GREEDY RANDOMIZED ADAPTIVE SEARCH PROCEDURE FOR TRAVELING SALESMAN PROBLEM

A Thesis

by

SEUNG HO LEE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2005

Major Subject: Industrial Engineering

GREEDY RANDOMIZED ADAPTIVE SEARCH PROCEDURE FOR TRAVELING SALESMAN PROBLEM

A Thesis

by

SEUNG HO LEE

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

May 2005

Major Subject: Industrial Engineering

ABSTRACT

Greedy Randomized Adaptive Search Procedure for Traveling Salesman Problem.

(May 2005)

Seung Ho Lee, B.E., Korea University

Chair of Advisory Committee: Dr. Sergiy Butenko

In this thesis we use greedy randomize adaptive search procedure (GRASP) to solve the traveling salesman problem (TSP). Starting with nearest neighbor method to construct the initial TSP tour, we apply the 2-opt and the path-relinking method for the initial tour improvement. To increase 2-opt search speed, fixed-radius near neighbor search and $don't - look$ bit techniques are introduced. For the same reason a new efficient data structure, the reverse array, is proposed to represent the TSP tour. Computational results show that GRASP gives fairly good solutions in a short time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

The traveling salesman problem (TSP) is one of the most common problems in combinatorial optimization. A number of prominent researchers have tried to attack this problem. The role of the TSP in the field is underlined by the fact that it is commonly accepted as the representative combinatorial optimization problem. Its practical importance is one of the reasons of such status. In fact, many significant real world problems can be formulated as instances of the TSP. The well known applications of the TSP include vehicle routing, circuit wiring, network connection, job sequencing. The definition of the TSP can be simply stated without any mathematical notation as follows. A salesman has to visit n cities once and only once and finish where he started. Given the cost of travel between each pair of cities, the salesman wants to find the minimum cost tour of cities. At a glance, the salesman's problem looks very straightforward and easy. However, the difficulty is revealed if the number of possible tours is considered. For an n-city symmetric problem (with the same pairwise distances regardless of the travel direction), there are $\frac{(n-1)!}{2}$ possible tours. Hence, for only $n = 10$, there exist more than 10^6 tours. In 1979, Garey and Johnson [9] proved that TSP is an *NP*-hard problem that can not yet be solved in polynomial time. Thus it is infeasible to follow complete enumeration of large size real-world TSP instances. Even if there is an exact method that guarantees an optimal solution, its running time is prohibitively excessive for large-scale problems. In order to handle such large problems, many heuristic methods have been developed.

An heuristic method, by definition, is any solution method involving computa-

---

This thesis follows the style and format of *SIAM Journal of Computing*.

tionally efficient strategies that should produce a solution at least close to the optimal one, even if it does not find the optimum. Though they do not necessarily find optimal solutions, heuristic methods give fairly good results. According to Feo and Resende, "The effectiveness of these methods depends on their ability to adapt to a particular realization, avoid entrapment at local optima, and exploit the basic structure of the problem, such as a network or a natural ordering among its components. Furthermore, restart procedures, controlled randomization, efficient data structures, and preprocessing are also beneficial" [6]. Combined with rapid development of computer technology, more successful heuristic methods are being introduced at a high pace. Heuristic algorithms for the TSP can be classified into two categories, tour construction algorithms and local search algorithms. The first tries to construct a good initial tour, and the second attempts to improve the tour already constructed. Nearest neighbor, insertion methods, greedy, and Christofides algorithm are some of the most promising known heuristics for the tour construction algorithm. 2-opt, 3-opt, simulated annealing, genetic algorithms, and Lin-Kernighan are the most competitive local search algorithms [4, 12].

Starting from any city, the nearest neighbor (NN) algorithm always takes the nearest not-yet-visited city. When there is no such city left, it means that all cities have been visited and the starting city has to be visited next. The running time of NN is $O(n^2)$, and Rosenkrantz, Stearns, and Lewis [18] show that if the costs between cities are nonnegative and satisfy the triangle inequality, the length of NN tour is less than $\frac{1}{2}\lceil \log n + 1 \rceil + \frac{1}{2}$ times the optimum.

Insertion methods start a tour construction by joining any two cities, and select the next city by the insertion rule. There are three different insertion methods depending on the insertion rule, farthest, nearest, and random insertion. The farthest insertion method always chooses the farthest city from any city in the current tour.

Similarly, the nearest insertion method picks the nearest city. The random insertion method inserts the city in random order. Rosenkrantz, Stearns, and Lewis [18] claim that the insertion methods have the tour cost less than $\lceil \log n \rceil + 1$ times the cost of the optimum.

In the greedy algorithm, the elements to be considered are not the cities but the edges between two cities. So the tour is built up by adding the shortest edge which does not create a degree 3 vertex or a cycle of length less than n. The straightforward implementation of the greedy algorithm requires $O(n^2 \log n)$ for the running time, but Bentley [1] reports $O(n \log n)$ time for the uniform inputs. With the assumption that all edges satisfy the triangle inequality, Ong and Moore [15] claim that the length of the greedy tour is less than $\frac{1}{2}\lceil \log n \rceil + 1$ times the optimum.

The algorithm of Cristofides [2] has the smallest worst-case bound among the mentioned algorithms, $\frac{3}{2}$ times the optimum. It starts with finding a minimum spanning tree T and constructing a minimum perfect matching M of the cities which have odd degree in T. Then $E(T) \cup M$ will be a connected multi-graph in which every vertex has even degree. Since this graph contains an Euler tour, a TSP tour can be found by taking shortcuts to avoid multiple visits. A modified Cristofides algorithm of Gabow and Tarjan [8] guarantee the $O(n^{2.5})$ running time.

2-opt is one of the most famous simple local search algorithms that was first proposed by Croes [5]. It deletes two edges, thus breaking the tour into two separated paths, and then reconnects these two paths to form another possible tour. Similarly, 3-opt exchanges three edges of the tour thus reforming the tour with 3 paths. The running time of the 2-opt consists of the improving move search time and the movement performing time. In the worst case, the improving move search time takes $O(n^2)$. However, the movement performing time can be reduced to $O(1)$ using some efficient data structures.

Lin-Kernighan [14] can be considered a special tabu search algorithm and a variable λ-opt algorithm. It uses a tabu list but has a much higher complexity. At each step, starting with $\lambda = 2$, λ edges from the current tour will be exchanged to make another possible tour. If better tour is found, then λ is increased by 1 and the search is continued to find another tour improvement. Generally, Lin-Kernighan algorithm is considered as one of the most effective methods. The modified Lin-Kernighan algorithm of Helsgaun [11] shows that the average running time is approximately $O(n^{2.2})$.

Feo and Resende [6] introduced an iterative restart approach called the greedy randomized adaptive search procedures (GRASP). Each iteration consists of two phases, a construction phase and a local search phase. The best solution found during the iteration will be reported as the final solution. The detailed procedure of GRASP will be discussed more in Chapter IV of this thesis. GRASP is one of the most promising heuristic methods and is a common approach used for solving many combinatorial optimization problems. It has been successfully applied to the various combinatorial optimization problems, such as set covering, production planning and scheduling, graph problems, and location problems. Festa and Resende [7] provide an annotated bibliography of the GRASP from 1989 to 1999.

Path-relinking, originally introduced by Glover [10], is a deterministic search process to examine neighbors between good solutions. PR method is based on the belief that the neighbor of a good solution can be also a good solution. By selecting each element of the starting solution as the guiding good solution, near neighbors could be considered. If some near neighbor solution is better than the starting solution, we will save it as new solution candidate and continue the search to the next near neighbor. When the search reaches the guiding solution, best solution found so far will be recorded as the new solution.

Recently, Laguna and Martí [13] showed that path-relinking intensifies the GRASP procedure. Many extensions, improvements, and applications have been reported for this hybrid heuristic method. In this thesis, GRASP with path-relinking is applied to solve the traveling salesman problem. Since GRASP has not been used for the TSP yet, this thesis can give a guideline of GRASP's efficiency in solving the TSP.

CHAPTER II

QUICK 2-OPT SEARCH TECHNIQUES

A. Steiglitz and Weiner technique

The Steiglitz and Weiner technique was introduced by Steiglitz and Weiner [20] based on the simple observation. In order to illustrate this technique, the following several notations need to be defined. A pair (x, y) represents the fact that x is the immediate predecessor of y in the tour order. The notation $< t_1, t_2, t_3, t_4 >$ denotes four cities involved in the 2-opt move, where the edges $\{t_2, t_3\}$ and $\{t_1, t_4\}$ replace the edges (t_1, t_2) and (t_4, t_3). Then we can write each 2-opt move of those four cities in two different notations, $< t_1, t_2, t_3, t_4 >$ and $< t_4, t_3, t_2, t_1 >$ depending on where the tour starts. As shown in Figure 1, under the counterclockwise orientation, we can express the 2-opt move as $< t_1, t_2, t_3, t_4 >$ starting from t_1. In the same way, if the tour starts from t_4, the move can be denoted by $< t_4, t_3, t_2, t_1 >$. In both of the two different notations, to be the improving move, it must be the case that either (a) $d(t_1, t_2) > d(t_2, t_3)$ or (b) $d(t_3, t_4) > d(t_4, t_1)$, or both, where $d(t_1, t_2)$ denotes the distance between t_1 and t_2. To make the search fast, we can reduce our attention to the case satisfying (a) $d(t_1, t_2) > d(t_2, t_3)$ without missing any improving move. Suppose we have an improving move with the four cities $< c_1, c_2, c_3, c_4 >$. Suppose further that $d(c_1, c_2) < d(c_2, c_3)$ is satisfied. Then this improving move will be missed at the first scan, because we considered only one condition $d(c_1, c_2) > d(c_2, c_3)$. However, when the same four cities are encountered starting with c_4, which is the other notation $< c_4, c_3, c_2, c_1 >$, we check the condition $d(c_4, c_3) > d(c_2, c_3)$. Since $< c_1, c_2, c_3, c_4 >$ (or $< c_4, c_3, c_2, c_1 >$) is an improving move, at least one of those conditions should be satisfied. With this property, we only need to find allowable candidates for t_3

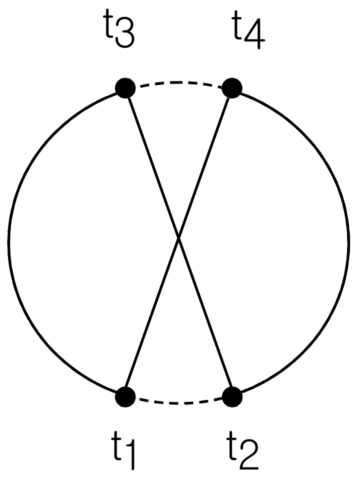


Fig. 1. 2-opt movement

satisfying (a) $d(t_1, t_2) > d(t_2, t_3)$. Steiglitz and Weiner [20] proposed storing a list of remaining cities for each city c in order of increasing distance from c. Then we can easily find all candidates x for t_3, which are cities from the beginning of t_2's list until $d(t_2, x) \geq d(t_1, t_2)$ is met. The drawback of this technique is that there are overhead $O(n^2 \log n)$ time and $O(n^2)$ space for sorting and saving the list. Johnson and McGeoch [12] reduced those overheads to $O(n^2 \log k)$ time and $O(nk)$ space by including only k nearest neighbors for each city.

B. *Don't-look* bit technique

In addition to the Steiglitz and Weiner technique, Bentley [1] proposed the *don't-look* bit technique. Let us consider the notation $< t_1, t_2, t_3, t_4 >$ and $< t_4, t_3, t_2, t_1 >$ again. We are now looking for t_3 satisfying (a) $d(t_1, t_2) > d(t_2, t_3)$ as introduced in the previous section. The basic idea of this method is that if no such improving move for a given t_1 has been found, and the neighbors of t_1 have not changed, then there is only a low probability to find an improving move for t_1 even on the next search. Thus, on the next search, t_1 will not be considered as a candidate for the improving move. Bentley [1] exploits this idea with a flag, called *don't-look* bit, for each city. At first, we start the improving move search with these flags all turned off. Whenever a search having $t_1 = c$ fails to find an improving move, the flag for the city c is turned on. Thus if its neighbors were not changed, c will be skipped for the t_1 candidates. However, when edges are deleted for the improving move, the flags of the corresponding 4 cities are turned off. Thus, when an edge having c as an endpoint is deleted, c can be considered as the candidate for t_1 again. An intuitive implementation of this *don't-look* bit technique is an array structure of the flag.

CHAPTER III

EFFICIENT 2-OPT DATA STRUCTURES

The data structure for the 2-opt search should support three operations, $Prev$, $Next$, and $Swap$. $Prev(t_2)$ and $Next(t_1)$ operations find the previous and the next city of t_2 and t_1, respectively. The other operation $Swap(t_1, t_2, t_3, t_4)$ is the realization of the notation $< t_1, t_2, t_3, t_4 >$ which is defined in Chapter II. It means that the edges $\{t_2, t_3\}$, $\{t_1, t_4\}$ substitute the edges (t_1, t_2), (t_4, t_3) in the tour. Suppose that we adopted an array or a linked list representation for the data structure. Then for the each $Swap$ operation, a path between t_2 and t_4 or a path between t_1 and t_3 should be reversed. However, since we are using an array structure, the only way to reverse the path is to exchange the cities in the path iteratively. In the worst case, the $Swap$ operation takes $\Theta(n)$ time. Therefore some appropriate data structure is required for the more efficient operation. In this chapter, we will review some previously proposed data structures, which enhance the $Swap$ operation to reduce overall running time.

A. Splay tree

Splay tree was invented by Sleator and Tarjan [19]. This data structure is essentially a binary tree having a city at each vertex. Splay tree has a special reversal bit indicating the direction of the subtree rooted at a vertex. Sleator and Tarjan [19] showed that the splay tree performs the worst-case $Swap$ operation in $O(\log n)$ time. However, the splay tree representation has not proved competitive in practice, because of its high overhead cost.

B. 2-level tree

The idea of 2-level tree was introduced by Chrobak, Szmacha, and Krawczyk [3]. 2-level tree divides the tour into approximately $\sqrt{n}$ segments whose members contain a pointer to their parent node. The parent node contains a reversal bit indicating whether the segment traverses in forward or reverse direction. Each member of the segment is maintained as a doubly-linked list and contains the index of the city it represents. By changing the reversal bit, we can reverse a path represented by the segment in constant time. Hence, the running time for the *Swap* operation is $O(\sqrt{n})$.

C. Satellite list

As another efficient data structure for 2-opt, Osterman and Rego [16] designed the satellite list. Figure 2 describes the basic concept of the satellite list structure and its *Swap* operation. The satellite list maintains two linked lists which indicate the clockwise and counterclockwise orientations of the tour. For example, suppose that 1-2-3-4-5-6-7 is a path of the TSP tour. Therefore the two linked lists indicating both orientations of the tour are the paths 1-2-3-4-5-6-7 (clockwise) and 7-6-5-4-3-2-1 (counterclockwise) as described in the first illustration of Figure 2. In the illustration, the two nodes connected with a dashed line indicate the same city and they are called *complement satellites* to each other. Suppose further $Swap(1,2,7,6)$ is performed; the edges $\{2,7\}$, $\{1,6\}$ substitute the edges $(1,2)$, $(6,7)$. This operation is a 180° flip of the linked lists as described in the second and third illustrations of Figure 2. The last illustration represents the reconstruction of the new tour path. In the C implementation, we can perform this *Swap* operation by changing the pointers of four linked list nodes (clockwise_1, clockwise_6, counterclockwise_2, counterclockwise_7) in a constant time. Osterman and Rego [16] designed a special array structure combining

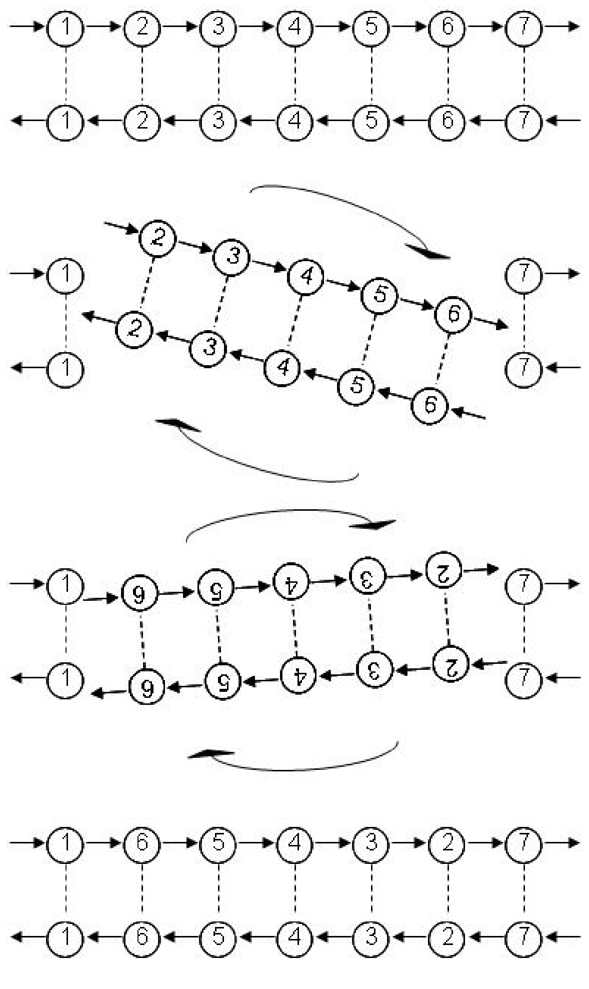
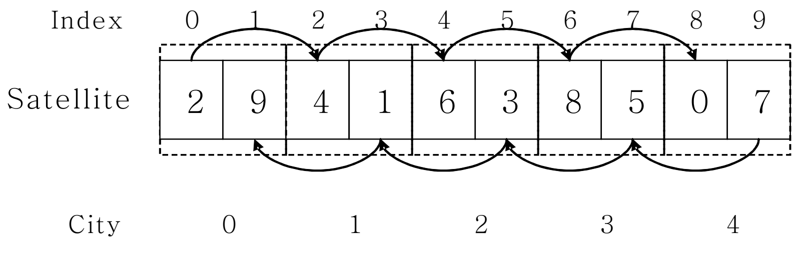
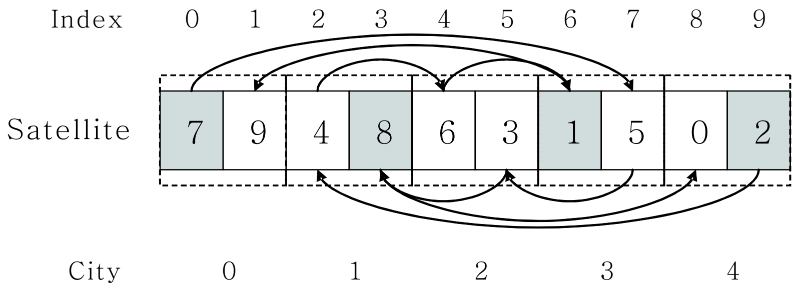


Fig. 2. Logical satellite list representation

those two linked lists. This satellite list array is the one dimensional array with length of $2n$, where n is the total number of cities. In the array, each of the evenly indexed element contains the next city's index starting from 0, and each of the oddly indexed element has the previous city's index. In this way, the satellite list array can be initially constructed representing both orientations of the TSP tour. The i^{th} element in the array indicates the index of the next or the previous city from the city $\lceil \frac{i}{2} \rceil$. If $i \; mod \; 2$ is 0, then the element represents the next city. Otherwise it denotes the index of previous city from the city $\lceil \frac{i}{2} \rceil$. For example, the 4^{th} element in the satellite list array will be the next city's index of the city 2, and the 5^{th} element will be the previous city's index of the city 2. Those 4^{th} and 5^{th} elements are the *complement satellites* which have the neighbor information of city 2. Under this convention, the tour 0-1-2-3-4 can be constructed to the satellite list array 2-9-4-1-6-3-8-5-0-7. In order to change the tour to the new tour 0-3-2-1-4, the edges (0,1), (3,4) should be removed and the edges {0,3}, {1,4} should be added. The concept of the *Swap* operation in the satellite list array is as follows. The satellite list array contains two tour representations of both orientations, which are (a) 0-1-2-3-4 and (b) 4-3-2-1-0. If we use the separated two linked list representation, the *Swap* operation will be the exchange of pointers. That is exchanging the pointer of 0 in tour (a) and the pointer of 4 in tour (b), and the pointer of 3 in (a) and the pointer of 1 in (b). Then the new two linked list will be (a) 0-3-2-1-4 and (b) 4-1-2-3-0. Similarly, in the satellite list array structure, the exchanges between array elements make the same tour reconstruction. Those are the swap between the even element of city 0 and the odd element of city 4, and the swap between the odd element of city 1 and the even element of city 3. Figure 3 demonstrates the above satellite list array *Swap* operation. After the swap operation, we can see that not all the even elements represent the next city's index. We should follow each element index to track the tour without knowing



(a) Tour0-1-2-3-4



(b) Tour0-3-2-1-4

Fig. 3. Physical satellite list representation

the orientation. By this reason, the satellite list is limited to the symmetric TSP. The splay tree (Sleator and Tarjan [19]) is claimed to handle the 2-opt movement operation in $O(\log n)$ time and the 2-level tree (Chrobak, Szymacha and Krawczyk [3]) is proved to have $O(\sqrt{n})$ *Swap* running time. Because of its symmetric design, the satellite list performs this operation easily in constant time, $O(1)$.

D. New efficient data structure, reverse array

For the 2-opt movement, the satellite list showed a good performance. However, the satellite list lost its advantage when it was used with the Steiglitz and Weiner technique. As described in Chapter II, the Steiglitz and Weiner technique stores a list of remaining cities for each city c in order of increasing distance from c. Then it finds all candidates x for t_3, which are cities of the t_2's sorted list. Since the Steiglitz and Weiner technique selects a candidate for t_3 from the stored sorting list sequentially, those candidates are not placed on the consecutive positions of the tour. Because of the satellite list's structure, it is impossible to find t_4 directly which is an immediate predecessor of randomly selected t_3 in the tour. Therefore $O(n)$ running time should be added to find t_4 for each candidate. Thus, we introduce the reverse array which shares the basic idea with the satellite list (Osterman and Rego [16]). In the reverse array, the two lists which denote both orientations of the tour will not be merged. The two arrays, the original array and the reverse array, are maintained separately. There is one more array which is named map array. The map array stores each city's index in the original array. For example, in Figure 4, the city 5 is stored at the index 2 of the original array, thus the 6th element of the map array will be 2. Using this map array, every city of the original tour can be accessed directly. We can perform the 2-opt move with those 3 arrays. At first, one candidate x for t_3 is selected from the

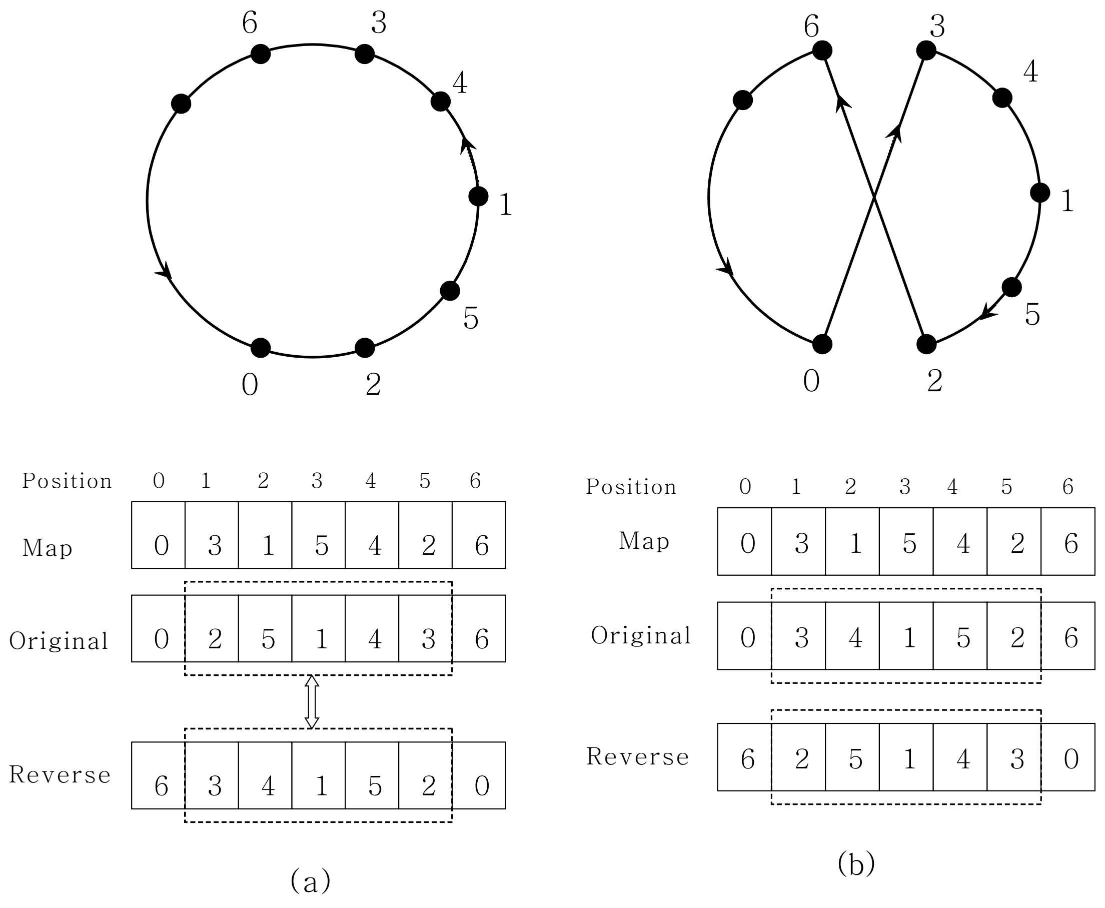


Fig. 4. 2-opt move and the reverse array representation

t_2's sorted list. Then x's index in the original array will be the value of x^{th} element in the map array and t_4 will be the $(x-1)^{th}$ element. Because we select the candidate elements for t_1 from the tour sequentially, the index of t_1 and t_2 are the number of iteration i and $i+1$. To complete the *Swap* operation, every element between t_2 and t_4 should be switched. In the reverse array, *Swap* can be done by the simple memory block swapping. The path between t_2 and t_4 is the memory block between the $(i+1)^{th}$ and the $(x-1)^{th}$ elements of the original array. The path is also the memory block between the $\{n-(x-1)+1\}^{th}$ and the $\{n-(i+1)\}^{th}$ elements of the reverse array where n is the size of the tour. By swapping these two memory blocks, the original and reverse array exactly represent the new tour and the running time is $O(1)$. This memory block copy operation is described in Figure 4. For a linked list structure, *Swap* operation has $O(n)$ running time and $O(\log n)$ for splay tree [19], and $O(\sqrt{n})$ for 2-level tree [3].

The only problem is the map array. Since the memory blocks between the original and reverse array have been exchanged, the map array has an incorrect information about the original array. To fix this problem, we use the map correction algorithm named fix-and-follow. The fix-and-follow algorithm corrects the invalid elements of the map array one by one. For example, we want to find the city c_1's index in the original array. First, we can get the c_1's index p_{c_1} from the map array which is the ${c_1}^{th}$ element of the map array. If the city c_1 is swapped before, the value of the $p_{c_1}^{th}$ element in the original array may be different from c_1. Otherwise, p_{c_1} is taken as the c_1's index. In the first case, suppose that c_2 is the value of the $p_{c_1}^{th}$ element in the original array. We know that the index information of c_2 in the map array is also incorrect and the actual index of c_2 is p_{c_1}. Thus, we can fix the index of c_2 by changing the value p_{c_2}, which is the ${c_2}^{th}$ element of the map array, to p_{c_1}. We keep following c_3, which is the $p_{c_2}^{th}$ element in the original array, until $c_n = c_1$. Algorithm 1

```
Data: City $c_1$
Result: Index of $c_1$ in the original array

$c_1 \leftarrow$ map[$c_1$]
while original[$i_3$] $\neq t_3$ do
    swap($i_3$, map[Original[$i_3$]])
end
map[original[$i_3$]] $\leftarrow i_3$
```

Algorithm 1: Fix-and-follow

is the pseudo-code of the fix-and-follow algorithm. As shown above, the fix-and-follow does not correct every swapped element of the map array, but it fixes only the elements that we already know and that we need to know. Therefore, the running time can be saved, and at the same time, the efficiency can be increased. Table 1 shows the average number of nodes corrected by the fix-and-follow procedure during a single 2-opt *Swap* operation of some selected instances from TSPLIB. From the computational results, the average number of corrected nodes for an n size instance is about $0.048n$. The correlation coefficient between the node size and the number of corrected nodes is 0.994. Thus we can say that there is a linear relationship.

Table 1. Average number of nodes corrected by fix-and-follow

Instance	Size of instance	Number of corrected node
burma14	14	0.6
gr17	17	0
gr21	21	0
fri26	26	0
bayg29	29	0.17
bays29	29	0.64
dantzig42	42	1.53
att48	48	1.16
gr48	48	0.85
hk48	48	1.2
eil51	51	1.23
berlin52	52	1.21
brazil58	58	1.67
eil76	76	2.06
gr96	96	2.32
eil101	101	2.54
gr120	120	2.84
bier127	127	3.79
ch130	130	3.57
gr137	137	3.83
ch150	150	4.47
d198	198	4.93
gr202	202	5.44

Table 1. Continued.

Instance	Size of instance	Number of corrected node
gr229	229	6.46
gil262	262	7.42
a280	280	7.26
fl417	417	9.41
gr431	431	14.49
d493	493	15.93
att532	532	15.28
ali535	535	15.88
d657	657	23.03
gr666	666	23.13
pr1002	1002	36.57
d1291	1291	58.27
fl1400	1400	37.33
fl1577	1577	67.99
d1655	1655	73.84
d2103	2103	106.56
pr2392	2392	106.16
fl3795	3795	180.5
fnl4461	4461	214.99

CHAPTER IV

GRASP PROCEDURE

The GRASP(Feo and Resende [6]) is an iterative procedure, where each iteration has two phases, the construction phase and the local search phase. In the construction phase, a feasible solution is constructed by choosing the next element randomly in the restricted candidate list (RCL). RCL contains only the r best elements selected by the greedy function. This RCL technique makes it possible to obtain a different solution at each iteration, while it does not compromise the power of adaptive greedy component. Since the solutions generated by the GRASP construction phase are not guaranteed to be the local optimum, it is recommended to apply the local search phase which is the second phase of the GRASP. In this thesis, the 2-opt search and the path-relinking (PR) method are applied for the local search phase. At the end of each GRASP iteration, the better solution substitutes the old solution to become the final solution when the given termination criterion is reached. The overall procedure of the GRASP is shown in Algorithm 2. The input data for Algorithm 2 include the stopping criteria. It can be the maximum number of iterations or the threshold value of the tour cost, or the running time. The output of Algorithm 2 is the best tour found during the iterations. Algorithm 3 is the detailed greedy randomized tour construction which is the construction phase of the GRASP. Selection of the r best elements for the RCL construction has been achieved by the nearest neighbor search method. The r number of nearest cities from the last selected city constitute the RCL. The size of RCL r is the input data for Algorithm 3. On the mark 1 of Algorithm 3, the variable r controls the number of candidate cities, which are the nearest cities. Thus, by adjusting the value of r, we can change the initially constructed tour quality. The smaller r we input, the better initial tour is constructed. In Chapter VI, we will

```
Data: Stop criterion
Result: Best tour $t^*$

while Stop criterion unsatisfied do
    $t \leftarrow$ GreedyRandomized()
    $t \leftarrow$ 2-optSearch($t$)
1   if E (Elite Set) not Filled then
        if t is not in E then
            $E \leftarrow E \cup t$
        end
    else
2       $t' \leftarrow$ RandomSelect($E$)
        $t \leftarrow$ PathRelinking($t, t'$)
        if Distance(t) < MaxDistance(E) then
            $t'' \leftarrow$ MostSimilar($E, t$)
            $E \leftarrow (E \backslash \{t''\}) \cup t$
        end
        $t^* \leftarrow$ Minimum(E)
    end
end
```

Algorithm 2: GRASP

```
Data: Number of elements in RCL $r$
Result: Initial tour $t$

$t_1 \leftarrow 0$
for $i$ = 1, ..., Number of Nodes-1 do
1   for $j$ = 0, ..., $r-1$ do
        RCL[j]$\leftarrow$ NearestNodeNotYetChosen($t_i$SortedList)
    end
    $t_{i+1} \leftarrow$ RandomSelect(RCL)
end
```

Algorithm 3: Greedy randomized construction

find the optimal size of r experimentally. After finishing the initial tour construction, the GRASP moves to the second phase, the local search phase. As mentioned before, we use the 2-opt search and the path-relinking method in the local search phase. Since the 2-opt search has high computational load, two advanced techniques and a specially designed data structure have been employed for the efficient implementation. Those are the Steiglitz and Weiner (Steiglitz and Weiner [20]), the *don't-look* bit (Bentley [1]) technique, and the reverse array data structure. These techniques are described in Algorithm 4. For the Steiglitz and Weiner technique, we need to keep the lists of remaining cities for each city c in order of increasing distance from c. This approach need the $\Theta(n^2)$ space and the $\Theta(n^2 \log n)$ setup time. Thus, Johnson and McGeoch [12] suggested to reduce those overheads by storing only k nearest neighbors for each city. The size of k is a control variable for the 2-opt improved tour quality.

```
Data: Initial tour $t$, Nearest neighbor to check $k$
Result: Improved tour $t'$

Set Bit[ ] = 1
for $i$ = 0, ..., Number of Nodes $-1$ do
    if Bit[i] is 1 then
        for $j$ = 0, ..., $k-1$ do
            $c_1 \leftarrow t_i$
            $c_2 \leftarrow t_{i+1}$
            $c_3 \leftarrow t_i$SortedList[j]
            $p_3 \leftarrow$ Fix-and-Follow($c_3$)
            $c_4 \leftarrow t_{p_3+1}$
            $t' \leftarrow$ReverseList$((c_1, c_2), \{c_3, c_4\})$
            if Cost($t'$)<Cost($t$) then
                return 2-optSearch($t'$)  /* Call 2-optSearch recursively
                    */
            end
        end
        Bit[i] $\leftarrow$ 0
    end
end
```

Algorithm 4: 2-opt search

```
Data: Tour $t$, Guide solution $t^*$
Result: Improved tour $t''$

$t' \leftarrow t$
$t'' \leftarrow t$
for $i$ = 0, ..., Number of Nodes $-1$ do
    if $t^*_i = t'_i$ then
        continue to next i
    end
    $p \leftarrow \mathrm{Map}[t^*_i]$
    $t'_p \leftrightarrow t'_i$
    if $Cost(t') < Cost(t'')$ then
        $t'' \leftarrow t'$
    end
end
```

Algorithm 5: Path-relinking

Johnson and McGeoch [12] claimed that $k = 40$ is the point of diminishing for TSPLIB instances. We will also determine the optimal size of k in Chapter VI of this thesis. The path-relinking uses the 2-opt neighborhood as a part of local search procedure. To keep the good guide solutions, we made a set of solutions which is called elite set. The size of elite set is another control variable. At first, the solutions from each end of iteration fill up the elite set. When the elite set is loaded, the path-relinking is performed between the current tour and the randomly selected guide solution from the elite set. As we can see at Algorithm 5, the path-relinking changes the position of each node to the position where it is placed in the guide tour. Indeed the path-relinking move is the same as the 2-swap move. The better solution found during the process is kept as an improved tour t''. If the cost of new tour t'' is smaller than the maximum cost of elite set solutions, the new tour t'' substitutes the most similar elite solution in the elite set. Thus, the number of solutions in the elite set is kept constantly, while the quality of the elite set is getting better. The whole process of the path-relinking is described at the mark 1 and 2 of Algorithm 2. The overall GRASP procedure is repeated until the given terminal conditions are satisfied. According to the computational results of Resende and Ribeiro [17], the GRASP indeed benefits greatly from the use of the path-relinking. We will see how the path-relinking intensifies the solution in Chapter VI of this thesis.

CHAPTER V

IMPLEMENTATION

To perform numerical experiments, the GRASP algorithm is coded in the C programming language. Since the program is written in ANSI C, it is portable to the multiple computer platforms. To implement the Steiglitz and Weiner technique, as introduced in Chapter II, Steiglitz and Weiner [20] proposed to store the list of remaining cities for each city c in order of increasing distance from c. Because $O(n^2)$ space is required to store the sorted list, it might be infeasible for the large problems. However, this space can be reduced to the size of $O(n)$ by limiting the number of the nearest neighbors stored in the sorted list. For the sorting, we applied the quick sort algorithm, which has the $O(n \log n)$ running time. The test instances are adopted from TSPLIB (http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/). There are several types of the distance data in the TSPLIB instances. If the distance data is not given explicitly, appropriate distance computations are required whose functions are introduced at TSPLIB. In this thesis, the distances between each pairs of cities are calculated once and saved in the memory during the preparation step. Thus, we can get the distance without further computation for the GRASP iteration. It helps to reduce the running time, but again it is impractical to save the distance data for the large instances ($>$ 5,000 cities). The following sections describe the C implementations of the reverse array data structure and the fix-and-follow technique employed in this thesis.

A. Reverse array

The reverse array operates the 2-opt improving move by swapping memory blocks between the original array and the reverse array. The memory block copy is executed

```
int tOrig[];   // Original array
int tReve[];   // Reverse array int
tTemp[];   // Temporary array for swap

tReveStartNode = tSizeofNode - tEndNode + 1;
tMemoryBlockSize = (tEndNode - tStartNode + 1) * sizeof(int);

memcpy(&tTemp, &tOrig[tStartNode], tMemoryBlockSize);
memcpy(&tOrig[tStartNode], &tReve[tReveStartNode],tMemoryBlockSize);
memcpy(&tReve[tReveStartNode], &tTemp, tMemoryBlockSize);
```

Fig. 5. C code for the reverse array implementation

using the C language function *memcpy*, which copies m bytes of the source to the destination. By using this technique, the *Swap* operation can be completed in $O(1)$ time. Figure 5 shows the C code implementation of the reverse array *Swap* operation.

B. Fix-and-follow

We introduced the fix-and-follow to correct the map array after the *Swap* operation. Because the memory block of the original array has been changed during the *Swap* operation, the map array contains an incorrect city information. Thus we need to find the actual information using the fix-and-follow method. Figure 6 is the C implementation of the fix-and-follow. In the *while* loop, we correct the map array until we get the right index of $t3$. At the end of the loop, we can find the actual index of $t3$. During the search period, the incorrect index information in the map array will be fixed only when they were found.

```
p3 = tMap[t3];   //t3 is candidate city

while( tOrig[p3] != t3 ) {
    pFollow = tMap[tOrig[p3]];
    tMap[tOrig[p3]] = p3;
    p3 = pFollow;
}

tMap[tOrig[p3]] = p3;
```

Fig. 6. C code for the fix-and-follow implementation

CHAPTER VI

COMPUTATIONAL RESULTS

As we mentioned in Chapter IV, there are three control variables for the GRASP process, the size of the restricted candidate list (RCL) r, the size of the elite set e, and the number of nearest neighbors k in 2-opt improving move search. In this chapter, we will see how these factors affect the final solution tour quality and the running time through computational result. Running times are based on the Pentium IV 2.2GHz CPU performance. The minimum running time does not indicate the total processing time, but the time when the smallest tour cost is first found. In the computational test, TSPLIB instances are divided into two groups which are the small and the large. If an instance is smaller than 1,000 cities, it is classified as small. Otherwise it belongs to large instances. The computational results are based on 100 trials and 10 trials for the small and the large instances, respectively. Both the small and the large cases have been iterated 10,000 times for each trial. The following formula has been used to compute the *gap* :

$$\text{Gap} = \frac{Bestsolution\ -\ Optimum}{Optimum}$$

In order to measure the effect of the number of nearest neighbors in 2-opt improving move search k, five different sizes of k from 10 to 50 have been tested for the selected TSPLIB instances. Table 2 shows the average *gap* and running time for each value k of this test. Similarly, we tested the same instances for the different size $k = np$, where n is the size of instance and p is the percent. In Table 3, p is increased from 10% to 50%. Thus the number of nearest neighbors k will be $k = n \times p$. As we can see in Table 2 and Table 3, the tour cost is not so dependent on the size of k.

But the bigger k we choose, the greater running time we get. Therefore $k = 40$ can be a reasonable choice for the number of nearest neighbors in 2-opt improving move search. This result supports Johnson and McGeoch [12], who claimed that $k = 40$ is the point of diminishing for TSPLIB instances. The same selected instances have been tested for different sizes of the elite set e to decide the appropriate size of elite set e. Test results indicate that the tour quality is not strongly related to the size of the elite set. However, since the running time increases along with the elite set size, $e = 5$ can be the best choice. Those results are shown in Table 4. The last control variable is the the size of the restricted candidate list (RCL) r. Using the same method and instances with the other variables, we changed the size of RCL r from 3 to 9, increased by 2. The outcomes of this trial are shown in Table 5, which indicates that as r is increased, *gap* and the running time are also increased. This result is quite intuitive because if r is extended, then relatively far neighbors can be included to the RCL. If r is too small, however, we can not get enough various tours for the path-relinking procedure. Thus we can conclude that $r = 3$ as the size of RCL gives sufficient flexibility to construct various tours and at the same time it guarantees high tour quality in a short running time. In order to test the factor interactions between the three control variables, we used three-factor ANOVA model I with $\alpha = 0.1$ Type I error. From the analysis, we can conclude that the three variables do not interact. Therefore we can use the best values of each control variable together to get the best solution. The more detail about the three-factor analysis can be found in Appendix A. Table 6 shows the computational result of each TSPLIB instance using the preferred values for the variables r, e, and k. Thus we set the values $r = 3$, $e = 5$, and $k = 40$. In Table 6, Pre time denotes the time spent to prepare the GRASP operation such as data file loading, distance computation, and sorting. The instance type specifies how the distance data is given. For each data type except the explicit data type, there is a special distance function defined by TSPLIB.

Table 2. Gap and running time for each k

Name	$k = 10$	$k = 20$	$k = 30$	$k = 40$	$k = 50$
a280	0.058	0.054	0.053	0.054	0.054
	3.8	3.9	3.9	3.9	4.2
ch150	0.032	0.032	0.032	0.032	0.032
	1.7	1.8	1.8	1.8	1.8
lin318	0.060	0.053	0.053	0.051	0.052
	5.5	5.7	5.8	5.8	5.7
gr431	0.057	0.055	0.055	0.054	0.054
	8.6	8.8	8.8	8.9	8.9
gr666	0.082	0.075	0.075	0.076	0.075
	17	17.3	17.4	17.5	17.6
rat783	0.082	0.082	0.082	0.082	0.082
	25.2	25.4	25.3	25.3	25.4
pr1002	0.089	0.083	0.085	0.082	0.085
	2.8	2.7	3	3.2	2.9
pcb1173	0.092	0.089	0.091	0.091	0.090
	4	3.1	3.6	3.7	3.6
rl1304	0.114	0.088	0.084	0.081	0.083
	4.5	4.8	5.2	4.4	5.3
u1432	0.087	0.089	0.087	0.086	0.087
	4.6	4.2	4.9	4	4.1
fl1577	0.119	0.105	0.093	0.086	0.087
	5.8	6.2	6.5	6.1	5.9
vm1748	0.097	0.087	0.084	0.082	0.083
	7.7	8.8	8.1	9.9	7.8
rl1889	0.113	0.090	0.088	0.090	0.089
	10	9.6	12	11.8	11.8
d2103	0.133	0.118	0.121	0.121	0.123
	9.4	9.6	10.5	10.4	11.1
pr2392	0.106	0.101	0.101	0.102	0.100
	13.6	13	13.2	12.3	12.5

Table 3. Gap and running time for each p%

Name	$p = 10$	$p = 20$	$p = 30$	$p = 40$	$p = 50$
a280	0.054	0.054	0.054	0.054	0.053
	3.9	3.9	3.9	3.9	4.2
ch150	0.032	0.032	0.031	0.030	0.031
	1.7	1.7	1.7	1.7	1.7
lin318	0.052	0.052	0.053	0.051	0.052
	5.4	5.5	5.5	5.5	5.5
gr431	0.054	0.054	0.053	0.054	0.053
	8.9	9	9.2	9.2	9.2
gr666	0.074	0.075	0.074	0.075	0.075
	17.9	18	18	18.1	18.1
rat783	0.082	0.082	0.082	0.082	0.082
	23.2	23.1	23	23.1	23.2
pr1002	0.083	0.082	0.084	0.082	0.081
	3.2	2.7	2.9	3.1	2.8
pcb1173	0.094	0.091	0.090	0.091	0.093
	3.8	3.6	3.7	3.8	3.6
rl1304	0.085	0.081	0.077	0.080	0.083
	5.7	4.9	4.7	5.4	4.9
u1432	0.087	0.088	0.090	0.087	0.087
	4.7	4.6	4.9	3.7	4.3
fl1577	0.063	0.062	0.063	0.064	0.062
	6.8	7.5	8.4	7.7	7.2
vm1748	0.082	0.083	0.083	0.083	0.082
	9.3	9.6	9.3	8.8	9.3
rl1889	0.085	0.086	0.085	0.086	0.087
	9.9	10.1	9.9	10.8	11
d2103	0.120	0.123	0.121	0.120	0.123
	10.8	9.1	12.2	11	10.7
pr2392	0.101	0.101	0.100	0.102	0.101
	11.9	12.3	11.5	11.6	12.9

Table 4. Gap and running time for each elite set size e

Name	$e = 5$	$e = 10$	$e = 15$	$e = 20$	$e = 25$
a280	0.054	0.054	0.054	0.054	0.054
	3.9	4	4.1	4.2	4.7
ch150	0.032	0.031	0.031	0.032	0.033
	1.7	1.8	1.8	1.9	1.9
lin318	0.052	0.053	0.052	0.053	0.053
	5.5	5.6	5.7	5.8	6
gr431	0.053	0.053	0.054	0.053	0.054
	9.4	9.5	9.8	9.9	10.2
gr666	0.074	0.075	0.074	0.074	0.074
	18.3	18.7	19	19.5	19.8
rat783	0.082	0.082	0.083	0.082	0.081
	26.3	26.7	27.2	27.5	28.1
pr1002	0.082	0.081	0.081	0.084	0.082
	4.1	4.1	4.2	4.2	4.4
pcb1173	1.091	0.092	0.091	0.089	0.090
	4.8	4.9	5	5	5.1
rl1304	0.081	0.084	0.080	0.084	0.082
	6.8	6.9	7	7	7.2
u1432	0.087	0.089	0.088	0.087	0.088
	6	6.1	6.1	6.2	6.3
fl1577	0.850	0.846	0.868	0.857	0.847
	2.2	2.2	2.3	2.3	2.4
vm1748	0.080	0.082	0.083	0.081	0.082
	11.7	11.9	12	12.1	12.2
rl1889	0.086	0.087	0.090	0.087	0.087
	13.9	14	14.2	14.2	14.4
d2103	0.756	0.754	0.754	0.754	0.754
	3.3	3.3	3.4	3.4	3.5
pr2392	0.100	0.101	0.102	0.101	0.100
	15.8	16	16.2	16.3	16.6

Table 5. Gap and running time for each RCL size r

Name	$r = 3$	$r = 5$	$r = 7$	$r = 9$
a280	0.049	0.061	0.061	0.057
	3.9	5.4	6.4	7.1
ch150	0.03	0.037	0.033	0.039
	1.7	2.3	2.7	3
lin318	0.051	0.056	0.056	0.058
	5.5	7.4	8.6	9.5
gr431	0.053	0.057	0.056	0.057
	8.8	12.2	14.5	16.2
gr666	0.08	0.085	0.083	0.082
	17	23.1	32.8	30.6
rat783	0.083	0.088	0.087	0.088
	22.9	31.5	37.3	41.7
pr1002	0.08	0.088	0.089	0.088
	3.9	5.5	6.5	7.4
pcb1173	0.089	0.099	0.098	0.098
	4.8	6.6	7.7	8.5
rl1304	0.087	0.092	0.096	0.1
	6.6	8.8	10.2	11.3
u1432	0.088	0.096	0.095	0.096
	5.9	8.1	9.7	10.8
fl1577	0.101	0.128	0.127	0.142
	7.1	9.8	11.5	12.8
vm1748	0.082	0.09	0.091	0.091
	11.8	15.4	17.5	19.2
rl1889	0.091	0.096	0.099	0.093
	13.7	18.2	21	23
d2103	0.117	0.127	0.123	0.123
	12.6	17.4	20.6	23.2
pr2392	0.1	0.111	0.113	0.111
	15.9	21.9	26	28.9

Table 6. Performance

Name	Cities	Type	Optimum	Gap			Time		
				Min	Ave	Max	Pre	Min	Ave
a280	280	EUC_2D	2579	0.021	0.036	0.047	0.0	0.0	4.0
ali535	535	GEO	202339	0.040	0.053	0.062	0.1	0.2	18.0
att48	48	ATT	10628	0.000	0.000	0.000	0.0	0.0	0.0
att532	532	ATT	27686	0.040	0.051	0.057	0.2	0.3	13.6
bayg29	29	MATRIX	1610	0.000	0.000	0.000	0.0	0.0	0.0
bays29	29	MATRIX	2020	0.000	0.000	0.000	0.0	0.0	0.0
berlin52	52	EUC_2D	7542	0.000	0.000	0.000	0.0	0.0	0.0
bier127	127	EUC_2D	118282	0.001	0.010	0.019	0.0	0.1	1.7
brazil58	58	MATRIX	25395	0.000	0.000	0.000	0.0	0.0	0.0
burma14	14	GEO	3323	0.000	0.000	0.000	0.0	0.0	0.0
ch130	130	EUC_2D	6110	0.003	0.012	0.021	0.0	0.0	1.6
ch150	150	EUC_2D	6528	0.007	0.016	0.026	0.0	0.0	1.7
d198	198	EUC_2D	15780	0.007	0.014	0.018	0.0	0.2	3.0
d493	493	EUC_2D	35002	0.034	0.045	0.050	0.1	0.2	12.6
d657	657	EUC_2D	48912	0.050	0.058	0.064	0.3	0.3	19.3
d1291	1291	EUC_2D	50801	0.056	0.077	0.087	1.1	3.6	56.2
d1655	1655	EUC_2D	62128	0.079	0.087	0.092	1.8	5.8	84.0
d2103	2103	EUC_2D	80450	0.109	0.111	0.114	3.0	3.2	129.2
dantzig42	42	MATRIX	699	0.000	0.000	0.000	0.0	0.0	0.0
eil51	51	EUC_2D	426	0.000	0.001	0.002	0.0	0.0	0.3
eil76	76	EUC_2D	538	0.002	0.012	0.020	0.0	0.0	0.7
eil101	101	EUC_2D	629	0.003	0.019	0.029	0.0	0.1	1.1
fl417	417	EUC_2D	11861	0.007	0.013	0.019	0.1	0.5	7.2
fl1400	1400	EUC_2D	20127	0.025	0.030	0.039	1.2	10.0	67.1
fl1577	1577	EUC_2D	22249	0.054	0.068	0.079	1.6	6.7	75.8
fri26	26	MATRIX	937	0.000	0.000	0.000	0.0	0.0	0.0
gil262	262	EUC_2D	2378	0.023	0.038	0.048	0.0	0.1	4.1
gr17	17	MATRIX	2085	0.000	0.000	0.000	0.0	0.0	0.0
gr21	21	MATRIX	2707	0.000	0.000	0.000	0.0	0.0	0.0
gr24	24	MATRIX	1272	0.000	0.000	0.000	0.0	0.0	0.0
gr48	48	MATRIX	5046	0.000	0.000	0.000	0.0	0.0	0.1
gr96	96	GEO	55209	0.000	0.003	0.011	0.0	0.0	1.0
gr120	120	MATRIX	6942	0.003	0.010	0.017	0.0	0.0	1.6
gr137	137	GEO	69853	0.002	0.011	0.020	0.0	0.1	1.8
gr202	202	GEO	40160	0.018	0.029	0.036	0.0	0.0	2.9
gr229	229	GEO	134602	0.014	0.025	0.034	0.0	0.0	3.2

Table 6. Continued

Name	Cities	Type	Optimum	Gap			Time		
				Min	Ave	Max	Pre	Min	Ave
gr431	431	GEO	171414	0.033	0.044	0.051	0.1	0.0	8.9
gr666	666	GEO	294358	0.052	0.064	0.071	0.2	0.1	17.9
hk48	48	MATRIX	11461	0.000	0.000	0.000	0.0	0.0	0.1
kroA100	100	EUC_2D	21282	0.000	0.001	0.004	0.0	0.0	1.0
kroB100	100	EUC_2D	22141	0.000	0.003	0.008	0.0	0.0	1.0
kroC100	100	EUC_2D	20749	0.000	0.001	0.005	0.0	0.0	0.8
kroE100	100	EUC_2D	22068	0.000	0.003	0.010	0.0	0.0	1.1
kroA150	150	EUC_2D	26524	0.000	0.015	0.021	0.0	0.0	1.7
kroB150	150	EUC_2D	26130	0.001	0.012	0.022	0.0	0.2	1.8
kroA200	200	EUC_2D	29368	0.007	0.016	0.026	0.0	0.1	2.7
kroB200	200	EUC_2D	29437	0.014	0.026	0.040	0.0	0.1	2.6
lin105	105	EUC_2D	14379	0.000	0.000	0.004	0.0	0.0	0.5
lin318	318	EUC_2D	42029	0.026	0.037	0.047	0.1	0.1	5.4
nrw1379	1379	EUC_2D	56638	0.071	0.075	0.078	1.3	4.9	69.8
p654	654	EUC_2D	34643	0.005	0.008	0.013	0.3	1.5	16.2
pcb442	442	EUC_2D	50778	0.037	0.047	0.052	0.1	0.0	8.9
pcb1173	1173	EUC_2D	56892	0.078	0.083	0.087	0.9	11.0	47.2
pcb3038	3038	EUC_2D	137694	0.093	0.097	0.099	6.8	44.2	243.3
pr76	76	EUC_2D	108159	0.000	0.002	0.007	0.0	0.0	0.8
pr107	107	EUC_2D	44303	0.000	0.002	0.009	0.0	0.0	1.0
pr124	124	EUC_2D	59030	0.000	0.000	0.003	0.0	0.0	0.5
pr136	136	EUC_2D	96772	0.004	0.012	0.025	0.0	0.0	1.4
pr144	144	EUC_2D	58537	0.000	0.000	0.000	0.0	0.0	0.3
pr152	152	EUC_2D	73682	0.000	0.003	0.008	0.0	0.0	1.7
pr226	226	EUC_2D	80369	0.000	0.003	0.008	0.0	0.1	2.7
pr264	264	EUC_2D	49135	0.001	0.016	0.031	0.0	0.1	3.6
pr299	299	EUC_2D	48191	0.019	0.034	0.041	0.0	0.1	4.6
pr439	439	EUC_2D	107217	0.020	0.037	0.046	0.1	1.0	9.3
pr1002	1002	EUC_2D	259045	0.063	0.073	0.078	0.7	0.0	37.2
pr2392	2392	EUC_2D	378032	0.090	0.094	0.098	4.2	12.1	157.3
rat99	99	EUC_2D	1211	0.001	0.011	0.021	0.0	0.0	1.0
rat195	195	EUC_2D	2323	0.025	0.039	0.051	0.0	0.0	2.3
rat575	575	EUC_2D	6773	0.057	0.066	0.071	0.2	0.0	13.2
rat783	783	EUC_2D	8806	0.063	0.074	0.078	0.4	0.6	23.9
rd100	100	EUC_2D	7910	0.000	0.003	0.009	0.0	0.0	1.0
rd400	400	EUC_2D	15281	0.040	0.051	0.058	0.1	0.2	8.2
rl1304	1304	EUC_2D	252948	0.062	0.069	0.078	1.1	19.3	67.4

Table 6. Continued

Name	Cities	Type	Optimum	Gap			Time		
				Min	Ave	Max	Pre	Min	Ave
rl1323	1323	EUC_2D	270199	0.061	0.066	0.072	1.2	27.8	72.1
rl1889	1889	EUC_2D	316536	0.065	0.076	0.080	2.6	35.6	135.8
rl5915	5915	EUC_2D	565530	0.095	0.102	0.107	35.0	142.6	779.8
rl5934	5934	EUC_2D	556045	0.100	0.108	0.110	36.0	82.9	829.1
si175	175	MATRIX	21407	0.001	0.002	0.003	0.0	0.0	2.1
si535	535	MATRIX	48450	0.004	0.006	0.008	0.1	0.3	12.2
si1032	1032	MATRIX	92650	0.002	0.005	0.006	0.4	1.8	34.3
st70	70	EUC_2D	675	0.000	0.000	0.001	0.0	0.0	0.1
swiss42	42	MATRIX	1273	0.000	0.000	0.000	0.0	0.0	0.0
ts225	225	EUC_2D	126643	0.001	0.005	0.008	0.0	0.0	2.7
tsp225	225	EUC_2D	3916	0.002	0.020	0.030	0.0	0.1	4.2
u159	159	EUC_2D	42080	0.000	0.008	0.018	0.0	0.0	1.9
u574	574	EUC_2D	36905	0.050	0.060	0.067	0.2	0.1	14.7
u724	724	EUC_2D	41910	0.054	0.067	0.073	0.3	0.2	20.8
u1060	1060	EUC_2D	224094	0.061	0.071	0.076	0.8	0.6	44.6
u1432	1432	EUC_2D	152970	0.076	0.081	0.084	1.4	0.6	59.1
u1817	1817	EUC_2D	57201	0.109	0.115	0.118	2.3	15.1	95.3
u2152	2152	EUC_2D	64253	0.102	0.114	0.119	3.2	30.3	122.1
u2319	2319	EUC_2D	234256	0.038	0.038	0.040	4.0	6.3	134.5
ulysses22	22	GEO	7013	0.000	0.000	0.000	0.0	0.0	0.0
vm1084	1084	EUC_2D	239297	0.050	0.064	0.068	0.8	3.4	50.2
vm1748	1748	EUC_2D	336556	0.067	0.076	0.078	2.1	8.6	116.2

CHAPTER VII

CONCLUSION

The main contribution of this thesis is that we introduced a new efficient data structure, the reverse array, which can be applied with the Steiglitz and Weiner technique. The simplicity of the algorithm is one of the important concerns in the heuristic method. In this aspect, we claim that the GRASP algorithm applied in this thesis is a successful algorithm. All the heuristic methods used in our GRASP procedure have simple algorithm structure. We applied the nearest neighbor search method for the construction phase of GRASP and the 2-opt and the path-relinking (PR) methods for the local search phase. Those well-known methods are easily understandable and implementable.

Computational experiments of the previous chapter have proven that GRASP guarantees a near optimal solution for most of the TSPLIB instances. Because GRASP has a randomized adaptive attribute, the solutions of each trial have large difference tour values.

The quick 2-opt search techniques and the efficient data structure really expedite the tour improvement. GRASP has a merit in the running time, especially with a large size instance. In comparison with Lin-Kernighan (LK), the running time increase rate is slow. Thus GRASP gives a solution within the 10% of optimum in a relatively short time. However, the tour improvement capability of the 2-opt search exceedingly decreases as the size of instance grows [21]. Therefore more intensified local search method needs to be employed. We leave this as a future work.

In the path-relinking, we performed the PR operation for every iteration's solution. But Resende and Ribeiro [17] suggested alternative schemes. Finding a more intensive PR scheme is another issue for the future work.

REFERENCES

[1] J. J. Bentley, *Fast algorithms for geometric traveling salesman problems*, ORSA Journal on Computing, 4 (Fall 1992), pp. 387–411.

[2] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Tech. Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1976.

[3] M. Chrobak, T. Szymacha, and A. Krawczyk, *A data structure useful for finding hamiltonian cycles*, Theoretical Computer Science, 71 (1990), pp. 419–424.

[4] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver, *Combinatorial optimization*, A Wiley-Interscience Publication., New York, 1998.

[5] G. Croes, *A method for solving traveling salesman problems*, Operations Res., 6 (1958), pp. 791–812.

[6] T. Feo and M. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization, 6 (1995), pp. 109–133.

[7] P. Festa and M. G. Rensende, *GRASP: An annotated bibliography*, Tech. Report 00.1.1, AT&T Labs Research, Floham Park, NJ, 2000.

[8] H. Gabow and R. Tarjan, *Faster scaling algorithms for general graph-matching problems*, J. Assoc. Comput. Mach, 38 (1991), pp. 815–853.

[9] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W.H. Freeman & Co., New York, 1979.

[10] F. Glover and M. Laguna, *Tabu search*, Kluwer Academic Publishers, Boston, 1997.

[11] K. Helsgaun, *An effective implementation of the Lin-kernighan traveling salesman heuristic*, European Journal of Operational Research, 126 (2000), pp. 106–130.

[12] D. Johnson and L. McGeoch, The traveling salesman problem: A case study in local optimization, in *Local search in combinatorial optimization*, E. Aarts and J. Lenstra, eds., John Wiley & Sons, Chichester, UK, 1997, pp. 215–310.

[13] M. Laguna and R. Martí, *GRASP and path relinking for 2-layer straight line crossing minimization*, INFORMS Journal on Computing, 11 (1999), pp. 44–52.

[14] S. Lin and B. Kernighan, *An effective heuristic algorithm for the traveling-salesman problems*, Operations Res., 21 (1973), pp. 498–516.

[15] H. Ong and J. Moore, *Worst-case analysis of two traveling salesman heuristics*, Operations Res. Lett., 2 (1984), pp. 273–277.

[16] C. Osterman and C. Rego, *The satellite list and new data structures for symmetric traveling salesman problems*, Tech. Report HCES-06-03, Hearin Center for Enterprise Science, School of Business Administration, University of Mississippi, University, MS, 2003.

[17] M. G. Resende and C. C. Ribeiro, *GRASP and path-relinking: Recent advances and applications*, Tech. Report , AT&T Labs Research, Floham Park, NJ, 2003.

[18] D. Rosenkrantz, R. Stearns, and P. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal of Computing, 6 (1977),

pp. 563–581.

[19] D. Sleator and R. Tarjan, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach, 32 (1985), pp. 652–686.

[20] K. Steiglitz and P. Weiner, *Some improved algorithm for computer solution of the traveling salesman problem*, in Proceedings of the 6th Annual Allerton Conf. on Communication, Control, and Computing, University of Illinois, Urbana, 1968, pp. 814–821.

[21] H. Tsai, J. Yang, Y. Tsai, and C. Kao, *An evolutionary algorithm for large traveling salesman problems*, IEEE Transactions on Systems, Man and Cybernetics, 34 (2004), pp. 1718–1729.

APPENDIX A

THREE-FACTOR ANALYSIS

For the three-factor analysis, we designed the experiment as 10 test runs with 10,000 iterations for each test run. The test instance $ch150$ is selected from TSPLIB. The three factors are the three control variables of GRASP, the size of the restricted candidate list (RCL) r, the size of the elite set e, and the number of nearest neighbors k in 2-opt improving move search. Table 7 contains the test results of this experiment. We set the Type I error $\alpha = 0.1$. Using the Kimball inequality for the family level of significance α, we calculated $\alpha_i = 0.015$ for each of the seven tests of the three-factor study. Thus, we can get each percentile value from the F distribution with $\alpha = 0.015$. Table 8 shows the ANOVA results. Since the test statistic for the three-factor interactions $F_{erk} = 1.272$ is less than F(0.985; 48, 900)=1.5144, we can say that there are no three-factor interactions between the three factors. For the same reason, there are also no two-factor interactions between each pair of variables.

Table 7. Experiments for the 3-factor analysis

Values	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
e5r3k10	0.015	0.009	0.01	0.027	0.018	0.023	0.013	0.015	0.013	0.015
e5r3k20	0.015	0.015	0.02	0.018	0.022	0.016	0.014	0.014	0.023	0.016
e5r3k30	0.017	0.009	0.023	0.023	0.009	0.015	0.015	0.016	0.018	0.02
e5r3k40	0.015	0.025	0.02	0.018	0.014	0.011	0.019	0.019	0.018	0.02
e5r3k50	0.019	0.008	0.017	0.018	0.023	0.018	0.021	0.02	0.021	0.023
e5r5k10	0.023	0.024	0.03	0.016	0.02	0.012	0.018	0.025	0.019	0.024
e5r5k20	0.022	0.018	0.023	0.022	0.023	0.022	0.013	0.017	0.023	0.02
e5r5k30	0.015	0.024	0.018	0.024	0.02	0.018	0.006	0.018	0.023	0.02
e5r5k40	0.017	0.018	0.026	0.017	0.008	0.019	0.023	0.022	0.024	0.024
e5r5k50	0.025	0.008	0.015	0.022	0.017	0.022	0.013	0.008	0.022	0.028
e5r7k10	0.015	0.019	0.014	0.016	0.01	0.011	0.026	0.023	0.022	0.02
e5r7k20	0.021	0.017	0.026	0.022	0.025	0.028	0.025	0.022	0.023	0.021
e5r7k30	0.02	0.015	0.015	0.02	0.025	0.024	0.019	0.023	0.017	0.021
e5r7k40	0.018	0.022	0.017	0.015	0.028	0.027	0.024	0.024	0.011	0.023
e5r7k50	0.017	0.026	0.023	0.026	0.02	0.018	0.019	0.022	0.021	0.024
e5r9k10	0.018	0.024	0.022	0.013	0.022	0.03	0.023	0.033	0.017	0.015
e5r9k20	0.022	0.018	0.015	0.024	0.013	0.014	0.015	0.023	0.024	0.024
e5r9k30	0.024	0.017	0.012	0.011	0.026	0.011	0.026	0.016	0.025	0.021
e5r9k40	0.021	0.02	0.014	0.017	0.027	0.02	0.013	0.019	0.024	0.018
e5r9k50	0.026	0.02	0.01	0.016	0.019	0.026	0.013	0.019	0.019	0.014
e10r3k10	0.013	0.016	0.016	0.019	0.019	0.016	0.021	0.013	0.021	0.013
e10r3k20	0.021	0.019	0.015	0.019	0.015	0.011	0.013	0.006	0.019	0.012
e10r3k30	0.014	0.02	0.018	0.016	0.016	0.017	0.019	0.022	0.017	0.009
e10r3k40	0.019	0.016	0.022	0.019	0.023	0.014	0.011	0.019	0.017	0.024
e10r3k50	0.013	0.017	0.015	0.009	0.02	0.018	0.016	0.016	0.02	0.019
e10r5k10	0.022	0.023	0.022	0.027	0.017	0.022	0.028	0.02	0.013	0.028
e10r5k20	0.015	0.025	0.016	0.018	0.026	0.021	0.023	0.02	0.023	0.019
e10r5k30	0.019	0.021	0.017	0.02	0.018	0.03	0.015	0.014	0.032	0.018
e10r5k40	0.021	0.017	0.011	0.023	0.022	0.024	0.023	0.017	0.023	0.014
e10r5k50	0.023	0.026	0.017	0.026	0.022	0.028	0.029	0.029	0.028	0.025
e10r7k10	0.017	0.024	0.021	0.02	0.015	0.017	0.013	0.018	0.022	0.022
e10r7k20	0.014	0.017	0.018	0.03	0.02	0.014	0.017	0.015	0.021	0.023
e10r7k30	0.008	0.019	0.016	0.011	0.022	0.012	0.025	0.021	0.021	0.028
e10r7k40	0.019	0.015	0.017	0.021	0.022	0.014	0.024	0.026	0.015	0.023
e10r7k50	0.024	0.018	0.028	0.01	0.016	0.011	0.024	0.022	0.022	0.024
e10r9k10	0.025	0.021	0.018	0.028	0.016	0.02	0.017	0.028	0.017	0.023
e10r9k20	0.023	0.026	0.019	0.02	0.016	0.021	0.014	0.022	0.02	0.017
e10r9k30	0.023	0.016	0.016	0.016	0.028	0.011	0.016	0.014	0.024	0.011
e10r9k40	0.017	0.023	0.015	0.025	0.022	0.025	0.02	0.013	0.013	0.02
e10r9k50	0.02	0.025	0.012	0.025	0.007	0.014	0.022	0.024	0.02	0.02

Table 7. Continued

Values	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
e15r3k10	0.019	0.022	0.016	0.014	0.021	0.02	0.014	0.013	0.022	0.016
e15r3k20	0.017	0.017	0.016	0.021	0.015	0.016	0.022	0.006	0.017	0.009
e15r3k30	0.009	0.021	0.023	0.015	0.021	0.017	0.016	0.018	0.019	0.017
e15r3k40	0.019	0.016	0.017	0.023	0.012	0.019	0.019	0.016	0.017	0.009
e15r3k50	0.014	0.017	0.013	0.016	0.018	0.013	0.012	0.016	0.02	0.014
e15r5k10	0.018	0.017	0.017	0.02	0.018	0.024	0.011	0.026	0.009	0.027
e15r5k20	0.023	0.026	0.023	0.023	0.028	0.02	0.019	0.026	0.023	0.021
e15r5k30	0.012	0.02	0.027	0.024	0.019	0.021	0.012	0.019	0.026	0.023
e15r5k40	0.013	0.024	0.029	0.026	0.018	0.019	0.018	0.025	0.011	0.014
e15r5k50	0.018	0.021	0.021	0.019	0.021	0.024	0.025	0.023	0.018	0.02
e15r7k10	0.019	0.023	0.022	0.022	0.021	0.018	0.014	0.019	0.02	0.024
e15r7k20	0.022	0.024	0.015	0.022	0.012	0.025	0.021	0.013	0.018	0.023
e15r7k30	0.028	0.025	0.02	0.019	0.021	0.019	0.025	0.01	0.021	0.028
e15r7k40	0.023	0.026	0.018	0.022	0.026	0.02	0.021	0.022	0.022	0.021
e15r7k50	0.011	0.023	0.018	0.012	0.027	0.022	0.022	0.007	0.023	0.024
e15r9k10	0.023	0.028	0.023	0.017	0.019	0.023	0.011	0.024	0.027	0.021
e15r9k20	0.025	0.026	0.021	0.025	0.015	0.015	0.023	0.024	0.028	0.027
e15r9k30	0.028	0.024	0.023	0.015	0.014	0.021	0.028	0.02	0.021	0.02
e15r9k40	0.025	0.016	0.028	0.029	0.023	0.02	0.023	0.027	0.019	0.002
e15r9k50	0.017	0.028	0.022	0.017	0.025	0.024	0.021	0.008	0.026	0.021
e20r3k10	0.025	0.017	0.021	0.017	0.018	0.018	0.018	0.009	0.015	0.02
e20r3k20	0.018	0.017	0.019	0.018	0.022	0.02	0.013	0.021	0.02	0.019
e20r3k30	0.014	0.01	0.015	0.017	0.014	0.013	0.013	0.025	0.022	0.015
e20r3k40	0.013	0.021	0.01	0.017	0.009	0.019	0.011	0.022	0.018	0.018
e20r3k50	0.009	0.016	0.008	0.017	0.017	0.018	0.011	0.019	0.017	0.015
e20r5k10	0.018	0.02	0.017	0.028	0.019	0.025	0.023	0.019	0.028	0.023
e20r5k20	0.028	0.021	0.021	0.021	0.02	0.018	0.01	0.024	0.019	0.019
e20r5k30	0.023	0.021	0.026	0.024	0.023	0.015	0.014	0.019	0.023	0.023
e20r5k40	0.018	0.024	0.025	0.025	0.022	0.024	0.018	0.016	0.028	0.024
e20r5k50	0.018	0.02	0.023	0.028	0.028	0.026	0.025	0.026	0.025	0.024
e20r7k10	0.009	0.021	0.03	0.023	0.022	0.016	0.02	0.02	0.019	0.027
e20r7k20	0.025	0.023	0.025	0.027	0.023	0.02	0.026	0.021	0.012	0.023
e20r7k30	0.017	0.02	0.026	0.029	0.019	0.026	0.019	0.027	0.012	0.026
e20r7k40	0.033	0.023	0.016	0.02	0.025	0.017	0.026	0.015	0.03	0.021
e20r7k50	0.011	0.022	0.023	0.022	0.024	0.013	0.021	0.02	0.009	0.016
e20r9k10	0.017	0.021	0.012	0.007	0.013	0.023	0.018	0.016	0.018	0.023
e20r9k20	0.017	0.023	0.025	0.02	0.016	0.02	0.017	0.015	0.014	0.019
e20r9k30	0.021	0.025	0.02	0.019	0.019	0.013	0.02	0.012	0.027	0.021
e20r9k40	0.018	0.013	0.023	0.024	0.024	0.02	0.021	0.023	0.013	0.021
e20r9k50	0.024	0.023	0.03	0.021	0.01	0.015	0.021	0.021	0.025	0.019

Table 7. Continued

Values	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
e25r3k10	0.017	0.02	0.011	0.019	0.014	0.019	0.015	0.021	0.026	0.016
e25r3k20	0.022	0.018	0.017	0.019	0.018	0.019	0.017	0.016	0.016	0.02
e25r3k30	0.019	0.022	0.017	0.016	0.015	0.022	0.014	0.015	0.011	0.012
e25r3k40	0.023	0.016	0.02	0.017	0.014	0.015	0.013	0.015	0.011	0.021
e25r3k50	0.023	0.019	0.021	0.019	0.019	0.012	0.014	0.018	0.019	0.014
e25r5k10	0.013	0.02	0.011	0.022	0.03	0.017	0.026	0.011	0.021	0.023
e25r5k20	0.023	0.021	0.024	0.022	0.016	0.027	0.023	0.031	0.016	0.02
e25r5k30	0.016	0.024	0.023	0.021	0.022	0.019	0.027	0.018	0.017	0.027
e25r5k40	0.023	0.019	0.025	0.017	0.016	0.025	0.023	0.024	0.024	0.024
e25r5k50	0.026	0.013	0.02	0.024	0.019	0.018	0.019	0.015	0.029	0.014
e25r7k10	0.022	0.019	0.015	0.02	0.022	0.027	0.019	0.018	0.027	0.017
e25r7k20	0.023	0.023	0.023	0.022	0.023	0.014	0.021	0.019	0.027	0.021
e25r7k30	0.024	0.019	0.018	0.028	0.021	0.008	0.023	0.016	0.015	0.018
e25r7k40	0.022	0.018	0.025	0.019	0.025	0.014	0.013	0.013	0.024	0.017
e25r7k50	0.024	0.019	0.031	0.022	0.014	0.016	0.03	0.016	0.025	0.02
e25r9k10	0.02	0.023	0.021	0.017	0.011	0.018	0.021	0.015	0.02	0.012
e25r9k20	0.02	0.018	0.025	0.019	0.018	0.021	0.019	0.019	0.025	0.023
e25r9k30	0.022	0.022	0.019	0.019	0.019	0.019	0.027	0.016	0.023	0.025
e25r9k40	0.025	0.015	0.021	0.021	0.012	0.009	0.018	0.031	0.025	0.027
e25r9k50	0.014	0.023	0.021	0.02	0.027	0.024	0.025	0.027	0.021	0.023

Table 8. ANOVA table

Source	Sum of Squares	DF	Mean Square	F	Percentile
SSe	8.05E-05	4	2.01E-05	0.929	3.1026
SSr	2.49E-03	3	8.30E-04	38.299	3.5067
SSk	6.13E-05	4	1.53E-05	0.707	3.1026
er	4.49E-04	12	3.75E-05	1.728	2.0976
ek	2.26E-04	16	1.41E-05	0.651	1.9321
rk	9.53E-05	12	7.95E-06	0.367	2.0976
erk	1.32E-03	48	2.76E-05	1.272	1.5144
Error	1.95E-02	900	2.17E-05		

VITA

Seung Ho Lee

226X Zachry Engineering Center
3131 TAMU
College Station, TX 77843-3131
mountlee@tamu.edu

EDUCATION

∗ Texas A&M University, College Station, TX

Master of Science in Industrial Engineering, May 2005

∗ Korea University, Seoul, Korea

Bachelor of Engineering in Industrial Engineering, February 1999

PROFESSIONAL EXPERIENCE

∗ 10DR Co., Ltd., Seoul, Korea, November 2002–May 2003

Freelancer Software Engineer

∗ A.P.Technology Inc., Seoul, Korea, February 1999–June 2002

Software Engineer