A New Genetic Algorithm for Traveling Salesman Problem and Its Application

by

Mr. Lee, Ka-Wai

supervised by

Dr. X. Cai

Submitted to the Department of Systems Engineering and Engineering Management
in partial fulfillment of the requirements for the degree of

Master of Philosophy

at the

THE CHINESE UNIVERSITY OF HONG KONG

June 1995

QA
402.5
L43
1995
ult

Abstract

This thesis consists of two parts. In part I, we propose, based on a new approach presented by Cai in 1991, two new crossover operators called LDPX (Local Dynamic Programming Crossover) and SPIR (single parent improved reproduction) for genetic algorithms for solving traveling salesman problem (TSP). We also develop a general-purpose TSP Solver called GASL which uses SPIR and LDPX as the core of genetic operators. Experimental results show that GASL can obtain optimal or near-optimal solutions for all TSPs we have tested. Comparison with other genetic algorithms with different crossover operators, such as partially mapped crossover (PMX) and edge recombination crossover (ER) has also been done. Part II of this thesis will focus on applying the TSP Solver to a practical problem - Flowshop Scheduling with Travel Time Between Machines (FSTTBM). The formulation of this problem will be described and the algorithm for the transforming of FSTTBM into TSP will be proposed. Then, GASL is applied to find a solution. A lot of randomly generated FSTTBMs have been solved by our proposed method and the results of the experiments show that the new method is very effective as compared with other well-known methods.

Acknowledgement

I would like to express my deepest gratitude and appreciation to my supervisor Dr. X. Cai, who gave me invaluable advices, comments, suggestions and a lot of great ideas, especially on the construction of the genetic crossover operator.

I also wish to thank all the staff of the Department, who have given me their valuable support during my research.

Contents

1 Introduction 6

1.1 Traveling Salesman Problem 6

1.2 Genetic Algorithms 8

1.3 Solving TSP using Genetic Algorithms 10

1.4 Outline of Work 12

Part I Algorithm Development 14

2 A Local DP Crossover Operator — LDPX 15

2.1 Review of DP for Solving TSP 15

2.2 On the Original LDPX 18

2.2.1 Gene Representation 18

2.2.2 The Original Crossover Procedure 19

2.3 Analysis 21

2.3.1 Ring TSP 21

2.3.2 Computational Results of Solving Ring TSP and Other TSP using LDPX 22

2.4 Augmentation of the Gene Set Representation 24

2.5 Enhancement of Crossover Procedure 25

2.6 Computational Comparison of the new proposed LDPX with the original LDPX 26

2.7 SPIR – An Operator for Single Parent Improved Reproduction . . . 26

3 A New TSP Solver **29**

4 Performance Analysis of the TSP Solver **33**

4.1 Computational results 34

4.2 Comparison between SPIR/LDPX, PMX and ER 35

4.3 Convergence Test of SPIR/LDPX 37

Part II Application **43**

5 Flowshop Scheduling Problem **44**

5.1 Brief Review of the Flowshop Scheduling Problem 44

5.2 Flowshop Scheduling with travel times between machines 45

6 A New Approach to Solve FSTTBM **47**

7 Computational Results of the New Algorithm for CPFSTTBM **53**

7.1 Comparison with Global Optimum 54

7.2 The Algorithm of SPIRIT 55

7.3 Comparison with SPIRIT 57

8 Conclusion **59**

Bibliography **61**

A Random CPFSTTBM problem Generation Algorithm **68**

Chapter 1

Introduction

1.1 Traveling Salesman Problem

The formulation of the Traveling salesman problem (TSP) is simple, which can be stated as [44]: *"Assume that a salesman wants to visit N cities. Given the cost $c(i,j)$ of traveling from city i to city j, $i,j = 1,2,...,n$, how could he visit every city exactly once and then come back to the starting city with minimum total cost of the tour?"*. This is a classic combinatorial optimization problem, a formal description of which originated as early as in 1832 in a book printed in Germany entitled *'The Traveling Salesman, how he should be and what he should do to get Commissions and to be Successful in his Business. By a veteran Traveling Salesman'.* [63].

It has been found, however, that many practical problems may fit into this simple model. For example, consider the problem of computer network wiring, which connects all computers in a ring topology by cables or optical fibers [15] [59]. If there are N computers and we are given the costs for cabling between any two computers, a problem will be how to build up the network so that the total cost is minimum. Obviously, this problem can be formulated as an N-city TSP. Another example of application of TSP is the process of manufacturing a circuit board [46]. In this process, many holes have to be drilled using drilling machines. After the coordinates and the sizes of all the holes to be drilled are known, the machine will

automatically drill them one by one following a pre-specified sequence. The problem is: what is the optimal sequence of the holes if one wants to minimize the total time spent on the drilling process? There are many other applications of TSP, including X-ray crystallography [5] , VLSI fabrication [40], etc., see [44] and [37].

Thus, determining solutions to many practical problems reduces to finding solutions to their corresponding TSP models. Nevertheless, it has been known that TSP is very hard to solve in nature. In fact, TSP has been classified as one of the most notoriously intractable combinatorial optimization problems. In theory, it has been proven that TSP is a problem which is NP-Hard in the strong sense [19] [38]. A simple observation may help explain the difficulty of this problem. Given an N-city TSP, it is easy to see that there will be $\frac{1}{2}(N-1)!$ possible tours. If it takes 1×10^{-5} second to evaluate the cost of a tour, we shall need 36 seconds to find the optimal tour for a 10-city TSP and we shall need 151 days for a 15-city problem. If N grows up to 30, we shall need 2.8×10^{18} years to search all combinations. It means that it is practically impossible to get the global optimum tour. Considering its inherent difficulty, researchers have been striving to devise effective approaches for solving TSP.

Traditionally, there are two classes of approaches proposed for solving TSP. The first class consists of methods based on mathematical programming, such as dynamic programming [29], branch-and-bound method [47] [42], and integer programming [49]. The other class consists of those so-called heuristic methods. Generally speaking, mathematical methods can guarantee the optimality of the solution obtained, but they often require excessive computational requirements. In fact, most of these methods are so computational cumbersome that they actually cannot be applied to solve any practical problem of moderate size. Heuristic methods include nearest neighbour [58], nearest insertion [58], farthest insertion [58], convex hull method [18], greedy algorithms [43], and Lin-Kernighan's 2-opt or 3-opt methods [45] [61] [50] [37]. They are efficient, but they usually offer no guarantee to find any solution of reasonably good quality. Analysis has shown that the quality of solutions

obtained by these methods are frequently worse than that of the optimal solution by a percentage varying from 2% to 36%. Besides, the performance of these methods are usually problem dependent [44]. For instance, farthest insertion has been shown to be good in approximating the solution of a 100-city TSP problem from Krolak/Felts/Nelson [41] but much worse in solving other Krolak/Felts/Nelson's problems [41].

Because of those limitations exhibited in the traditional approaches, a third class of solution methods have emerged in recent years, which are the so-called inter-disciplinary approaches. These include simulated annealing (SA) [39], genetic algorithms (GA) [25], and taboo search [20]. These new methods have shown great promise in solving some very complicated combinatorial problems including some large-scale traveling salesman problems, although they are still under investigation and development because of the various open issues that need to be resolved. For a comprehensive review, see ([44] [25] [36]). A primary objective of this thesis is to resolve one of the critical problems in using GA for TSP. For this reason, we will now review this methodology in some details.

1.2 Genetic Algorithms

John Holland invented Genetic algorithms (GA) in 1975 when he published his book Adaptation in Natural and Artificial Systems [31]. GA have since become a topic of active research, see [25]. In general, GA are a class of searching methods based on the analogy to the natural evolution process. In searching for the optimum, a genetic algorithm always maintains a population of individuals. Each individual represents a potential solution and is associated with a value to represent its fitness. Then, the population undergoes an evolution process. During this process, fitter individuals will be selected into a parent pool for reproducing the next generation. The selection is biased towards those individuals that have higher fitness values. Offsprings (new solutions) will be generated by performing certain genetic operations, such as crossover and mutation, over the individuals drawn from the parent

pool. Then, a new generation will be produced. This evolution process continues until the population consists of individuals which are optimum or near optimum.

Classically, an individual is represented by a binary string, e.g. x_1= (01011001) and x_2= (01110101), which are often termed chromosomes. When a crossover operation is to be carried out over a pair of parents, a cutting point will be selected randomly, and the chromosomes of the parents will be both split at that point and then the segments of those chromosomes will be exchanged to give the offspring. For example, suppose parents are x_1 and x_2 as described above. If the cutting point is 3, then the offsprings will be y_1=01010101 and y_2=01111001. Mutation alters directly one or more elements of a chromosome. For example, a mutation occurring at the element 6 of the parent x_1=10111001 will generate y_1=10111101.

In general, a canonical genetic algorithm has the following steps:

1. Initialize Population

   Randomly generate an initial population POPULATION(0), and set k=0. The population consists of a number of individuals. Each individual is associated with a fitness value. The number of individuals is the population size $PSIZE$.

2. Generate the parent pool for reproduction

   Selecting individual i from POPULATION(k) into the parent pool with probability:

$$P_i = \frac{f(i)}{\sum_{j=1}^{PSIZE} f(j)},$$

   where $f(j)$ is the fitness value of individual j.

3. Crossover

   Randomly select two individuals x_1, x_2 from the parent pool. Generate a random cutting point. Exchange the splitted chromosomes of the two parents. After crossing, the offspring y_1 and y_2 generated will be put into the POPULATION(k+1). However, not all individuals in POPULATION(k+1)

come from crossover operation. In fact, some of them are randomly selected from POPULATION(k) so that population sizes will be maintained constantly throughout the evolution. The ratio of the number of individuals generated by crossover to $PSIZE$ is known as the crossover probability.

4. Mutation

   Randomly choose an individual from POPULATION(k+1) and alter randomly an element of the chromosome of the individual with probability, P_{mut}.

5. If the standard deviation of POPULATION(k+1) is less than certain prespecified small amount, the algorithm stops; otherwise set k=k+1, and go to step 2.

1.3 Solving TSP using Genetic Algorithms

GA have been successfully applied in solving complicated problems where no effective methods are available to solve the problems exactly in a reasonable time span. Examples include optimization of gas pipeline [22], Blind knapsack problem [24], etc., see [25] [6]. Inspired by these successful applications, there have been a lot of efforts expended to applying GA for solving TSP. However, these attempts have all encountered a major difficulty, namely, the production of illegal offspring generated from the traditional crossover operation. This is illustrated as follows:

For a traveling salesman problem, an easy and natural representation of the chromosome of a route (an individual) is the sequence of cities in this route. For example, x_1= (3,1,2,5,4) and x_2= (2,5,3,4,1) give the sequences of cities in two solutions, which can be used as the chromosomes of the two solutions. Then, if a traditional crossover operation applies with a cutting point equal to 2, namely:

$$x_1 = (3,1 \mid 2,5,4)$$
$$x_2 = (2,5 \mid 3,4,1)$$

Then the offspring generated will be:

$$y_1 = (3,1,3,4,1)$$
$$y_2 = (2,5,2,5,4)$$

It can be seen that offspring y_1 misses cities 2 and 5 and repeats cities 1 and 3 while the other one y_2 misses cities 1 and 3 and repeats cities 2 and 5. Both of them are illegal (infeasible) tours.

Because the problem as shown above always exists in the traditional crossover operator, a number of special-purpose crossover operators have been invented for TSP. They are Partially mapped crossover operator [23], Order crossover operator [12], Cyclic crossover operator [52], Edge recombination crossover operator [65].

A famous crossover operator is partially mapped crossover, proposed by Goldberg and Lingle in 1985 [23]. In this crossover operator, a subsequence of a tour is selected by choosing two random cutting points. For instance, consider the following two parents x_1 and x_2 with two random cut points at positions 3 and 7:

$$x_1 = (1,2,3 \mid 4,5,6,7 \mid 8,9)$$
$$x_2 = (4,5,2 \mid 1,8,7,6 \mid 9,3)$$

The first step of PMX is to build up a swapping list from the elements between the cutting points. For the above parents, the swapping list will be $(1 \leftrightarrow 4, 8 \leftrightarrow 5, 7 \leftrightarrow 6, 6 \leftrightarrow 7)$. Then, we change the elements of the two parents based on this swapping list.

For instance, the first city of x_1 will change from 1 to 4 while the second city of x_1 is 2 remains unchanged because city 2 does not appear in the swapping list. Finally, two offspring y_1 and y_2 are generated as follows:

$$y_1 = (4,2,3 \mid 1,8,7,6 \mid 5,9)$$
$$y_2 = (1,8,2 \mid 4,5,6,7 \mid 9,3)$$

The above example shows that offsprings generated will inherit segments of chromosome between two cutting points from parents.

Although PMX [23] guarantees to generate legal offspring, our computational experiment on PMX shows that the results were not so satisfactory. In our experiment, a genetic algorithm with PMX, called GA(PMX), is used to solve four standard TSPs "Eilon50", "Eilon75", "Eilon100" [17], "oli30" [52]. The 'Time' is the computing time of GA(PMX) on a DEC Alpha workstation model AXP3800 - 200MHz.

Source	N	Optimal	GA(PMX)	% above Opt	Time (sec)
Oliver	30	420	536	27.6%	41
Eilon	50	425	643	51.3%	184
Eilon	75	538	1109	106.1%	562
Eilon	100	629	1846	193.5%	262

From the result shown above one can see that GA(PMX) is in fact still far from satisfaction. Similar conclusion may apply to other operators.

The main difficulty in crossover operations as stated above as well as the unsatisfactory performance of the existing operators constitute our motivation to explore new crossover operation schemes. This will be the main work of this thesis.

1.4 Outline of Work

My work in this thesis is primarily based on a new approach proposed by Cai in 1991 [8], with a new idea for doing crossover operation for TSP. The idea is to invent a kind of local dynamic programming (DP) procedure to explore the gene sets of

the parents so as to find the best offspring that the parents can deliver. This idea will be elaborated in details in Chapter 2 below.

My main work in this thesis can be grouped into two parts. In part I, my work is to analyze the approach of Cai, propose an enhanced scheme based on the analysis, develop a general-purpose TSP solver based on the enhanced scheme, and evaluate the performance of the Solver by computational experiments. I will show that the enhanced scheme can get around an unsolved problem that exists in Cai's original approach. I will report on the computational results that I have obtained in a series of computational experiments in evaluating the performance of our TSP solver based on the standard testing problems. As a result of our work, a new crossover operator, called LDPX (Local Dynamic Programming Crossover), has been developed. Moreover, a new genetic operator, which generates offspring based on the gene information of a single parent, called SPIR (single parent improved reproduction), has also been proposed. A new genetic algorithm based on SPIR and LDPX have been constructed. Experimental results have shown that the performance of our TSP solver is surprisingly good, which can obtain optimal or near-optimal solutions for all problems we have tested. Its performance is much better than those GA with other operators like Edge Recombination operator [65].

Part II of this thesis will focus on applying the TSP solver for a practical problem - Flowshop Scheduling with Travel Times Between Machines (FSTTBM). I will first describe the formulation of this problem, and show that this problem can be transformed into a TSP model. Algorithms for performing the transformation will be proposed. Then, we apply our TSP Solver to solve this problem. A series of computational experiments have also been carried out, the results of which show that our proposed method is very effective as compared with other heuristic methods.

Part I

Algorithm Development

Chapter 2

A Local DP Crossover Operator — LDPX

2.1 Review of DP for Solving TSP

In this chapter, a crossover operator based on DP will be developed. Let us review this idea of using DP to solve TSP first.

Dynamic programming (DP) is a multi-stage decision approach, which was invented by Bellman [4]. The basic idea of DP [4][16] [33] is to formulate the problem as a multi-stage decision problem and then determine the optimal decisions stage by stage. There are, in general, a number of states at each stage. Each state is associated with an optimal value which is evaluated based on the so-called principle of optimality.

DP has been applied to solve TSP, for example, [29]. The traditional approach in doing this is to separate an N-city TSP problem into an N-stage decision problem. First, we arbitrarily choose any one city as the last city visited for the decision problem. Without loss of generality, let us choose city N. At stage k, a state g is defined as a subset of k cities which are selected from cities $\{1, 2, ..., N-1\}$. For example, $g = (2, 3, 6)$ and $g = (2, 3, 6, 7)$ are states in stage 3 and stage 4 respectively. For each state g, an objective function $S(i, g)$ is defined as the optimal cost of the

tour starting from city $i \in g$, visiting all other cities in g once and only once and at last, visiting city N. In summary, we have the following set of states for each stage:

Stage 1:	$\{1\}$	$\{2\}$	$\{3\}$	$\ldots$	$\{N-1\}$
Stage 2:	$\{1,2\}$	$\{1,3\}$	$\{1,4\}$	$\ldots$	$\{N-2,N-1\}$
Stage 3:	$\{1,2,3\}$	$\{1,2,4\}$	$\{1,2,5\}$	$\ldots$	$\{N-3,N-2,N-1\}$
	$\vdots$				
Stage N-1:	$\{1,\ldots,N-1\}$				

At stage $N-1$, the decision problem is to find the optimal cost starting from every city via all other cities and ending at city N. At the last stage, stage N, the decision problem is to find the optimal cost starting from city N via all cities and ending at city N. The following is a recursive relationship for calculating the optimal cost, S^*.

Given N cities, and let $c(i,j)$ be the cost of traveling from city i to city j, we have:

$$
\begin{aligned}
S^*(i,\{i\}) &= c(i,N) && \text{, where } i = 1,2,...,N-1 \\
\forall g, S^*(i,g) &= \min_{u \in g-\{i\}} \{c(i,u) + S^*(u, g-\{i\})\} && \text{, where } i \in g \\
S^* &= \min_{u \in \{1,...,N\}} \{c(N,u) + S^*(u,\{1,2,...,N-1\})\}
\end{aligned}
$$

Since there are $\binom{N-1}{k}$ different subsets at stage k, the total number of states throughout the searching will be $\sum \binom{N-1}{k}$, where $k = 1,2,...,N-1$. It means that the total number of states to be evaluated using DP is $O(2^{N-1})$. It shows that the complexity of using DP to solve TSP will exponentially increase with respect to N, the number of cities.

The algorithm is as follows:

INPUT: Distance matrix of a TSP

DECLARATION:

$S^*(i,g)$:	The optimal cost for subset g.
S^* :	The overall optimal cost.
$BestPath(i,g)$:	The optimal solution of $S^*(i,g)$.
$BestTour$:	The overall optimal tour.

```
BEGIN
    FOR i = 1,2,...,N-1 DO
        BEGIN
            S*(i,{i}) = c(i,N)
            BestPath(i,{i}) = (i,N)
        END
    END FOR
    FOR j = 1,...,N-1 do
        FOR EACH g ⊆ {1,2,...,N-1} with |g| = j do
            FOR EACH i ∈ g
                BEGIN
                    S*(i,g) = min_{u∈g-{i}} {c(i,u) + S*(u, g-{i})}
                    Let u* be the city that achieves this minimum.
                    BestPath(i,g) = {i} concatenates BestPath(u*, g-{i})
                END
            END FOR
        END FOR
    END FOR
    S* = min_{1≤u≤N-1} {c(N,u) + S*(u,{1,2,...,N-1})}
    Let u* be the city that achieves this minimum.
    BestTour = BestPath(u*,{1,2,...,N-1})
    OUTPUT BestTour
END
```

The algorithm above, builds up the values of $S^*(i,g)$ from $|g| = 1$, $|g| = 2$, ... until the $S^*(i,\{1,2,\ldots,N-1\})$ are obtained for $i = 1,2,\ldots,N-1$. Finally, it searches for the optimal tour connecting the last city and the first city.

2.2 On the Original LDPX

Cai's original LDPX [8] consists of two basic parts, one is a scheme of defining the gene sets, while the other is a local DP to explore the gene sets to search for the best offspring.

2.2.1 Gene Representation

Cai argues that the basic genetic information donated by parents is the sub-tours of those parents. Therefore, a crossover operator must extract the best genes contained in these sub-tours to its offspring.

For example, if a parent x= (1,2,3,4,5) is considered, the genetic information describing this individual will be (1,2), (2,3), ..., (1,2,3,4,5). Each parent has its own sets of genes. By considering those sub-tours, we can form a set from each parent, which is termed *gene set*. In general, we define a gene set G of parent x_1=(i_1,i_2,...,i_n) as:

$$\left\{\begin{array}{lllll} (i_1,i_2) & (i_2,i_3) & (i_3,i_4) & \ldots & (i_{n-1},i_n) \\ (i_1,i_2,i_3) & (i_2,i_3,i_4) & \ldots & i_{n-2},i_{n-1},i_n) & \\ \vdots & & & & \\ (i_1,i_2,\ldots,i_n) & & & & \end{array}\right\} \quad (2.1)$$

For example, if x_1= (3,1,2,5,4) and x_2=(2,5,3,4,1) are considered, the corresponding gene sets will be defined as follows:

$$\left\{\begin{array}{llll} (3,1) & (1,2) & (2,5) & (5,4) \\ (3,1,2) & (1,2,5) & (2,5,4) & \\ (3,1,2,5) & (1,2,5,4) & & \\ (3,1,2,5,4) & & & \end{array}\right\}$$

and

$$\left\{\begin{array}{llll} (2,5) & (5,3) & (3,4) & (4,1) \\ (2,5,3) & (5,3,4) & (3,4,1) & \\ (2,5,3,4) & (5,3,4,1) & & \\ (2,5,3,4,1) & & & \end{array}\right\}$$

If offspring is generated from these gene sets during crossover, it will inherit the characteristic of its parents. That is the basic idea of the original LDPX.

2.2.2 The Original Crossover Procedure

In general, two parents will have two families of gene sets. The fundamental idea of Cai's original crossover procedure is to mix up these two families of gene sets into G, and then use a dynamic programming procedure to explore these sets so as to find the best offspring. Specifically, the gene sets are first classified into different groups in such a way that group 1 contains gene sets with 1 city, group 2 contains gene sets with 2 cities, ... and the last group contains gene sets with all cities. Then, a DP procedure is used to generate the best offspring from these gene sets. The procedure will find out the optimal cost of the objective function $S^*(i,g)$ for each gene $g \in G$ where $S^*(i,g)$ is the optimal cost traveling from city $i \in g$ via all other cities in g once and only once. At stage k the DP procedure limits its search to the gene sets with k cities only.

When $k = 1$, all $S^*(i,g)$ with $|g| = 1$ will be evaluated as 0. Therefore we have, $S^*(1,\{1\}) = 0$, $S^*(2,\{2\}) = 0$, ..., $S^*(n,\{n\}) = 0$. When $k = 2$, $S^*(i_1,\{i_1,i_2\})$ will be evaluated as $c(i_1,i_2) + S^*(i_2,\{i_2\})$ where $c(i_1,i_2)$ is the cost from city i_1 to city i_2. In general, a recursive relationship of the objective function $S^*(i,g)$ is defined as follows:

$$S^*(i,g) = \min \left\{ \begin{array}{c} c(i,i_1) + S^*(i_1, g - \{i\}) \\ c(i,i_2) + S^*(i_2, g - \{i\}) \\ \vdots \\ c(i,i_t) + S^*(i_t, g - \{i\}) \end{array} \right\}$$

However, $g - \{i\}$ does not always exist in the gene set G because it is only a subset of the whole combination. For a set not in G, we call it infeasible or illegal and its objective function will be set to be ∞. For example, for the x_1 and x_2 as given above, stage 2 of the DP procedure will consider the gene sets - { (3,1), (1,2), (2,5), (5,4), (5,3),(3,4), (4,1) }, ... and stage 5 will consider the gene sets { (3,1,2,5,4), (2,5,3,4,1) }. All other genes not in the gene sets are regarded as infeasible or illegal [8]. The following is an example to illustrate the idea in which $g = (2,5,3,4)$ is the gene set considered and $S(3,(2,5,3,4))$ is defined as the cost of visiting the cities in the set $(2,5,3,4)$ once and only once, subject to the condition that the starting city is city 3:

$$S(3,(2,5,3,4)) = \min \left\{ \begin{array}{c} c(3,2) + S(2,(2,5,4)) \\ c(3,5) + S(5,(2,5,4)) \\ c(3,4) + S(4,(2,5,4)) \end{array} \right\}$$

$S(5,(2,5,4))$ has been set to be ∞ because (2,4) is not an legal gene sets.

In summary, we can derive a recursive relationship as follows:

Let $g = (i_1, i_2, ..., i_t)$ be a gene set of cardinality t, and let $S^*(i,g)$ be the minimal cost of the tour which visits the t cities in g by starting from $i \in g$ subject to the condition that the remaining elements $g - \{i\}$ consist of a legal gene set as given by the parents. Then, we have:

$$S^*(i,g) = \min \left\{ \begin{array}{c} c(i,i_1) + S^*(i_1, g - \{i\}) \\ c(i,i_2) + S^*(i_2, g - \{i\}) \\ \vdots \\ c(i,i_t) + S^*(i_t, g - \{i\}) \end{array} \right\}$$

$$S^*(i,g) = \infty \text{ , if } g - \{i\} \text{ is infeasible.}$$

Using this recursive relationship, the solution procedure first moves from $t = 1, 2, .., $ until n when it finds the minimal overall cost, then moves from $t = n, n-1, ...,$ until 1 to find the optimal tour, namely, the best offspring that is given by the parents.

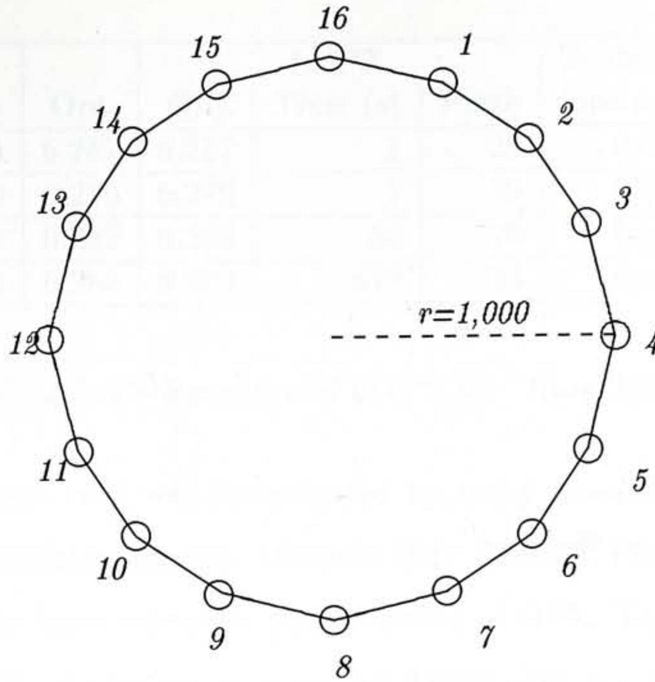
2.3 Analysis

The crossover procedure as described above is an innovative idea. Computational experiments have shown that this approach greatly outperformed Lin's 2-opt with multiple starts, see [8]. However, [8] only drafts the fundamental idea. A number of important issues remain to be resolved to make the approach an applicable one. One of these is that the original LDPX neglects the cost between the last city and the first city. This phenomenon was actually revealed when I compared the performance of the LDPX in solving a special case TSP - "Ring TSP" and other TSPs. Experimental results showed that it works fine for "Ring TSP", but it does not give satisfactory results for general TSP. This will be illustrated below.

The second major issue is on the algorithmic aspect of the original LDPX. It requires much working space to memorize each element at each stage in the dynamic programming procedure. Besides, much time is spent on searching whether a set is a legal gene set. In the following we will describe a genetic operator to solve these problems. This operator will generate an offspring based on the gene sets provided by a single parent.

2.3.1 Ring TSP

Ring TSP is a special TSP whose cities are located on the perimeter of a circle evenly. The cost from one city to other city is defined as the distance between those cities. An example with 16 cities is depicted below.



In general, the distance between two cities i and j is given below:

$$c(i,j) = 2r\sin(\frac{(j-i)\pi}{n}) \text{ , where } 1 \leq i < j \leq n$$

where n is the number of cities, and r is the radius of the circle. In our experiment, r is chosen to be 1,000. Without loss of generality, we define the cost between a pair of cities i and j as $c(i,j)$.

For any TSP with $n \geq 3$, the solution is a ring - $(1, 2, \cdots, n)$ and the minimal cost of an optimal tour is $2nr\sin(\frac{\pi}{n})$ which will close to 6,283, the perimeter, when n is very large.

2.3.2 Computational Results of Solving Ring TSP and Other TSP using LDPX

We obtained the following results in solving Ring TSP using a genetic algorithm with LDPX, where PSize is the size of the population in the genetic algorithm.

From Table 2.1 we can see that the LDPX has a very good performance in solving the TSP. Let us now apply it to solve other TSPs. The problem instances we used in our experiments came from a TSP Library [57]. The TSP Library (TSPLIB) collects

n	Opt.	Obj.	LDPX Time (s)	PSize	% above optimal
20	6,257	6,257	1	20	0.0%
50	6,279	6,279	7	20	0.0%
100	6,282	6,282	50	20	0.0%
200	6,283	6,283	579	20	0.0%

Table 2.1: Performance of LDPX for "Ring TSP"

some TSPs which have been well investigated by many researchers. In our experiments here, four problems, namely, Oliver30 [52], Eilon50, Eilon75, and Eilon100 [17], from this library were solved by the crossover - LDPX. The experiments were carried out on a DEC Alpha workstation AXP3800 with one 200MHz CPU. The results are summarized in Table 2.2.

			GA(LDPX)			Percentage
Problem Source	Size	Optimal	Obj.	Time (s)	PSize	of error
Oliver	30	420	424	12	20	0.95%
Christofides/Eilon	50	425	438	98	50	3.06%
Christofides/Eilon	75	538	567	610	100	5.39%
Christofides/Eilon	100	629	650	1,018	100	3.34%

Table 2.2: Performance of LDPX for TSP in TSPLIB

From Table 2.2 we can see that, although LDPX worked well in solving "Ring TSP" (the exact optimum solutions were obtained in every test), its performance was not so satisfactory in solving those TSPs from the TSP library.

The main reason is that the original LDPX neglects considering the cost between the last city and the first city. This makes the solution procedure focus on searching for a path rather than a cycle as required by TSP. This problem is not critical in the ring TSP since an optimal path in the ring TSP is always $(1, 2, \cdots, n)$ while the optimal cycle is also $(1, 2, \cdots, n)$. However, for a general TSP, the solution obtained will carry certain error, as shown in Table 2.2. This indicates that the original LDPX as described in 2.2 should be modified to account for the cost between the

last city and the first city of a TSP. To do this, in the following sub-sections we will first augment the gene set representation in Section 2.2.1 so as to account for information on the connection between the last city and the first city provided by the parents, and then introduce modification to the original LDPX to make use of the information.

2.4 Augmentation of the Gene Set Representation

Considering the gene sets of a parent $x_1 = (1,2,3,4,5)$:

$$\left\{ \begin{array}{llll} (1,2) & (2,3) & (3,4) & (4,5) \\ (1,2,3) & (2,3,4) & (3,4,5) & \\ (1,2,3,4) & (2,3,4,5) & & \\ (1,2,3,4,5) & & & \end{array} \right\}$$

Clearly, the gene information on the connection of the last city 5 and the first city 1 is not contained in these sets. In order to account for this information, we may augment the gene sets as follows:

$$\left\{ \begin{array}{lllll} (1,2) & (2,3) & (3,4) & (4,5) & \mathbf{(5,1)} \\ (1,2,3) & (2,3,4) & (3,4,5) & \mathbf{(4,5,1)} & \mathbf{(5,1,2)} \\ (1,2,3,4) & (2,3,4,5) & \mathbf{(3,4,5,1)} & \mathbf{(4,5,1,2)} & \mathbf{(5,1,2,3)} \\ (1,2,3,4,5) & \mathbf{(2,3,4,5,1)} & \mathbf{(3,4,5,1,2)} & \mathbf{(4,5,1,2,3)} & \mathbf{(5,1,2,3,4)} \end{array} \right\}$$

In general, we have a new definition of gene sets for the parent $x_1=(i_1,i_2,\ldots,i_n)$:

$$\left\{ \begin{array}{lllll} (i_1,i_2) & (i_2,i_3) & (i_3,i_4) & \ldots & (i_n,i_1) \\ (i_1,i_2,i_3) & (i_2,i_3,i_4) & \ldots & & i_n,i_1,i_2) \\ \vdots & & & & \\ (i_1,i_2,\ldots,i_n) & \ldots & & & (i_n,i_1,\ldots,i_{n-1}) \end{array} \right\} \tag{2.2}$$

2.5 Enhancement of Crossover Procedure

There are a number of possible options to modify the original LDPX so that it can account for the connection between the last city and the first city. We suggest to use the following one:

First, select randomly a reference city i_{end}. Then, define $S^*(i,g)$ as the minimal cost of visiting $g = \{i_1, i_2, \ldots, i_t\}$ subject to the condition that the tour starts from $i \in g$, visits all other cities in $g - \{i\}$ once and only once, and ends at city i_{end}. A recursive relationship for calculating S^* is given below:

$$\left\{ \begin{array}{ll} S^*(i,g) = \min \left\{ \begin{array}{l} c(i,i_1) + S(i_1, g - \{i\}) \\ c(i,i_2) + S(i_2, g - \{i\}) \\ \vdots \\ c(i,i_t) + S(i_t, g - \{i\}) \end{array} \right\} & \text{,if } i \neq i_{end} \\ S^*(i,g) = \infty & \text{,if } i = i_{end} \text{ or } g - \{i\} \text{ is an infeasible gene set. (Note the feasibility is based on the augmentation of the gene sets as defined in section 2.4)} \end{array} \right\}$$

The minimal overall cost will be:

$$S^* = \min_{i \in G_n} \{c(i_{end}, i) + S(i, G_n)\} \quad ,$$

where G_n denotes the gene set with cardinality equal to n.

Without ambiguity, from now on we will call the approach above LDPX. According to the principle of optimality of dynamic programming [16] [4], the offspring generated by this LDPX approach will be the best one that can be delivered from the parents subject to the condition that all the subsets g are the gene sets defined by the parents.

2.6 Computational Comparison of the new proposed LDPX with the original LDPX

We have incorporated, respectively, the new LDPX proposed above and the original LDPX into a genetic algorithm. The following computational results show the performance of the genetic algorithm with the new LDPX and the genetic algorithm with the original LDPX, which were obtained by solving two problems from Christofides and Eilon [17] with numbers of cities equal to 50 and 75 respectively. The population size was chosen to be 100.

			Original		The New LDPX	
Problem Source	n	Opt.	Obj.	Time	Obj.	Time (s)
Christofides/Eilon	50	425	438	98 s	425	96 s
Christofides/Eilon	75	538	567	610 s	538	569 s

From the results we can see that the GA with the new LDPX obtained the exact optimal solutions in both of the problem instances (which are given in [17]). Compared with the new GA, the GA with the original LDPX which, however, failed to find the optimal solutions, needed almost the same computing time.

2.7 SPIR – An Operator for Single Parent Improved Reproduction

The LDPX described above uses the gene sets provided by a couple of parents. From our computational experiments we found that most of the time required by the genetic algorithm with this crossover operator was spent on validating whether a sub tour was a legal gene set, particularly at the early stage of the evolution of the genetic algorithm. In fact, at the early stage of the evolution when individuals in the population have not been very fit, it is still unnecessary to cross over the genes of a couple of parents. In other words, at this stage, even a single individual has much room for improvement and it is unnecessary to spend time to consider the crossover

of the genes of two parents. This is our motivation to propose the operator SPIR, which generates an offspring based on the genes provided by a single parent.

SPIR is similar to LDPX, except that it only explores the gene sets defined by a single parent. In the case of single parent, by letting $t \in \{2, 3, \cdots, n\}$ denote the cardinality of a gene set and s denote the starting position, we can define $g(t, s)$ as the gene set that contains t elements starting from the position s. For example, assuming that $x_1 = (7, 1, 6, 2, 4, 5, 3)$ is the single parent under consideration, we have

$$
\begin{aligned}
g(2,1) &= (7,1) \\
g(3,6) &= (5,3,7) \\
g(4,6) &= (5,3,7,1) \\
&\vdots \\
&etc.
\end{aligned}
$$

With this indexing scheme to represent the gene sets, the procedure to check whether a gene is legal or not is easy. Let the parent be $(p(1), p(2), ..., p(j), ...p(n))$, where $p(j)$ is the city at position j. Then, the genes $g(t, s)$, $g(t-1, s)$ and $g(t-1, s+1)$ given by the parent are as follows:

Starting position $= s$

$$g(t-1, s) = \left(p(s) \quad p(s+1) \quad \cdots \quad p(s+t-2) \right)$$

$$g(t, s) = \left(p(s) \quad p(s+1) \quad \cdots \quad p(s+t-1) \right)$$

Starting position $= s+1$

$$g(t-1, s+1) = \left(p(s+1) \quad p(s+2) \quad \cdots \quad p(s+t-1) \right)$$

It can be seen from the above that $g(t,s) - \{p(j)\}$ is a legal gene set if and only if $j = s$ or $j = s + t - 1$. Any other j will lead to an illegal gene set. That is:

$$\begin{aligned}
g(t,s) - \{p(s)\} &= g(t-1, s+1) \\
g(t,s) - \{p(s+1)\} &= \text{Infeasible} \\
&\vdots \\
g(t,s) - \{p(s+t-2)\} &= \text{Infeasible} \\
g(t,s) - \{p(s+t-1)\} &= g(t-1, s)
\end{aligned}$$

In summary, the recursive relationship for the SPIR operator, where $S^*(j, g(t,s))$ denotes the minimal cost of visiting the cities in the set $g(t,s)$ by starting at the city $p(j)$ and ending at a city i_{end} (see Section 2.5) is as follows:

At stage t=1, For $1 \leq i \leq N$ and $p(i) \neq i_{end}$,

$S^*(i, g(1,i)) = c(p(i), i_{end})$

At stage t={2,...,N},

For $j = s$,

$$S^*(p(j), g(t,s)) = \min \left\{ \begin{array}{l} c(p(s), p(s+1)) + S^*(s+1, g(t-1, s+1)) \\ c(p(s), p(s+t-1)) + S^*(s+t-1, g(t-1, s+1)) \end{array} \right\}$$

For $j = s + t - 1$,

$$S^*(j, g(t,s)) = \min \left\{ \begin{array}{l} c(p(s+t-1), p(s)) + S^*(s, g(t-1, s)) \\ c(p(s+t-1), p(s+t-2)) + S^*(s+t-2, g(t-1, s)) \end{array} \right\}$$

For $s < j < s + t - 1$, $S^*(j, g(t,s)) = \infty$

The optimal cost will be:

$$S^* = \min_{1 \leq j \leq N} \left\{ \min_{i \in g(N,j)} \{c(i_{end}, i) + S(i, g(N,j))\} \right\}$$

Chapter 3

A New TSP Solver

We develop, in this chapter, a new GA for solving TSP which incorporates the operators LDPX and SPIR. The new GA uses the strategy that SPIR is activated at its early stage while LDPX is used at its late stage. The motivation is that at the late stage of the evolution, individuals have been so fit that an improvement from just a single parent is hard to obtain. The overall architecture of the new TSP Solver is depicted below:

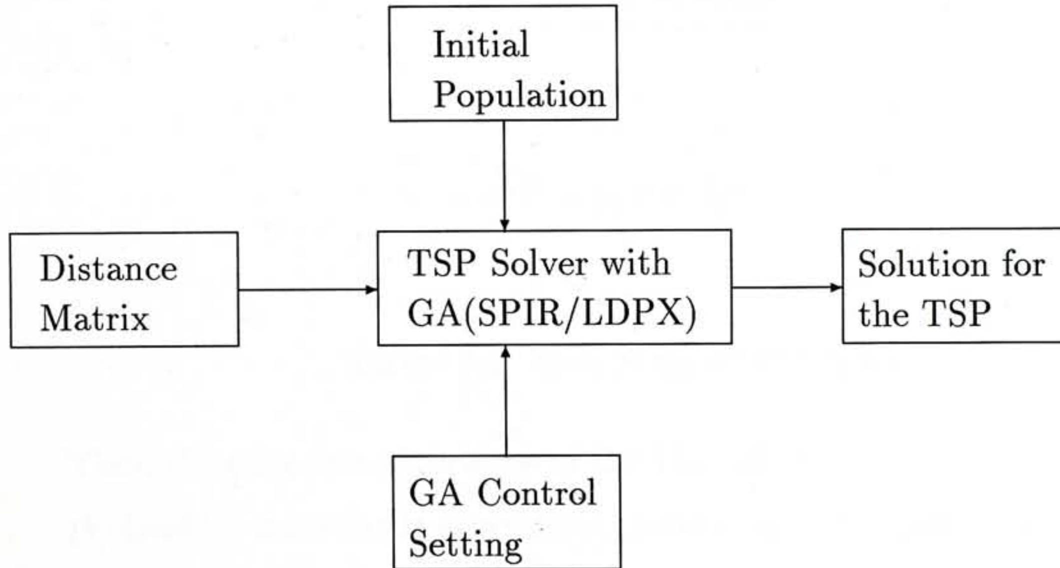


Figure 3-1: Input and Output of TSP Solver

Specifically, the TSP Solver takes in three sets of data: (1) an initial population which is randomly generated, (2) a cost matrix which stores the costs between each

pair of cities, and (3) a set of GA control parameters such as population size, the number of individuals selected to perform reproduction, terminating condition, etc. The main steps of the TSP Solver are illustrated below where δ_t is a small number:

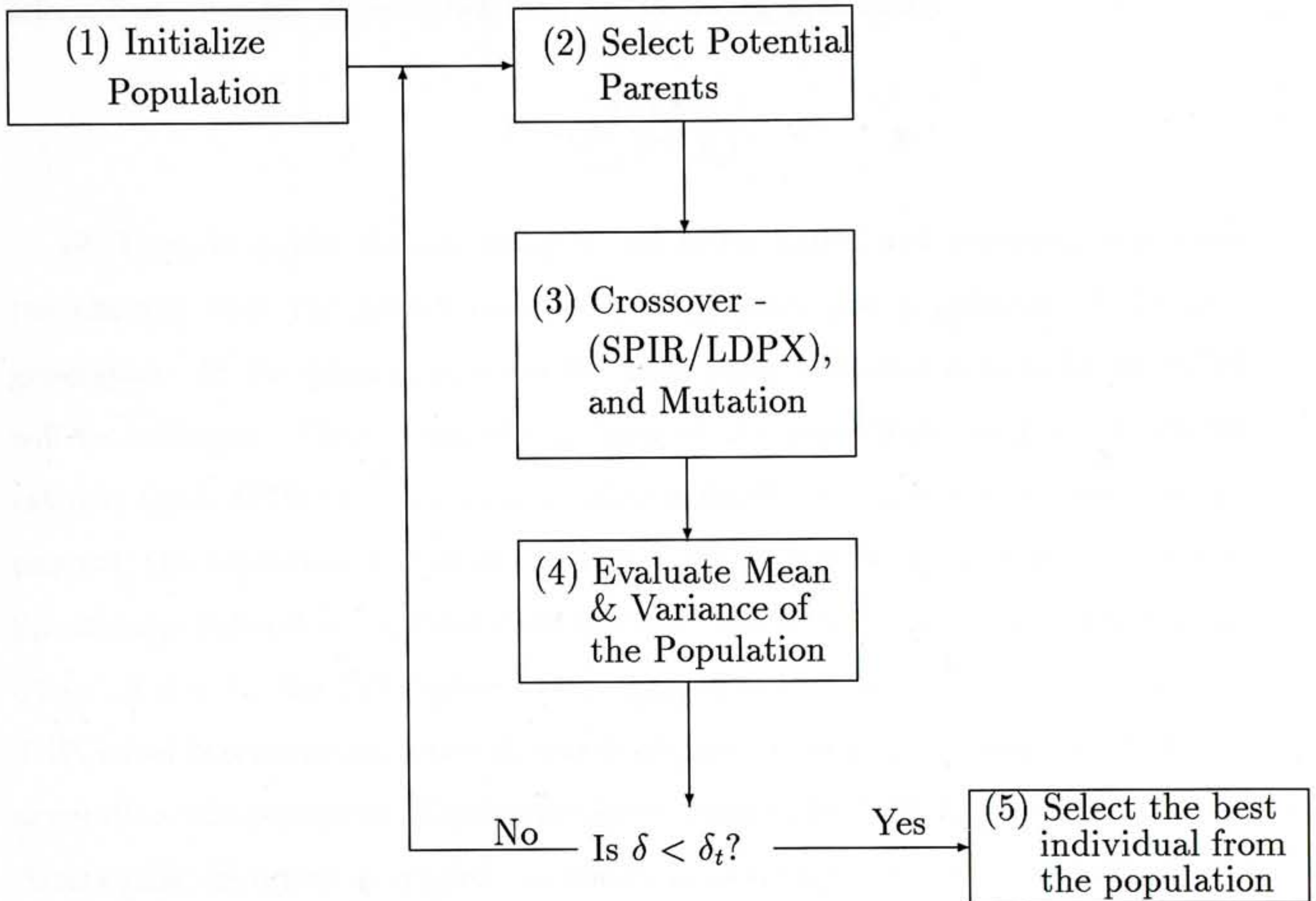


Figure 3-2: Main Steps of TSP Solver

Thus, there are four main steps in the TSP solver:

(1) Initially, it randomly generates a number (equal to a pre-specified population size $PSIZE$) of feasible tours to form the population of the first generation. The cost of each and every individual is evaluated.

(2) At each generation, it randomly selects a number of individuals to form a parent pool. The selection is biased towards the individuals which are fitter (with lower total cost). Specifically, let $F(X_i)$ be the cost of an individual X_i, and let $F_{\max}$ be the maximum of $F(X_j)$, $j = 1, 2, ..., PSIZE$, in the population. The raw

fitness value of the individual X_i is defined as $F'(X_i) = F_{\max} - F(X_i) + 1$, and the scaled fitness value is defined as (here we adopt the scaling scheme of [25]) $F''(X_i) = aF'(X_i) + b$, where the coefficients a and b are chosen in such a way that the mean value of the scaled fitness $\overline{F''}$ is equal to the mean value of the raw fitness $\overline{F'}$ and that maximum value of the scaled fitness $\max_i\{F''(X_i)\}$ is equal to $C_{mult} \times \overline{F'}$, where C_{mult} is equal to 2 (suggested by [25]). An individual X_i will be selected to enter the parent pool with the following probability:

$$P_i = \frac{F''(X_i)}{\sum_i F''(X_i)}.$$

(3) Then, it applies the genetic operators SPIR/LDPX and mutation to produce the offpring from the parent pool so as to generate the population of the next generation. At the beginning, when the population is randomly distributed, SPIR will be activated. Then, when the variance of the population reaches a threshold (at that time, SPIR has been hard to make a significant improvement over a single parent), the algorithm will switch to LDPX which will be used until convergence. Specifically, define δ as the ratio of standard deviation to the mean of the population. Then , if $\delta < \delta_s$, the TSP Solver switches from SPIR to LDPX, and if $\delta < \delta_t$, the TSP Solver is terminated, where δ_s and δ_t are given control parameters. During each generation, the number of offspring produced using SPIR/LDPX is equal to $PSIZE$. Afterwards, mutation is applied. It selects individual from the new offspring, and then exchanges randomly the positions of two cities. For example, if cities 2 and 5 are selected, then:

$$(1, \mathbf{2}, 3, 4, \mathbf{5}) \rightarrow (1, \mathbf{5}, 3, 4, \mathbf{2})$$

In TSP Solver, each new offspring should have a probability to perform mutation and the probability is known as *mutation probability*. Usually, this probability is set to be very small such as 0.01 in our computation test in next chapter.

(4) The value of δ will be evaluated at every generation. The evolution continues until δ of the current population has been smaller than δ_t.

The above TSP Solver has been developed as a software package using C. It can now solve any TSP problem, given the problem parameters. The software package has been extensively tested using numerous standard testing problems of varying size from a well-known TSP library. The computational experiments have shown that our Solver has an excellent performance in solving various problems. The computational results are reported in the next Chapter.

Chapter 4

Performance Analysis of the TSP Solver

To evaluate the performance of our TSP Solver, a large number of TSPs of varying size have been solved. In the following sections we shall report our results obtained in solving those standard problems from a TSP library TSPLIB [57]. Since the best solutions for these problems have also been provided in the Library, the evaluation of the quality of the solutions obtained by our Solver can be based on these known solutions. In addition to the comparison with these known results, our Solver were also compared with other approaches. One is a GA with a crossover operator called edge recombination (ER) [65], which was chosen because it was shown to be the most efficient operator in solving TSPs and thus have been adopted by many researchers, (see, for example, Genitor [64] and Tolkien [2]). Another approach that was used to compare with our Solver uses PMX (partially mapped crossover) [23] as the crossover operator. In our computational experiments, all the approaches, including our Solver, were applied to solve the same problem with the same parameters under the same conditions (such as the stopping condition). All the experiments were carried out in a DEC workstation AXP3800 with one 200MHz CPU. The settings of the parameters in the our GA, which have been tested and adjusted so that our TSP Solver will obtain good results, are given in Table 4.1.

Problem Name	Pop. Size	Parent Pool Size SPIR	LDPX	Mutation probability	δ_s	δ_t
oli30	20	16	16	1.0%	5.0	0.02
eil50	20	16	16	1.0%	5.0	0.02
eil75	20	16	16	1.0%	2.0	0.15
eil100	20	16	16	1.0%	2.0	0.10
lin105	20	16	16	1.0%	5.0	4.0
kroA100	20	16	16	1.0%	10.0	6.0
kroC100	20	16	16	1.0%	2.0	1.8
kroD100	20	16	16	1.0%	2.0	0.7
kroA150	20	16	16	1.0%	5.0	3.5
kroB150	20	16	16	1.0%	5.0	1.1
kroA200	20	16	16	1.0%	5.0	1.3
kroB200	20	16	16	1.0%	5.0	3.2
lin318	20	16	16	1.0%	3.0	2.0
pr439	20	16	16	1.0%	20.0	15.0

Table 4.1: GA(SPIR/LDPX) Parameters Setting

4.1 Computational results

We now report the results in comparing the solutions obtained by our Solver with the best known solutions. In the Table 4.2, all the problems and their corresponding optimal solutions were extracted from TSPLIB, in which eil50, eil75 and eil100 came from Eilon [17], oli30 from [52], kroA100, kroA150, kroA200, kroB150, kroB200, kroC100, kroD100 from [41], lin105, lin318 from [45], and pr439 from [53].

From the computational results in Table 4.2 we can see that the TSP solver obtained, in reasonable time, optimal or near-optimal solutions for all the problems tested. Note that the parallel structure of the GA in the TSP Solver has not been utilized. It is expected that the computing time required by the GA can be reduced substantially if it is implemented in parallel.

Problem Name	Size	Optimum	SPIR/ LDPX	Time (min)	% above optimum
Oliver et. al. - oli30	30	420	420	0.1	0.0 %
Christofides/Eilon - eil50	50	425	425	0.7	0.0 %
Christofides/Eilon - eil75	75	538	538	1.0	0.0 %
Christofides/Eilon - eil100	100	629	629	4.8	0.0 %
Krolak/Felts/Nelson - kroA100	100	21,282	21,282	3.3	0.0 %
Krolak/Felts/Nelson - kroC100	100	20,749	20,749	3.2	0.0 %
Krolak/Felts/Nelson - kroD100	100	21,294	21,294	5.0	0.0 %
Lin/Kernighan - lin105	105	14,379	14,379	5.5	0.0 %
Krolak/Felts/Nelson - kroA150	150	26,524	26,528	9.5	0.0 %
Krolak/Felts/Nelson - kroB150	150	26,130	26,188	13.6	0.2 %
Krolak/Felts/Nelson - kroA200	200	29,368	29,503	44	0.5 %
Krolak/Felts/Nelson - kroB200	200	29,437	29,446	66	0.0 %
Lin/Kernighan - lin318	318	42,029	42,446	223	1.0 %
Padberg/Rinaldi - pr439	439	107,217	107,868	560	0.6 %

Table 4.2: Computational results of GA(SPIR/LDPX)

4.2 Comparison between SPIR/LDPX, PMX and ER

In this section, we compare our Solver with two GAs, one with the crossover operators ER [65], and the other with the crossover operator PMX [23]. In Table 4.3, 'Time' is the execution time in minutes while 'Obj.' is the objective function values obtained by using different crossover operators.

From the results given in Table 4.3 we can see that ER and SPIR/LPDX gave better results than those obtained by PMX. In order to compare the performance of ER and SPIR/LDPX, the results obtained by them are further compared with the optimal solutions, see Table 4.4.

From Table 4.4 it can be seen that, for small size TSPs such as Oliver's 30 cities TSP, both ER and LDPX were able to find the exact optimal solutions. However, for complex problems of large size, ER failed to obtain good solutions. An example is Lin's 318-city TSP. The solution obtained by ER is 59.5% above the optimum for this problem. For almost all the problems tested (except the last one), our Solver with

Problem Name (n)	Optimal	PMX		ER		SPIR/LDPX	
		Obj.	Time	Obj.	Time	Obj.	Time
Oliver et. al. (30)	420	536	0.7	420	0.2	420	0.1
Christofides/Eilon (50)	425	643	3.1	430	0.9	425	0.7
Christofides/Eilon (75)	538	1,109	9.4	544	2.3	538	0.7
Christofides/Eilon (100)	629	1,846	4.4	659	4.5	629	4.8
Krolak/Felts/Nelson A(100)	21,282	52,398	84.9	23,962	4.1	21,282	2.3
Krolak/Felts/Nelson C(100)	20,749	43,485	96.6	21,498	4.6	20,749	3.2
Krolak/Felts/Nelson D(100)	21,294	56,545	28.8	21,714	4.5	21,294	5.0
Lin/Kernighan (105)	14,379	45,430	29.0	14,626	5.0	14,379	3.0
Lin/Kernighan (318)	42,029	233,176	529	67,024	64	42,446	223

Table 4.3: Comparison of PMX, ER and SPIR/LDPX

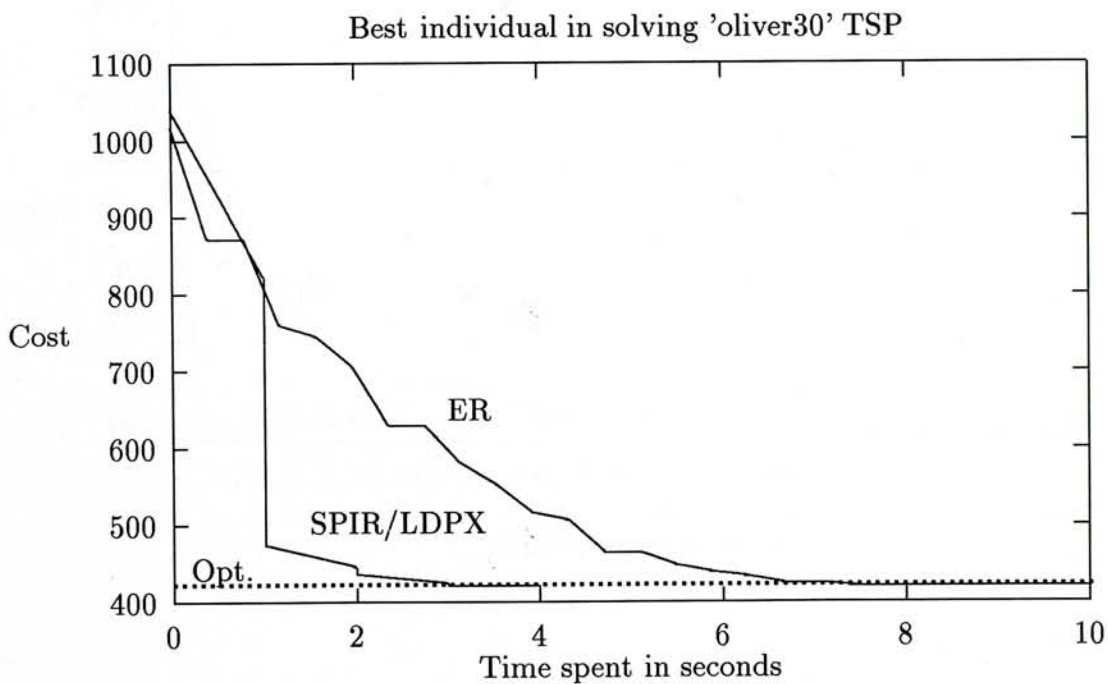
Problem Name (n)	Optimal	ER Obj.	% above optimum	SPIR/LDPX Obj.	% above optimum
Oliver et. al. (30)	420	420	0.0 %	420	0.0 %
Christofides/Eilon (50)	425	430	1.2 %	425	0.0 %
Christofides/Eilon (75)	538	544	1.1 %	538	0.0 %
Christofides/Eilon (100)	629	659	4.8 %	629	0.0 %
Krolak/Felts/Nelson A(100)	21,282	23,962	12.6 %	21,282	0.0 %
Krolak/Felts/Nelson C(100)	20,749	21,498	3.6 %	20,749	0.0 %
Krolak/Felts/Nelson D(100)	21,294	21,714	1.9 %	21,294	0.0 %
Lin/Kernighan (105)	14,379	14,626	1.7 %	14,379	0.0 %
Lin/Kernighan (318)	42,029	67,024	59.5 %	42,446	1.0 %

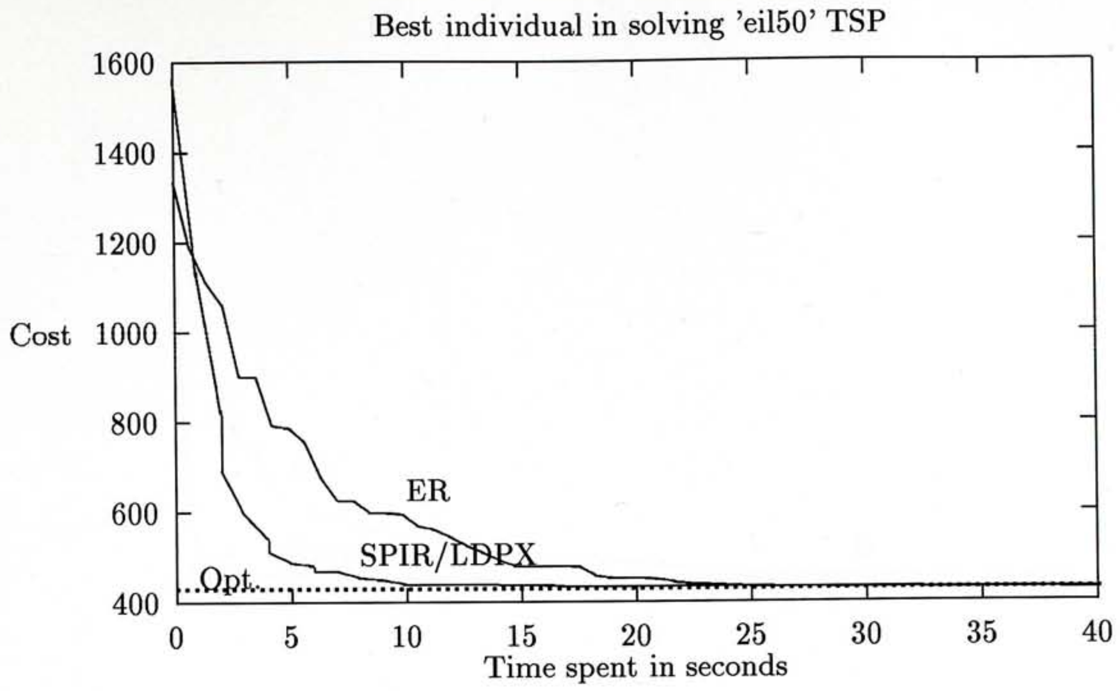
Table 4.4: % above optimum for ER and SPIR/LDPX

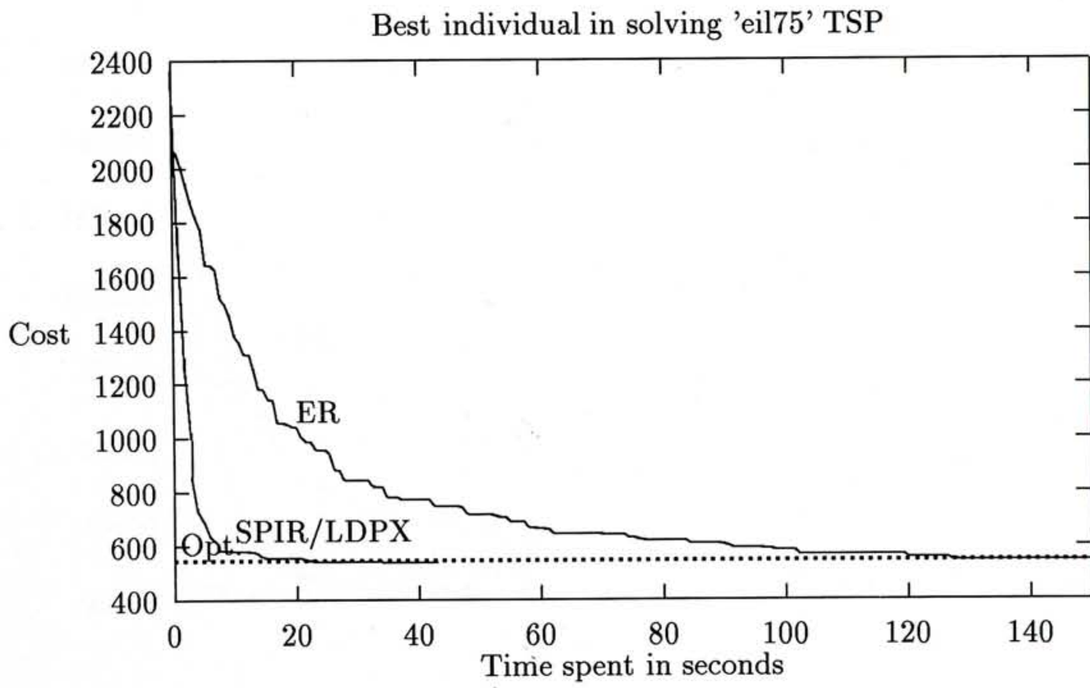
the SPIR/LDPX consistently delivered the exact optimal solutions. Even for the one (Lin's 318-city) for which it failed to find the optimum, the solution it derived is quite near the optimum. These results indicate that SPIR/LDPX outperforms ER, particularly for large TSPs. Since ER has been shown to be a very effective crossover operator for solving TSP, the computational results suggest that our SPIR/LDPX are very powerful operators.

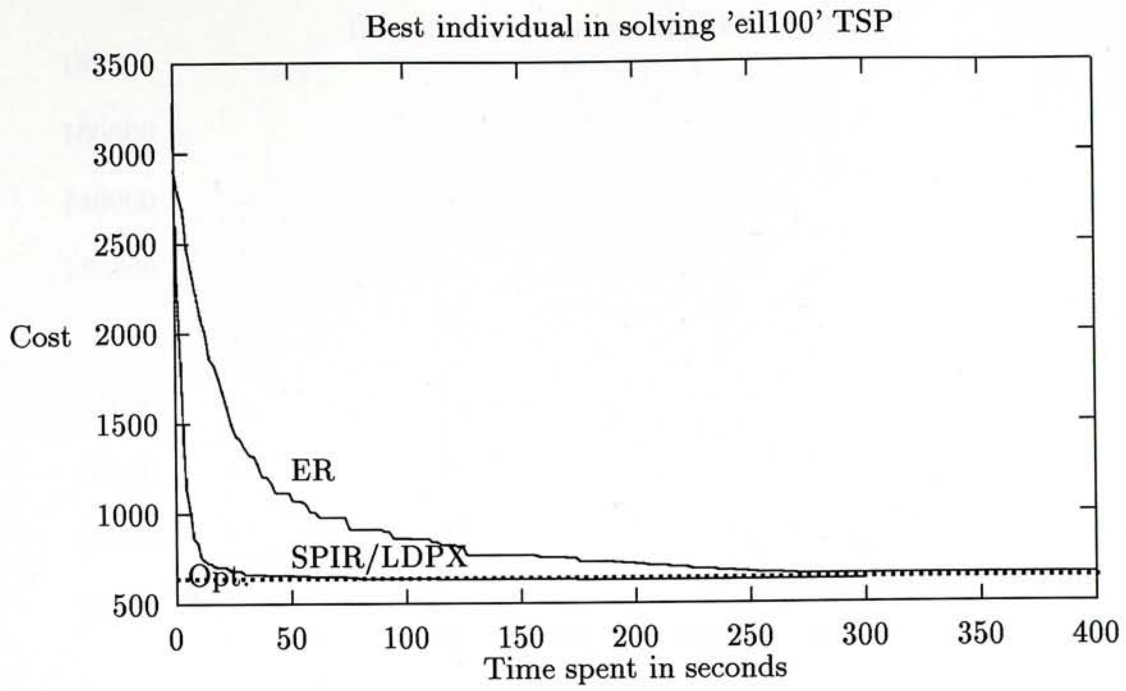
4.3 Convergence Test of SPIR/LDPX

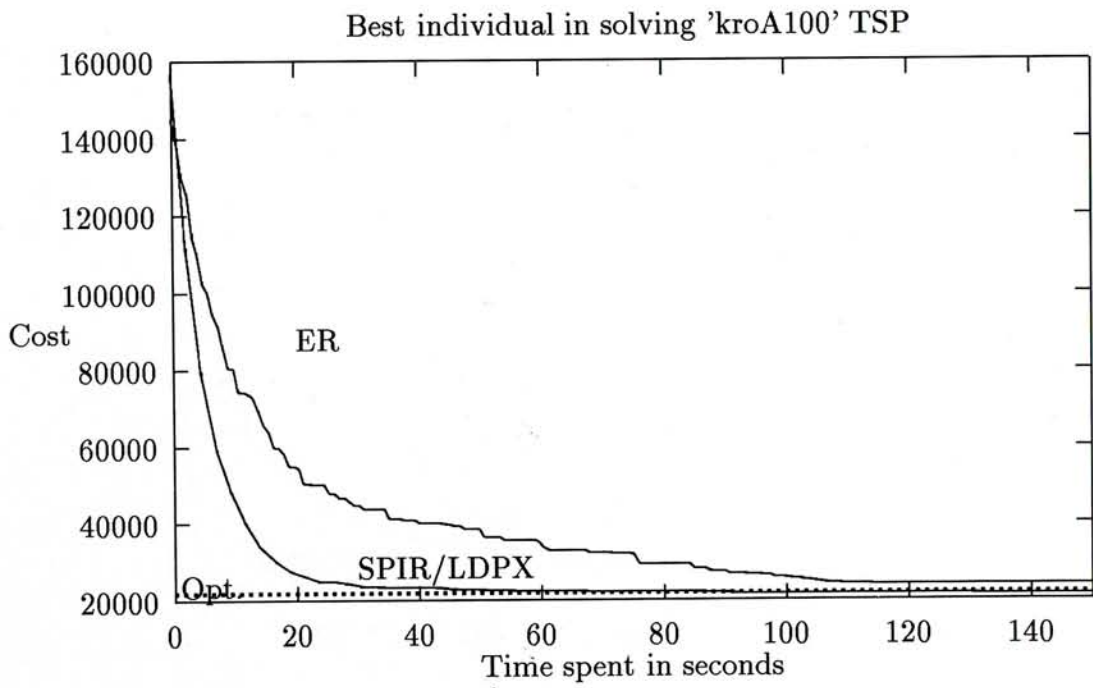
Table 4.3 and 4.4 summarise the final results. To demonstrate the real convergence rate of our Solver, we have recorded some data during the convergence procedure in solving the testing problems. These results are displayed in the graphs below, in comparison with ER. In these graphs, the x-axis represents the computing time (in seconds) while y-axis represents the lowest cost in the population. The dotted lines represent the optimal solutions.

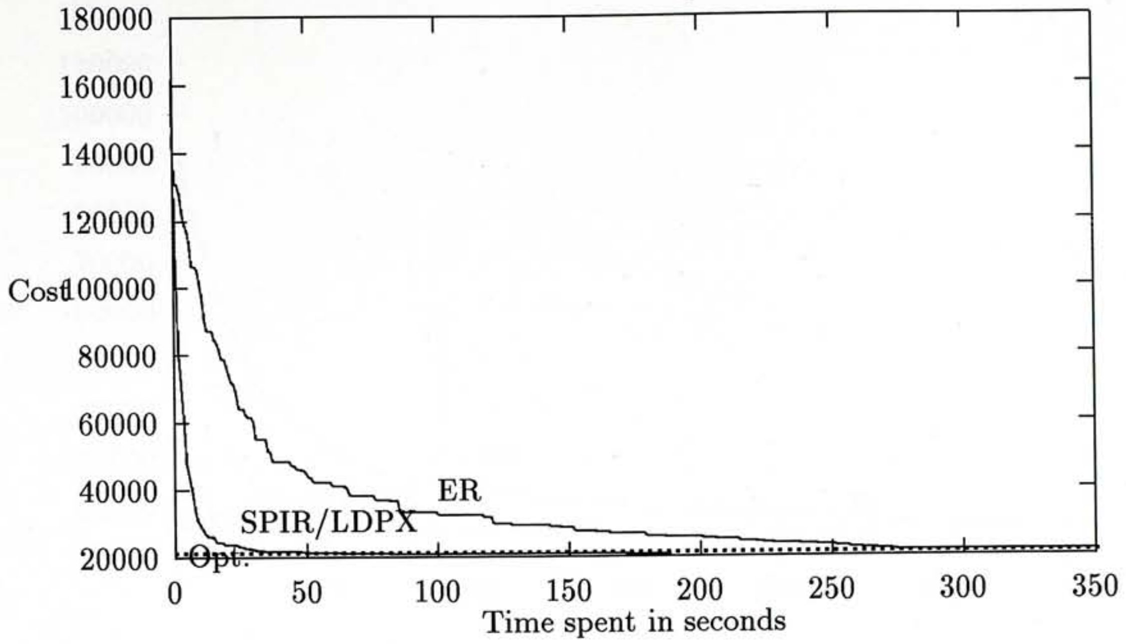


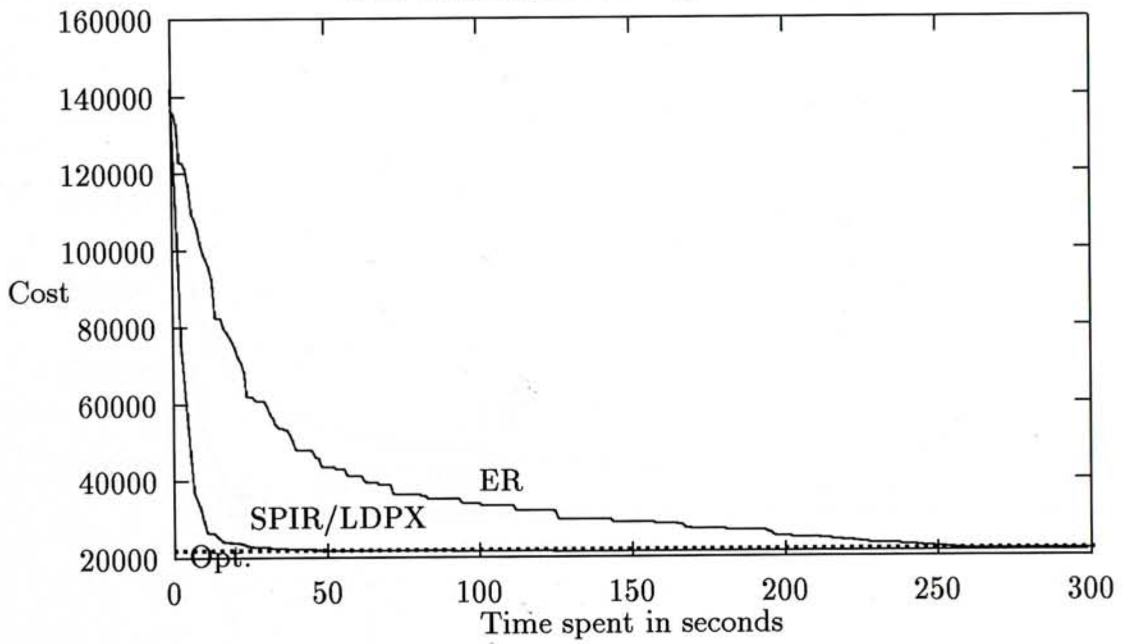

Best individual in solving 'eil50' TSP
1600
1400
1200
Cost 1000
800
600
400
ER
SPIR/LDPX
Opt.
0
5
10
15
20
25
30
35
40
Time spent in seconds

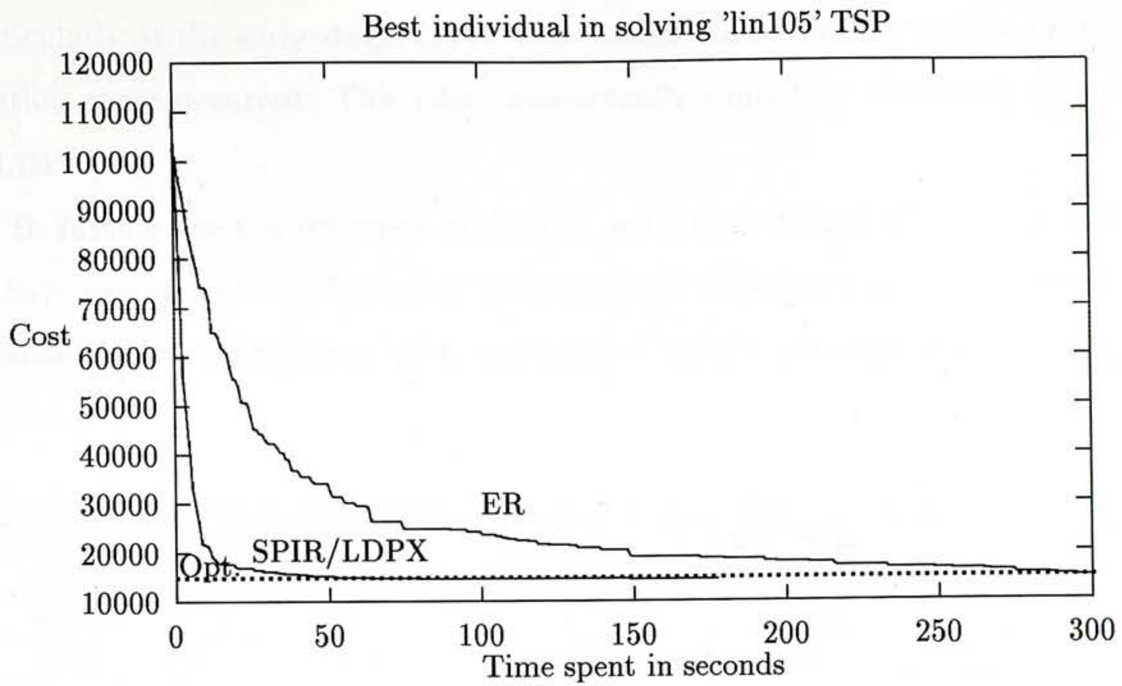

Best individual in solving 'eil75' TSP
2400
2200
2000
1800
1600
Cost 1400
1200
1000
800
600
400
ER
SPIR/LDPX
Opt.
0
20
40
60
80
100
120
140
Time spent in seconds

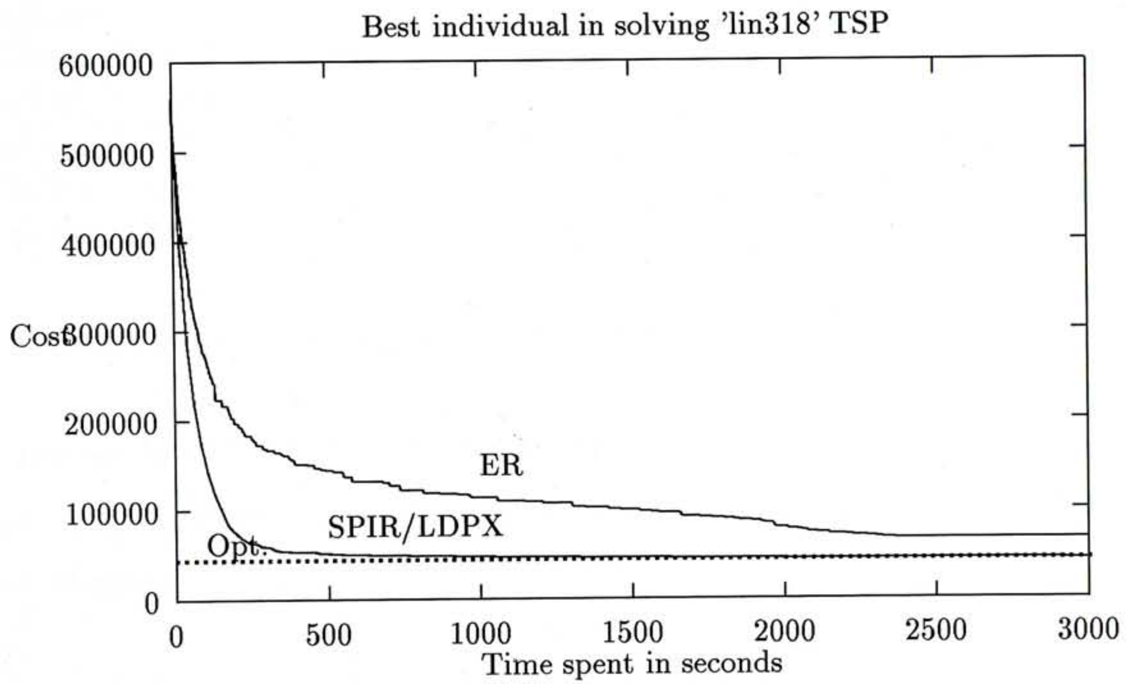

Best individual in solving 'eil100' TSP
3500
3000
2500
Cost 2000
1500
1000
500
ER
SPIR/LDPX
Opt.
0
50
100
150
200
250
300
350
400
Time spent in seconds


Best individual in solving 'kroA100' TSP
160000
140000
120000
100000
Cost
80000
60000
40000
20000
ER
SPIR/LDPX
Opt.
0
20
40
60
80
100
120
140
Time spent in seconds


Best individual in solving 'kroC100' TSP
180000
160000
140000
120000
Cost100000
80000
60000
40000
20000
ER
SPIR/LDPX
Opt.
0
50
100
150
200
250
300
350
Time spent in seconds


Best individual in solving 'kroD100' TSP
160000
140000
120000
100000
Cost
80000
60000
40000
20000
ER
SPIR/LDPX
Opt.
0
50
100
150
200
250
300
Time spent in seconds





From all these graphs, one can see that our SPIR/LDPX is much faster than ER. Moreover, two observations can be made: (1) SPIR/LDPX converged quickly,

particularly at the early stage; (2) In some cases, like oliver30, a sudden drop in the solution curve occurred. This effect was actually caused by the switch from SPIR to LDPX.

To further see the influence of the GA with and without the operator LDPX, we have carried out another set of computational experiments, in which the results obtained by the TSP Solver with and without LDPX are compared, see Table 4.5 below.

Problem	Size	Optimum	SPIR/LDPX	Time (min)	SPIR	Time (min)
oli30	30	420	420	0.1	420	0.1
eil50	50	425	425	0.7	425	1.3
eil75	75	538	538	1.0	539	1.1
eil100	100	629	629	4.8	630	4.4
kroA100	100	21,282	21,282	3.3	21,292	2.9
kroC100	100	20,749	20,749	3.2	20,769	3.3
kroD100	100	21,294	21,294	5.0	21,309	2.0
lin105	105	14,379	14,379	5.5	14,379	6.7
kroA150	150	26,524	26,528	9.5	26,528	14.1
kroB150	150	26,130	26,188	13.6	26,296	12.4
kroA200	200	29,368	29,503	44	29,715	47
kroB200	200	29,437	29,446	66	29,800	33
lin318	318	42,029	42,446	223	42,516	188
pr439	439	107,217	107,868	560	108,988	149

Table 4.5: Comparison of results from GA(SPIR/LDPX) and GA(SPIR)

The results in Table 4.5 indicate that the operator LDPX did improve the solutions. This shows that crossover of genes provided by more parents does generate fitter offspring.

Part II

Application

Chapter 5

Flowshop Scheduling Problem

5.1 Brief Review of the Flowshop Scheduling Problem

Suppose that n jobs are to be processed on m machines: M_1, M_2,...,M_m, in order of $M_1 \to M_2 \to ... \to M_m$. The processing time of job i on machine j is known to be $p_j(i)$, $i = 1, ..., n$, $j = 1, ..., m$. The problem of flow shop scheduling (FSP) is to find a sequence to process the jobs on each machine so as to minimize the overall completion time [11].

Johnson [35] formulated the initial FSP model in 1954. Since then, FSP has been investigated by many researchers [67] [34] [56] [66] [3] [7]. The general FSP with $m \geq 3$ has been shown to be NP-hard in the strong sense. Since there are no efficient methods available which can find exact optimal solutions for this difficult problem, a wide variety of algorithms, including heuristics, have been suggested to find approximate solutions. (For a comprehensive review, see [7] [66]). In the following we will consider a generalized version of the FSP, which has been formulated in the recent years motivated by some problems arising in manufacturing systems.

5.2 Flowshop Scheduling with travel times between machines

A generalized model of FSP has been investigated in recent years [1] [48] [55], which considers the situations where, for each $j = 1, 2, ..., m-1$, there is a transporter T_j between machines M_j and M_{j+1}, which picks up a completed job from machine M_j, travels to machine M_{j+1} in time t_j, unloads the job and then returns to machine M_j in time r_j. Machine M_j will be blocked by a completed job unless the transporter is available to remove the job from the machine. The problem is also to determine a sequence π to process the n jobs on each machines so as to minimize the makespan F_{max} which is defined as the overall completion time of all jobs.

As usual, we consider the problem under the following assumptions:

1. All jobs are available to process at time zero;

2. A job, once started, may not be interrupted;

3. Each machine can process at most one job at any time;

4. Set-up time for a job has been included in its processing time;

5. Travel times t_j and r_j are job independent;

6. Loading and unloading times of a transporter have been included in its travel time t_j;

7. The sequences to process the jobs on all machines are same; and

8. There is no intermediate storage space available to hold partially completed jobs.

For convenience, let us call the model described above FSTTBM. When only two machines are involved, FSTTBM has been shown solvable by some efficient

procedures [1] [48] [55]. However, the problem in general is NP-hard in the strong sense [60]. This is understandable since the problem without travel times between machines has been NP-hard in the strong sense.

Chapter 6

A New Approach to Solve FSTTBM

We have described the model of FSTTBM in Chapter 5. In this Chapter, we propose a new approach to solve FSTTBM which makes use of our TSP Solver described in part I. The new approach involves two steps. The first step is to transform FSTTBM into a TSP model while the next step is to use our TSP Solver to find a solution. Specifically, we will address a problem where a job, once started on machine 1, must be processed by all machine without any idle time. The problem is called continuous processing FSTTBM or CPFSTTBM [28]. In this problem, a job shall wait by the first machine before starting its processing until it can be continuously processed by all machines.

We will propose an algorithm to transform CPFSTTBM into a TSP model where jobs must be processed continuously through all the machines. To illustrate, let us construct a Gantt chart for two jobs J_a and J_b, see figure 6-1. In this Gantt chart, we let S_i be the earliest starting time of job J_i on machine 1. Therefore, the two dotted lines at S_a and S_b correspond to the starting time of jobs J_a and J_b on the first machine M_1 respectively.

Since all jobs are available to be processed at the beginning, S_b can be adjusted to the most left in the Gantt chart so as to minimize the makespan, under the

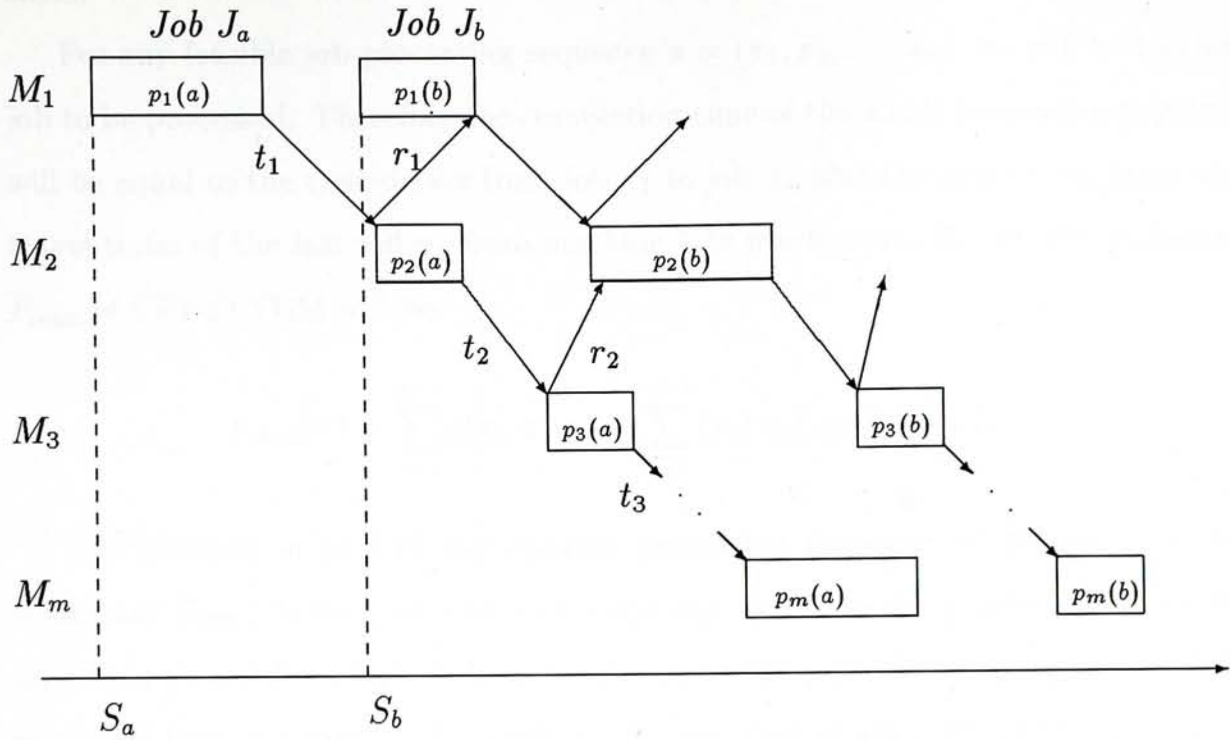


Figure 6-1: Gantt Chart for Two Consecutive Job in CPFSTTBM

following two constraints:

1. *Each machine can process at most one job at any time.*

2. *Machine M_j will be blocked by a completed job unless the transporter is available to remove the job from the machine.*

If S_b is set too early, the previous job J_a may not be completed on machine M_2 while the job J_b has arrived at machine M_2. Because of constraint (1), job J_b will have to wait until job J_a has been completed on machine M_2. Moreover, job J_b has to consider the availability of the transporter T_1. These considerations apply to all machines. In general, S_b should be set to a minimum value such that job J_b can be continuously processed on all machines after job J_a. Accordingly, define the minimum delay of job J_b as $d(a, b) = \min(S_b - S_a)$.

With the definition of minimum delay, we can regard jobs in CPFSTTBM as cities in TSP and the minimum delay between two jobs as the distance between two

cities.

For any feasible job processing sequence $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, π_n will be the last job to be processed. Therefore the completion time of the whole processing sequence will be equal to the time delays from job π_1 to job π_n plus the processing times and travel times of the last job π_n from machine 1 to machine m. Hence, the makespan $F_{\max}$ of CPFSTTBM will be:

$$F_{\max}(\pi) = \sum_{i=1}^{n-1} d(\pi_i, \pi_{i+1}) + \sum_{s=1}^{m-1} (p_s(\pi_n) + t_s) + p_m(\pi_n)$$

The problem is to find the optimal processing sequence π^* to minimize the makespan $F_{\max}$. In fact, we can add a dummy job J_0 to the problem, and let the minimum time delay of job J_0 by other job J_i be the total time required for J_i to be processed from machine M_1 to machine M_m and that of job J_i by J_0 be zero. Then, the CPFSTTBM will be equivalent to an $(n+1)$-city TSP with the cost matrix equal to $C = \{c_{ij}\}$ where $i, j = 0, 1, ..., n$. The matrix is as follows:

$$\left\{ \begin{array}{ll} c_{0i} = 0 & , \forall i \neq 0 \\ c_{ij} = d(i,j) & , \forall i,j \neq 0, i \neq j \\ c_{i0} = \sum_{s=1}^{m-1} (p_s(i) + t_s) + p_m(i) & , \forall i \neq 0 \\ c_{ii} = \infty & , \forall i \end{array} \right\} \tag{6.1}$$

Now, we outline the procedure to evaluate the minimum time delay $d(i,j)$ of J_j from J_i.

Evaluate $d(i,j)$ - The Minimum Time Delay

Let $S_s(a)$ be the start time of processing job J_a on machine M_s. Then, we have the following recursive equations:

$$S_1(a) \;=\; S_a$$

$$S_s(a) \;=\; S_{s-1}(a) + p_{s-1}(a) + t_{s-1}\ ,\ \text{where } s = 1, ..., m$$

After simplification,

$$S_s(a) = S_a + \sum_{k=1}^{s-1}(p_k(a) + t_k)\ ,\ \text{where } s = 1, ..., m \tag{6.2}$$

Now, we show how to compute $d(i, j)$ under the following constraints:

Constraint 1 - One machine can process only one job at any time

We assume that every machine should not process more than one job at a time. Thus, job b should not be started processing on one machine until job a had finished processing on that machine. Mathematically, we can represent this constraint by the following inequality:

$$S_s(a) + p_s(a) \leq S_s(b)\ ,\ s = 1, 2, \cdots, m \tag{6.3}$$

Noting (6.2), We have:

$$S_a + \sum_{k=1}^{s-1}(p_k(a) + t_k) + p_s(a) \leq S_b + \sum_{k=1}^{s-1}(p_k(b) + t_k) \tag{6.4}$$

This should be valid for all s. Therefore,

$$S_b \geq S_a + \max_{1 \leq s \leq m} \left\{ \sum_{k=1}^{s-1}(p_k(a) - p_k(b)) + p_s(a) \right\} \tag{6.5}$$

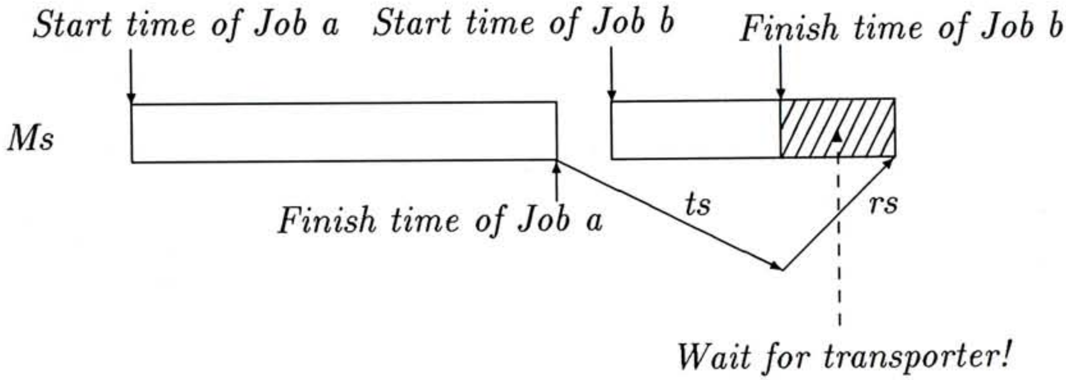
Constraint 2 - Availability of Transporter

Once a job J_a has been completed on machine M_s, a transporter T_s will pick up the job to the next machine M_{s+1} and then return to M_s. The total time required by the transporter will be $t_s + r_s$, during which the transporter is not available for the next job J_b. If job J_b has been completed within this period of time, it will have to wait until the transporter T_s is available. As a result, job J_b cannot be processed continuously.

Let us now determine the condition on the earliest starting time of job J_b on machine M_s considering the availability of transporter T_s. Suppose that job J_a is completed on machine M_s at time $S_s(a)+p_s(a)$ and then, transporter T_s will spend t_s+r_s units of time to remove job J_a to machine M_{s+1}, and return to machine M_s. Therefore, transporter T_s will be available after time $S_s(a)+p_s(a)+t_s+r_s$ for job J_b. Thus, we have the following inequality:

$$S_s(a)+p_s(a)+t_s+r_s \leq S_s(b)+p_s(b) \text{ , where } s=1,...,m \tag{6.6}$$

The following diagram depicts the case if the above inequality does not hold.



From (6.6) and (6.2), we get the following inequality:

$$S_b \geq S_a + \max_{1\leq s\leq m-1}\left\{\sum_{k=1}^{s}(p_k(a)-p_k(b))+t_s+r_s\right\} \tag{6.7}$$

Combining the two constraint inequalities (6.5) and (6.7), we have:

$$S_b \geq S_a + \max\left\{\begin{array}{l} \max_{1\leq s\leq m}\left\{\sum_{k=1}^{s-1}(p_k(a)-p_k(b))+p_s(a)\right\} \\ \max_{1\leq s\leq m-1}\left\{\sum_{k=1}^{s}(p_k(a)-p_k(b))+t_s+r_s\right\} \end{array}\right\} \tag{6.8}$$

The minimum of (S_b-S_a) is equal to,

$$\max\left\{\begin{array}{l} \max_{1\leq s\leq m}\left\{\sum_{i=1}^{s-1}(p_i(a)-p_i(b))+p_s(a)\right\} \\ \max_{1\leq s\leq m-1}\left\{\sum_{i=1}^{s}(p_i(a)-p_i(b))+t_s+r_s\right\} \end{array}\right\}$$

and therefore the minimum time delay of job J_b from job J_a will be,

$$d(a,b) = \min(S_b - S_a) = \max \left\{ \begin{array}{l} \max_{1\le s\le m} \left\{\sum_{i=1}^{s-1}(p_i(a) - p_i(b)) + p_s(a)\right\} \\ \max_{1\le s\le m-1} \{\sum_{i=1}^{s}(p_i(a) - p_i(b)) + t_s + r_s\} \end{array} \right\} \tag{6.9}$$

According to (6.9), a procedure for transforming the CPFSTTBM to TSP is as follows:

INPUT: n: number of jobs
m: number of machines
$p_i(j)$: processing time of job i on machine j
t_j: transportation time from machine j to machine $j+1$
r_j: transportation time from machine $j+1$ to machine j
BEGIN
DECLARE ARRAY $c(n,n)$
FOR $i = 0$ TO n
FOR $j = 0$ TO n
IF $i = j$ THEN $c(i,j) = \infty$
ELSE IF $i = 0$ THEN $c(i,j) = \sum_{k=1}^{m-1}(p_k(i) + t_k) + p_m(i)$
ELSE IF $j = 0$ THEN $c(i,j) = 0$
ELSE $c(i,j) = d(i,j)$ as described in Eq. (6.9)
END IF
END FOR
END FOR
END
OUTPUT: ARRAY c

The above algorithm has been developed as a software module which can be incorporated into our new TSP Solver. It can now solve any CPFSTTBM problem, given the problem parameters. This software module has been extensively tested using numerous testing problems of varying number of jobs, machines and processing times. The computational experiments have shown that our new method has an excellent performance in solving various CPFSTTBM problems. The computational results are reported in the next Chapter.

Chapter 7

Computational Results of the New Algorithm for CPFSTTBM

To evaluate the performance of our new algorithm for CPFSTTBM, numerous problem instances have been solved. In the following sections we shall report our results obtained in solving some problems which are generated randomly. The details of the random CPFSTTBM generation procedures are described in appendix A. Since the optimal solutions for these problems are not known, complete enumeration for problems with small number of jobs was also carried out as comparison. For large problems, we compared the solutions obtained by using our new algorithm with an efficient method called SPIRIT (Sequencing Problem Involving a Resolution by Integrated Taboo search techniques) [66]. All the experiments were carried out in a DEC workstation AXP3800 with one 200MHz CPU with the parameters setting of GA(SPIR/LDPX) as given in the Table 7.1.

Problem Name	Population Size	Parent Pool Size SPIR	LDPX	Mutation probability	δ_s	δ_t
10x10	20	16	16	0.8%	5.0	0.5
12x12	20	16	16	0.8%	5.0	0.2
50x10	20	16	16	0.8%	5.0	0.2
100x12	20	16	16	0.8%	5.0	0.2
200x12	20	16	16	0.8%	5.0	0.2

Table 7.1: GA(SPIR/LDPX) Parameter Setting for FSTTBM

7.1 Comparison with Global Optimum

Table 7.2 and Table 7.3 report the computational results obtained in using our algorithm to solve 10 instances of 10-machine-10-job-CPFSTTBM and 10 instances of 12-machine-12-job-CPFSTTBM which were randomly generated. The 'Problem No.' is the random seed number used for the generation procedure. Furthermore, the best solution of each problem was found out by complete enumeration 'Comp. Enum.' and is reported in column 'Opt.'. In addition, the execution time, 'Time' required to obtain the solution by using our new algorithm was recorded, which is the total time spent by the transforming algorithm and the TSP Solver.

Problem	Comp. Enum.		TSP solver		% of
No.	Opt.	Time	Obj.	Time	Error
1	837	32.7 s	837	0.3 s	0.00 %
2	796	32.7 s	796	0.3 s	0.00 %
3	813	32.7 s	813	0.2 s	0.00 %
4	833	32.7 s	833	0.3 s	0.00 %
5	814	32.7 s	814	0.3 s	0.00 %
6	809	32.7 s	809	0.3 s	0.00 %
7	829	32.7 s	829	0.3 s	0.00 %
8	817	32.7 s	817	0.3 s	0.00 %
9	800	32.7 s	800	0.3 s	0.00 %
10	845	32.7 s	845	0.2 s	0.00 %

Table 7.2: Results on 10-machine-10-job CPFSTTBM

Problem	Comp. Enum.		TSP solver		% of
No.	Opt.	Time	Obj.	Time	Error
1	1034	4880 s	1034	0.4 s	0.00 %
2	1036	4880 s	1036	0.3 s	0.00 %
3	1037	4880 s	1037	0.6 s	0.00 %
4	1011	4880 s	1011	0.3 s	0.00 %
5	1003	4880 s	1003	0.4 s	0.00 %
6	1012	4880 s	1012	0.4 s	0.00 %
7	1017	4880 s	1017	0.4 s	0.00 %
8	1062	4880 s	1062	0.3 s	0.00 %
9	1007	4880 s	1007	0.3 s	0.00 %
10	1069	4880 s	1069	0.3 s	0.00 %

Table 7.3: Results on 12-machine-12-job CPFSTTBM

It can be seen from Table 7.2 and Table 7.3, that the new algorithm can find the global optimum of all the CPFSTTBM instances we have tested.

7.2 The Algorithm of SPIRIT

For CPFSTTBM of large size, it has been impossible to find the optimal solution by complete enumeration, so we compare our method with an efficient method for flowshop scheduling problem called SPIRIT [66]. In Widmer's paper [66], SPIRIT has been compared with other heuristic methods including: Slope order index by Palmer [54], Gupta's algorithm [27], CDS [10], rapid access with close order search RACS [14] and, NEH [51], and shown to be able to find solutions of higher quality. For a detailed discussion, see [66].

SPIRIT consists of two main steps. The first step is to find an initial solution using insertion method and then, the second step is to refine the solution by a taboo search technique [20].

The insertion method used for CPFSTTBM is sketched below: (It is similar to the one used for FSP but the cost function incorporates the constraints of CPFSTTBM described in Chapter 6.)

```
INPUT:  n: number of jobs
        m: number of machines
        p_j(i): processing time of job i on machine j
        t_j: transportation time from machine j to machine j+1
        r_j: transportation time from machine j+1 to machine j

DECLARE OPT=(): optimal sequence of jobs.
DECLARE UNSEQ=(1,2,...,n): unsequenced jobs.

Step 1: Find two jobs J_a and J_b such that the makespan of the two jobs is minimum.
        OPT = (a,b), UNSEQ=UNSEQ -a - b
Step 2: WHILE |UNSEQ| > 0 DO
            Choose a job J_i from UNSEQ randomly.
            Insert into OPT such that the makespan of OPT is minimum.
            UNSEQ=UNSEQ -i
        END WHILE

OUTPUT: OPT
```

SPIRIT uses the taboo search algorithm for FSP [21] to refine the solution obtained by the insertion method. The algorithm used in our experiment is as follows.

```
INPUT: n: number of jobs
       m: number of machines
       p_j(i): processing time of job i on machine j
       t_j: transportation time from machine j to machine j+1
       r_j: transportation time from machine j+1 to machine j
       nbmax: max. number of iterations between 2 improvements
       T: taboo list storing ((a,i),(b,j))
          (It remembers the optimal move which swapping Job a
          at position i with Job b at position j)
       TSize: size of the taboo list T

BEGIN
    Find an initial feasible sequence s using Insertion Method
    nbmax=n
    TSize=7
```

```
    nbiter=0 (Iteration counter)
    BI=0 (The iteration in which the best makespan has been found)
    BS=π (The best sequence)
    BV=F_max(π) (The makespan value of BS)

    WHILE nbiter-BI<nbmax DO
        nbiter=nbiter+1
        Find the best neighbour π* of π which is not taboo by swapping
        two cities.
        π = π*
        Store the move into the taboo list T
        IF |T| > TSize THEN
            Remove the oldest move from the taboo list T
        END IF
        IF F_max(π) <BV THEN
            BV=F_max(π)
            BI=nbiter
            BS=π
        END IF
    END WHILE

    OUTPUT BS, BV
END
```

7.3 Comparison with SPIRIT

We report in Table 7.4 the computational results of using SPIRIT and GA(SPIR/LDPX). The number of jobs varies from 50 up to 200. The 'Random seed' is the random seed number for CPFSTTBM generation using the procedure in appendix A. 'Time' is the execution time including the time spent on the TSP Solver as well as the transforming procedure. The parameters setting of the TSP Solver has been described before in Table 7.1.

Table 7.4 shows that for CPFSTTBM of large size, our method can also get satisfactory results within reasonable execution time. Compared with SPIRIT, our algorithm obtained better solutions. Note that SPIRIT has been shown to outperform other heuristic methods for FSP.

No. of Jobs, No. of Machines, Random seed	SPIRIT		GA(SPIR/LDPX)		% better than SPIRIT
	Obj.	Time (s)	Obj.	Time (s)	
50,20,1	3,514	1.9	3,429	9.1	2.4 %
50,20,2	3,608	2.0	3,455	11.2	4.2 %
50,20,3	3,612	1.8	3,476	15.1	3.8 %
100,20,1	6,163	28.2	5,982	68.6	2.9 %
100,20,2	6,204	28.5	6,026	155.8	2.9 %
100,20,3	6,236	28.5	6,155	97.2	1.3 %
200,20,1	11,245	470	11,010	702	2.1 %
200,20,2	11,322	475	11,279	648	0.4 %

Table 7.4: Computational Results of TSP Solver and SPIRIT

Chapter 8

Conclusion

We have successfully improved the original LDPX by enhancing the gene sets and the crossover procedure. We have also proposed a new genetic operator, Single Parent Improved Reproduction (SPIR), which is suitable at the early stage of the evolution when individuals in the population have not been very fit.

By employing SPIR and LDPX in the genetic algorithm, we have developed a general-purpose TSP Solver software package called GASL. The solver has been tested by solving TSPs of varying size. For TSPs of small size, the solver can obtain the exact optimal solutions whereas for TSPs of large size such as the problem pr439, it can obtain near-optimal solutions. We have also compared GASL with other genetic algorithms which use partial mapped crossover (PMX) and edge recombination crossover (ER). The results show that GASL is consistently better than GA(PMX) as well as GA(ER), although ER is known to be a powerful crossover operator.

In part II, we have examined the problem of flowshop scheduling with travel time between machines, a more realistic model of flowshop scheduling problem. We have described the formulation of the model and have proposed an algorithm to transform CPFSTTBM into a TSP model. A software module has been developed and incorporated into the TSP Solver. Computational results have been obtained in solving problems with up to 200 jobs, which show that our new method can obtain

the exact optimal solutions or near optimal solution. In addition, computational experiments have been carried out to compare our method with SPIRIT [66], an effective heuristic method for FSP, shows that our method obtained better results.

Bibliography

[1] A. Agnetis, C. Arbib and K. E. Stecke, *Optimal two machine scheduling in a flexible flow system*, Proceedings of 2nd International Conference on Computer Integrated Manufacturing, Rensselaer Polytechnic Institute, 1990.

[2] Anthony Y.C. Tang *TOLKIEN: Toolkit for genetics-based applications*, M. Phil. Dissertation, Computer Science Department, The Chinese University of Hong Kong, 1994.

[3] K.R. Baker, *Introduction to Sequencing and Scheduling*, Wiley, New York, 1974.

[4] Bellman, Richard and Stuart Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.

[5] R.G. Bland and D.F. Shallcross, *Large Traveling Salesman Problems Arising From Experiments in X-Ray Crystallography: A Preliminary Report in Computation*, Operations Research Letters, Vol.8, pp. 125-128, 1989.

[6] L. Booker, *Improving Search in Genetic Algorithms*, in Lawrence Davis (Ed.), Genetic Algorithms and Simulated Annealing. Morgan Kaufmann, pp.61-73, 1987.

[7] D. Booth and S. Turner, *Comparison of heuristics for flow shop sequencing*, OMEGA, International Journal of Management Science 15/1, 1987.

[8] X. Cai, *The Crossover Operator for Genetic Algorithms*, - a working paper, 1991.

[9] X. Cai and K.W. Lee, *A genetic algorithm for flowshop scheduling with travel times between machines*, The Second European Congress on Intelligent Techniques and Soft Computing, Aachen, Germany, October, 1994.

[10] H.G. Campbell, R.A. Dudek and M.L. Smith, *A heuristic algorithm for the n-job, m-machine sequencing problem*, Management Science, Vol.16, B630-B637, 1970.

[11] R.W. Conway, W.L. Maxwell and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[12] L. Davis *Applying Adaptive Algorithms to Epistatic Domains*, Proceedings of the International Joint Conference on Artificial Intelligence, pp. 162-164, 1985.

[13] L. Davis *Job shop scheduling with genetic algorithms*, Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 136-140, 1985.

[14] D.G. Dannenbring, *An evaluation of flow shop sequencing heuristics*, Management Science, Vol.23, pp.1174-1182, 1977.

[15] R.C. Dixon, *Lore of the Token Ring*, in IEEE Network Magazine, vol. 1, pp. 11-18, Jan. 1987.

[16] Dreyfus, E. Stuart and M.L. Averill, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1977.

[17] S. Eilon, C. Watson-Gandy and N. Christofides, *Distribution Management: Mathematical Modeling and Practical Analysis*, Operational Research Quarterly Vol.20, p.309, 1969.

[18] S. Eilon, C. Watson-Gandy and N. Christofides, *Distribution Management*, Griffin, London, 1971.

[19] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman, San Francisco, 1979.

[20] F. Glover, C. McMillan and B. Novick, *Interactive decision software and computer graphics for architectural and space planning*, Annals of Operations Research 5, 1985.

[21] F. Glover, *Futher path for integer programming and links to artificial intelligence*, Computers and Operations Research Vol.13, No.5, 1986.

[22] D.E. Goldberg, *Computer-aided gas pipeline operation using genetic algorithms and rule learning*, (Doctoral dissertation, University of Michigan). Disserrtation Abstracts International, 44(10), 3174B. (University Microfilms No. 8402282), 1983.

[23] D. E. Goldberg, R. Lingle, *Alleles, Loci, and the TSP*, in Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 154-159, 1985.

[24] D.E. Goldberg and R.E. Smith, *Blind inferential search with genetic algorithms.* Paper presented at the ORSA/TIMS Joint National Meeting, Miami, FL, 1986.

[25] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, Reading, MA, 1989.

[26] J. Grefenstette, R. Gopal, B. Rosmaita, D. van Gucht, *Genetic algorithms for the traveling salesman problem*, Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 160-165, 1985.

[27] J. N. D. Gupta, *A search algorithm for the generalised flow-shop scheduling problem*, Computer and Operations Research, Vol.2, pp. 83-90, 1975.

[28] J. N. D. Gupta, *Flowshop Schedules with Sequence Dependent Setup Times*, Jornal of the Operations Research, Vol.29, No.3, pp.206-219, September 1986.

[29] M. Held and R.M. Karp, *A Dynamic Programming Approach to Sequencing Problems*, J. SIAM Vol.10, pp.196-210, 1962.

[30] M. Herdy, *Application of the Evolution Strategy to Discrete Optimization Problems*, Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN), Dortmund, Germany, pp. 188-192, 1990.

[31] J.H. Holland, *Adaptation in natural and artificial systems.* Ann Arbor: The University of Michigan Press, 1975.

[32] A. Homaifar and S. Guan, *A New Approach on the Traveling Salesman Problem by Genetic Algorithm*, Technical Report, North Carolina A & T State University, 1991.

[33] Howard, A. Ronald, *Dynamic Programming*, Management Science, Vol.12, pp.317-345, 1966.

[34] E. Ignall and L. Schrage, *Application of the Branch and Bound Technique to Some Flow Shop Scheduling Problems*, Operations Research Vol.13, 3, May 1965.

[35] S.M. Johnson *Optimal Two- and Three-stage Production Schedules with Set Up times included*, Nov. Res. Log. Quart., Vol.1, 1954.

[36] M.E. Johnson, *Simulated annealing (SA) & optimization : modern algorithms with VLSI, optimal design, & missile defence applications*, American journal of mathematical and management sciences, vol. 8, nos. 3-4, American Sciences Press, Syracuse, N.Y., 1988.

[37] D.S. Johnson, *Local Optimization and the Traveling Salesman Problem*, in M.S. Paterson (Editor), Proceedings of the 17th Colloquium on Automata, Languages, and Programming, Springer-Verlag, Lecture Notes inComputer Science, Vol. 443, pp. 446-461, 1990.

[38] Karp, *Reducibility among combinatorial problems*, Miller and Thatcher eds., Complexity of computer computations, Plenum Press, NY, p.85, 1972.

[39] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, *Optimization by Simulated Annealing*, Science, 220, pp.671-680, 1983.

[40] B. Korte, *Applications of Combinatorial Optimization*, talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.

[41] P. Krolak, W. Felts, and G. Marble, *A Man-Machine Approach toward Solving the Traveling-Salesman Problem*, CACM Vol.14, pp.327-334, 1971.

[42] E. L. Lawler and D.E. Wood, *Branch-and-Bound Methods: A Survey*, Operations Research, Vol.14, pp.699-719, 1966.

[43] E. L. Lawler, *Combinatorial optimization: Networks and matroids.* New York: Holt, Rinehart and Winston, 1976.

[44] E. L. Lawler, et al., *The Traveling Salesman Problem*, Wiley-Interscience Publication, 1985.

[45] S. Lin and B. W. Kernighan, *An Effective Heuristic Algorithm for the Traveling Salesman Problem*, Operations Research, pp. 498-516, 1973.

[46] J.D. Litke, *An Improved Solution to the Traveling Salesman Problem with Thousands of Nodes*, Communications of the ACM, Vol.27, No.12, pp.1227-1236, 1984.

[47] J.D.C. Little, et al., *An Algorithm for the Traveling Salesman Problem*, Opns. Res. Vol.11, pp.972-989, 1963.

[48] P. L. Maggu, G. Das and R. Kumar, *On equivalent job-for-job block in $2 \times n$ sequencing problem with transportation times*, Journal of Operational Research Society Japan, Vol. 24, pp. 136-146, 1981.

[49] P. Miliotis, *Integer Programming Approaches to the Traveling Salesman Problem*, Math. Progr. Vol.10, pp. 367-378, 1976.

[50] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer, *Evolution Algorithms in Combinatorial Optimization*, Parallel Computing, Vol.7, pp. 65-85, 1988.

[51] M. Nawaz, E.E. Enscore and I. Ham, *A heuristic algorithm for the m-machine, n-job flow shop sequencing problem*, OMEGA, International Journal of Management Science, Vol.11, No.1, 1983.

[52] I. M. Oliver, D. J. Smith, and J. R. C. Holland *A Study of Permutation Crossover Operators on the Traveling Salesman Problem*, in Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 224-230, 1987

[53] M. Padberg and G. Rinaldi, *Optimization of a 532-City Symmetric Travelling Salesman Problem*, Technical Report IASI-CNR, Italy, 1986.

[54] D. S. Palmer, *Sequencing jobs throudh a multi-stage process in the minimum total time - a quick method of obtaining a near optimum*, Operational Research Quarterly, Vol.16, pp.101-107, 1965.

[55] S. S. Panwalkar, *Scheduling of a two machine flowshop with travel time between machines*, Journal of Operational Research Society, Vol. 42, pp. 609-613, 1991.

[56] S. Reddi and C. Ramamoorthy, *On the Flow Shop Sequencing Problem with No Wait in Process*, Operational Research Quarterly Vol.23, 3 Sept. 1972.

[57] G. Reinelt, *TSPLIB - A Traveling Salesman Problem Library*, ORSA Journal on Computing, Vol.3, No.4, pp.376-384, 1991.

[58] D. Rosenkrantz, R. Stearns and P. Lewis, *Approximate Algorithms for the Traveling Salesperson Problem*, Proceedings of the 15th Annual IEEE Symposium of Switching and Automata Theory, pp.33-42, 1974.

[59] F.E. Ross, *FDDI - A Tutorial*, in IEEE Commun. Magazine, col 24, pp. 10-15, May 1986.

[60] H. I. Stern and G. Vitner, *Scheduling parts in a combined production transportation work cell*, Journal of Operational Research Society, Vol. 41, pp. 625-632, 1990.

[61] N.L.J. Ulder, E.H.L. Aarts, H.J. Bandelt, P.J.M. van Laarhoven, E. Pesch *Genetic Local Search Algorithms for the Traveling Salesman Problem*, in Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN), Dortmund, Germany, pp. 109-116, 1990.

[62] J.M. Van Deman, K.R. Baker, *Minimizing Mean Flowtime in the Flow Shop with No Intermediate Queues*, AIIE Transactions, Vol.6, pp.28-34, 1974.

[63] B. F. Voigt, *Der Handlungseisende, wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein*, Von einem alten Commis-Voyageur, Ilmenau. (Republished (1981) Verlag Berd Schramm, Kiel.)

[64] D. Whitley, and J. Kauth *GENITOR: a different genetic algorithm*, Proceeding of the Rocky Mountain Conference on Artificial Intelligence, Denver, CO, pp. 118-130, 1988.

[65] D. Whitley, T. Starkweather and D'Ann Fuquay, *Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator*, in Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Inc., pp. 133-140, 1989.

[66] M. Widmer and A. Hertz, *A new heuristic method for flow shop sequencing problem*, European Journal of Operational Research, Vol. 41, pp. 186-193, 1989.

[67] D. A. Wismer, *Solution of the Flowshop Scheduling Problem with No Intermediate Queues*, Operations Research, Vol. 20, pp. 689-697, 1972.

Appendix A

Random CPFSTTBM problem Generation Algorithm

CPFSTTBM problems were generated with the following three conditions:

$$10 \leq t_j \leq 20 \text{ , where } j = 1, ..., m-1$$
$$5 \leq r_j \leq 10 \text{ , where } j = 1, ..., m-1$$
$$10 \leq p_j(i) \leq 50 \text{ , where } i = 1, ..., n; j = 1, ..., m$$

where t_j is the time required for the transporter T_j to move a job from machine j to machine $j+1$ and r_j is the time required for the transporter T_j to come from machine $j+1$ back to machine j. Each job J_i, has its own processing time on different machine j which is denoted by $p_j(i)$.

The random number generator *rand()* is defined as follows:

$$
\begin{aligned}
X_0 &= Random\ Seed\ No \\
X_{n+1} &= (97 * X_n + 37) \bmod 65536
\end{aligned}
$$

Figure A-1: Random Number Generation Algorithm

Using the random generator, an m-machine-n-job FSTTBM is generated in three steps:

```
Step 1:    FOR i = 1 TO m - 1
                   t_i = 10 + rand() mod 11
           END FOR

Step 2:    FOR i = 1 TO m - 1
                   r_i = 5 + rand() mod 6
           END FOR

Step 3:    FOR i = 1 TO n
                   FOR j = 1 TO m
                           p_j(i) = 10 + rand() mod 41
                   END FOR
           END FOR
```

Figure A-2: Algorithm for FSTTBM Generation

For example, a 5-machine-10-job FSTTBM generated using seed number 1 is as follows:

t_j	12	10	19	19	
r_j	7	6	7	6	
p_{1j}	50	34	14	23	11
p_{2j}	34	42	50	49	31
p_{3j}	33	24	18	48	28
p_{4j}	45	41	21	45	36
p_{5j}	36	32	43	15	26
p_{6j}	22	45	41	47	32
p_{7j}	10	14	22	16	20
p_{8j}	39	28	20	11	29
p_{9j}	29	44	47	48	43
p_{10j}	42	23	32	29	19

Table A.1: Example of 5-machine-10-job FSTTBM

CUHK Libraries
000733954