k v_l$, and $v_m v_n$ by $v_i v_l$, $v_j v_m$, and $v_k v_n$.

The algorithm that we use for the 2-Opt and 3-Opt local optimization is an adaption of the method from the Simulated Annealing algorithm of Numerical Recipes [Press et al. 1992]. In this algorithm we perform N iterations, and at each iteration we randomly select whether a 2-Opt or a 3-Opt should be attempted. We then attempt a 2-Opt or a 3-Opt accordingly. Thus we have the following algorithm:

Algorithm LocalOptimization(T)

- (1) Let $N = |V(G)|$.
- (2) For $i = 1$ to N do
 - (2.1) Flip a fair coin C.
 - (2.2) If C is Heads perform a 3-Opt(T).
 - (2.3) Else perform a 2-Opt(T).

Figure 3.5

3-Opt

Figure 3.6

Martin's Local Optimization Algorithm

The algorithm we've implemented picks a segment of the tour and looks for edges along this segment that when used by the 2-Opt operator will result in a shorter tour. If two such edges are found then a 2-Opt is performed. This algorithm is order N^2 . See Figure 3.7.

Algorithm **Martin-Opt**(T)

- (1) Randomly select a segment S of the tour T with $N/2$ vertices. Let the segment S selected be labeled $S_{v_1}, S_{v_2}, \dots, S_{v_{N/2}}$.
- (2) For $i = 1$ to $(N/2)-3$ do
 - (2.1) For $j = i+2$ to $(N/2)-1$ do
 - (2.1.1) If the result of performing a 2-Opt on $S_{v_i}, S_{v_{i+1}}$ and $S_{v_j}, S_{v_{j+1}}$ decreases the weight, then perform the 2-Opt and let S be the new segment as a result of the 2-Opt.

Figure 3.7

Discussion

It seems that the population size needs to increase as the TSP instances get larger (i.e. has more vertices). Certainly, Davis et al. [Davis et al. 1987] state that diversity is important in the initial population. In the TSP, if the problem instances get larger and the population size remains fixed it seems apparent that the diversity represented by tours in the graph decreases exponentially (since the number of tours is increasing exponentially).

CHAPTER FOUR

Comparison of Various Genetic Operators

In this chapter we present results of running the GA using two different genetic operators; namely, the Bui-Moon crossover operator and the Genetic Edge Recombination Operator (GERO) as presented by Whitley, et al. First we discuss the options that we have used in running the software package LibGA.

Steady State Model and the Generational Model

In the generational model an iteration consists of selecting $N/2$ pairs with replacement and mating each pair which results in two children, for a total of N children that make up the subsequent population:

```
Algorithm GenerationalGA()
(1) Let  $i = 0$  and let  $P_i$  be the initial population.
(2) While stopping criteria not met for  $P_i$  do
(2.1) Let  $P_{i+1} = \{\}$ .
(2.2) For  $j = 1$  to  $N/2$  do
(2.2.1) Randomly select two parents A and B in population  $P_i$ .
(2.2.2) Reproduce A and B to produce children  $C_1$  and  $C_2$ .
(2.2.3) Mutate  $C_1$  and  $C_2$ .
(2.2.4) Perform local optimization on  $C_1$  and  $C_2$ .
(2.2.5) Let  $P_{i+1} = P_{i+1} \cup \{C_1, C_2\}$ .
(2.3) Let  $i = i+1$ 
```

In line (2.2.1) of the **GenerationalGA** above, the selection is done according to a chosen selection method. Recall, that in general, the selection method is a random process such that parents having “better” fitness values will be more likely to be chosen for reproduction (these same comments apply for line (2.2) of the **SteadyStateGA** described below).

The steady state model, however, considers one iteration a selection of 2 parents for mating of which the children replace the weakest members of the current population to make up the subsequent population:

```
Algorithm SteadyStateGA()
(1) Let  $i = 0$  and let  $P_i$  be the initial population.
```

- (2) While stopping criteria not met for P_i do
 - (2.1) Let $P_{i+1} = P_i$.
 - (2.2) Randomly select two parents A and B from population P_i .
 - (2.3) Reproduce A and B to produce children C_1 and C_2 .
 - (2.4) Mutate C_1 and C_2 .
 - (2.5) Perform local optimization on C_1 and C_2 .
 - (2.6) Replace the two weakest members of population P_{i+1} with C_1 and C_2 .
 - (2.7) Let $i = i+1$.

Elitism

One of the options that the LibGA software package makes available is called elitism which we have utilized in all of our test cases. From the LibGA package we quote, “Elitism has two actions. For a generational GA, elitism makes two copies of the best performer in the old pool and places them in the new pool, thus ensuring the most fit chromosome survives. The other action works with both models. In this case, elitism picks the best two chromosomes from the parents and children. Thus, if a child is not as fit as either parent, it will not be placed in the new pool (i.e. among any two parents and their two children, the best two chromosomes are placed in the new pool). Selecting elitism in LibGA performs both actions.”

Diversification or “Dropping the Bomb”

It is often the case, during a run of the GA, that the variance of the population will go to zero. This suggests that the entire population is the same (that is, each individual is the same tour) or that some of the tours may be different, but exhibit the same tour length (although we believe the former case to be the most probable). Through our experimentation, we have noticed that there is usually a high probability that the variance of the population will go to zero. When this happens we say we have reached a “local minima” in the search space (except for some rare cases where the global minimum is reached), and when this occurs, it is difficult for the GA to evolve any further (since during reproduction, if both parents have exactly the same chromosomes, the children will turn out exactly like the parents). In order to help the GA escape local minima we have devised a means of introducing a high mutation rate. When the variance of the population drops to zero the mutation rate is changed to one minus the previous mutation rate until the variance is no longer zero. This has the effect of mutating individuals to force a change in the population fitness. Hence the name of the

method diversification, or “Dropping the Bomb”.

Stopping Criteria

We run the algorithms for a specified number of seconds, or in some cases a specified number of iterations.

Effects of Local Optimization

Our initial tests report on the effect of the local optimization step when the GA is applied to the TSP. We test the instance LIN105 from the TSPLIB. It is a graph of 105 vertices and is shown in the appendix. For each local optimization method and each of the two crossover operators (Bui-Moon and Edge Recombination), we performed 100 runs using the steady state model. We let each run perform exactly 8000 iterations. The selection method used is rank based and the selection pressure is set to 1.6. Elitism is set to TRUE. The population size is 100 and the initial population is selected randomly. No Bui-Moon preprocessing was performed (recall that the Bui-Moon preprocessing step, finds an ordering that should be used for the Bui-Moon crossover). Therefore, the Bui-Moon crossover operator that we used in

Table 4.0
GA Fixed Parameters (for Tables 4.1-3)

Fixed Parameters:	
<i>GA Model:</i>	Steady State
<i>Population Size:</i>	100
<i>Selection Method:</i>	Rank Based
<i>Selection Pressure:</i>	1.6
<i>Stopping Criteria:</i>	8000 Iterations
<i>Initial Population:</i>	Selected Randomly
<i>Elitism:</i>	True
<i>Diversification:</i>	True
Bui-Moon Crossover Fixed Parameters:	
<i>Ordering:</i>	Default (No Preprocessing)
<i>Crossover Points:</i>	5

Table 4.1
Effects of local optimization on LIN105

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 8000 Iterations	Average Percent Above Optimal after 8000 Iterations	Number of Times Optimal Occurred in 100 Runs
None	Bui-Moon	0.123	114.3	0
	Edge Recombination	0.118	122.1	0
Martin	Bui-Moon	0.224	3.8	2
	Edge Recombination	0.223	2.5	5
2Opt/3Opt	Bui-Moon	0.233	4.8	0
	Edge Recombination	0.182	4.6	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.209	3.7	0
	Edge Recombination	0.204	2.5	1
2Opt	Bui-Moon	0.156	6.7	0
	Edge Recombination	0.166	6.3	0
3Opt	Bui-Moon	0.186	9.5	0
	Edge Recombination	0.182	6.2	0

this test uses the default ordering $\sigma = v_1, v_2, v_3, \dots, v_{100}$ and 5 crossover points. For quick reference, these fixed parameters are shown in Table 4.0.

We present our results in Table 4.1. It is clear from the Table that local optimization is imperative regardless of which GA operator is used. If no local optimization is performed, the results after 8000 iterations of the steady state model GA yield very poor approximations that are more than twice the weight of the optimal tour. However, with local optimization the results are only two to nine percent (on average) more than the optimal tour length. In fact, in some of the cases, the optimal tour was found. Table 4.1 shows that after 8000

Table 4.2
Effects of local optimization on D198

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 8000 Iterations	Average Percent Above Optimal after 8000 Iterations	Number of Times Optimal Occurred in 50 Runs
None	Bui-Moon	0.150	180.7	0
	Edge Recombination	0.177	183.7	0
Martin	Bui-Moon	0.236	8.5	0
	Edge Recombination	0.232	6.0	0
2Opt/3Opt	Bui-Moon	0.281	6.5	0
	Edge Recombination	0.292	3.6	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.228	3.9	0
	Edge Recombination	0.215	3.9	0
2Opt	Bui-Moon	0.201	5.3	0
	Edge Recombination	0.189	4.6	0
3Opt	Bui-Moon	0.278	10.4	0
	Edge Recombination	0.231	6.9	0

iterations of the GA performing the “Martin” local optimization step combined with Edge Recombination resulted in finding the optimal tour 5 out of 100 times. These results may be misleading since the “Martin” local optimization performs possibly a factor N more tries at performing a 2-Opt than the 2-Opt or 3-Opt local optimization procedures. That is, “Martin” local optimization performs $O(N^2)$ tries at performing a 2-Opt along the picked segment, whereas the 2-Opt and 3-Opt perform $O(N)$ tries. The intent of this exercise and the results of Table 4.1 was to show that local optimization is imperative if any real progress is to be made when using the GA for the TSP with our parameters for the GA operators. We note, however, that for LIN105 and for all versions of local optimization the Edge Recombination operator gives better answers than the Bui-Moon operator. Furthermore, in those cases

Table 4.3
Effects of local optimization on PR299

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 8000 Iterations	Average Percent Above Optimal after 8000 Iterations	Number of Times Optimal Occurred in 50 Runs
None	Bui-Moon	0.167	438.4	0
	Edge Recombination	0.187	484.1	0
Martin	Bui-Moon	0.304	50.6	0
	Edge Recombination	0.381	24.4	0
2Opt/3Opt	Bui-Moon	0.346	44.2	0
	Edge Recombination	0.339	10.8	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.317	12.4	0
	Edge Recombination	0.307	10.4	0
2Opt	Bui-Moon	0.266	28.5	0
	Edge Recombination	0.270	11.1	0
3Opt	Bui-Moon	0.431	48.1	0
	Edge Recombination	0.412	28.0	0

where local optimization is performed the results are obtained in roughly the same amount of time.

Next we perform the same experiment, but on graph D198, which is almost twice as large graph LIN105. The results are reported in Table 4.2. In order to reduce the run time, we have reduced the number of runs from 100 to 50. But we kept all other parameters the same. In Table 4.2 we notice that we see no optimal solution found in any of the cases. This is not surprising because we reduced the number of runs from 100 to 50, and more importantly, the population size stayed the same, namely 100, while we almost doubled the size of the graph.

Table 4.4
No local optimization but increased number of iterations.

Graph	Crossover Method	Average System Time (seconds) Required for 16000 Iterations	Average Percent Above Optimal after 16000 Iterations	Number of Runs
LIN105	Bui-Moon	0.474	69.6	100
	Edge Recombination	0.306	67.1	100
D198	Bui-Moon	0.380	122.2	50
	Edge Recombination	0.367	134.7	50
PR299	Bui-Moon	0.452	311.2	50
	Edge Recombination	0.374	368.8	50

We have again run this same experiment with graph PR299 keeping the population size at 100. The results are shown in Table 4.3. For both D198 and PR299, the Edge Recombination operator gives better results, usually faster, than the Bui-Moon operator, when both have local optimization.

We also ran the case with no local optimization for twice the number of iterations as above to see if we obtained any better results than what is shown in Tables 4.1-3. The results are still discouraging and reinforce the claim that local optimization is imperative if any satisfactory results are to be obtained. The results of no local optimization but an increased number of iterations are shown in Table 4.4.

For all three graphs, the average system time shown in Table 4.4 is usually greater than the average system time for local optimization methods, but the results are remarkably bad. Although in this case Bui-Moon gives better results than Edge Recombination on the two bigger problems.

Effect of Changing from the Steady State Model to the Generational Model

There are many possible combinations of parameters that can be adjusted in the GA. We wanted to compare the GA using the Steady State model, the model that we used for the runs shown in tables 4.1 through 4.4, with the GA when we use the Generational model. We originally picked the Steady State model since it was used in the published results from Whitley and Bui-Moon. Keeping all parameters the same except for the difference between the Steady State model and the Generational model, we only needed to know how many iterations we should run the Generational model to make a fair comparison. An iteration means something quite different in the two models. We decided that it would be a fair comparison if we picked the number of iterations such that the number of reproductions in either model stays the same. Recall, in the Steady State model, an iteration is the reproduction of one pair of parents. Whereas, in the Generational model an iteration is a turn over of the entire population. Thus, an iteration in the Generational model consists of $N/2$ reproductions that produce N children. These N children constitute the subsequent population. We picked the number of iterations so that the number of reproductions in either model stayed the same.

The number of reproductions in the Steady State model R_s given the number of iterations performed I_s is,

$$R_s = I_s \quad (4.1)$$

And the number of reproductions in the Generational model R_g given the number of iterations performed I_g is,

$$R_g = I_g \frac{(\text{Population Size})}{2} \quad (4.2)$$

Letting $R_s = R_g$ and solving for I_g we obtain,

$$I_g = \frac{2I_s}{(\text{Population Size})} \quad (4.3)$$

The population size that we used in the LIN105, D198 and PR299 graphs from Tables 4.1-3 was 100. And the number of iterations I_s of the Steady State model that was used in those experiments was 8000 per run. Substituting 8000 for I_s and 100 for the *Population Size* in Equation (4.3) we would expect $I_g = 160$ to be the equivalent number of iterations to use

Table 4.5
Generational Model LIN105 (160 Iterations)

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 160 Iterations	Average Percent Above Optimal after 160 Iterations	Number of Times Optimal Occurred in 100 Runs
None	Bui-Moon	0.045	144.2	0
	Edge Recombination	0.059	106.4	0
Martin	Bui-Moon	0.171	5.0	0
	Edge Recombination	0.150	4.2	1
2Opt/3Opt	Bui-Moon	0.081	33.7	0
	Edge Recombination	0.089	17.4	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.144	4.4	0
	Edge Recombination	0.113	3.6	1
2Opt	Bui-Moon	0.099	24.3	0
	Edge Recombination	0.078	12.4	0
3Opt	Bui-Moon	0.081	39.0	0
	Edge Recombination	0.102	29.9	0

in our experiments on the same graphs for the Generational model. So we ran the same cases as before for 160 iterations. The results for the LIN105 graph are shown in Table 4.5.

An immediate observation is that the running time for this case is substantially lower than for the same case in the Steady State model as shown in Table 4.1. Also notice that the average tour weight is substantially worse than the same case in the Steady State model of Table 4.1. We just argued, however, that 160 iterations should be “equivalent” to the 8000 iterations of the Steady State model. There are two things (related) that might be cause the tour quality to be worse but the times to be faster.

Table 4.6
Generational Model LIN105 (320 Iterations)

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 320 Iterations	Average Percent Above Optimal after 320 Iterations	Number of Times Optimal Occurred in 100 Runs
None	Bui-Moon	0.098	84.3	0
	Edge Recombination	0.067	53.0	0
Martin	Bui-Moon	0.234	4.9	0
	Edge Recombination	0.280	4.0	0
2Opt/3Opt	Bui-Moon	0.134	9.2	0
	Edge Recombination	0.141	5.2	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.190	3.5	1
	Edge Recombination	0.219	2.6	5
2Opt	Bui-Moon	0.118	8.4	0
	Edge Recombination	0.118	6.4	0
3Opt	Bui-Moon	0.147	16.7	0
	Edge Recombination	0.170	11.4	0

First, the Steady State model can be viewed as an insertion sort over $N/2$ iterations (since at every iteration the offspring are inserted into the currently (sorted) population). Thus, over $N/2$ iterations we have done $O(N^2)$ operations. However, in the Generational model $N/2$ selections/reproductions are made whose offspring are **appended** (an $O(1)$ operation) to the new population (which starts out empty). After the $N/2$ selections/reproductions, the new population is sorted using the Quick Sort algorithm which is known to be an $O(N\log N)$ (on average) algorithm. This may explain the reason why the Generational model is faster than the Steady State model over the “equivalent” number of iterations.

In the preceding paragraph we argued why the running time of the Generational model

Table 4.7
Generational Model D198 (320 Iterations)

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 320 Iterations	Average Percent Above Optimal after 320 Iterations	Number of Times Optimal Occurred in 100 Runs
None	Bui-Moon	0.139	138.2	0
	Edge Recombination	0.133	99.6	0
Martin	Bui-Moon	0.247	11.5	0
	Edge Recombination	0.256	7.9	0
2Opt/3Opt	Bui-Moon	0.248	25.7	0
	Edge Recombination	0.240	14.1	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.275	6.4	0
	Edge Recombination	0.272	4.8	0
2Opt	Bui-Moon	0.249	17.9	0
	Edge Recombination	0.174	9.8	0
3Opt	Bui-Moon	0.244	29.8	0
	Edge Recombination	0.268	26.1	0

might be faster than the Steady State model over the “expected” same number of iterations. We will now explain why we think the tour quality of the tours produced by Generational model are substantially worse. At each iteration in the Steady State model a set of parents are picked that may produce a child whose tour quality is better than any of the other individuals in the current population (because of local optimization this is even more likely at the onset of the algorithm). The child is then available to be selected after only one iteration of the Steady State model. Consider, however, the scenario in the Generational model. When two parents are selected and they produce a child whose tour quality is better than any of the other tours in the current population, the child is placed in the subsequent population and is not available for selection until at most $N/2-1$ selections of individuals from

Table 4.8
Generational Model PR299 (320 Iterations)

Local Optimization Method	Crossover Method	Average System Time (seconds) Required for 320 Iterations	Average Percent Above Optimal after 320 Iterations	Number of Times Optimal Occurred in 100 Runs
None	Bui-Moon	0.209	410.1	0
	Edge Recombination	0.171	337.4	0
Martin	Bui-Moon	0.327	77.5	0
	Edge Recombination	0.284	39.7	0
2Opt/3Opt	Bui-Moon	0.410	92.8	0
	Edge Recombination	0.322	64.0	0
50% Martin 50% 2Opt/3Opt	Bui-Moon	0.420	43.3	0
	Edge Recombination	0.356	27.8	0
2Opt	Bui-Moon	0.339	68.6	0
	Edge Recombination	0.304	45.3	0
3Opt	Bui-Moon	0.402	117.4	0
	Edge Recombination	0.794	101.3	0

the current population. Therefore subsequent selections and reproductions can not take advantage of this individuals "better" tour quality until at most $N/2-1$ selections later. This may explain why the Generational model produces tours of substantially worse tour quality than that of the Steady State model over the "expected" same number of iterations.

In Tables 4.6-8 we present results of running the Generational model GA on the three graphs LIN105, D198, and PR299. We have doubled the number of iterations per run from 160 to 320. This results in running times that are comparable to the running times that we observed with the Steady State model. However, in general, the quality of the tours are still not as good as the quality tours that we observed in the Steady State model. Also note that

in each model, local optimization vastly improves the results and Edge Recombination gives consistently better results than Bui-Moon.

In summary, our results indicate that local optimization is imperative with any implementation of the Genetic Algorithm for the TSP. Furthermore, the Edge Recombination operator combined with 50% Martin-Opt and 50% 2-Opt/3-Opt seems to generally perform the best overall of the cases with which we have experimented.

CHAPTER FIVE

Discussion of Other Approximation Algorithms

In this chapter we introduce some well known approximation algorithms and briefly describe and discuss the known bounds.

Nearest Neighbor Approximation Heuristic

The Nearest Neighbor (NN) approximation heuristic at each iteration visits the nearest vertex that has not yet been visited, returning to the starting vertex after all vertices have been visited. This heuristic does not have a constant ratio bound. In fact, the ratio bound is $O(\log N)$, where $N = |V(G)|$. For arbitrarily large N there exist TSP instances such that the nearest neighbor solution is at least $\Theta(\log N)$ times as long as the optimal tour [Reinelt 1994].

The NN heuristic for the TSP can be implemented as follows:

Algorithm **TSP-NN**(G)

Assumption: G is complete.

- (1) Let $A = V(G)$.
- (2) Let v_1 be an arbitrarily selected vertex from A .
- (3) Let $A = A - \{v_1\}$ and let $i = 2$
- (4) While A is not empty do
 - (4.1) Scan A for the closest vertex to v_{i-1} . Let v_i be that vertex.
 - (4.2) Let $A = A - \{v_i\}$.
- (5) Return the tour given by v .

There is a variation on the TSP-NN tour construction heuristic which is known as the Greedy algorithm. The greedy algorithm builds a tour by picking the least weighted edge

Table 5.1a
Results for Nearest Neighbor (Random Start)

Graph	Average System Time (seconds)	Average Percent Above Optimal	Number of Runs	Number of Times Optimal Occurred
LIN105	0.000027	30.5	1000	0
D198	0.000040	22.3	1000	0
PR299	0.000087	28.4	1000	0

Table 5.1b
Results for Nearest Neighbor (Deterministic Start)

Graph	System Time (seconds)	Best Percent Above Optimal	Number of Runs	Number of Times Optimal Occurred
LIN105	0.001611	17.8	105	0
D198	0.007319	11.7	198	0
PR299	0.023210	19.2	299	0

from G and adding it to the current set of edges picked such that a degree three vertex would not be formed. The Greedy algorithm is $O(N^2 \log N)$ since it requires that the edges are sorted. The **TSP-NN** algorithm as implemented above is $O(N^2)$.

We have implemented the **TSP-NN** algorithm given above and run it on the three graphs that we used in the previous chapter. The results are shown in Table 5.1a. The results in Table 5.1a show results for the average behavior of running the **TSP-NN** algorithm 1000 times by randomly picking a start vertex. However, the **TSP-NN** algorithm is deterministic and we could run the **TSP-NN** algorithm using each vertex in the graph as the start vertex. In this case we report the total time that it takes to perform all of the runs for that graph (i.e. the time it takes to run the **TSP-NN** algorithm N times where run $i = 1 \dots N$ uses vertex v_i as the arbitrarily selected vertex in line (2) of algorithm **TSP-NN**). These results are shown in Table 5.1b. Note: The running time in this deterministic case is $O(N^3)$.

The results given here for **TSP-NN** are much better than the GA results without any local optimization and are obtained much more quickly than any of our GA results in the previous chapter. From the class notes of Schmeichel we have the following theorem:

Theorem 5.1 [Rosenkrantz, Sterns, Lewis, 1977]. Let τ be the tour found by the NN algorithm. Then, $w(\tau) \leq \frac{1}{2} \left(1 + \lceil \log_2 |V| \rceil \right) w(\tau^*)$ where τ^* is the optimal tour.

By Theorem 5.1, the theoretical upper bound on graphs with 105, 198, and 299 vertices as in LIN105, D198 and PR299 is 4.0, 4.5, and 5.0 times the optimal respectively. The results of Table 5.1b indicate that in these cases, the **TSP-NN** algorithm performs much

better than the known theoretical upper bound.

Minimum Spanning Tree Approximation Heuristic

The next heuristic we present for approximating the TSP gives a much better upper bound than the nearest neighbor method. In fact the upper bound is a constant ratio to the optimal tour weight. The TSP Minimum Spanning Tree (TSP-MST) method first finds a minimum spanning tree T of the graph G . Using T we construct a tour which will have weight no more than twice the cost of the optimal weight tour of G . In order to describe this heuristic fully we first give some definitions.

Definition 5.1. A **walk** in graph G is an alternating sequence $v_1 e_1 v_2 e_2 \cdots v_{k-1} e_{k-1} v_k$ of vertices and edges of G such that edge e_i is incident to vertices v_i and v_{i-1} . Vertices and edges of G may be repeated in a walk. We denote such a walk by $v_1 v_2 \cdots v_{k-1} v_k$. Furthermore, any sequence of vertices in a complete graph clearly corresponds to a walk. In the algorithms below when vertices are removed from a sequence denoting walk W , a new walk W' is clearly formed.

Definition 5.2. A **trail** of G is a walk of G with the additional property that no edge is repeated.

Definition 5.3. A **circuit** of G is a trail of G with the additional property that the trail begins and ends with the same vertex (i.e. $v_1 = v_k$).

Definition 5.4. An **Eulerian circuit** of G is a circuit of G that contains every edge in G .

Definition 5.5. A connected graph G is said to be **Eulerian** if it contains an Eulerian circuit.

Using definitions 5.1 through 5.5 there is a well known theorem [Euler 1736] that states that a graph G is Eulerian if and only if G is connected and every vertex has even degree.

Algorithm **TSP-MST**(G)
 Assumption: G is complete.
 (1) Let $T = \mathbf{MST}(G)$.

- (2) Construct G' by duplicating every edge in T so that every vertex in G' has even degree.
- (3) Find an Eulerian circuit C of G' .
- (4) Construct the tour τ by keeping the first instance of a vertex v in the circuit C and removing the rest of the instances of v in C , except the initial and terminal vertices of C are unchanged.
- (5) Return the tour τ .

Theorem 5.2. Algorithm **TSP-MST** finds a tour τ of G that has weight no more than twice the weight of the optimal tour τ^* in G .

Proof.

It is clear that τ is a tour of G . Since T is an MST of G it has weight less than $w(\tau^*)$. Thus, by duplicating each edge in T to construct G' , the weight of the Eulerian circuit C of G' has weight less than $2w(\tau^*)$ since every edge of T is used exactly twice. Let the Eulerian circuit C of G' be $i_1 \alpha_1 i_2 \alpha_2 \cdots i_{n-1} \alpha_{n-1} i_n \alpha_n i_1$ where each α_j is a (possibly empty) sequence of vertices and $\tau = i_1 i_2 \cdots i_{n-1} i_n$. Thus i_j is the first instance of i_j in $i_1 \alpha_1 i_2 \alpha_2 \cdots i_{j-1} \alpha_{j-1} i_j$. In constructing τ it is clear that the weight of the walk $i_j \alpha_j i_{j-1}$ is at least as much as the weight of $i_j i_{j-1}$ because of the triangle inequality. That is, by the triangle inequality, traveling from vertex i_j to vertex i_{j-1} through a sequence of vertices α_j is at least as much as the weight of traveling from vertex i_j directly to vertex i_{j-1} . Thus $\tau < 2w(\tau^*)$.

Algorithm **TSP-MST** is time constrained by the need to find the MST of G in step (1). Finding a MST can be performed in time $O(N^2)$ using Fibonacci heaps (see Chapter One where the algorithm was presented). Finding a Eulerian circuit and the removal of multiple instances of vertices can be performed in $O(E)$ (where E is the size of the edge set of G , and for a complete graph is $O(N^2)$). Thus algorithm **TSP-MST** is an $O(N^2)$ algorithm.

Table 5.2a
Results for Minimum Spanning Tree (Random Start)

Graph	Average System Time (seconds)	Average Percent Above Optimal	Number of Runs	Number of Times Optimal Occurred
LIN105	0.000019	47.8	1000	0
D198	0.000059	23.3	1000	0
PR299	0.000096	40.1	1000	0

Table 5.2b
Results for Minimum Spanning Tree (Deterministic Start)

Graph	System Time (seconds)	Best Percent Above Optimal	Number of Runs	Number of Times Optimal Occurred
LIN105	0.002333	41.8	105	0
D198	0.010210	21.2	198	0
PR299	0.031230	34.4	299	0

We implemented the **TSP-MST** heuristic as given above (using Fibonacci heap data structures) and ran it on the three graphs presented in the previous chapter. We ran the algorithm on each instance 1000 times by picking a random start vertex for each run and took the average system time and percent above optimal as we did in Table 5.1a. The results are shown in Table 5.2a.

However, the **TSP-MST** algorithm (like the **TSP-NN** algorithm) is deterministic and we could run the **TSP-MST** algorithm using each vertex in the graph as the start vertex. In this case we report the total time that it takes to perform all of the runs for that graph (i.e. the time it takes to run the **MST-TSP** algorithm N times where run $i = 1 \dots N$ uses vertex v_i as the first vertex select by line (3.1) of the **MST** algorithm (see the **MST** algorithm on page 4 of Chapter One). These results are shown in Table 5.2b. Note: The running time in this deterministic case is $O(N^3)$.

It should also be noted that the results in Table 5.2a (as far as the quality of the tours obtained) agree very closely to the results obtained by Reinelt [Reinelt 1994]. As noted by Reinelt, the **TSP-MST** heuristic has “very poor performance”. The quality of the tours obtained by the **TSP-NN** heuristic narrows the gap between the Best Percent Above Optimal by about a factor of two on average; even though the guaranteed upper bounds on the **TSP-MST** heuristic are much better than the **TSP-NN** heuristic, the results of **TSP-NN** are better than **TSP-MST**.

Christofides Approximation Heuristic

The algorithm of Christofides is similar to the TSP-MST algorithm. Instead of duplicating the minimum weight spanning tree T , the method finds a minimum complete matching M on the odd vertices of T .

Theorem 5.3. Algorithm **Christofides** finds a tour τ of G that has weight no more than one and a half times the optimal tour τ^* of G .

Proof.

Algorithm **Christofides**(G)

Assumption: G is complete.

- (1) Let $T = \text{MST}(G)$.
- (2) Let M be the minimum complete matching in G of the vertices in T which have odd degree (there are an even number of odd vertices).
- (3) Construct G' by letting $V(G') = V(G)$ and the edge set of G' be composed of the edges in T and the edges in M .
- (4) Find an Eulerian circuit C in G' .
- (5) Construct the tour τ by keeping the first instance of a vertex v in the circuit C and removing the rest of the instances of v in C , except the initial and terminal vertices of C are unchanged.
- (6) Return the tour τ .

The weight of the minimum spanning tree T of G is less than the weight of the optimal tour of G . Since G' is constructed by combining the edges of T with those of M we want to show that the weight $w(M)$ of the minimum matching is less than one half the weight of an optimal tour $w(\tau^*)$ of G . The weight $w(C)$ of the Eulerian circuit will thus be less than 1.5 times $w(\tau^*)$ and therefore the weight of the tour τ found from C will be less than 1.5 times $w(\tau^*)$.

Our goal now is to show that the weight $w(M)$ of the minimum matching is less than one half $w(\tau^*)$. Let $2m$ be the number of odd degree vertices of the minimum spanning tree T . And let $i_1 i_2 i_3 \dots i_{2m}$ be the sequence that these odd degree vertices appear in τ^* . Consider the following two matchings of the odd degree vertices:

$$M_1 = \{(i_1, i_2), (i_3, i_4), \dots, (i_{2m-1}, i_{2m})\}$$
$$M_2 = \{(i_2, i_3), (i_4, i_5), \dots, (i_{2m}, i_1)\}$$

Notice that since M is a minimum matching of the $2m$ vertices, we have $w(M) \leq w(M_1)$ and

$w(M) \leq w(M_2)$. Figure 5.1 shows the relationship between the optimal tour and the matchings M_1 and M_2 . Therefore, because of the triangle inequality $w(\tau^*) \geq w(M_1) + w(M_2)$.

Figure 5.1

And since $w(M_1) + w(M_2) \geq 2w(M)$ we obtain $w(\tau^*) \geq 2w(M)$. That is $w(M) \leq \frac{1}{2}w(\tau^*)$. Thus the Eulerian circuit C has weight less than $w(\tau^*) + \frac{1}{2}w(\tau^*) = 1\frac{1}{2}w(\tau^*)$.

Simulated Annealing

We have implemented a simulated annealing algorithm obtained from Numerical Recipes [Press et al. 1992]. Like the Genetic Algorithm, the Simulated Annealing Algorithm mimics a process that has been found in nature. The Simulated Annealing algorithm mimics the process of cooling a metal (or chemical) to its state of least energy. We can imagine that an optimal solution to the Traveling Salesman Problem is a condition of least energy in that any non-optimal solution will cost as least as much as the optimal solution. In this sense, the non-optimal solutions of more weight have a higher energy state than the optimal solution. Thus, in the adaptation of simulated annealing to the Traveling Salesman Problem we equate the current tour weight with energy. The Simulated Annealing Algorithm begins with a temperature and at each iteration lowers the temperature of the system. The temperature affects how the system can move from one energy state to the next. For the Traveling Salesman Problem, the “system” is simply the current solution. The initial temperature and the rate at which the temperature is lowered is called the *annealing schedule*. The probability that a change in the system, at a temperature $T > 0$, from a state at energy E_1 to a state at energy E_2 is:

$$P(E_2 - E_1) = e^{\left(\frac{-(E_2 - E_1)}{T}\right)} \quad (5.1)$$

and is derived from the Boltzmann probability distribution [Press et al. 1992].

Notice that if $E_2 < E_1$ the probability of $P(E_2 - E_1)$ is greater than one, so that any change in the system to a lower energy state is always taken (regardless of the temperature). However, any uphill change in the system (that is when $E_1 > E_2$) is taken with a probability that will depend on the temperature of the system. As the system cools the probability of making an uphill change in the energy is less.

Algorithm **TSP-SA**(*G*)

Assumption: *G* is complete.

- (1) Let τ be an arbitrarily chosen tour of *G*.
- (2) Let the temperature $T = 1.0$ and $\Delta T = 0.97$.
- (3) While stopping criteria not met do
 - (3.1) For $i=1$ to $100 * |V(G)|$ do
 - (3.1.1) Let τ' be the result of randomly performing either a 2-Opt or a 3-Opt on τ .
 - (3.1.2) If $w(\tau') < w(\tau)$ then let $\tau = \tau'$.
 - (3.1.3) Otherwise $\tau = \tau'$ with probability $P(\tau' - \tau)$ given by equation 5.1.
 - (3.2) Let $T = T * \Delta T$.
- (4) Return τ .

Our implementation of the Simulated Annealing Algorithm **TSP-SA** is shown above. In this implementation T is initially set to 1.0. At each iteration the temperature drops by some known scale factor (ΔT). An iteration consists of $100 * |V(G)|$ tries at performing either a 2-Opt or a 3-Opt. If the energy state from either maneuver is lowered, then the maneuver is performed with 100 percent probability. If however, the state increases, the probability that the maneuver is performed is given by equation 5.1.

Table 5.3
Results for Simulated Annealing

Graph	Seconds of System Time	Average Percent Above Optimal	Number of Runs	Number of Times Optimal Occurred
LIN105	0.24	0.5	100	27
D198	0.36	0.8	50	0
PR299	0.38	3.1	50	0

In Table 5.3 we report the results of the Numerical Recipes Simulated Annealing Algorithm on the three graphs LIN105, D198, and PR299 as presented in the previous chapter.

The temperature scaling factor used is 0.97. That is, every iteration the temperature T is lowered by a factor of 0.97.

Summary

In this chapter we have looked at five different approaches for approximating solutions to the TSP. Of the five approaches we obtained experimental results with three of them: Nearest Neighbor (TSP-NN), Minimum Spanning Tree (TSP-MST), and the Simulated Annealing (TSP-SA) approach. Of the three approaches, the first two are deterministic and exhibit well known ratio bounds. On the other hand, the Simulated Annealing approach is a randomized procedure based on the same fundamental ideas that the Genetic Algorithm is based: that is, trying to mimic some process found in nature.

The Nearest Neighbor and the Minimum Spanning Tree approach do very well considering the amount of time that they take to find an approximation. However, the quality of the tours is noticeably worse than those tours produced by the Genetic Algorithm or Simulated Annealing. The Simulated Annealing approach and the Genetic Algorithm certainly do better if one is willing to spend the time required. Furthermore, the Simulated Annealing approach appears to give better results at a cost of somewhat more time than the best Genetic

Table 5.4a
Best Tour Quality of Heuristics Discussed

Heuristic	Graph		
	LIN105	D198	PR299
Nearest Neighbor	17.8	11.7	19.2
Minimum Spanning Tree	41.8	21.2	34.4
Simulated Annealing	00.5	00.8	03.1
Genetic Algorithm	02.5	03.9	10.4

Table 5.4b
Corresponding Time

Heuristic	Graph		
	LIN105	D198	PR299
Nearest Neighbor	0.001611	0.007319	0.023210
Minimum Spanning Tree	0.002333	0.010210	0.031230
Simulated Annealing	0.240000	0.360000	0.380000
Genetic Algorithm	0.204000	0.215000	0.307000

Algorithm approach. Table 5.4a and 5.4b summarize the results of the approaches discussed in this chapter in comparison to the Genetic Algorithm.

CHAPTER SIX

Summary and Other Interesting Works

In this section we will discuss results from a paper by Johnson, et. al. which will appear in a chapter of the book entitled *Local Search in Combinatorial Optimization* sometime in 1997 [Johnson et al. 1995]. We will also discuss (briefly) some recent results by Sanjeev Arora in which he presents an algorithm that finds, in **nearly** linear time, a $1+\epsilon$ approximation to the Euclidean TSP.

Discussion of the Johnson et al. Paper

In the paper of Johnson et. al. (from here on - the paper), the authors site that problems with 100 or fewer vertices are routinely solved within a few minutes on a workstation. Instances in the range of 1,000 vertices can usually be solved in a few hours (or days). In the paper they concentrate on results for instances with significantly more than 1,000 vertices. The paper suggests that the TSP is not a typical combinatorial optimization problem, since the TSP seems significantly harder to solve to optimality than other typical combinatorial optimization problems.

In section one of the paper a general introduction is given in which the paper summarizes that, in general, most theoretical research on the TSP is done with the special case of the TSP's that satisfy the triangle inequality. And, in general, the class of heuristics widely used can be broken into two classes. The two general approaches are: (1) tour construction, and (2) local optimization. Tour construction heuristics build a tour from scratch using *successive* (usually greedy) *augmentation* techniques that terminate once the tour is constructed. These usually get within 10-15% of optimal. The classical local optimization approaches do somewhat better. The simple 3-Opt gets to within 3-4% of the optimal solution and the variable-op algorithm of Lin Kernighan to within 1-2% of the optimal solution. Also, these approaches also seem to have growth rates of $o(N^2)$ in the 1,000-1,000,000 vertex range. The classical local optimization heuristics begin with a single tour and iteratively make changes to make the tour better.

We find the following interesting statement on page four of the paper: "The approaches

from (1) and (2) above leave small room for improvement for the new algorithms such as, simulated annealing, neural networks, etc. However, one new approach, genetic algorithms, does have something to contribute if one is willing to pay a large, although still, $O(N^2)$ running time price.” This statement is intriguing since in our limited experiments with the three graphs LIN105, D198, and PR299 we have found that there is a substantial amount of time required in order to achieve “good” results from Simulated Annealing and the Genetic Algorithm approaches. Because we limited ourselves to the three graphs, it is not clear that the time required of the Genetic Algorithm approaches that we used is $O(N^2)$.

Section two of the paper discusses theoretical bounds for the TSP in which two important theorems are presented. These two fundamental theorems “constrain the behavior of *any* heuristic for the TSP.” They are:

Theorem 6.1 [Sahni & Gonzalez, 1976]. Assuming $P \neq NP$, no polynomial-time TSP heuristic can guarantee $A(I)/OPT(I) \leq 2^{p(N)}$ for any fixed polynomial p and all instances I .

This theorem indicates that as N grows, the approximation ratio grows exponentially with N . The approximation ratio is certainly not constant over all N . That is, for the general TSP, no polynomial-time algorithm can achieve a constant ratio bound for all N (unless $P = NP$). Fortunately, and as mentioned in Chapter One, the result of Theorem 6.1 does not apply when one restricts attention to the subclasses of the TSP that satisfy the triangle inequality (those instances that satisfy the triangle inequality are also known as Metric TSPs). And, for the case of the instances I of the TSP that satisfy the triangle inequality we have the following theorem:

Theorem 6.2 [Arora, Lund, Motwani, Sudan, & Szegedy, 1992]. Assuming $P \neq NP$, there exists an $\epsilon > 0$ such that no polynomial-time heuristic can guarantee $A(I)/OPT(I) \leq 1+\epsilon$ for all instances I satisfying the triangle inequality.

Note that the ϵ in Theorem 6.2 is certainly less than 0.5 since we know that Christofides algorithm produces a tour of weight no greater than 1.5 times the optimal. In the paper of Johnson et al., it is indicated that current proof techniques do not seem capable of showing that ϵ is even as large as $1/100$.

Figure 6.1

Recall, in our assumptions that we listed in Chapter One, we assumed that we would work with TSP instances that satisfied the triangle inequality. In fact we actually worked with the set of instances that satisfied the Euclidean distance metric, which is a **subset** of the instances that satisfy the triangle inequality (see Figure 6.1). And if we restrict our attention to those instances of the TSP which satisfy the Euclidean distance metric, Sanjeev Arora has given a polynomial-time approximation scheme (PTAS) for the Euclidean TSP [Arora 1996] which we will discuss in a moment.

First, recall that an **approximation scheme** for an optimization problem, such as the TSP, is an algorithm that takes as input not only an instance I of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ , the scheme finds an approximation $A(I)$ such that $A(I)/OPT(I) \leq 1+\epsilon$. An approximation scheme is called **polynomial-time approximation scheme** if the scheme runs in polynomial time in the size of its input. Furthermore, a polynomial-time approximation scheme is a **fully polynomial-time approximation scheme** if the running time does not increase by more than a constant factor when ϵ is decreased. That is, an approximation scheme is a **fully polynomial-time approximation scheme** if the running time is polynomial in $1/\epsilon$ as well as the size of its input [Cormen et al. 1990].

Although Arora has given a PTAS for the Euclidean TSP, his algorithm is not a fully polynomial-time approximation scheme. And, unfortunately, we have the following result [Lawler et al. 1985]:

Theorem 6.3. If $P \neq NP$ then there can be no fully polynomial-time approximation scheme for the TSP, even if instances are restricted to points in the plane in the Euclidean metric.

In a recent paper by Sanjeev Arora [Arora 1996], “Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems”, he has presented a polynomial-time approximation scheme for the Euclidean TSP which is $n^{O(1/\epsilon)}$. That is it finds a $(1+\epsilon)$ approximation to the TSP in time $n^{O(1/\epsilon)}$. Thus the results of Arora are pushing the limits of PTAS algorithms for the Euclidean TSP. Arora’s algorithm for the Euclidean TSP “recursively partitions the plane so that very few edges of the tour cross each line of the partition”. The way in which the partition is constructed guarantees a near optimal tour (i.e. a $1+\epsilon$ approximation). In a more recent paper [Arora 1997] he gives an improved version of the algorithm with running time $n(\log n)^{O(1/\epsilon)}$. Lawler et al. eluded to such a scheme in their chapter on Approximation Schemes. Also Arora mentions that schemes similar to his have been proposed before.

In the middle of section two Johnson et al. discuss the Held-Karp lower bound. When comparing results of various algorithms it is important to know the value of the optimal tour length. However, when this is unavailable and infeasible to compute, we compromise and settle for a well known lower bound to the optimal solution, namely, the Held-Karp lower bound. The Held-Karp lower bound can be computed exactly using linear programming. That is, “the bound is a solution to the standard linear programming relaxation of the TSP.” And when it is too difficult to calculate using a linear programming method, it can be approximated (Johnson calls this the lower bound on the lower bound!). In Chapter One, we presented an algorithm for approximating the Held-Karp lower bound.

At the end of section two, Johnson et al., present four well known tour construction heuristics. Three of the four tours presented have already been discussed in Chapter Five of this paper: Nearest Neighbor (NN), Greedy (GR), and Christofides (CHR). The other tour construction heuristic discussed by Johnson et al. in section two, Clarke-Wright (CW), has not. We only implemented the Nearest Neighbor (NN) approach in our paper. So we will discuss briefly the results obtained by Johnson, et al. in their implementation of NN.

Recall, that the instances that Johnson et al. chose to work with were those instances that

had 1,000 to 1,000,000 vertices. There are two main sources of instances between 1,000 and 1,000,000 vertices. The randomly created instances, and those instances from TSPLIB. The 15 largest two-dimensional Euclidean instances that they selected from the TSPLIB averaged roughly 26% above the Held-Karp lower bound. Our averages above the optimal for the three instances was 27.1% for the random start case, and 16.2% when we chose the best of the N tours generated (for the three graphs we presented of 105, 199, and 299 vertices). The random start case that we performed is certainly in line with the results of Johnson et al. Also, they site that the results between the randomly generated instances and those from the TSPLIB are “surprisingly consistent between the two instance sources”. They found that results from randomly generated Euclidean graphs averaged less than 24% above the Held-Karp lower bound. Their results for randomly generated graphs are very close to their result of 27% above optimal for the largest 15 graphs in TSPLIB. Johnson et al. do not state whether their run for Nearest Neighbor uses a random start vertex or that they selected the best of several tours. Although we think it is safe to assume that they used a random start vertex since their results are very similar to our results for the random start vertex case. And also Johnson et al., talk about the running time of the Nearest Neighbor algorithm as $O(N^2)$ and if they had run the deterministic case, as we have done, the run time is $O(N^3)$.

Although we did not implement the Christofides heuristic, it is worth mentioning the results that Johnson et al. obtained. On random instances they get just below 10% on average above the Held-Karp lower bound. The Christofides heuristic that they perform uses a more sophisticated short-cutting technique than simply moving to the next unvisited vertex in the Eulerian tour (they mention that if the more sophisticated short-cutting technique was not performed, then they would get about 15% above the Held-Karp lower bound on average). However, the running time for Christofides’ heuristic grows faster than N^2 as expected, whereas the running times for the other three methods (NN, CW and GR) grow “subquadratically”.

In section three of the Johnson et al. paper we find a discussion on the theoretical bounds of 3-Opt and 2-Opt and also some experimental results as well as a discussion on how these local optimization methods might “be sped up using parallelism”. Their implementation of 2-Opt and 3-Opt is significantly different than the implementations that we used. Recall that the implementation that was used in the GA was to randomly select candidate edges for a 2-

Opt or 3-Opt maneuver. If the result of performing the 2-Opt or 3-Opt operation on these randomly chosen candidate edges results in a better tour, then the maneuver was accepted. Otherwise the 2-Opt or 3-Opt operation was rejected. In our implementation of the 2-Opt and 3-Opt procedures we were following more closely what is done in Simulated Annealing. The 2-Opt and 3-Opt algorithms that Johnson et al. implemented works as follows:

- (1) Generate a starting tour using a randomized Greedy heuristic. In this randomized version, the two available edges of least weight are candidates in the selection and the edge of least weight has a $2/3$ probability of being used.
- (2) Generate a nearest neighbor list for each vertex.
- (3) The local optimization chooses the first improving move that is found using the nearest neighbor list.
- (4) The algorithm halts when no more improving moves can be made.

The results of the 2-Opt using this algorithm average about 5% to 6% above the Held-Karp lower bound for all random Euclidean instances between 100 and 1,000,000 vertices. The 3-Opt does about twice as good as the 2-Opt, averaging from 2.5% to 3.0% for the same instances.

Section six discusses the Genetic Algorithm and *iterated local optimization* algorithms that have been adopted from them. The section opens with a brief introduction to the Genetic Algorithm. In this introduction an outline of the processing steps of the Genetic Algorithm is given in a figure. One of the steps in this figure is a local optimization step and to this Johnson et al. note that “local optimization steps appear to be essential if competitive results are to be obtained for the TSP, and we shall restrict our attention in what follows to genetic algorithms that use them.” We also have evidence that local optimization steps are imperative if one wants to obtain satisfactory results from the Genetic Algorithm.

Johnson et al. describe a genetic algorithm proposed by Brady in 1985 [Brady 1985] that they believe to be the first genetic algorithm that uses the processing steps they outlined in the figure during the introduction to this section. In this genetic algorithm, Brady’s crossover step was as follows: examine the two parents until a subset of the vertices of G is found in a subtour of both parents but in a different ordering. Then replace the longer subtour by the

shorter subtour to obtain a child. Johnson et al. note that this approach of looking for a subset of vertices but in a different order is limited since finding such a subset will become less likely as N increases. Brady utilized the 2-Opt method for his local optimization step.

Johnson et al. then discuss an alternative approach used by Suh and Van Gucht [Suh and Van Gucht 1987a, 1987b]. In this approach the crossover used is called the *heuristic* crossover from Grenfenstete [Grenfenstete et al. 1985]. This crossover operator is much like the Whitley et al. Edge Recombination operator that we studied. This operator works as follows: (1) begin by randomly picking a start vertex v , (2) let the next vertex be the closest unvisited vertex that is a neighbor of v in either of the parent tours (if there is none, then randomly select an unvisited vertex as the next vertex). After crossover, a truncated 2-Opt local optimization (one that does not guarantee local optimality) was used. Suh and Van Gucht ran this GA on cases with less than 200 vertices and with a population size of 100. Johnson et al. report that the results of Suh and Van Gucht are comparable to those for the 3-Opt local optimization algorithm. However, Johnson et al. indicate that the running times are far in excess of the .1 seconds or less required by their 3-Opt neighbor-list implementation for graphs that have the same number of vertices.

The last algorithm that Johnson et al. discuss is praised by them as a “major leap in genetic algorithm performance”. This algorithm is the Mühlenbein Algorithm [Mühlenbein et al. 1988 and 1989]. It is a parallel implementation of the Genetic Algorithm with a “more sophisticated mating strategy, together with what appears to be an even better crossover”. The parallelization of the algorithm is obtained by assigning one tour to each processor. Processors are connected in a grid such that each processor has four neighbors. Each iteration of the algorithm consists of each processor (the receiver) mating its tour with some other processor’s tour (the donor) biased towards shorter tours. The donors were selected among the four neighboring processors and the best tour overall. Once a donor has been selected, the crossover step works by selecting a subpath between 10 and $N/2$ vertices from the donor and then extending the path similar to the *heuristic* method described above. However, when adding a next vertex, vertices are biased towards those in the receiver in the following manner: if v is the current vertex in the tour being extended, use a neighbor of v in the receiver tour that has not yet been visited. If no such vertex exists then use a neighbor of v in the donor tour that has not yet been visited. If again, no such vertex exists, then

simply select the next unvisited vertex. The Mühlenbein Algorithm local optimization step uses a truncated 2-Opt procedure (i.e. one that does not guarantee local optimality), with one caveat: before running the local optimization procedure, all subpaths of four or more vertices that are present in both parents are *locked*, preventing the 2-Opt from breaking the subpath.

Johnson et al. report the results given by Mühlenbein et al. for running this algorithm on a network of 64 Transputers. In this case they ran the algorithm on the TSPLIB instance ATT532 and obtained an average of 0.19% above the optimal tour length. Johnson et al. note that this is much better than the average of 0.94% above the optimal obtained by the Lin-Kernighan algorithm that they implemented. However the running time of the Mühlenbein Algorithm is roughly 1 to 3 sequential hours compared to 2 seconds for a run of the Lin-Kernighan Algorithm. They also note that if they run the Lin-Kernighan Algorithm for 500 runs, taking about 10 minutes of processing time, they can obtain tours that are 0.15% above the optimal. Also Johnson et al. note that this version of the Genetic Algorithm is in the same ballpark of what can be accomplished by their best simulated annealing variant. In further studies, Braun in 1991 went back to the original Mühlenbein Algorithm that performed both 2-Opt and an extension of a simplified version of 3-Opt to local optimality and implemented these procedures efficiently using neighbor-lists (which was not done in the Mühlenbein et al. implementation) [Braun 1991]. Johnson et al. report that results obtained by Braun in 1991 indicate for the 431 vertex TSPLIB instance he was obtaining the optimal tour in about one-half of his runs and took about 35 minutes on a Sun workstation.

Of the studies of the genetic algorithms reviewed by Johnson et al. they state that “for a given amount of sequential time, the performance of genetic algorithms can be improved by using a more powerful local optimization algorithm...” and that, “performance can also be improved by increasing the population size.” To this Johnson et al. ask the question: which is more important, increasing the population size or using a more powerful local optimization procedure? And they answer the question: “In the context of the TSP, it now appears that using a more powerful local optimization procedure is by far the more important factor.” With this, Johnson et al., finish up section six with a discussion on the Martin, Otto, and Felten Algorithm [Martin, Otto, and Felten, 1991 and 1992] and the Iterated Lin-Kernighan Algorithm.

The Martin-Otto-Felten Algorithm is very similar to the simulated annealing approach. It uses a population size of one and relies on mutation to drive the search process. A child is obtained from the parent by a mutation. The mutation is a “double-bridge” 4-Opt (not described in the Johnson et al. paper). Selection is then solely based on whether the parent or the child should move to the next generation. The selection is done as in the Simulated Annealing Algorithm. That is, if the child is shorter than the parent it is selected, otherwise it is selected with probability $e^{-\Delta/T}$ where Δ is the amount by which the child exceeds the length of its parent and T is the current temperature. For the local optimization step, the Martin-Otto-Felten Algorithm uses a 3-Opt procedure utilizing neighbor lists. Johnson et al. state that the results of this algorithm are “quite promising”. They site that for the LIN318 instance of TSPLIB it found optimal solutions in 4-6 minutes, and for the ATT532 instance it got within 0.07% of optimal in 1-1.5 hours.

The last algorithm discussed is the Iterated Lin-Kernighan. This algorithm is a version of the Martin-Otto-Felten Algorithm with the 3-Opt local optimization procedure replaced by the Lin-Kernighan local optimization procedure.

Recall, that Johnson et al. stated in section one that good results can be achieved from the Genetic Algorithm if one is willing to spend more, although still $O(N^2)$, time. As a last attempt we decided to see if we could obtain better results by running the Genetic Algorithm on PR299 for a longer period of time (more iterations). The results are averaged over 50 runs and are shown in Table 6.1. The results are reported for the two Genetic Algorithm models, Steady State and Generational, utilizing two different population sizes. In the previous section we ran the Steady State model GA for 8000 iterations and the Generational model GA for 320 iterations. In this experiment we have tripled the number of iterations. We also ran each case with a larger population size to see if population size has any significant effect on the tour quality.

In each case the 50% Martin-Opt and 50% 2-Opt/3-Opt method was used as the local optimization procedure, and the Edge Recombination operator was used as the crossover operator. All other parameters remained the same as they were when we compared results of the various local optimization and crossover operator in Chapter Four. In Chapter Four,

Table 6.1
PR299 w/ Increased Iterations and Increased Population Size

Genetic Algorithm Model	Population Size	Iterations	Average System Time (seconds) Required	Average Percent Above Optimal
Generational	100	960	0.714003	11.2
Steady State	100	24000	0.766409	10.8
Generational	200	960	1.714184	10.8
Steady State	200	24000	1.136929	9.8

the Steady State model obtained 10.4% above optimal (see Table 4.3) on the same graph after only 8000 iterations (this small difference in average percent above optimal is attributed to a different seed value for the randomizer). This “no difference” in tour quality between 8000 iterations and 24000 iterations indicates that the procedure is converging too quickly to a local optimal solution from which the algorithm can not escape. On the other hand, note that the results of the Generational Model after 960 iterations produces tours that average 11.2% above optimal. This is substantially better than the results given in Table 4.8 of 27.8% above optimal when the algorithm was run for only 320 iterations. The last two lines in Table 6.1 reflect the same algorithms as the previous two except for a change in the population size from 100 to 200. For the Steady State model, an increase of population size produced, on average, 10% better results at a cost of about 50% more processing time. For the Generational model, the increase in population size produces tours about 4% better at a cost of almost 150% more processing time.

Conclusions

In this paper we have investigated the Genetic Algorithm in various forms and its methodology for approximating solutions for the Traveling Salesman Problem (TSP). We have also briefly examined other methods for approximating solutions to the TSP, such as Nearest Neighbor, Minimum Spanning Tree, and Simulated Annealing. We performed experiments on three graphs from a library of test instances called TSPLIB. The three graphs we used in our experiments were LIN105, D198 and PR299 of 105, 198 and 299 vertices respectively. If one wants to implement the Genetic Algorithm for the TSP our results obtained from the experiments that we performed can provide some guidance on which way

to go from here.

First, our results indicate that the Steady State model overall produces better results than the Generational model in roughly the same amount of time. For the Generational model we even doubled the number of iterations from what we thought might be the equivalent number of iterations for the Steady State model. These conclusions for the comparison between the Steady State model and the Generational model are based on results obtained in Chapter Four, Tables 4.1-3 and 4.6-8.

Also from Tables 4.1-3 and 4.6-8, in either model it is evident that a local optimization procedure is imperative in obtaining quality approximations to the TSP using Genetic Algorithms. The dependence on a local optimization procedure for obtaining quality tours in a genetic algorithm is also iterated by Johnson et al. as discussed in the first part of this chapter. For our own experiments, in Tables 4.1-3, we find that our version of 2-Opt produces better results than our version of 3-Opt in a shorter amount of time regardless of which crossover method has been selected. However, the best overall results for our experiments were obtained using the 50% Martin-Opt and 50% 2-Opt/3-Opt local optimization procedure. Johnson et al. indicate that the performance of genetic algorithms can be greatly improved by finding an effective local optimization procedure. In our experiments, we certainly have not chosen the most effective local optimization that we could possibly have. Thus more work could be done in this area. And certainly using a more effective local optimization procedure, as those discussed by Johnson et al. that used neighbor lists, should be further investigated.

Furthermore, results from Chapter Four indicate that in experimenting with the two crossover operators that we discussed in Chapter Three, we have found the Edge Recombination operator generally produced better results than the Bui-Moon crossover operator, regardless of the local optimization procedure that was used. Thus, if one wants to implement the Genetic Algorithm for approximating the TSP, our recommendation is to use a the Steady State model combined with the Edge Recombination operator and the 50% Martin-Opt and 50% 3-Opt/2-Opt local optimization procedure. In our experiments, as evident in Chapter Four from Tables 4.1-3 and 4.6-8, this combination provided the best results overall. The Edge Recombination operator is very similar to the crossover operator

discussed by Johnson et al. in the implementation of the Mühlenbein Algorithm. The results of Bui-Moon are fairly new, and although we did not implement the proposed preprocessing step, we believe that further comparisons should be performed using their proposed preprocessing step.

In Chapter Five we compared the results obtained in Chapter Four to other approaches. Notably, the traditional approaches that we ran produced approximations in substantially less time than the Genetic Algorithm. However the quality of the tours produced were substantially worse than the tours obtained by the Genetic Algorithm. However, one approach that we experimented with in Chapter Five, the Simulated Annealing Algorithm from Numerical Recipes, produced noticeably better results than the Genetic Algorithm in roughly the same amount of time. The Simulated Annealing Algorithm is certainly a contender for an alternative approach to the Genetic Algorithm. Its implementation was very straight forward, requiring fewer parameters than the Genetic Algorithm, and it produced very good results.

From the Johnson et al. paper we find many variants of the genetic algorithm for the Traveling Salesman Problem. A key algorithm discussed by Johnson et al. is the Mühlenbein Algorithm. I believe it is important because it demonstrates that parallelization may play an important role in the success of the Genetic Algorithm. Also an important algorithm discussed by Johnson et al. is the Martin, Otto, Felten Algorithm. This algorithm is a variant on the genetic algorithm that is very similar to simulated annealing.

From the review of the paper of Johnson et al., it appears that future algorithms for approximating solutions to the TSP should be most concentrated in incorporating an effective local optimization procedure. Furthermore, such a future algorithm may most likely be a hybrid of the genetic algorithm and simulated annealing approaches designed for parallel implementation.

APPENDICES

Appendix A: TSPLIB

In this appendix we show plots of the three graphs LIN105, D198, and PR299 from the TSPLIB that we used for our experiments in the previous chapters. In all of the plots the optimal tour is plotted with the graph. For the D198 graph, Figure A1.2a shows the graph with all of the vertices. In graph D198 the initial vertex is located at the very bottom left hand corner, and is relatively distant from the other vertices. Thus we have also given a plot of D198 without its initial vertex shown in Figure A1.2b. This was done so that one may make out the positions of the vertices a little better.

Figure A1.1. TSPLIB graph LIN105 with optimal tour.

Figure A1.2a. TSPLIB graph D198 with optimal tour.

Figure A1.2b. TSPLIB graph D198- v_1 with D198 optimal tour.

Figure A1.3. TSPLIB graph PR299 with optimal tour.

Appendix B: LibGA

In this appendix we give the extra code for the crossover operators that were implemented for LibGA.

Whitley Genetic Edge Recombination Operator

```
/*-----
| Helper variables for Edge crossover (Whitley, et. al.)
|-----*/
static int *dc1 = NULL; /* Degree array for generating child 1 */
static int *dc2 = NULL; /* Degree array for generating child 2 */
static int *pa1 = NULL; /* Previous array for parent 1 */
static int *na1 = NULL; /* Next array for parent 1 */
static int *pa2 = NULL; /* Previous array for parent 2 */
static int *na2 = NULL; /* Next array for parent 2 */

void X_ex_in_neig( int *, int , int *, int * );

/*-----
| Edge crossover (Whitley, et. al.)
|-----*/
X_ex(ga_info, parent_1, parent_2, child_1, child_2)
    GA_Info_Ptr ga_info;
    Chrom_Ptr parent_1, parent_2;
    Chrom_Ptr child_1, child_2;
{
    int i,j,k,l,dhigh,na[4],idx,v,n;

    /* Make sure datatype is compatible */
    if(ga_info->datatype != DT_INT_PERM)
        UT_error("X_order1: bad data type");

    /* Cannot yet deal with heterozygous parents */
    if(parent_1->length != parent_2->length)
        UT_error("crossover: heterozygous parents");

    l = parent_1->length;

    /* Initialize arrays */
    if( dc1 == NULL ) {
        dc1 = (int*)malloc(sizeof(int)*l);
        dc2 = (int*)malloc(sizeof(int)*l);
        pa1 = (int*)malloc(sizeof(int)*l);
        pa2 = (int*)malloc(sizeof(int)*l);
        na1 = (int*)malloc(sizeof(int)*l);
        na2 = (int*)malloc(sizeof(int)*l);
    }

    /* Assign values to succ and pred arrays for each parent */
    for(i=0;i<l;i++) {
```

```

        v = (int)parent_1->gene[i]-1;
        na1[v] = (int)parent_1->gene[(i+1)%l]-1;
        pa1[na1[v]] = v;
        v = (int)parent_2->gene[i]-1;
        na2[v] = (int)parent_2->gene[(i+1)%l]-1;
        pa2[na2[v]] = v;
    }

    /* Find highest degree of all the vertices */
    dhigh = 2;

    for(i=0;i<l;i++) {
        dcl[i]=2;
        if( na1[i]!=na2[i] && na1[i]!=pa2[i] ) dcl[i]++;
        if( pa1[i]!=na2[i] && pa1[i]!=pa2[i] ) dcl[i]++;
        if( dcl[i] > dhigh ) dhigh = dcl[i];
        dc2[i]=dcl[i];
    }

    /* Generate random starting point at vertex of degree dhigh */
    idx = RAND_DOM(0,l-1);
    while( dcl[idx] < dhigh ) idx = (idx+1)%l;

    for(i=0;i<l-1;i++) {
        child_1->gene[i] = (Gene_Type)idx+1;
        /* Set the degree to -1, meaning visited */
        dcl[idx]=-1;
        n=0;
        X_ex_in_neig( dcl, na1[idx], na, &n);
        X_ex_in_neig( dcl, pa1[idx], na, &n);
        if( na2[idx]!=na1[idx] && na2[idx]!=pa1[idx] )
            X_ex_in_neig( dcl, na2[idx], na, &n);
        if( pa2[idx]!=na1[idx] && pa2[idx]!=pa1[idx] )
            X_ex_in_neig( dcl, pa2[idx], na, &n);
        if( n == 0 ) {
            idx = RAND_DOM(0,l-1);
            while( dcl[idx] == -1 ) idx = (idx+1)%l;
            nNewEdges++;
        } else if( n==1 ) {
            idx = na[0];
        } else {
            idx = na[RAND_DOM(0,n-1)];
        }
    }

    child_1->gene[i] = (Gene_Type)idx+1;
    dcl[idx]=-1;

    idx = RAND_DOM(0,l-1);
    while( dc2[idx] < dhigh ) idx = (idx+1)%l;

    for(i=0;i<l-1;i++) {
        child_2->gene[i] = (Gene_Type)idx+1;
        dc2[idx]=-1; /* Set vertex to visited */
        j=0;

```

```

X_ex_in_neig( dc2, na1[idx], na, &j);
X_ex_in_neig( dc2, pa1[idx], na, &j);
if( na2[idx]!=na1[idx] && na2[idx]!=pa1[idx] )
    X_ex_in_neig( dc2, na2[idx], na, &j);
if( pa2[idx]!=na1[idx] && pa2[idx]!=pa1[idx] )
    X_ex_in_neig( dc2, pa2[idx], na, &j);
if( j == 0 ) {
    idx = RAND_DOM(0,l-1);
    while( dc2[idx] == -1 ) idx = (idx+1)%l;
    nNewEdges++;
} else if( j==1 ) {
    idx = na[0];
} else {
    idx = na[RAND_DOM(0,j-1)];
}
}
child_2->gene[i] = (Gene_Type)idx+1;
dc2[idx]=-1;
return OK;
}

void X_ex_in_neig( int *d, int v, int *n, int *l ) {
    int i;
    if( d[v] == -1 ) return;
    d[v]--;
    for(i=0;i<(*l);i++) {
        if( v == n[i] ) return;
        if( d[n[i]] > d[v] ) {
            n[i]=v;
            (*l)=i+1;
            return;
        }
    }
    if( d[n[i]] < d[v] ) return;
}
n[i] = v;
(*l)++;
}

```

Bui-Moon Crossover

```

/*-----
| Helper variables for Bui crossover
|-----*/
static int *picked = NULL; /* Picked Array for vertices */
static int *xpoint = NULL; /* cross over point array */
static int *child1 = NULL; /* child1 successor array */
static int *child2 = NULL; /* child1 successor array */

/*-----
| Bui Crossover
|-----*/
X_bui(ga_info, parent_1, parent_2, child_1, child_2)
    GA_Info_Ptr ga_info;
    Chrom_Ptr parent_1, parent_2;

```

```

    Chrom_Ptr  child_1, child_2;
{
    int i,j,k,l,t,idx,v,n,nxp;

    /* Number of crossover points */
    nxp = 5;

    /* Make sure datatype is compatible */
    if(ga_info->datatype != DT_INT_PERM)
        UT_error("X_order1: bad data type");

    /* Cannot yet deal with heterozygous parents */
    if(parent_1->length != parent_2->length)
        UT_error("crossover: heterozygous parents");

    l = parent_1->length;

    /* Initialize arrays */
    if( pal == NULL ) {
        pal = (int*)malloc(sizeof(int)*l);
        pa2 = (int*)malloc(sizeof(int)*l);
        na1 = (int*)malloc(sizeof(int)*l);
        na2 = (int*)malloc(sizeof(int)*l);
        child1 = (int*)malloc(sizeof(int)*l);
        child2 = (int*)malloc(sizeof(int)*l);
        picked = (int*)malloc(sizeof(int)*l);
        xpoint = (int*)malloc(sizeof(int)*(nxp+2));
    }

    /* Assign values to the succ and pred arrays for each parent tour */
    for(i=0;i<l;i++) {
        v = (int)parent_1->gene[i]-1;
        na1[v] = (int)parent_1->gene[(i+1)%l]-1;
        pal[na1[v]] = v;
        v = (int)parent_2->gene[i]-1;
        na2[v] = (int)parent_2->gene[(i+1)%l]-1;
        pa2[na2[v]] = v;
        picked[i] = 0;
        child1[i] = -1;
    }

    /* Generate the cross over points randomly */
    for( i=1; i<=nxp; i++ ) {
        j = RAND_DOM(1,l-1);
        for( k=1; k<i; k++ )
            if( xpoint[k] > j ) {
                t = xpoint[k]; xpoint[k] = j; j = t;
            }
        xpoint[i] = j;
    }
    xpoint[0] = 0;
    xpoint[nxp+1] = l;

    for(i=0;i<=nxp;i+=2)
        for(j=xpoint[i];j<xpoint[i+1];j++) {

```

```

        v = na1[j];
        child1[j]=v;
        picked[v]=1;
    }

    for(i=1;i<=nxp;i+=2)
        for(j=xpoint[i];j<xpoint[i+1];j++) {
            v = na2[j];
            if( !picked[v] ) child1[j]=v;
            picked[v]=1;
        }

    . for(i=0;i<=nxp;i++)
        for(j=xpoint[i];j<xpoint[i+1];j++)
            if( child1[j] == -1 ) {
                if( i%2 == 0 ) {
                    v = pa1[j];
                    if( picked[v] ) v = na2[j];
                    if( picked[v] ) v = pa2[j];
                } else {
                    v = pa2[j];
                    if( picked[v] ) v = na1[j];
                    if( picked[v] ) v = pa1[j];
                }
                if( !picked[v] ) child1[j] = v;
                picked[v]=1;
            }

    for(i=0;i<l;i++)
        if( child1[i]==-1 ) {
            idx = RAND_DOM(0,l-1);
            while( picked[idx] ) idx = (idx+1)%l;
            child1[i]=idx;
            picked[idx]=1;
        }

    resolve_cycles( child1, l );

    /* Reset arrays needed for child2 */
    for(i=0;i<l;i++) {
        picked[i]=0;
        child2[i]=-1;
    }

    /* Generate cross over points randomly */
    for( i=1; i<=nxp; i++ ) {
        j = RAND_DOM(1,l-1);
        for( k=1; k<i; k++ )
            if( xpoint[k] > j ) {
                t = xpoint[k]; xpoint[k] = j; j = t;
            }
        xpoint[i] = j;
    }
    xpoint[0] = 0;
    xpoint[nxp+1] = 1;

```

```

    for(i=0;i<=nxp;i+=2)
        for(j=xpoint[i];j<xpoint[i+1];j++) {
            v = na1[j];
            child2[j] = v;
            picked[v] = 1;
        }

    for(i=1;i<=nxp;i+=2)
        for(j=xpoint[i];j<xpoint[i+1];j++) {
            v = na2[j];
            if( !picked[v] ) child2[j]=v;
            picked[v] = 1;
        }

    for(i=0;i<=nxp;i++)
        for(j=xpoint[i];j<xpoint[i+1];j++)
            if( child2[j] == -1 ) {
                if( i%2 == 0 ) {
                    v = pa1[j];
                    if( picked[v] ) v = na2[j];
                    if( picked[v] ) v = pa2[j];
                } else {
                    v = pa2[j];
                    if( picked[v] ) v = na1[j];
                    if( picked[v] ) v = pa1[j];
                }
                if( !picked[v] ) child2[j] = v;
                picked[v] = 1;
            }

    for(i=0;i<l;i++)
        if( child2[i]==-1 ) {
            idx = RAND_DOM(0,l-1);
            while( picked[idx] ) idx = (idx+1)%l;
            child2[i]=idx;
            picked[idx]=1;
        }

    resolve_cycles( child2, 1 );

    j = RAND_DOM(0,l-1);
    k = RAND_DOM(0,l-1);

    for(i=0;i<l;i++) {
        child_1->gene[i] = (Gene_Type)j+1;
        child_2->gene[i] = (Gene_Type)k+1;
        j = child1[j];
        k = child2[k];
    }

    return OK;
}

```

Resolve Cycles

```
/* vertex visited array */
static int *visited = NULL;

void resolve_cycles( int *v, int l ) {
    int a,b,t,i,n;

    /* Perform initialization */
    if( visited == NULL ) visited = (int*)malloc(sizeof(int)*l);
    for(i=0;i<l;i++) visited[i]=0;

    /* Get the first cycle */
    a = i = RAND_DOM(0,l-1); n = 1; visited[i] = 1;
    while( v[i] != a ) i = v[i], n++, visited[i] = 1;

    /* Add subsequent cycles */
    while( n<l ) {

        /* Find an unvisited city */
        i = RAND_DOM(0,l-1);
        while( visited[i] ) i=(i+1)%l;

        /* Find its cycle */
        b = i; visited[i] = 1; n++;
        while( v[i] != b ) i = v[i], n++, visited[i] = 1;

        /* Join cycle */
        t = v[a]; v[a] = v[i]; v[i] = t;
    }
}
```

REFERENCES

- S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy, 1992, "Proof Verification and Hardness of Approximation Algorithms", *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pages 14-23.
- S. Arora, 1997, "Nearly Linear Time Approximation Schemes for Euclidean TSP and other Geometric Problems", private communication, email: arora@cs.princeton.edu.
- S. Arora, 1996, "Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems", private communication, email: arora@cs.princeton.edu.
- D. Beasley, D.R. Bull and R.R. Martin, 1993, "An Overview of Genetic Algorithms: Part 2, Research Topics", *University Computing*, 15(4) pages 170-181. Available by ftp from ENCORE in file: GA/papers/over93-2.ps.gz.
- D. Beasley, D.R. Bull and R.R. Martin, 1993, "An Overview of Genetic Algorithms: Part 1, Fundamentals", *University Computing*, 15(2) pages 58-69. Available by ftp from ENCORE in file: GA/papers/over93.ps.gz.
- J.L. Bentley, 1992, "Fast Algorithms for Geometric Traveling Salesman Problems", *ORSA Journal on Computing* 4, pages 387-411.
- R.M. Brady, 1985, "Optimization Strategies Gleaned from Biological Evolution", *Nature* 317, pages 804-806.
- H. Braun, 1991, "On Solving Travelling Salesman Problems by Genetic Algorithms", in "Parallel Problem Solving from Nature", H.-P. Schwefel and R. Männer (eds.), *Lecture Notes in Computer Science* 496, pages 129-133, Springer-Verlag, New York, NY.
- T.N. Bui and B.R. Moon, 1994, "A New Genetic Approach for the Traveling Salesman Problem", *Proceedings of the IEEE Conference on Evolutionary Computation* 1, pages 7-12.

- A.L. Corcoran and R.L. Wainwright, 1993, "LibGA: A User-Friendly Workbench for Order-Based Genetic Algorithm Research", *Proceedings of the 1993 ACM SIGAPP Symposium on Applied Computing*, Indianapolis, Indiana, February 14-16.
- T.H. Cormen, C.E. Leiserson and R.L. Rivest, 1990, *Introduction to Algorithms*, MIT Press.
- L. Davis (ed.), 1991, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, NY.
- L. Davis, (ed.), 1987, *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publishers.
- L. Euler, 1736, "Solutio Problematis ad Geometriam Situs Pertinentis", *Commentarii Academiae Petropolitanae* 8, pages 128-140 (in Latin).
- D.B. Fogel, 1994, "An Introduction to Simulated Evolutionary Optimization", *IEEE Trans. Neural Networks* 5(1), pages 3-14.
- D.E. Goldberg, 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub. Co., Reading, MA.
- D.E. Goldberg, 1994, "Genetic and Evolutionary Algorithms Come of Age", *Communications of the ACM* 37(3), pages 113-119.
- J. Grefenstette and J.E. Baker, 1989, "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism", *Third International Conference on Genetic Algorithms*, pages 20-27.
- J. Grefenstette, R. Gopal, B.J. Rosmaita and D. Van Gucht, 1985, "Genetic Algorithms for the Traveling Salesman Problem", *Proceedings of the International Conference on Genetic Algorithms and their Applications*, pages 160-168.

- J. Grefenstette, 1987, "Incorporating Problem Specific Knowledge in Genetic Algorithms", *Genetic Algorithms and Simulated Annealing*, L. Davis (ed.).
- J. Heitkoetter and D. Beasley (eds.), 1994, "The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)", *USENET: comp.ai.genetic*, available via anonymous ftp from [rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic/](ftp://rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic/), about 90 pages.
- M. Held and R.M. Karp, 1970, "The traveling-salesman problem and minimum spanning trees: Part II", *Mathematical Programming* 1, pages 6-25.
- D.S. Johnson, L.A. McGeoch and E.E. Rothberg, 1996, "Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound", *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 28-30.
- D.S. Johnson and L.A. McGeoch, 1995, The Traveling Salesman Problem: A Case Study in Local Optimization. Chapter in *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (eds.), John Wiley and Sons, New York, NY. (To appear in 1997?).
- J.R. Koza, 1992, *Genetic Programming: On the Programming of Computers by means of Natural Selection*, MIT Press, Cambridge, MA.
- E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.), 1985, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, NY.
- O. Martin, S.W. Otto and E.W. Felten, 1991, "Large-step Markov Chains for the Traveling Salesman Problem", *Complex Systems* 5, pages 299-326.
- O. Martin, S.W. Otto and E.W. Felten, 1992, "Large-step Markov Chains for the TSP Incorporating Local Search Heuristics", *Operations Research Letters* 11, pages 219-224.

- Z. Michalewicz, 1992, *Genetic algorithms - Data Structures = Evolution Programs*, Springer-Verlag, New York, NY.
- H. Mühlenbein and J. Kindermann, 1988, "The Dynamics of Evolution and Learning - Towards Genetic Neural Networks", *Connectionism in Perspective*, R. Pfeiffer (ed.), North-Holland, Amsterdam, pages 173-198.
- H. Mühlenbein, M. Georges-Schleuter and O. Krämer, 1988, "Evolution Algorithms in Combinatorial Optimization", *Parallel Computing* 7, pages 65-85.
- C.H. Papadimitriou and K. Steiglitz, 1982, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.
- W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery (eds.), 1992, *Numerical Recipes in C: The Art of Scientific Computing*, Second Edition, Cambridge University Press., Cambridge, MA.
- G. Reinelt, 1994, "The Traveling Salesman Problem: Computational Solutions for TSP Applications", *Lecture Notes in Computer Science* 840, Springer-Verlag, New York, NY.
- G. Reinelt, 1995, "TSPLIB95", available via anonymous FTP from <ftp://ftp.iwr.uni-heidelberg.de/pub/tsplib/>.
- D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis, 1977, "An Analysis of Several Heuristics for the Traveling Salesman Problem", *J. SIAM Computing* 6, pages 563-581.
- S. Sahni and T. Gonzalez, "P-complete Approximation Problems", *Journal of the ACM* 23, pages 555-565.
- J.Y. Suh and D. Van Gucht, 1987b, "Incorporating Heuristic Information into Genetic Search", *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 100-107.

- J.Y. Suh and D. Van Gucht, 1987a, "Distributed Genetic Algorithms", Report No. 225, Computer Science Department, Indiana University, Bloomington, IN
- H. Tamaki, H. Kita, N. Shimizu, K. Maekawa and Y. Nishikawa, 1994, "A Comparison Study of Genetic Codings for the Traveling Salesman Problem", *Proceedings of the IEEE Conference on Evolutionary Computation* 1, pages 1-6.
- A. Volgenant and R. Jonker, 1982, "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation", *European Journal of Operational Research* 9, pages 83-89.
- T. Volgenant and W.B. van den Hout, 1990, *TSP1 and TSP2: Symmetric Traveling Salesman Problem for Personal Computer*, Institute of Actuarial Science & Econometrics, University of Amsterdam.
- A. Volgenant and R. Jonker, 1983, "The symmetric traveling salesman problem and edge exchanges in minimal 1-trees", *European Journal of Operational Research* 12, pages 394-403.
- D. Whitley, T. Starkweather and D'Ann Fuquay, 1989, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombinator", *Third International Conference on Genetic Algorithms*, pages 133-140.
- D. Whitley, 1989, "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best", *Third International Conference on Genetic Algorithms*, pages 116-121.