Solving Traveling Salesman Problem With a Non-complete Graph

by

Mahsa Sadat Emami Taba

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2009

Author’s Declaration

Abstract

One of the simplest, but still NP-hard, routing problems is the *Traveling Salesman Problem* (TSP). In the TSP, one is given a set of cities and a way of measuring the distance between cities. One has to find the shortest tour that visits all cities exactly once and returns back to the starting city. In state-of-the-art algorithms, they all assume that a complete graph is given as an input. However, for very large graphs, generating all edges in a complete graph, which corresponds to finding shortest paths for all city pairs, could be time-consuming. This is definitely a major obstacle for some real-life applications, especially when the tour needs to be generated in real-time.

The objective, in this thesis, is to find a near-optimal TSP tour with a reduced set of edges in the complete graph. In particular, the following problems are investigated: which subset of edges can be produced in a shorter time comparing to the time for generating the complete graph? Is there a subset of edges in the complete graph that results in a better near-optimal tour than other sets? With a non-complete graph, which improvement algorithms work better?

In this thesis, we study six algorithms to generate subsets of edges in a complete graph. To evaluate the proposed algorithms, extensive experiments are conducted with the well-known TSP data in a TSP library. In these experiments, we evaluate these algorithms in terms of tour quality, time and scalability.

Acknowledgements

I would like to express my great gratitude to my supervisor, Professor Edward Chan for his constant support and guidance. He introduced me to the field of combinatorial optimization and he enormously helped in the evolution and refinement of the ideas presented in this thesis by his constant feedback and supervision.

I would also like to thank my readers, Professors Shai Ben-David and Jonathan Buss, for their valuable comments that have significantly improved this thesis. I would also like to thank to the School of Computer Science in University of Waterloo.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Solving Traveling Salesman Problem With a non-complete Graph

One of the NP-hard routing problems is the Traveling Salesman Problem (TSP). In combinatorial optimization, TSP has been an early proving ground for many approaches, including more recent variants of local optimization techniques such as simulated annealing, tabu search, neural networks, and genetic algorithms.

The TSP is defined as follows: given a complete graph $G = (V, E, w)$, where V and E are sets of vertices (cities) and edges, respectively, and w: $E \rightarrow \mathcal{R}^{\geq 0}$**,** that is, w is a length function from the set of edges to a set of non-negative real numbers**.** A solution to the TSP is the cheapest Hamiltonian cycle of G. A Hamiltonian cycle is a cycle that visits each node in a graph exactly once.

From mail delivery to network architect, the TSP is applicable to many real-life problems. In all these instances, the cost or distance between locations, whether they are cities, buildings or nodes in a network, is known. With this information, the primary goal is to find an optimal tour. One of the main assumptions regarding the TSP is that the input is a complete graph. Computing a complete graph can be done with the following method. For each $s \in V$, one can compute a shortest path tree rooted at s, for example, with the Dijkstra's algorithm. Let us call this the *brute-force* method. This method may not be very fast, especially when the graph is large. In addition, some complete graphs are too large to fit in the main memory and the algorithms such as Dijkstra's algorithm cannot handle it. In [14] and [15] the authors investigate the problem of how to evaluate a collection of shortest path queries on massive graphs that are too big to fit in the main memory.

In state-of-the-art algorithms, they all assume that a complete graph is given as an input. There has not been any work that uses a non-complete graph as an input to find the TSP. In this thesis, we try to solve the TSP without having a complete graph. More specifically, we attempt to find, with a subset of edges in the graph, a non-optimal TSP tour with its quality comparable to the optimal tour. Our main focus is to find out having a subset of edges, how much it affects the tour quality compare to having a complete graph. Our goal is to figure out which subset of edges can be produced in a shorter time comparing to the time for generating the complete graph while producing a near-optimal TSP tour using that subset? Several algorithms are studied for generating subsets of edges in a complete graph. A subset of edges is considered to be *good* if the subset has the two following aspects:

- They can be computed very fast
- They can be used to produce a good or near-optimal TSP tour.

The input to these introduced non-complete graph generating algorithms is a graph with a set of vertices (their coordinates) and a way of measuring between the vertices (which is usually Euclidean distance). The output of these algorithms is a subset of edges (the source and destination vertices' coordinates of each edge and the weight of that edge). This subset can be used as an input to any TSP algorithms unlike the typical input to TSP algorithms which is a complete graph (a matrix of all pair wise distances). In this thesis we have used the *Greedy* algorithm to generate the TSP tour. Since the input to the Greedy algorithm is not a complete graph, the result may not be a complete TSP tour. If the result is not a complete tour and is a set of disjoint simple paths, we use *an Initial Tour Construction* algorithm to generate the TSP tour. After generating the TSP tour, using the same subset of edges that we used to generate the TSP tour, we improve the tour using a modified version of the 2-Opt improvement algorithm. We have introduced three modified versions of the 2-Opt algorithm. In these three algorithms, we use the same subset of edges that we used in generating the TSP tour to improve the TSP tour.

The rest of the thesis is organized as follows. Chapter 2 reviews the existing literature on NP-complete rout query problems especially the Traveling Salesman Problem. Chapter 3 presents our six introduced non-complete graph generating algorithms. Chapter 4 gives details regarding the TSP tour construction algorithms that we have used in this thesis. Chapter 5 explains the three introduced modified versions of the TSP. Chapter 6 gives details on our implementation and the TSP instances that we have used. Chapter 7 provides some experimental results of the proposed algorithms. Chapter 8 gives our final conclusions and suggested future work.

1.2 Terminology

Before proceeding to the description of algorithms, let us study the definitions of frequently used terms. Terms that are not defined here are common concepts in graph theory (such as vertices, edges, paths, and trees), which can be found in any graph theory resource [12].

Definition 1. Simple Graph

A simple graph is an undirected graph that has no loops and no more than one edge between any two different vertices.

Definition 2. Complete Graph

A complete graph is a simple graph in which every pair of distinct vertices is connected by an edge. The complete graph on n vertices has n vertices and $n(n\text{-}1)/2$ edges. Complete graphs have the feature that each pair of distinct vertices has an edge connecting them.

Definition 3. Shortest Path

Let P_{uv} be a path from u to v in G; then v is *reachable* from u, and u is *back-traceable* from v. All vertices reachable from u including itself in G are u's *descendants,* denoted as **des**(u). Consequently, all vertices back-traceable from v including itself in G are v's *ancestors*, denoted as **anc**(v).

A path P_{uv} is said to be a *shortest path*, denoted as SP_{uv}, if it is not longer than any other possible path P^*_{uv}. Given any $v \in \mathbf{V}$, v could have more than one shortest path from a source *s* in *G*, and all *v*'s shortest paths are of the same shortest distance. The shortest distance from *u* to *v* in *G* is denoted as d_{uv}.

Definition 4. Shortest-Path Tree (SPT)

Given a digraph $G = (V, E, w)$, a tree rooted at a source vertex *s*, denoted as T_s, is an SPT if $\forall v \neq s \in$ **des**(*s*), T_s contains a SP_{sv}. Due to the structure of trees, $\forall v \neq s \in$ **des**(*v*), T_s contains only one shortest path *SPsv*

Definition 5. Minimum Spanning Tree

Given a digraph $G = (V, E, w)$, an acyclic subset $T \subseteq E$ is a minimum spanning tree if it connects all of the vertices and whose total weight

$w(T) = \sum_{(u,v)\in T} w(u,v)$

is minimized. Since *T* is acyclic and connects all of the vertices, it must form a tree.

Definition 6. Hamiltonian Cycle

A Hamiltonian cycle is a cycle in graph which visits each vertex exactly once.

Definition 7. Linear Programming (LP)

One way of formulating a problem is using *Linear Programming*. It is formulating a problem with maximizing or minimizing an objective. This objective is captured by a linear function with certain variables. These variables are used to specify competing constraints and limited resources of the problem. We use LP to formulate TSP in section 2.1.3.

Definition 8. Integer Linear Programming

Integer Linear programming is a special case of linear programming where variables are required to be integers.

In addition to the notation we have introduced in this section so far, there are some routines used in the description of the upcoming algorithms. We use a minimum-priority queue *minpq* and a maximum-priority queue *maxpq* in most of the algorithms in this thesis. Entries in *minpq* and *maxpq* are stored in the format of <*key*, *value*>. Priority queue is ranked based on the value of *key* and *value* contains the information that is related to the *key*. In some algorithms, *value* could be a pair of data, <*value1*, *value2*>. *minpq* and *maxpq* support two methods. *Add*(*key*, *value*) and *Extract*(). The first method adds one entry with key of *key* and value of *value*. If *key* is already existed in the priority queue then it will be replaced by the entry only if the new *key* is smaller than the old one. *Extract*() method selects and removes the entry with minimum *key* in *minpq* or the entry with maximum *key* in *maxpq*.

Chapter 2

Survey on Related Work

2.1 NP-Complete Route Query Problems

The structure of this survey is, to a certain degree, inspired by [59] and the notation has been taken from the same book.

2.1.1 Vehicle Routing Problem

The *vehicle routing problem* (*VRP*) has received an immense attention from the scientific community during the last three to four decades as it often plays a vital role in the design of distribution systems.

The VRP deals with the problem of designing routes for a set of capacitated vehicles that are to service a set of geographically dispersed customers at the least cost. The vehicles are often assumed to have a common home base, called the *depot*. The cost of traveling between each pair of customers and between the depot and each customer is given. In real-life contexts restrictions such as time windows for when the service can commence make up important side-constraints to the problem. Typical real-world applications include, among others, goods delivery and pickup, school bus routing, street cleaning, transportation of handicapped persons, routing of salespeople and of maintenance units.

Several objectives can be considered for the vehicle routing problem. Typical objectives are:

- Minimization of the global transportation cost, dependent on the global distance or the global time traveled.
- Minimization of the number of vehicles required to serve all the customers.
- Balancing of the routes, for travel time and vehicle load.

Dantzig and Ramser [25] introduced the VRP more than 45 years ago. They described a real-world application and proposed the first *mathematical programming formulation* and algorithmic approach for the solution of the problem. A few years later, Clarke and Wright [19] proposed an effective greedy heuristic. Following that many models and exact and heuristic algorithms were proposed for the optimal and approximate solution of the different versions of the VRP. A number of surveys already exist [1][4][11][17][18][26][27][31][59][61][62] and we refer the reader to these surveys for an in-depth overview of this fertile area of research.

2.1.2 Other Classes of VRP

The objective of this section is to give an introduction to some of the core problems in the field of vehicle routing.

Traveling Salesman Problem

Traveling Salesman Problem is one of the most important versions of the VRP. The simplicity and applicability of this problem has led to decades of research on this problem. In this thesis, we mainly focus on this class of the VRP. We introduce this problem in more details in section 2.1.3. In sections 2.2, 2.3 and 2.4, we survey three different methods that have been taken to gear this problem. For more explanation of the TSP, we refer the reader to [54] , [27] and [57].

Compared to the TSP, more general routing problems like the capacitated vehicle routing problem or the pickup and delivery problem with time windows are much harder to solve, both heuristically and exactly.

Capacitated Vehicle Routing Problem

The basic version of the VRP is *capacitated* VRP (CVRP). It is the simplest and most studied member of the family. In the CVRP, all the customers are known in advance. The vehicles are identical and based at a single central depot, and only the capacity restriction for the vehicles are imposed. The objective is to minimize the total cost to serve all the customers.

Distance-Constrained Vehicle Routing Problem

The first variant of CVRP is *Distance-Constrained* VRP (DVRP) where for each route the capacity constrained is replaced by a maximum length (or time) constraint. In particular, the total length of the arcs of each route cannot exceed the vehicle's max length. If the vehicles are different, then the maximum route lengths might be different. Normally the cost and the length matrices coincide; hence the objective of the problem is to minimize the total length of the routes. The case in which both the vehicle capacity and the maximum distance constraints are considered, the problem is called *Distance-Constrained* CVRP (DCVRP).

Vehicle Routing Problem with Time Windows

The VRP *with time windows* (VRPTW) is the extension of the CVRP in which capacity constraints are imposed and each customer i is associated with a time interval $[a_i, b_i]$, called a *time window*. The service for each customer must start within the associated time window, and the vehicle must stop at the customer location for service time period. The vehicle should arrive and leave within the window. In case of early arrival at the location of customer the vehicle is allowed to wait until the time instant to serve that customer may start.

One of the real-life applications of VRPTW is Attended Home Delivery Services [2]. Attended delivery may be necessary for security reasons (e.g. electronics), because goods are perishable (e.g. groceries), because goods are physically large (e.g. furniture), or because a service is performed (e.g. repair or product installation).

Pickup and Delivery Problem

A subclass of vehicle routing problems is VRP *with Pickup and Delivery* (*VRPPD*). In this class of problems each customer i is associated with two quantities d_i and p_i, to be delivered and picked up. Sometimes only one demand quantity $d_i = d_i - p_i$ is used for each customer i (thus being possibly negative). The problem is to find routes for each vehicle such that all pickups and deliveries are served and such that the pickup and delivery corresponding to one request is served by the same vehicle and the pickup is served before the delivery. A number of additional constraints are often enforced, the most typical being capacity and time window constraints.

Heterogeneous Vehicle Routing Problem

In particular, the following characteristics were modified:

The vehicle fleet is composed by an unlimited number of vehicles for each type. The fixed costs of the vehicles are not considered and the routing costs are vehicle-independent. The survey by Baldacci et al. [5] gives an overview of approaches from the literature to solve heterogeneous VRPs. In particular, they classify the different variants described in the literature and, as no exact algorithm has been presented for any variants of heterogeneous VRP. They also review the lower bounds and the heuristic algorithms proposed.

Most integer programming formulations of the basic VRP, use binary variables as vehicle flow variables to indicate if a vehicle travels between two customers in the optimal solution. In this way, decision variables combine assignment constraints, modeling vehicle routes, with commodity flow constraints, modeling movements of goods. Another important type of formulation for Heterogeneous VRPs can be obtained by extending the Set Partitioning (SP) model of the VRP and associates a binary variable with each feasible route.

Dynamic Vehicle Routing Problem

Madsen et al. provide a comprehensive survey in regard to the *dynamic vehicle routing problem* (*DVRP*) [46] . The major technological advances during the recent years mean that the majority of new vehicles are equipped with advanced GPS/GIS systems. Hence, the distribution companies are now able to monitor the vehicles' position and status at any given time.

The basic VRP deals with customers who are known in advance to the planning process. Furthermore, all other information such as the driving time between the customers and the service times at the customers are used to be known prior to the planning. This provides that perfect set-up for applying advanced mathematical based optimization methods such as set partitioning. However, when dealing with real-life applications the information often tends to be uncertain or even unknown at the time of the planning. The traditional VRP can be said to be static as well as deterministic. In contrast to this, the *DVRP* considers a VRP in which a subset (or the full set) of customers arrive after the day of operation has begun. The DVRP will have to be able to consider how to include the new requests into the already designed routes.

In many DVRP, the vehicles service two types of requests:

1. *Advance requests,* which can also be referred to as static customers as these requests for service has been received before the routing process was begun.

2. *Immediate requests*, which can also be referred to as dynamic customers as these will appear in real-time during the execution of the routes.

Ideally, the new customers should be inserted into the already planned routes without the order of the non-visited customers being changed and with minimal delay. However, in practice, the insertion of new customers will usually be a much more complicated task and will imply either partial or full re-planning of the non-visited part of the route.

Inventory Routing

The following description has been encouraged from [9]. Inventory routing problems are among the most important and challenging extensions of vehicle routing problems, in which inventory control and routing decisions have to be made simultaneously. The objective is to minimize the total cost such as the sum of inventory holding and transportation costs, while avoiding stock-outs and respecting storage capacity limitations. In the literature there is a large variety of inventory routing problems and they have a few common characteristic. Inventory routing problems all share some basic characteristics. They all consider an inventory component in which products are shipped from a supplier to one or more customers by means of capacitated, vehicles. Costs are defined as the distance traveled by the vehicles. They are included in the objective function. The supplier has to manage product inventory at customers to ensure that customers do not experience a stock-out.

Bertazzi et al. [9] pointed out a variety of other characteristics that may significantly change the structure of a particular inventory routing problem, such as:

- The planning horizon can be finite or infinite;

• Inventory holding costs may or may not be considered;

• Inventory holding costs may be charged at the supplier only, at the supplier and the customers, or at the customers only;

• The production and consumption rates can be deterministic or stochastic;

• Production and consumption take place at discrete time instants or take place continuously;

• Production and consumption rates are constant over time or vary over time;

• The optimal delivery policy can be chosen from among all possible policies or has to be chosen from among a specific class of policies.

2.1.3 Traveling Salesman Problem

The problem comes in different versions based on how we define the distances between cities. If the distance from city *i* to city *j* is the same as the distance from city *j* to city *i* for all cities *i* and *j*, then the problem is said to be *symmetric*. The problem is said to be *asymmetric* if this property does not hold. If the cities are located in $\mathbb{R}^d$ and the distance between two cities is the *Euclidean* distance then the problem is said to be *Euclidean*. In this thesis we will concentrate on *symmetric* TSP. However, our work can be applied to asymmetric TSP as well. The *symmetric* TSP is useful in many practical applications like X-ray crystallography [10][9] and VLSI chip fabrication [41].

TSP can be formulated as a mathematical model in the following way. Given a complete digraph $G = (V, A, w)$. We define binary decision variable x_{ij} that is set to one if and only if arc (i, j) is used in the solution. The problem can be formulated as the following integer linear programming model:

$$\min \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} x_{ij} \tag{2.1}$$

Subject to:

$$\sum_{j \in V \setminus \{i\}} x_{ij} = 1 \qquad \forall i \in V \tag{2.2}$$

$$\sum_{i \in V \setminus \{j\}} x_{ij} = 1 \qquad \forall j \in V \tag{2.3}$$

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \qquad \forall S \subset V \tag{2.4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \tag{2.5}$$

The objective (2.1) minimizes the arc costs, equations (2.2) and (2.3) ensure that one arc leaves each node and one arc enters each node, equation (2.4) eliminates sub-tours by forcing connection (an edge to be used) between every pair of separate subsets. This formulation is impractical for large scale problems, because the number of sub-tour elimination constraints is of order 2^n (if V has n vertices (cities), the number of subsets is 2^n).

One important restriction that often imposed on instances is the *triangle inequality*. Which is for all i, j, k, $1 \leq i, j, k \leq N$, $d(c_i, c_j) \leq d(c_i, c_k) + d(c_k, c_j)$. It says that the direct path between two cities is always the shortest route. Most of the theoretical works on TSP heuristics assume that the triangle inequality holds.

A generalization of the TSP introduces more than one salesman which is called *m-traveling salesman problem* (*m-TSP*). In the m-TSP we are given *n* cities, *m* salesmen and one depot or home base. All cities should be visited exactly once on one of *m* tours, starting and ending at the depot. The tours are not allowed to be empty. If distances satisfy the triangle inequality then it is easy to see that the distance of the shortest TSP tour on the *n* cities plus the depot always is less than or equal to the distance of the shortest m-TSP solution for any *m*.

Any m-TSP with *n* cities can be formulated as a TSP with *m* + *n* cities. One first creates *m* copies of the depot node. The distances between depot nodes is then set to a sufficiently large number while the distances between the depot nodes and ordinary nodes are copied from the m-TSP. The large distance between depot nodes ensures that no salesmen tours are empty. Notice that the resulting TSP does not obey the triangle inequality. In view of the fact that the m-TSP is so closely related to the TSP, it has not studied widely in the literature. Bektas [6] surveyed the literature regarding heuristics and exact methods of this problem.

TSP is one of the most studied NP-hard problems and many solution methods for this problem have been introduced in the literature. More general routing problems like the capacitated vehicle routing problem or the pickup and delivery problem with time windows turn out to be much harder to solve, both heuristically and exactly, compared to the TSP. Solution methods for the TSP provide a significant development in solution methods for more general routing problems.

In the next three sections, we describe the three various approaches that have been taken in order to tackle the TSP.

2.2 Exact methods

Exact methods guarantee that the optimal solution is found if the method is given sufficiently time and space. A simple enumeration is out of the question, so exact methods must use more clever techniques. The worst-case running time for NP-hard problems are still going to be high though. We cannot expect to construct exact algorithms that solve NP-hard problems in polynomial time unless NP = P. For some classes of problems, there are hopes of finding algorithms that solve problem instances occurring in practice in reasonable time though.

2.2.1 Branch and Bound Algorithm

A branch and bound algorithm searches the complete space of solutions for a given problem for the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the algorithm to search parts of the solution space only implicitly.

Branch and bound algorithms are non-heuristic, in the sense that they maintain a provable upper and lower bound on the (globally) optimal objective value; they terminate with a certificate proving that the suboptimal point found is ε-suboptimal. Branch and bound algorithms can be (and often are) slow, however. In the worst case they require effort that grows exponentially with problem size, but in some cases the methods converge with much less effort.

In TSP, branch and bound methods are normally based on some relaxation of the integer linear programming model of TSP. The lower bound for the length of the cycle is determined by an appropriate relaxation, e.g. by discarding the sub-tour elimination constraints. This results in an *assignment problem*, and it is called an *assignment relaxation*. The assignment problem is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. It consists of finding a maximum weight matching in a weighted bipartite graph (is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are independent sets). In its most general form, the problem is as follows:

There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimized

The branch and bound with assignment relaxation is simple but inefficient because of its bounding technique. The tree of sub-problems may become too large. A stricter lower bound can be calculated using other relaxations, e.g. a minimal spanning tree relaxation, resulting in relatively limited branching. These and other branch and bound variants are reported in [42].

In [43] Laporte and Norbert present a complete and detailed analysis of the branch-and-bound algorithms up until the late 1980s. Since CVRP generalizes TSP, many exact approaches for the CVRP are inherited from the extensive and successful work done for the exact solution of the TSP. Branch and bound algorithms used basic combinatorial relaxations, such as the *Assignment Problem* (AP), the degree-constrained *Shortest Spanning Tree* (SST). These algorithms were, until the late 1980s, the most effective exact approaches for the CVRP. Recently more complicated bounds were proposed which increased the size of the problems that can be solved by branch and bound algorithms.

There is no exact comparison of the branch and bound algorithms in the literature since the authors either considered a slightly different problem or solved a completely different set of instances.

2.2.2 Branch and Cut Algorithms

As stated in [59]: "The *linear relaxation* of an integer linear program *IP* is the linear program obtained from IP by dropping the condition that all variables have to be integers. Therefore the optimal value Z_{LP} of the relaxation is a lower bound to the optimal value Z_{IP} of the integer linear program ($Z_{LP} \leq Z_{IP}$). If the number of constraints of an integer linear program is small enough so that its linear relaxation can be fed into an LP solver, a classical method to solve it is branch-and bound with linear programming bound. On the other hand, when the number of linear constraints of LP is large or when they have exponential size, then the constraint system cannot fed into an LP solver and a *cutting plane* technique has to be used to solve the linear program. Let IP be an integer program and LP(∞) be its linear relaxation having a very large number of constraints, the cutting plane algorithm works as follows:

For $h \geq 0$, let LP(h) be a linear program consisting of a subset of reasonable size of the constraints in LP(∞). Solve LP(h), if the solution is feasible for IP, then it is optimal solution. Otherwise assume there is a black box algorithm called *separation algorithm,* which gives at least one constraint of LP(∞) that is violated by the solution of LP(h), if one exists. Then LP($h+1$) is obtained by adding the violated constraints to LP(h). For every $h \geq 0$, if $Z_{LP(h)}$ is the optimal value of LP(h) we have $Z_{LP(h)} \leq Z_{LP(h+1)} \leq Z_{LP(\infty)} \leq Z_{IP}$ *.”*

The branch and cut method solves the linear program without the integer constraint using the regular simplex algorithm. When an optimal solution is obtained, and this solution has a non-integer value for a variable that is supposed to be integer, a *cutting plane* algorithm is used to find additional linear constraints which are satisfied by all feasible integer points but violated by the current fractional solution. If such an inequality is found, it is added to the formulation, such that resolving it will yield a different solution which is hopefully "less fractional". This process is repeated until either an integer solution is found (which is then known to be optimal) or until no more cutting planes are found. We may normally end with an optimal solution however, in practice we may not have an exact separation algorithm and it may return no violated inequality although there are some. If we have not terminated with an optimal solution to IP, we *branch*. We decompose the problem into two new problems, i.e., adding upper and lower bounds to a variable whose current value is fractional. The problem is split into two versions, one with the additional constraint that the original variable is greater than or equal to the next integer greater than the intermediate result, and one where this variable is less than or equal to the next lesser integer. Then we solve each new problem recursively by the same method and the optimal solution to the original problem will be the better of these two solutions. Such an integration of enumeration with cutting plane is the core of the branch and cut method.

This method has been successful in finding optimal solutions of large instances of a closely related problem, the *Symmetric Traveling Salesman Problem* (STSP). However, compare to TSP, the amount of research carries out on branch and cut applied to CVRP is still quite limited.

Similar to in branch and bound algorithms, the central problem of branch and cut is that the tree generated by the branching procedure becomes too large and termination seems unlikely within a reasonable amount of time.

2.3 Classical heuristics

Given the fact that TSP is a NP-hard problem, one approach to tackle this problem is to look for heuristics that find *near*-optimal tours fast. Heuristics are methods that typically relatively quickly can find a feasible solution with reasonable quality. There are no guarantees about the solution quality though, it can be arbitrarily bad. The heuristics are tested empirically and based on these experiment, comments about the quality of the heuristic can be made. Heuristics are typically used for solving real-life problems because of their speed and their ability to handle large instances.

Several families of heuristics have been proposed for the TSP. These can be classifies into two main classes: *classical heuristics*, developed mostly between 1960 and 1990, and *metaheuristics* whose grown has occurred in the last decade. Most standard construction and improvement procedures in use today belong to the first class. These methods perform a relatively limited exploration of the search space and typically produce good quality solutions within modest computing times. Moreover most of them can be easily extended to account for the diversity of constraints encountered in real-life contexts. Therefore, they are still widely used in commercial packages.

2.3.1 Attributes of a good heuristic

We start this section by an explanation on the four essential attributes of a good heuristic that Cordeau et al. [20] described for software transferability and end-user adoption. Most heuristics are usually measured against two criteria: *accuracy* and *speed*. However *simplicity* and *flexibility* are also essential attributes of good heuristics. Let us explain these four criteria.

Accuracy

Accuracy measures the degree of departure of a heuristic solution value from the optimal value. Since optima and sharp lower bounds are usually unavailable. In the case of the TSP, the optima and lower bounds are commonly not available. Based on this reason, most comparisons have to be made with best known values. Analyzing heuristic results is fraught with difficulties. It is usual that authors report results obtained for the best combination of algorithmic parameters, or for the best of several runs. The problem increases when authors in presenting their results do not use the same techniques such as rounding. This can lead to hugely different results.

Another issue related to accuracy is consistency. Users will prefer a heuristic that performs well all the time rather than one that may perform even better most of the time but very poorly on other occasions. Such solutions are enough to discredit the algorithm.

Finally, users will often prefer an algorithm that produces a good solution at an early stage, and then displays solutions of increasing quality throughout the execution to an algorithm that comes up with only a final answer, possibly after a long computing time. This gives users a better feel of how much additional effort is worth investing given the evolution rate of the solution value.

Speed

The importance of computation speed depends on the planning level at which the problem is solved and on the required degree of accuracy. At one extreme, real-time applications such as express courier pickup and delivery or ambulance redeployment require fast, sometimes almost instantaneous, action. At the other extreme, in long term planning decisions made every several months, such as fleet sizing, there would be no problem in devoting several hours or even several days of computing time. Most applications can be placed somewhere between these two extremes. It does not seem unreasonable to invest ten or twenty minutes of computing time on a routing problem that must be solved daily. Interactive systems must of course react much more quickly.

Simplicity

Some heuristics are rarely implemented because they are just too complicated to understand and to code. In addition, heuristics should be reasonably robust to ensure that they work properly, even if not every single detail is implemented. Many algorithmic descriptions fail to provide too much or insufficient detail. Simple codes, preferably short and self-contained, stand a better chance of being adopted, although a minimum of complexity is to be expected for good results.

Algorithms that contain too many parameters are difficult to understand and unlikely to be used. This problem is prevalent in most metaheuristics developed over the past twenty years. In their search for better solutions, researchers have increased the number of parameters contained in their algorithms far beyond what can be deemed reasonable, particularly in view of the fact that relatively few instances are used in the tests. Not only should the number of algorithmic parameters be limited, but these should also make sense to the end-user. As suggested by Cordeau et al. [20], there are two easy ways around the production of parameters. One is to set them once and for all at some meaningful value, especially if tests show that the algorithm is rather insensitive to a particular parameter choice. Another possibility is to make use of parameters that self adjust during the course of the algorithm.

Flexibility

A good heuristic should be flexible enough to be able accommodate the various side constraints encountered in a majority of real-life applications. In the literature most papers focus on the accuracy and speed and it is not clear how changes can be made to deal with further constraints. The result may affect the performance of the algorithm.

In general, classical TSP heuristics can be classified into two categories: *Constructive heuristics* gradually build a feasible solution while keeping an eye on solution cost, but they do not contain an improvement phase. *Improvement methods* attempt to upgrade any feasible solution by performing a sequence of edge or vertex exchanges. However, the distinction between constructive and improvements methods is often blurred since some constructive algorithms include improvements steps.

Generally the heuristic approaches to the TSP can be divided into two classes: *constructive* heuristics and *improvement* heuristics. We describe more of these two approaches in sections 2.3.2 and 2.3.3.

2.3.2 Constructive Methods

These heuristics build a solution from scratch and then the process will continue until a feasible solution has been found. The process is usually a greedy algorithm. Moreover the constructive algorithms are important since they can be used to generate the initial tours needed by local search algorithms.

Johnson and McGeoch [38] stated in their paper that “Whereas the successive augmentation approach performs poorly for many combinatorial optimization problems, in the case of the TSP many tour construction heuristics do surprisingly well in practice. The best typically get within roughly 10-15% of optimal in relatively little time.”

The remainder of this subsection we discuss four important tour construction heuristics and their algorithmic behaviors.

Nearest Neighbor

In this algorithm, TSP tour is constructed by beginning with the starting city and iteratively visiting the nearest neighbor of the last city added to the tour from cities that have not been visited yet. The running time for this algorithm is $O(N^2)$. However, if the distance metric satisfies the triangle inequality, then the best guarantee, in terms of tour quality, is $\mathrm{NN}(I)/\mathrm{OPT}(I) \leq (0.5)(\lceil \log_2 N \rceil + 1)$. However, Rosenkrantz et al. [55] found instances for which the ratio grows as $\Theta(\log N)$.

Greedy

In this heuristic, the input of the algorithm is a complete graph with cities as vertices and the shortest distance between each pair of citied as edges. The tour builds up one edge at a time starting with the shortest edge, and repeatedly adding the shortest among the remaining *available* edges. An edge is called *available* if it is not yet in the tour and if adding it would not create a degree-3 vertex or a cycle of length less than total number of vertices.

This algorithm can be implemented with running time $\Theta(N^2 \log N)$. As you may have noticed, this algorithm is slower than the nearest neighbor algorithm. Like the nearest neighbor algorithm, it can be shown that for all instances satisfying triangle inequality, worst-case tour quality is $\mathrm{Greedy}(I)/\mathrm{OPT}(I) \leq (0.5)(\lceil \log_2 N \rceil + 1)$ [47] however, the worst examples known for Greedy only make the ratio grow as $(\log N)/(3 \log \log N)$ [28].

Clarke-Wright savings heuristic

The *Clarke-Wright savings heuristic* (Clarke-Wright or simply CW for short) is derived from a more general vehicle routing algorithm [19]. In this algorithm we choose a random city as the *hub*. The salesman returns to the hub after each visit to another city. As a result we have multiple graphs and in each of them every non-hub vertex is connected by two edges to the hub. For each pair of non-hub cities, let the *savings* be the amount by which the tour length would be reduced if the salesman went directly from one city to the other, bypassing the hub. We compute the saving for all the non-hub city pairs. In the next step we order all the savings and then apply them in non-increasing order of savings. Similar to greedy algorithm, we ignore the savings in which performing the bypass will create a cycle of non-hub vertices or cause a non-hub vertex to become adjacent to more than two other non-hub vertices. The construction process terminates when only two non-hub cities remain connected to the hub, in which case we have a true tour.

As with Greedy, this algorithm can be implemented to run in time $\Theta(N^2 \log N)$. The best performance guarantee currently known (assuming the triangle inequality) is $\mathrm{CW}(I)/\mathrm{OPT}(I) \leq (\lceil \log_2 N \rceil + 1)$ (a factor of 2 higher than that for Greedy) [47], but the worst examples known yield the same $(\log N)/(3 \log \log N)$ ratio as obtained for Greedy [28].

Christofides

Christofides' algorithm [16] is the one heuristic that has a constant worst-case performance ratio of just 3/2 assuming the triangle inequality. Johnson and McGeoch [38] explained how to construct a TSP tour using Christofides heuristic as follows:

"First, we construct a minimum spanning tree T for the set of cities. Note that the length of such a tree can be no longer than $\mathrm{OPT}(I)$, since deleting an edge from an optimal tour yields a spanning tree. Next, we compute a minimum-length matching M on the vertices of odd degree in T. It can be shown by a simple argument that assuming the triangle inequality this matching will be no longer than OPT(I)/2. Combining M with T we obtain a connected graph in which every vertex has even degree. This graph must contain an Euler tour, i.e., a cycle that passes through each edge exactly once, and such a cycle can be easily found. A traveling salesman tour of no greater length can then be constructed by traversing this cycle while taking shortcuts to avoid multiply visited vertices. (A *shortcut* replaces a path between two cities by a direct edge between the two. By the triangle inequality the direct route cannot be longer than the path it replaces.)"

Besides providing a better worst-case guarantee, this heuristic finds better TSP tours in practice than Nearest Neighbor, Greedy, and Clarke-Wright. However in terms of time consumption, Christofides heuristic takes $\Theta(N^3)$ time which is substantial comparing to the other three heuristics. Gabow & Tarjan [29] modify Christofides heuristic in a way that takes $O(N^{2.5})$ time holding the same worst-case guarantee. They obtained this outcome by using a scaling-based matching algorithm and stopping once the matching is guaranteed to be no longer than $1 + (1/N)$ times optimal.

For readers interested in the several other tour-construction heuristics are referred to more extensive studies such as those of Bentley [7] [8], Reinelt [53] and Junger, Reinelt, and Rinaldi [40].

2.3.3 Improvement Methods

In this subsection, we describe *Variable-Opt*, *2-Opt* and *3-Opt* algorithms which are the most famous classical local optimization algorithms for the TSP.

Johnson and McGeoch [38] stated in their paper that "classical local optimization techniques for the TSP yield even better results (than constructive methods), with the simple 3-Opt heuristic typically getting with 3-4% of optimal and the ‘‘variable-opt’’ algorithm of Lin and Kernighan [45] typically getting with 1-2%.” In addition to that, these algorithms provide the essential building blocks in adapting tabu search, simulated annealing, etc. to the TSP.

Other local optimization algorithms that have been introduced in the literature are *k*-Opt algorithms, for some fixed $k > 3$, which allows exchanges of as many as *k* edges, as well as *2.5*-Opt [8], *Or*-Opt [48], *Dynasearch* [52], *GENI and GENIUS* [30].

Variable-Opt Algorithm

In [45] Lin and Kernighan introduced an improvement which involves swapping pairs of edges to make a new tour. It is a generalization of *k*-Opt. Lin–Kernighan is adaptive and at each step decides how many edges between cities need to be switched to find a shorter tour. It starts with a non-optimal but feasible solution. It is not optimal because there are *k* edges in the tour that are *out of place*; to make the tour optimal, these edges should be replaced by *k* new edges. The problem is simply to identify *k* out of place edges and *k* new edges. In their algorithm profit or gain is defined as the difference of the length of the old edge from length of the new edge.

They start with a random initial solution and an empty set which will include a proposed set of exchanges. In the first step, they select the most-out-of-place pair which is the pair that maximize the improvement of the proposed set of exchanges and add it to the proposed set of exchanges. They repeat this step until no more gain can be made (according to an appropriate stopping rule). In the next step, they select a number of proposed exchanges which have the best improvement and apply them. They repeat these two steps until no more improvement is found.

2-Opt Algorithm

2-Opt Algorithm is considered as a local improvement algorithm for the TSP based on simple tour modifications. Given a feasible tour, the algorithm then repeatedly performs a sequence of operations, so long as each reduces the length of the current tour, until a tour is reached for which no operation yields an improvement (a *locally optimal* tour).

The 2-Opt algorithm was first proposed by Croes [22]. The 2-Opt algorithm deletes two edges, thus breaking the tour into two paths, and then reconnects those paths in the other possible way (Only if the sum of the length of the newly added edges is less than the sum of the length of the deleted edges). In Figure 2.1 the initial tour has been shown on the left. This has been improved by deleting edges “v1v5” and “v2v6” and adding edges “v1v2” and “v5v6”.

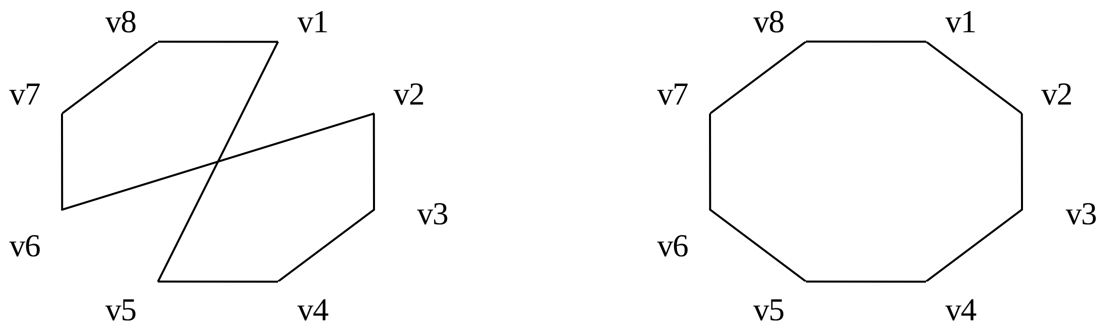


Figure 2.1: A 2-Opt move: original tour on the left and the resulting tour on the right.

3-Opt Algorithm

In 3-Opt [44] the exchange replaces up to three edges of the current tour. See Figure 2.2

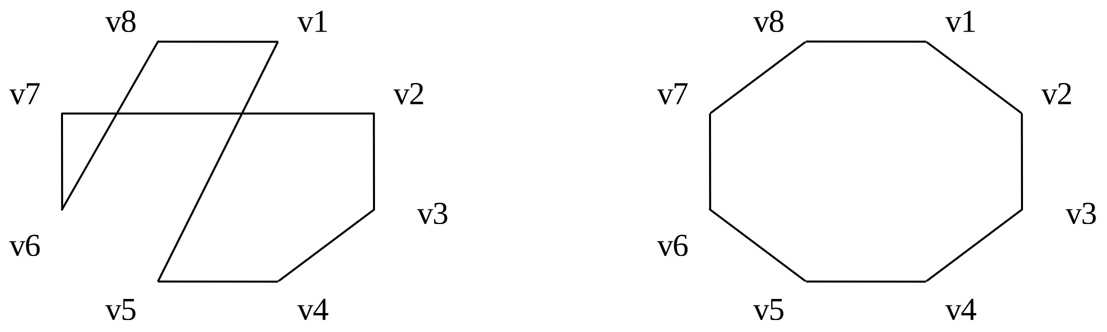


Figure 2.2: A 3-Opt move: original tour on the left and a possible resulting tour on the right.

Theoretical bounds on local search algorithms

In terms of tour quality using local search algorithms, in case of arbitrary instances, these algorithms are constrained by the two theorems mentioned earlier in section 2.1.3.

In 1977 Papadimitriou & Steiglitz [50] proved that if P $\neq$ NP, no local search algorithm that takes polynomial time per move can guarantee $A(I)/\text{OPT}(I) \leq C$ for any constant C, even if an exponential number of moves is allowed. The situation is much worse in 2-Opt, 3-Opt and k-Opt with $k<3N/8$. In 1978 they showed that there exist instances that have a single optimal tour but exponentially many locally optimal tours, each of which is longer than optimal by an exponential factor [51]. This result does not apply to graphs satisfying the triangle inequality.

In 1994 Chandra et al. [24] showed in their results that assuming the triangle inequality the best performance guarantee for 2-Opt is a ratio of $(1/4)\sqrt{N}$ and for 3-Opt is

(1/4)$N^{1/6}$. In general, satisfying the triangle inequality, the best performance guarantee for k-Opt is at least (1/4)$N^{1/2k}$. For restricted instances where cities are points in $\mathbf{R}^d$ for some fixed d and distances are computed according to Euclidean norm, the worst-case performance ratio is O (log$N$).

On the other hand, using a good heuristic to generate our starting tours, we can obtain significantly better worst-case behavior. For example, using Christofides to generate the starting tour, then the tour resulted by 2-Opt will never be worse than 3/2 times optimal (assuming the triangle inequality). That is because the worst-case ratio of Christofides algorithm is 3⁄2.

In terms of number of moves in local search algorithms before reaching the optimal tour, this number can be pretty large for 2-Opt and 3-Opt algorithms. Chandra et al. [24] provide the worst-case bound on the number of moves for 2-Opt, 3-Opt and k-Opt for any fixed k before halting. This number can be Θ ($2^{N/2}$) for 2-Opt. However these results are based on the assumption of a random starting tour.

In addition to the number of moves, we also need to consider the time per move which is the time to find an improving move and the time to apply the move. Both depend on the used data structure used. For example, employing an array data structure, one the cost of evaluating a move is Θ (1) while the cost of performing the move is $\Theta(N)$.

As mentioned in [38], results show that the 2-Opt and 3-Opt algorithms perform much better in practice than the theoretical bounds might indicate.

2.4 Metaheuristics

A special class of heuristics that has received special attention in the last two decades is metaheuristics. Metaheuristics provides general frameworks for heuristics that can be applied to many problem classes. High solution quality is often obtained using metaheuristics.

In metaheuristics, the emphasis is on performing a deep exploration of the most promising regions of the solution space. These methods typically combine sophisticated neighborhood search rules, memory structures, and recombination of solutions, the quality of solutions produced by these methods in much higher than that obtained by classical heuristics, but the price to pay is increased computing time. Moreover, the procedures usually are context dependent and require finely tuned parameters, which may make their extension to other situations difficult. In a sense metaheuristics are no more than sophisticated improvement procedures, and they can simply be viewed as natural enhancements of classical heuristics.

In a major departure from classical approaches, metaheuristics allow deteriorating and even infeasible intermediary solutions in the course of the search process. The best known metaheuristics developed typically identify better local optima that earlier heuristics, but they also consumes more time.

Gendreau et al. [31] provide a good overview of the six main types of metaheuristics that have been applied to the VRP. The following overview of these metaheuristics had been inspired from their paper.

Simulated Annealing (SA), Deterministic Annealing (DA) and Tabu Search (TS), start from an initial solution, and move at each iteration to a solution in the neighborhood until a stopping condition is satisfied. Care must be taken to avoid cycling. Genetic Algorithms (GA) examines at each step a population of solutions. Each population is derived from the preceding one by combining its best elements and discarding the worst. Ant Systems (AS) is a constructive approach in which several new solutions are created, at each iteration, using some of the information gathered at previous iterations. As was pointed out by Taillard et al. [59], TS, GA and AS are methods that record, as the search proceeds, information on solutions encountered and use it to obtain improved solution. Neural Networks (NN) is a learning mechanism that gradually adjusts a set of weights until an acceptable solution is reached. The rules governing the search differ in each case and these must also be tailored to the shape of the problem at hand. Also, a fair amount of creativity and experimentation is required. Gendreau et al. [31] survey some of the most representative applications of local search algorithms to the VRP. In their survey they conclude that the best of these methods are able to find excellent and sometimes optimal solutions to instances with a few hundreds customers, albeit at a significant cost in computation time. Tabu Search now emerges at the most effective approaches. Procedures based on pure Genetic Algorithms and on Neural Networks are clearly poorer in quality than others, while those based on Simulated or Deterministic Annealing and on Ant Systems are not quite competitive. Considering the performance improvements obtained with successive implementations of any given approach, it appears, however, that hybrid AS and GA may. In the future, we might be able of matching the effectiveness of existing TS heuristics, since there approaches have not been fully exploited. Another observation is that the data sets currently used a benchmarks are made up of instances that are too small to differentiate sharply between various implementation of some of these metaheuristics, TS in particular. Data sets corresponding to larger instances are thus required. Also, one may wonder how these metaheuristics would perform on the much larger instances that often encountered in practical applications. Given their computing requirements, heuristics with such a level of sophistication may be unable to solve satisfactory large instances in any reasonable amount of time, especially if real-time applications are contemplated. With respect to the classical heuristics, metaheuristics are rather time-consuming but they also provide much better solutions. Typically, classical methods yield solution values between 2% and 10% above the optimum (or the best known solution value); while the result of best metaheuristics is often less than 0.5%.

As stated in [38]: "The successes for traditional approaches leave less room for new approaches like tabu search, simulated annealing, etc. to make contributions. Nevertheless, at least one of the new approaches, genetic algorithms, does have something to contribute if one is willing to pay a large, although still $O(N^2)$, running time price."

2.5 Experimental Evaluation of TSP

There is a large body of work on TSP algorithms. In this section we try to address briefly the experimental results on TSP. In [36] the authors provide an extensive informal discussion with regard to analyze algorithms experimentally. It is based on lessons learned by the author over the course of more than a decade of experimentation, survey paper writing, refereeing, and lively discussions with other experimentalists.

2.5.1 Theoretical results

Sahni & Gonzalez present a theorem which provides a good constraint for the behavior of *any TSP* heuristic [56]. The theorem is as follows:

Theorem: Let $A(I)$ be the length of TSP tour generated using heuristic A and instance I and OPT (I) symbolizes the length of an optimal TSP tour. Assuming P $\neq$ NP, no polynomial-time TSP heuristic can guarantee $A(I)/\mathrm{OPT}(I) \leq 2^{p(N)}$, for any fixed polynomial p and all instances I.

In the theorem they consider no restriction on the instances. However, most applications impose restrictions such as triangle inequality on instances. In [1] the authors introduced a theorem which provides a much more limited constraint for TSP heuristics.

Theorem: Assuming P $\neq$ NP, there exists an $\varepsilon > 0$ such that no polynomial-time TSP heuristic can guarantee $A(I)/\mathrm{OPT}(I) \leq 1 + \varepsilon$ for all instances I satisfying the triangle inequality. On the other hand, it has not been even shown how large ε can be.

Based on these two theorems, the question arises that what kinds of performance guarantees can be provided by polynomial-time TSP heuristics? As observed by Rosenkrantz, Stearns, and Lewis [55], there are at least three simple polynomial-time tour generation heuristics, *Double Minimum Spanning Tree, Nearest Insertion,* and *Nearest Addition* that have worst-case ratio 2 under the triangle inequality. That is, they guarantee $A(I)/\mathrm{OPT}(I) \leq 2$ under that restriction. Although the worst-case performance of these three algorithms is good in theory, the four tour construction heuristics (*Nearest Neighbor, Greedy, Clarke-Wright, Christofides*) discussed in [38] provide a much looser worst-case in theory and perform much better in practice.

2.5.2 Lower Bound

When evaluating the empirical performance of TSP methods, the precise optimal tour length is often not available since for large instances we typically do not *know* the optimal tour length. As a result, in studying large instances, authors often compare heuristic results to something we can compute: the lower bound on the optimal tour length due to Held and Karp [33][34]. In their paper they have defined *1-tree* as a tree having vertex set {2, 3, ..., n}, together with two distinct edges at vertex 1. Their approach exploits the fact that a minimum-weight 1-tree is easy to compute. Also a tour is simply a 1-tree in which each vertex has degree 2 and if a minimum-weight 1-tree is a tour, then it is a tour of minimum weight. They point out that the weight of a minimum tour is greater than or equal to the weight of a minimum-weight 1-tree. Their bound is produced using the linear programming relaxation of the TSP. For moderate size instances, it can be computed exactly using linear programming. However, the computation is non-trivial since the number of constraints in the linear program is exponential in the number of cities. It is mentioned by Johnson and McGeoch [38] that the Held-Karp bound itself appears to provide a consistently good approximation to the optimal tour length. From a worst-case point of view, the Held-Karp bound can never be smaller than (2/3)OPT(*I*), assuming the triangle inequality [63][58]. In practice, it is typically far better than this, even when the triangle inequality does not hold.

2.5.3 Experimental Results

Up until 1980, the number of cities in the largest TSP instance solved to optimality was 318 [23]. Padberg & Rinaldi [49] in 1987 solve a TSP instance with 2392 cities. Taking 3-4 years of computing time, Applegate et al. [3] in 1994 could find the optimal TSP tour for instances as large as 7397 cities. The largest Euclidean instance containing 24,978 cities has been solved using branch-and-cut method by the research team of Applegate, Bixby, Cvátal, Cook and Helsgaun [65]. The same authors find the optimal TSP tour for instances as large as 85900 cities. According to them, heuristic methods for the TSP have been applied to an instance with more than 1.9 million cities [66].

[67] “8th DIMACS Implementation Challenge” provides a good source of state-of-the-art algorithms in the area of TSP heuristics. This challenge intends to provide a continually updated picture of the state-of-the-art in the area of TSP heuristics. In [39] and [37] the authors give extensive Experimental Analysis of Heuristics for the symmetric and asymmetric TSP respectively. They show that compare to the optimal result for TSPLIB instances the typical excess over the Held-Karp bound is less than 1% and the maximum excess is 1.74%.

One drawback of the information provided by the published papers is the high level description of the algorithms and results. This makes it difficult to compare different algorithms especially when it comes to running time comparisons. Besides, the worst-case nature of many theoretical results is too pessimistic to tell us much about typical algorithmic behavior. Considering these problems, Johnson and McGeoch [38] provide an extensive case study on the TSP. In their paper they focused on the relationship between what is known theoretically about the algorithms they studied and what can be observed empirically. They were successful to provide the reader with a useful source of more widely applicable ideas. The results in their paper show that the TSP tour resulted by the *Christofides* algorithms is about 9.5 to 9.9 percent over the Held_karp lower bound. For *Clarke-Wright* algorithms the produced TSP tour is between 9.2-12.2 percent over the Held_karp lower bound. *Greedy* algorithm resulted in TSP tours with about 14.2-19.5 percent over the Held_karp lower bound and finally the *Nearest Neighbor* algorithm found TSP tours between 23.3-25.6 percent over the Held_karp lower bound.

In state-of-the-art algorithms, they all assume that a complete graph is given as an input. There has not been any work that uses a non-complete graph as an input to find the TSP. Because of this, only the part of our experimental results that uses 100% of the edges in the complete graph is comparable to the state-of-the-art experimental results. Although our main focus in this thesis is to find out having a subset of edges how much the tour quality decreases compare to having a complete graph.

Chapter 3

Non-Complete Graph Generating Algorithms

The existing work on the TSP assumes a complete graph as an input. However, for very large graphs, generating all edges in the complete graph could be time-consuming.

In this chapter, we introduce a few algorithms for generating edges to solve the TSP. We shall evaluate these algorithms, in Chapter 7, in term of the tour quality and cost. For each algorithm proposed, we prove its correctness and analyze its output complexity.

3.1 X Nearest Neighbor (XNN) Algorithm

Intuitively we can generate a complete graph and then find the best TSP tour. The difficulty in this approach is that generating a complete graph requires issuing *N* (number of cities) SPT queries. In large graphs it may take a long time to generate a complete graph. In order to reduce this time while not losing essential information, we consider edges generated from XNN queries. Given a source *s*, an *XNN* query is defined as the top

X nearest neighbors to the source *s*. An XNN query can be evaluated by generating an SPT rooted at *s*. The process is terminated when the *X* nearest neighbors are located.

3.1.1 Algorithm Description

The input of XNearestNeighbor algorithm is a graph containing the vertices including cities and non-cities, a source node and an integer which is the number of nearest neighbors that we want to produce. The structure of this algorithm is based on the Dijkstra's shortest path algorithm [21]. We have modified this algorithm in a way that instead of producing the shortest paths from the source vertex to all the remaining vertices, the algorithm constructs a shortest paths tree from the source vertex to *X* nearest neighbor cities (we need to check to see if a closed vertex is a city).

Like Dijkstra's shortest path algorithm, we keep track of information for each vertex, which includes the shortest distance from *src* so far, the parent node in the shortest path, and whether it is closed or not. Nodes are stored in a minimum priority queue *pq*, usually binary heap, ordered by their distance from *src*. Initially all vertices are open and their distance from source *src* is infinity except *src* itself which is 0. In each iteration, the vertex with the minimum shortest distance so far is closed and relaxed. During the process, whenever a vertex is closed, if the vertex in not the source vertex, we add the edge connecting the source to this vertex to *xnnpq* priority queue and then the relaxation process in Dijkstra's algorithm [21] should be applied to any adjacent vertex in the graph. The relaxation and closing process keeps going until *X* number of cities has been closed.

Algorithm: XNearestNeighbor(*g*, *src* , *x*)

Input: *g* is the graph of vertices that may include cities; *src* is the source node, *x* is the number of the nearest neighbors to be found

Output: a min-priority queue (*xnnpq*) containing *x* nearest neighbors

```
1:  Initialize a min priority queue xnnpq
2:  Initialize a min priority queue pq
3:  src.distance = 0
4:  pq.update (0 , src)
5:  nn = 0
6:  while (pq is not empty & nn <= x) do
7:          v0 = pq.extractMin()
8:          if(v0.closed == false) then
9:                  v0.closed = true
10:                 if (v0.isCity == true)
11:                         nn++
12:                         if (nn > 1) then
13:                                 xnnpq.add(v0.distance, edge (src → v0) )
14:                         end if
15:                 end if
                    {relaxation}
16:                 for all v1 ∈ adjacent(v0) do  {relaxation}
17:                         d = v0.distance + edge (v0 → v1).length
18:                         if (v1.distance > d) then
19:                                 v1.distance = d
20:                                 v1.predecessor = v0
21:                                 pq.add (d , v1)
22:                         end if
23:                 end for
24:         end if
25: end while
```

In order to generate a non-complete graph we need to issue XNN query for every vertex (city) in the graph.

3.1.2 Proof of Correctness

To prove that the XNearestNeighbor algorithm is correct, it is only necessary to prove that it is equivalent to the Dijkstra's algorithm. The only modification to the Dijkstra's algorithm is that we keep track of the edges connecting source and each vertex that will be closed (step). The rest of the algorithm is the same as the Dijkstra's algorithm.

3.1.3 Complexity Analysis

In each iteration, one open vertex with the minimum distance is extracted and closed. Then in the relaxation process, shortest distances are updated for all open neighbors of the newly closed vertex. The distances can be maintained in a priority queue, which is usually implemented by heaps. Given the *O(log V)* (*V* is the number of vertices (cities) in the graph) complexity of updating heaps, and *O(V)* updates in handling a graph, the time complexity of computing *X* nearest neighbor is *O(V log V)* (where *V* is the number of nodes (cities)). This time complexity is for a single source problem. If we want all-pairs *X* nearest neighbors, this algorithm need to be executed *V* times, so the time complexity is $O(V^2 \log V)$.

3.2 Top Shortest X Percentage (TSXP) Algorithm

The main characteristic of the subset of the edges generated from XNN algorithm is that the subset contains the top *X* nearest neighbors for each city. This means that probably most of the edges in this subset are among the shortest path edges from the complete graph. In order to have a better understanding on how the subset of edges generated from XNN algorithm compare to the subset of edges containing shortest path edges in the complete graph, we introduced TSXP algorithm.

3.2.1 Algorithm Description

This algorithm computes the complete graph (finds all shortest path distances for all the cities in the graph) using Dijkstra's algorithm and put all the shortest path distances in a minimum priority queue. In the next step selects the top *X*% shortest edges from the priority queue.

3.2.2 Complexity Analysis

If we want all-pairs shortest paths (a complete graph), Dijkstra's algorithm need to be executed V times, so the time complexity is $O(V^2 \log V)$ (where V is the number of nodes (cities)).

3.3 Minimum Spanning Tree (MST) Algorithm

Minimum Spanning Tree is a tree that connects all of the vertices and whose total weight is minimized. This makes it a good candidate to generate a subset of edges of the complete graph edges. The edges produced in MST have a major characteristic which is connecting all the cities.

3.3.1 Algorithm Description

Algorithm: MST(*g*, *src*)

Input: *g* is the graph of vertices that may include cities; *src* is a random source node
Output: a min priority queue (*mstpq*) containing minimum spanning tree edges

```
1:  Initialize a min priority queue mstpq
2:  Initialize a min priority queue pq
3:  src.distance = 0
4:  pq.update (0 , src)
5:  while (pq is not empty) do
6:      v0 = pq.extractMin()
7:      if (v0.closed == false)
8:          v0.closed = true
9:          if ((v0.isCity == true) & (v0 != src ))
10:             nextvertex = v0
11:             prevertex = v0.predecessor
12:             d = 0
13:             while (prevertex.isCity != true) do
14:                 d = d + (prevertex, nextvertex).length
15:                 nextvertex = prevertex;
16:                 prevertex = prevertex.predecessor
17:             end while
18:             d = d + (prevertex, nextvertex).length
19:             mstpq.add(d, edge (prevertex $\rightarrow$ v0))
20:         end if
21:         for all adjacent nodes v1 of v0 do
22:             if (v0.isCity == true) then d=0 else d= v0.distance
23:                 newdist = d+(v0,v1).length
24:             end if
25:             if ((v1.distance > newdist) & (!v1.close))
26:                 v1.distance = newdist
27:                 v1.predecessor = v0
28:                 pq.add (v1.distance, v1)
29:             end if
30:         end for
31:     end if
32: end while
```

The introduced algorithm is the Prim's algorithm. It operates much like Dijkstra's algorithm for finding shortest paths in a graph. At each step, the shortest edge that connects tree to a closed city is added to the tree. We continue until the tree spans all the cities in the graph.

The output of the algorithm is a priority queue called *mstpq*. It contains all the edges of the minimum spanning tree.

3.3.2 Complexity Analysis

The performance of this algorithm depends on how we implement the min priority queue *mstpq*. Java's priority queue provides $O(log(V))$ time for the dequeing method. The body of **while** loop in step 5 is executed V times, and since each *extractMin*() operation takes $O(log(V))$ time, the total time for all calls to *extractMin*() is $O(V\ log(V))$. The **while** loop in step 13 is executed at most V-2 times. The **for** loop in step 21 is executed $O(V)$ times. Thus the total time for the MST algorithm is $O(V\ log(V)+ V^2) = O(V^2)$.

3.4 Hilbert Value Clustering (HVC) Algorithm

Another way of choosing a subset of edges from a complete graph is using clustering. We group vertices as clusters and then choose a random vertex from each cluster. The next step is to compute the SPT for each of the selected vertices. The edges in generated SPTs not only connect the source vertex to vertices in its own cluster but also connect the source vertex to the vertices in other clusters. This way we try to make sure that for each pair of clusters, there exists at least one edge that connects the clusters.

3.4.1 Algorithm Description

This algorithm first clusters the graph using Hilbert values. We use Hilbert space-filling curve. It first described by the German mathematician David Hilbert in 1891 [35]. Figure 3.1 shows the 3 steps of building Hilbert Curve. In addition to its mathematical importance, Hilbert space-filling curve has applications to dimension reduction. To compute the Hilbert value we have used Butz algorithm [13] . It is a mapping function using several bit operations such as shifting, exclusive OR, etc. It takes cities' coordinates as input and returns the Hilbert value as an output and we put it in a minimum priority queue called *pq* (step 4-9). This way we have converted 2 dimensional values (coordinates) to 1 dimensional.

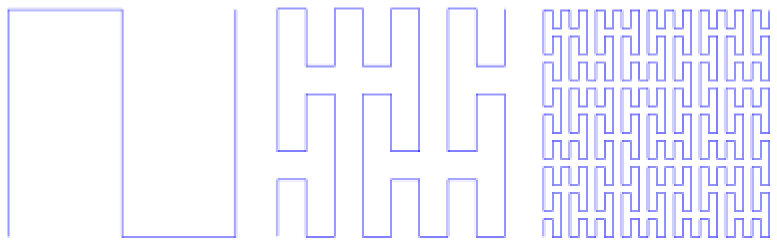


Figure 3.1: 3 steps toward building the Hilbert curve

The example below shows how we cluster cities using their Hilbert values.

Example: In the Figure 3.2 below we have a set of Hilbert values belong to a set of cities. They have been shown in a line from value 0 to value 1 (Hilbert value is always a double value between 0 and 1).

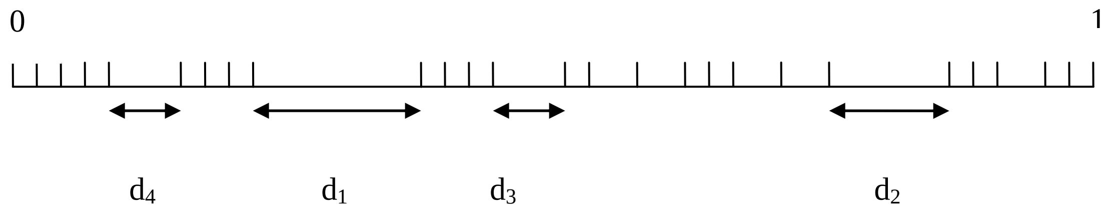


Figure 3.2: Example of Hilbert values and the differentiating between them

If the number of desired clusters is 2, using d_1 we can cluster the Hilbert values to 2 clusters. Hilbert values from 0 to the left city in d_1 make the first cluster and Hilbert values from the right city in d_1 to 1 make the second cluster. If the number of desired clusters is 3, we use d_1 and d_2. The first cluster is from 0 to the left city in d_1 and the second cluster is from right city in d_1 to the left city in d_2 and the third cluster is from the right city in d_2 to 1. If we want to have *n* clusters we choose the top *n-1* maximum differentiation values to cluster cities.

In steps 10-21 we compute the differentiate of each sequent eateries (Hilbert value implies a city) in *pq* and add the differentiate as a key and the pair of vertices as value to the maximum priority queue called *difpq*. The reason that we compute the differentiate between the sequent entries is that we try to cluster nodes that are close to each other based on their Hilbert value. The top entry in *difpq* is demonstrating the pair of cities that have the maximum difference in their Hilbert value. In steps 23-27 we poll the maximum value from *difpq* and add the first city (*v1*) from the pair of cities (*v1, v2*) to the array of cities called *selectedCities*. Each city in the pair (*v1, v2*) belongs to a different cluster. This way we make sure that from each cluster we have selected a city.

The output will be a subset of selected cities (*selectedCities*). The number of selected cities is equal to the number of the clusters (*c*). Each city in this subset belongs to a different cluster. Using *selectedCities* we generate a subset of edges. In the next step we construct the shortest path tree using Dijkstra's algorithm for each city in the *selectedCities* and add the edges in SPT to the subset of edges.

Algorithm: HilbertValueCluster(*g, c*)

Input: *g* is the graph of vertices that may include cities; *c* is the number of clusters
Output: *selectedCities* subset of cities

```
1:  Initialize a min priority queue pq
2:  Initialize a max priority queue difpq
3:  Initialize an array of cities selectedCities
4:  for all vertices v do
5:          if (v0.isCity == true)
6:              hv = HilbertValue(v.coordinates)
7:              pq.add(hv, v)
8:          end if
9:  end for
10: keyvalue = pq.extractMin()
11: hv1 = keyvalue.key
12: v1 = keyvalue.value
13: while (pq is not empty) do
14:     keyvalue = pq.extractMin()
15:     hv2 = keyvalue.key
16:     v2 = keyvalue.value
17:     difference = hv2– hv1
18:     difpq.add(difference,(v1,v2))
19:     hv2 = hv1
20:     v2 = v1
21: end while
22: n = 0
23: while n < c do
24:     keyvalue = difpq.extractMax()
25:     selectedCities.add(keyvalue.value.v1)
26:     n++
27: end while
```

3.4.2 Complexity Analysis

The performance of this algorithm depends on how we implement the priority queues *pq* and *difpq*. Java's priority queue provides *O(log(V))* time for the enqueing and dequeing methods. The body of **for** loop is executed at most *V* times. The first **while** loop is executed at most *V* times and since each extractMax()operation takes *O(log(V))* time, the total time for all calls to extractMax() is *O(V log(V))*. The second **while** loop is executed *c* times which is the number of clusters. It can be at most equal to *V*. So the maximum time to execute the second **while** is *O(V log(V))*. Thus the total time for the HilbertValueCluster algorithm is *O(V log(V))*.

3.5 Shortest Path Tree Leaves Hilbert Value Clustering (LHVC) Algorithm

In this algorithm, a number of SPTs are generated and their edges form the subset. We first choose a random vertex and apply Dijkstra's algorithm to generate an SPT. In the next step we choose the SPT leaves and cluster them using their Hilbert values. The clustering algorithm is the same as algorithm introduced in 3.4. Then, for each cluster, we choose a random city and construct an SPT. We add the edges in all these SPTs to form the subset of edges.

3.6 Random SPT (RSPT) Algorithm

In order to be able to compare the introduced edge-generating algorithms, we have introduced RandomSPT algorithm. This algorithm generates SPT edges for random vertices in the graph. This provides a good worst case scenario in order to generate a subset of edged from the complete graph.

3.6.1 Algorithm Description

Algorithm: RandomSPT(*g*, *src*)

Input: *g* is the graph of vertices that may include cities; *src* is a random source node
Output: min a priority queue (*sptpq*) containing all shortest path edges

```
1:  Initialize a min priority queue sptpq
2:  Initialize a min priority queue pq
3:  src.distance = 0
4:  pq.update (0 , src)
5:  while (pq is not empty) do
6:      v0 = pq.extractMin()
7:      if(v0.closed == false)
8:          v0.closed = true
9:          if (v0.isCity == true)
10:             sptpq.add(v0.distance, edge (src → v0)
11:         end if
12:         {relaxation}
13:         for all adjacent nodes v1 of v0 do
14:             d = v0.distance + edge (v0 → v1).length
15:             if (v1.distance > d)
16:                 v1.distance = d
17:                 v1.predecessor = v0
18:                 pq.add (d , v1)
19:             end if
20:         end for
21:     end if
22: end while
```

This algorithm finds the shortest path tree for a random source in a graph using Dijkstra's algorithm. In order to generate a subset of edges from the complete graph, we need to compute the *RandomSPT* for a number of random cities as source node (*src*).

3.6.2 Complexity Analysis

The time complexity of this algorithm is the same as the Dijkstra's algorithm which is $O(V \log V)$(where v is the number of vertices (cities)). Assume that we compute this algorithm for R random sources. R can be at most equal to V. The total time complexity will be $O(V^2 \log V)$.

Chapter 4 TSP Tour Construction Algorithms

4.1 Greedy Algorithm

4.1.1 Algorithm Description

In this algorithm, we view an instance as a non-complete graph. A TSP tour is then simply a Hamiltonian cycle in this graph, i.e., a connected collection of edges in which every city has degree 2.

In the following algorithm, we try to build up a Hamiltonian cycle one edge at a time, starting with the shortest edge, and repeatedly adding the shortest remaining available edge, where an edge is *available* if it is not yet in the tour and if adding it would not create a degree-3 vertex or a cycle of length less than N. Since instead of a complete graph as the input information, we only have a subset of edges in the complete graph, the result of the algorithm might not be a Hamiltonian cycle and be a set of disjoint simple paths.

Algorithm: Greedy (*mainpq, pathTable, num_cities*)

Input: *mainpq* -- a min-priority queue containing any subset of edges in the complete graph.
pathTable -- At the end of the algorithm *pathTable* will contain all the simple paths generated from the algorithm.
num_cities -- the number of cities.
Output: Either a tour of n cities or a set of disjoint simple paths

```
1:  while (mainpq is not empty & a complete tour has not been found) do
2:          edge e = mainpq.extractMin()
3:          Case 1: e intersects with some simple path
4:                  Subcase 1: e intersects with exactly one simple path
5:                          (a) e intersects with one of the endpoints
6:                          add e to the simple path
7:                          (b) e intersects with both of the endpoints {adding e will make a
                            cycle}
8:                          if the simple path includes all the cities in the graph then
9:                                  add e to the simple path and make the TSP tour
10:                         end if
11:                 Subcase 2: e intersects with two simple paths
12:                         (c) e intersects with endpoints of the simple paths
13:                         connect two simple paths using e
14:                         check if the resulting path is a complete tour or not.
15:         Case 2: e doesn't intersect with any simple path
16:                 Create a new simple path containing e
17: end while
```

4.1.2 Proof of Correctness

Prove by Induction:

Induction hypothesis: after each iteration of the **while** loop, there is either a complete tour or a set of disjoint simple paths. We prove the hypothesis by induction on the number *i*>= 0 of iteration of the while loop.

Basis: $i = 0$. Initially we have zero simple path which satisfies the condition that we have a set of simple paths

Induction: $i > 0$. Assume that iteration i-1 satisfies the hypothesis that we have a set of simple paths or a tour.

Now we need to prove that after iteration i we are going to have a set of simple paths or a tour. If we already have a tour in iteration i-1 then the condition to enter the while loop will not be satisfied. If we have a set of simple paths in iteration i-1 then one of the following cases will be accepted.

We prove the induction by exhaustive analysis. The two cases in the algorithm correspond to all possible cases of an extracted edge and its relationship with the existing set of disjoint simple paths. We conclude that the hypothesis holds after the i^{th} iteration.

4.1.3 Complexity Analysis

For time complexity, the while loop steps 1 can be done in at most *O(E)* times, where *E* is the number of edges in the subset of edges of the complete graph. Maximum value of *N* is the number of edges in the complete graph. In step 2 *extractMin*() method takes *O(log(E))* time for the dequeing priority queue. Therefore, the overall time complexity is *O(E log(E))*.

4.2 Initial Tour Construction Algorithm

If the greedy algorithm above result in a set of disjoint simple paths. We built an initial TSP tour by connecting the simple paths randomly together.

4.2.1 Algorithm Description

Algorithm: InitialTour (*simplePaths, individualCities*) {Constructing an arbitrary initial TSP tour}

Input: *simplePaths* -- a linked list of disjoint simple paths
individualCities – a linked list of individual cities (the cities that do not belong to any simple path)
Output: A TSP tour that connects all the cities (cities included in *simplePaths* and *individualCities*)

```
1:  Initialize tour T to an empty tour
    {saving the starting vertex of the tour}
2:  startCity = first vertex of the first simple path in simplePaths
3:  sp1 = fist simple path in simplePaths
4:  add the cities and edges of sp1 to T
5:  simplePaths.remove(sp1)
6:  for each simple path sp2 ∈ simplePaths do
7:          add the cities and edges of sp2 to T
8:          add edge (last city in sp1 → first city in sp2) to T
9:          sp1 = sp2
10:         simplePaths.remove(sp2)
11: end for
12: for each city c ∈ individualCities do
13:         add edge (last city in T → c ) to T
14:         add c to T
15: end for
16: add edge (last city in T → startCity ) to T
```

This algorithm constructs an arbitrary initial TSP tour by iterating through all disjoint simple paths and connecting them together and then iterating through all individual cities and adding them to the tour.

4.2.2 Complexity Analysis

For time complexity, the first iteration can be done in at most *O(S)* times, where *S* is the number of disjoint simple paths. Maximum value of *S* can be *V/2* where *V* is the number of cities in the TSP tour.

The second iteration can be done in at most *O(V)* times where *V* is the number of cities in the TSP tour. Therefore, the overall time complexity of this algorithm is *O(V)*.

Chapter 5 Modified 2-Opt Improvement Algorithms

In this chapter we propose some modifications to the 2-opt algorithm in a way that it results in the best possible improvement while having a non-complete graph to work on instead of a complete graph.

5.1 Original 2-Opt Algorithm

5.1.1 Algorithm Description

The following algorithm is called the original 2-opt algorithm because unlike the other 2-opt algorithms introduced in this thesis (which in each iteration consider an edge from the subset of edges in the complete graph and finds the matching edges for that), this algorithm deletes two distinct edges from the tour, thus breaking the tour into two paths, and then reconnects those paths in the other possible way (only if the newly adding edges exist in the subset of edges of the complete graph).

Algorithm: OriginalOptAlg (***initialTour***, ***subsetOfEdges***) {improving the initial TSP tour}

Input: *initialTour* -- a TSP tour that connects all the cities
subsetOfEdges – a subset of edges in the complete graph
Output: A TSP tour that connects the same cities that the initial tour connects however with a shorter length.

```
1:  Initialize tour T by initialTour
    {edges e1,e2,e3 and e4 are four distinct edges & v1, v2, v3 and v4 are distinct}
2:  for each edge e1 (v1,v2) $\in$ T do
3:      for each edge e2 (v3,v4) $\in$ T do
4:          maxSaving = 0
5:          if ( edge e3 (v1,v3) $\in$ subsetOfEdges ) & ( edge e4 (v2,v4) $\in$
               subsetOfEdges) then
6:              saving = ( e1.length + e2.length ) - ( e3.length + e4.length )
7:          end if
8:          if saving > maxSaving then
9:              maxSaving = saving
10:             maxedge1 =  e3
11:             maxedge2 =  e4
12:         end if
13:     end for
14:     if maxSaving > 0 then {apply the maximum saving and updating the tour}
15:         update tour T by deleting e1 and e2 and adding maxedge1 and maxedge1
16:     end if
17: end for
```

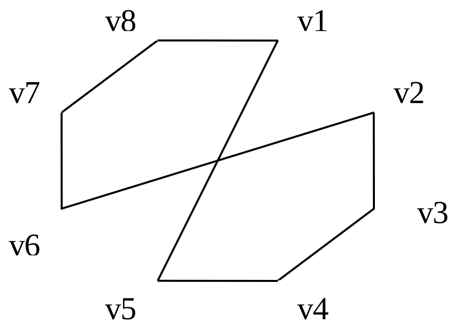
5.1.2 Complexity Analysis

For time complexity, the outer for loop in step 2 can be done in at most *O(E)*, where *e* is the number of edges in the TSP tour. The for loop in step 3 can be done in at most *O(E)*. Step 15 updates the tour consequently updates a hashtable. The maximum number of update to the hashtable is *V*. So the time complexity is *O(V)*. Therefore, the overall time complexity is $O(E^2 + E\ V)$.

5.2 Modified 2-Opt Algorithm

5.2.1 Algorithm Description

The following algorithm is a modified version of the 2-Opt algorithm introduced by Croes [22]. The approach of this 2-Opt algorithm is to compute savings for all pairs of edges in the tour. For each pair, if the saving is more than zero, applies it. The following algorithm has not been given a complete graph. Instead, it is given a subset of edges in the complete graph. Knowing this fact, computing the saving for all pairs of edges is unnecessary. In the following algorithm, we have taken another approach in applying 2-Opt algorithm. For each edge that is not already in the tour, there exist two other edges that can be matched with that particular edge. Adding the edge to the tour results in four possible formations in which only two are valid. See Figure 5.1 Assume that the tour shown in (a) is the initial tour. Edge (v1, v2) is an edge in the subset of edges of the complete graph. In order to apply 2-Opt and adding (v1, v2) to the tour, we have to find its matching cases. Considering the adjacent edges of v1 and v2 which are (v8, v1), (v1, v5), (v3, v2) and (v2, v6), there is four different cases of adding (v1, v2) to the tour. Two of them will result in a non-complete tour and two of them will result in a complete tour. These two valid cases are called **matching cases** of edge (v1, v2). The following 2-Opt algorithm computes the saving for the two matching cases. The case with more saving (if the saving is positive) will be considered in applying 2-Opt.



(a)

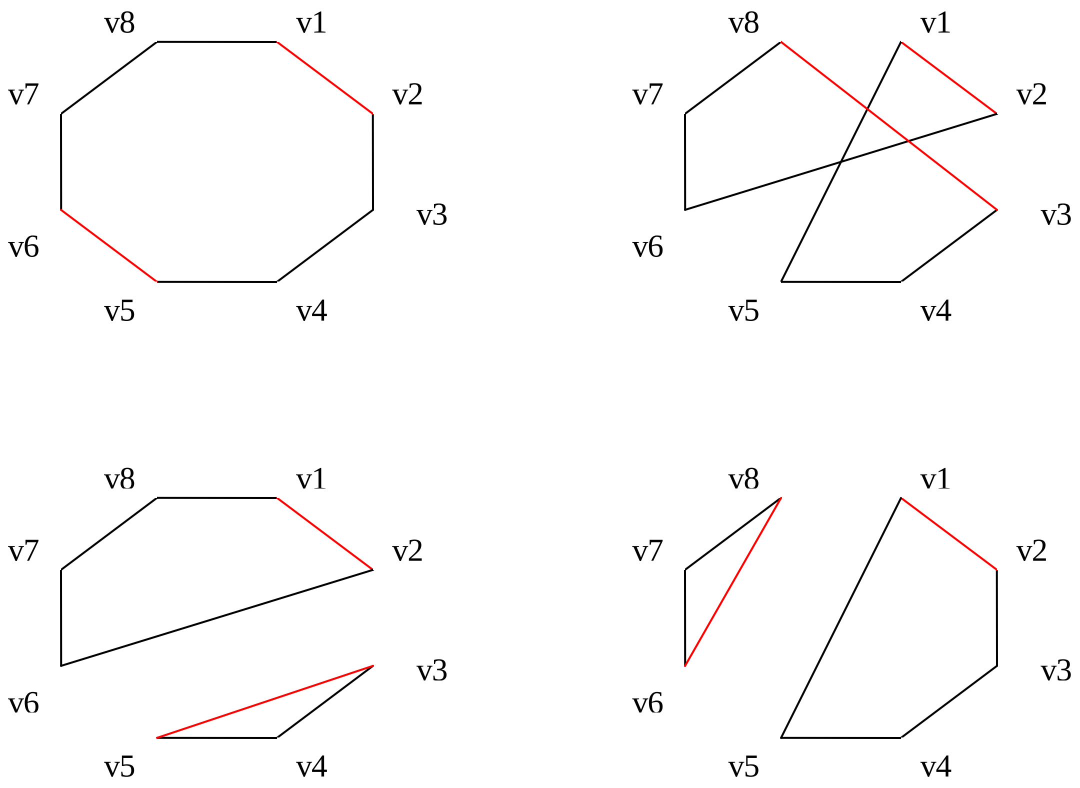


Figure 5.1: Adding edge (v1, v2) to the tour and its four possibilities

In applying the 2-Opt moves, different ordering of edges examined may result in different final tours. There is no particular ordering that guarantee the shortest final tour. Since applying a 2-Opt move may change the matching cases of the next 2-Opt move.

To provide illustration of the vertices' name used in the following algorithm, we use the vertices' name in Figure 5.1.

Algorithm: ModifiedOptAlg(*initialTour*, *subsetOfEdges*) {improving the initial TSP tour}

Input: *initialTour* -- a TSP tour that connects all the cities
subsetOfEdges **--** a subset of edges in the complete graph
Output: A TSP tour that connects the same cities that the initial tour connects however with a shorter length.

```
1:  Initialize tour T by initialTour
    {edges e1,e2,e3,e4,e5,e6 and e7 are distinct edges }
2:  for each edge e1 (v1,v2) ∈ subsetOfEdges do
3:        maxSaving = 0
4:        if e1 ∈ T then continue
5:        end if
6:        Let edges e2(v5,v6) and e3(v3,v8) be the two matching cases of e1
          {case 1: compute the saving of adding e1, e2 and deleting e4(v1,v5), e5(v2,v6)}
7:        if e2 ∈ subsetOfEdges then
8:              saving1 = (e1.length + e2.length) - (e4.length + e5.length)
9:        else saving1 = 0
10:       end if
          {case 2: compute the saving of adding e1, e3 and deleting e6(v1,v8), e7(v2,v3)}
11:       if e3 ∈ subsetOfEdges then
12:             saving2 = (e1.length + e3.length) - (e6.length + e7.length)
13:       else saving2 = 0
14:       end if
          {apply the maximum saving and updating the tour}
15:       if saving1 ≥ saving2 & saving1>0 & saving2>0  then
16:             update tour T by deleting e4 and e5 and adding e1 and e2
17:       end if
18:       if saving2 >saving1 & saving1>0 & saving2>0  then
19:             update tour T by deleting e6 and e7 and adding e1 and e3
20:       end if
21: end for
```

5.2.2 Proof of Correctness

Prove by Induction:

Induction hypothesis: after each iteration of the for loop, there is a complete tour that has equal or less length than the previous iteration

Basis: i=0. Initially we have a TSP tour that connects all the cities which satisfies the condition that we have a complete tour.

Induction: i>0. Assume that iteration i-1 satisfies the hypothesis that we have a complete tour that has equal or less length than the previous iteration.

Now we need to prove that after iteration i we are going to have a complete tour that has equal or less length than (i-1)th iteration: One of the two cases will be accepted.

We prove the induction by exhaustive analysis. The two cases in the algorithm correspond to all possible cases of updating the tour using the new added edges. If the newly added edge does not exist in the subset of edges then the saving is zero. If the newly added edge exists in the subset of edges then we compute the saving. Between the two cases the one with higher saving will be selected. If the higher saving is positive then the tour will be updated to a complete tour with less distance. If the higher saving is not positive then the tour will not be updated and the length of the tour won’t change. We conclude that the hypothesis holds after the ith iteration.

5.2.3 Complexity Analysis

For time complexity, the for loop steps 2-20 can be done in at most *O(N)* times, where *N* is the number of edges in the generated edge set (which is the subset of edges in the complete graph). Maximum value of *N* is the number of edges in the complete graph which is *V*(V-1)* where *V* is the number of nodes (cities) in the TSP tour. Steps 15 and 18 are updating the tour which results in updating a hashtable. The maximum number of updates to the hashtable is *V*. So the time complexity of this step is *O(V)*. Therefore, the overall time complexity is *O(N V)*.

5.2.4 Data Structure

hashTable (Vertex , AdjacentVertices)

In the 2-Opt algorithm, in each iteration we may remove two edges and add two new edges. If we use linkedList as data structure for arcs in the tour then we may have some difficulties for adding and removing edges in the middle of the linkedList (because after updating the tour, the sequence of the edges in the tour will change). Moreover we like to avoid using linkedList. So I have defined a different data structure for arcs in the tour. It is a hashTable that contains vertices' (cities) coordinates as key and their two adjacent vertices (cities) in the tour as its value.

5.3 Saving List 2-Opt Algorithm

5.3.1 Algorithm Description

The following algorithm is a modified version of the original 2-Opt algorithm. The original 2-Opt algorithm deletes two edges, thus breaking the tour into two paths, and then reconnects those paths to form a complete tour.

For each edge in the subset of edges (step 6), this version of 2-Opt algorithm computes the saving of adding the edge and its matching edge to the tour (and deleting two other edges). Updating the tour by adding an edge to the tour gives rise to two valid possible tours. The one with more saving (if the saving is positive) will be selected and put in a priority queue (steps 20 & 24). The iteration goes on for all the edges in the subset of edges.

In the next step, the algorithm removes savings from the max-priority queue and checks whether it is valid or not. A saving is *valid* if applying 2-Opt to the complete TSP tour using that saving will not result in a non-complete tour. If it is valid then the 2-Opt will be applied according to the extracted saving. It continues until the max-priority queue is empty (steps 29-34).

The algorithm will repeat the above iteration from computing the savings of updating the tour using each edge in the subset of edges to applying the saving only if they are *valid*. This loop (step 4) continues until there is no saving more than zero in updating the tour using edges in the subset of edges (step 27).

To provide illustration of the vertices' name used in the following algorithm, we use the vertices' name in Figure 5.1.

Algorithm: OptAlgSavingList (***initialTour***, ***subsetOfEdges***)

Input: *initialTour* -- a TSP tour that connects all the cities
subsetOfEdges **--** a subset of edges in the complete graph
Output: A TSP tour that connects the same cities that the initial tour connects however with a shorter length.

```
1:  Initialize tour T by initialTour
2:  Initialize max priority queue savingList
    {edges e1,e2,e3,e4,e5,e6 and e7 are distinct edges }
3:  flag = true
4:  while (flag)
5:      maxSaving = 0
6:      for each edge e1 (v1,v2) ∈ subsetOfEdges do
7:          if e1 ∈ T then continue
8:          end if
9:          Let edges e2(v5,v6) and e3(v3,v8) be the two matching cases of e1
        {case 1: compute the saving of adding e1,e2 and deleting e4(v1,v5), e5(v2,v6)}
10:         if e2 ∈ subsetOfEdges then
11:         saving1 = (e1.length + e2.length) - (e4.length + e5.length)
12:         else saving1 = 0
13:         end if
        {case 2: compute the saving of adding e1,e3 and deleting e6(v1,v8), e7(v2,v3)}
14:         if e3 ∈ subsetOfEdges then
15:             saving2 = (e1.length + e3.length) - (e6.length + e7.length)
16:         else saving2 = 0
17:         end if
18:         if saving1 ≥ saving2 & saving1>0 & saving2>0  then
19:             maxSaving = saving1
20:             savingList.add (maxSaving, (e1, e2))
21:         end if
22:         if saving2 >saving1 & saving1>0 & saving2>0  then
23:             maxSaving = saving2
24:             savingList.add (maxSaving, (e1, e3))
25:         end if
26:     end for
27:     if savingList is empty then flag = false
28:     end if
29:     while (savingList is not empty)
30:         saving = savingList.extractMax()
31:         if saving is valid then
32:             update tour T by applying the saving
33:             remove the two added edges to the tour from subsetOfEdges
34:         end if
35:     end while
36: end while
```

We check the *validity* of each saving before applying it to the tour. It is because a saving may not be *valid* after applying previous update or updates to the tour. The following example illustrates why and how we check the *validity* of a saving.

In this example, the initial tour is shown in (a) Figure 5.2. Assume that we compute two savings in this tour in order to apply the 2opt algorithm. The first saving is deleting edges "*ah*" and "*pi*" and adding edges "*ai*" and "*ph*" (Figure 5.2 (b)). The second saving is deleting edges "*dm*" and "*el*" and adding edges "*dl*" and "*me*" (Figure 5.2 (c)). The savings will be put in a max-priority queue in order to be applied in a non-increasing order. Assume the first saving described above has higher saving than the second one. After applying the first saving the tour will be updated to Figure 5.2 (b). Updating the tour, results in updating the data structure which records predecessor and successor of each vertex in the tour. Applying the second saving to this updated tour will result in a non *valid* tour showed in Figure 5.2 (d). As you can see, the tour has been divided to two disjoint tours which is not a *valid* tour. The conclusion is that we can not just compute the savings and apply them by order. Each saving has to be checked whether it is valid for the updated tour or not. Now this question arises that how we can check the validity of a saving? It can be done using the data structure which records predecessor and successor of each vertex. In this example, before applying the saving ,which deletes edges "*dm*" and "*el*" and adds edges "*dl*" and "*me*", we check the predecessor and the successor of the four included vertices "*m,e,d,l*". According to the updated tour which is shown in Figure 5.2 (b) the predecessor and the successor for each vertex is as below:

Vertex name (predecessor, successor): *m* (*d,n*), *e* (*f,l*), *d* (c,m), *l* (*e,k*).

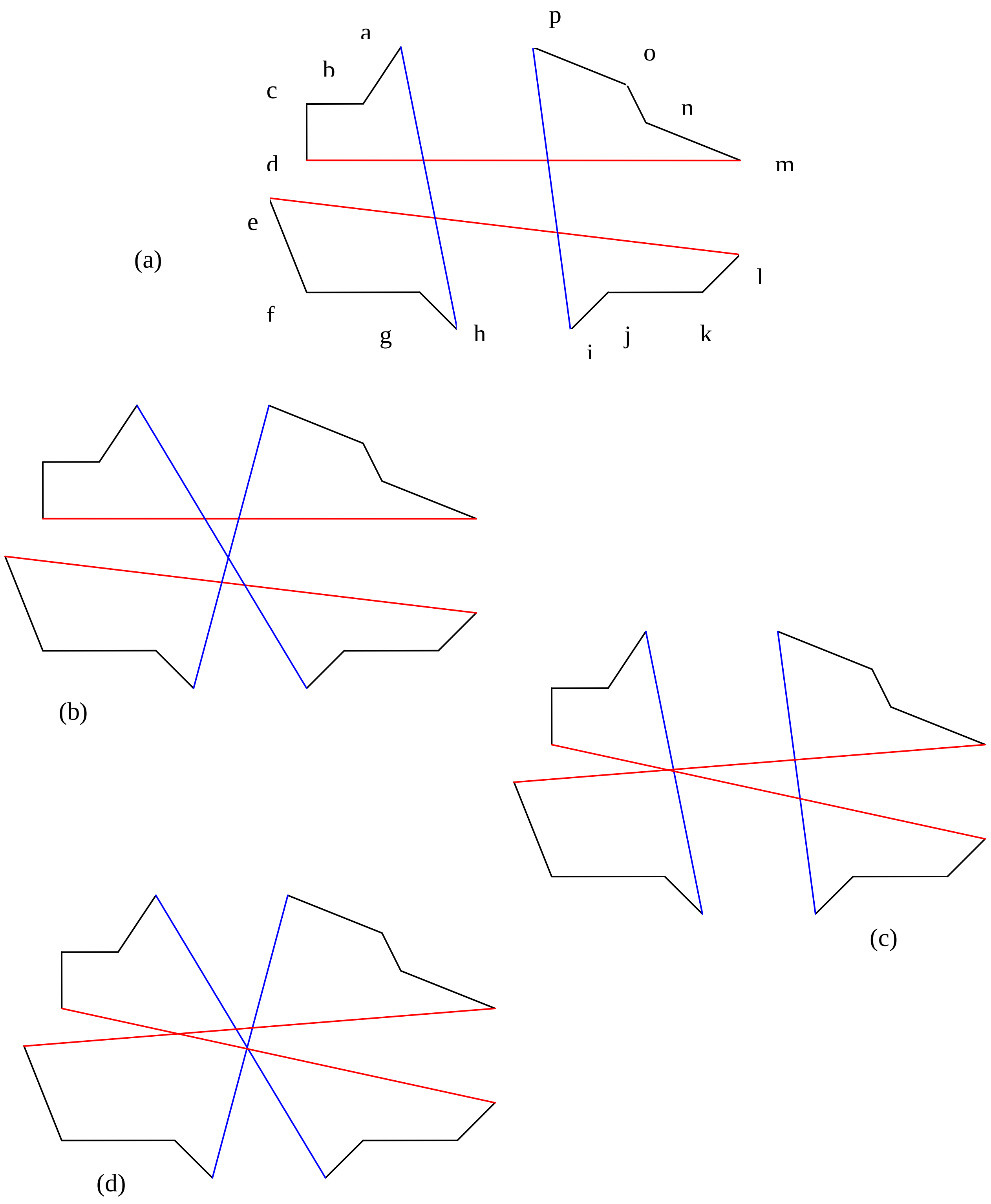


Figure 5.2: Necessity of checking validity of the saving before applying the saving

A saving is valid when *from* and *to* vertices in the edges, which are going to be added to the tour, both are either predecessor or successor of the vertices included in the saving. In this example, the edges that will be added to the tour saving are: "*dl*" and "*me*". Now we see that "*d*" is the predecessor for "*m*" and "*l*" is the successor for "*e*" so this pair is not *valid*. (if "*d*" and "*l*" were both predecessor (or successor) of "*m*" and "*e*", this would be a valid pair.)

5.3.2 Proof of Correctness

Prove by Induction:

Induction hypothesis: after each iteration of the for loop, there is a complete tour that has equal or less length than the previous iteration

Basis: i=0. Initially we have a TSP tour that connects all the cities which satisfies the condition that we have a complete tour.

Induction: i >0. Assume that iteration i-1 satisfies the hypothesis that we have a complete tour that has equal or less length than the previous iteration.

Now we need to prove that after iteration i we are going to have a complete tour that has equal or less length than $(i\ -1)^{th}$ iteration: One of the following cases will be accepted.

We prove the induction by exhaustive analysis. The two cases in the algorithm correspond to all possible cases of updating the tour using the new added edges. If the higher saving is positive then the tour will be updated to a complete tour with less distance. If the higher saving is not positive then the tour will not be updated and the length of the tour won't change. We conclude that the hypothesis holds after the i^{th} iteration.

5.3.3 Complexity Analysis

The **while** loop in step 4 can be done in O(N) times (where N is the number of edges in the generated edge set). It is because we apply at least one of the savings in the *savingList* will be applied (The fist saving is always a *valid* saving). The maximum value of N is the number of edges in the complete graph which is $V*(V-1)$ where V is the number of vertices (cities) in the TSP tour.

For time complexity, The **for** loop in step 6 to 26 can be done in at most $O(N)$ times. The **while** loop in step 29 can be done in at most $O(N)$ times. Step 32 updates the tour consequently updates a Hashtable. The maximum number of update to the Hashtable is V. So the time complexity is $O(V)$. Therefore, the overall time complexity is $O(N^2 + N^2V)$.

Chapter 6

Implementation

The testing data source, the implementation details on the data structures and the algorithms are discussed in this chapter.

6.1 Test Instances

In our experimental results we use instances that not only obey the triangle inequality but also are geometric in nature and distances are computed under a standard metric which is the Euclidean. Many of the applications of the symmetric TSP are of this sort, and most recent published studies have concentrated on them.

The instances that we use in our experiments, they have been solved to optimally. So we have the optimal distance for each of them and we can compare the TSP tour resulted using our algorithms with the optimal result.

Table 6.1: Test Instances

Group Name	Group Range	Number of Instances	Instances Name
2	$10^1 – 10^2$	9	"berlin52","eil76","rat99", "kroA100","kroB100", "kroC100", "kroD100", "kroE100", "rd100".
2.5	$10^2 – 10^{2.5}$	30	"eil101","lin105","pr107","pr124", "bier127","ch130","pr136","pr144", "ch150","kroA150","kroB150", "pr152", "u159", "rat195", "d198", "kroA200","kroB200","ts225", "tsp225","pr226", "gil262","pr264", "a280","pr299","lin318","rd400", "fl417", "pr439","pcb442", "d493".
3	$10^{2.5} – 10^3$	7	"u574", "rat575", "p654", "d657", "u724", "rat783","dsj1000".
3.5	$10^3 – 10^{3.5}$	19	"pr1002","u1060","vm1084", "pcb1173","d1291","rl1304", "rl1323","nrw1379","fl1400", "u1432", "fl1577", "d1655", "vm1748","u1817","rl1889", "d2103","u2152","u2319", "pr2392".

The instances used in our experiments derived from a database of standard instances called TSPLIB [64]. It contains instances with as many as 85,900 cities, including many from printed circuit board and VLSI applications, as well as geographical instances based on real cities. For instances larger than 2392 cities, we got out of memory error. We divided instances, which we used to run the tests, into four groups as shown in Table 6.1. Two of the above instances have been shown in figure below.

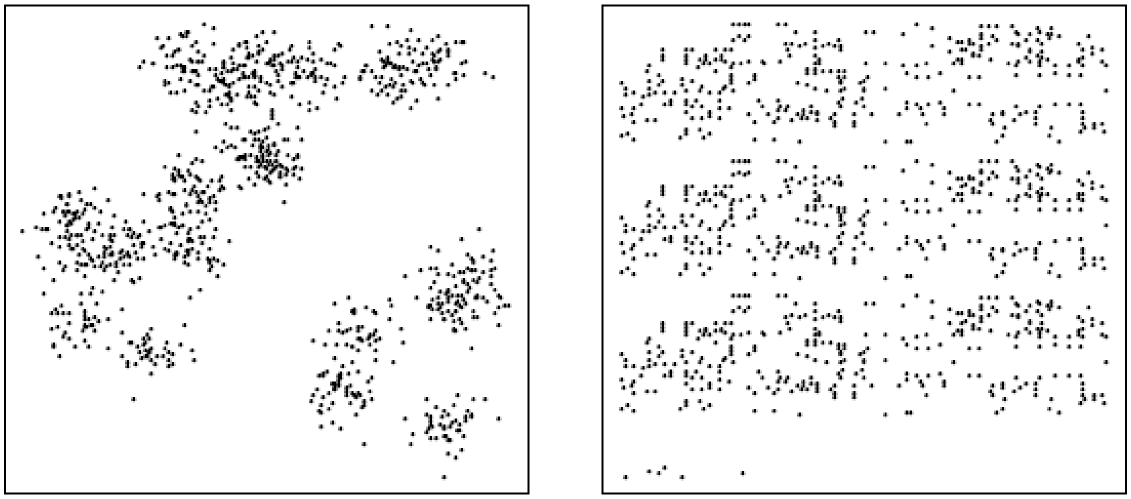


Figure 6.1: TSPLIB instances dsj1000 (left) and pr1002 (right).

Group “2” contains instances having 50-100 cities. Group “2.5” contains instances with 100-500 numbers of cities. Group “3” encloses instances having 500-1000 cities. Group “3.5” includes instances from 1000 number of cities to “2392” number of cities. The largest instance that has been tested in our experiments is “pr2392”. The instances larger than this encountered the out of memory error.

6.2 Implementation Detail

In this section, the implementation details about the edge-generating algorithms, the modified 2-Opt algorithms and the data structures are introduced.

6.2.1 Details in Edge-generating Algorithms

In our edge-generating edges algorithms we encounter some important modifications in the implementation. We discuss them in this subsection.

Including implicit edges

In our edge-generating algorithms we tried to make the most use out of the produced information. Producing SPT, not only we add the edges of the tree to the subset of edges, but also we add the edges connecting each vertex and its ancestors to the subset. Having more edges in the subset of edges is helpful both in tour construction and tour improvement algorithms.

Updating the priority queue

In the introduced edge-generating algorithms (XNN Algorithm, TSXP Algorithm, HVC Algorithm, LHVC Algorithm, MST Algorithm, RSPT Algorithm), we produce SPT or XNN based on the Dijkstra's algorithm. This algorithm uses a priority queue which stores edges as *values* and rank those edges based on their distances from the source (*key*). During computation of SPT or XNN, if we found a shorter distance from the source to a vertex than before, then we add the distance and the particular vertex to the priority queue. Therefore the priority queue may need to be updated.

Using Java's priority queue, we are not able to update an element in the priority queue with the same *value* and a different *key*. So if we just add the new element to the priority queue without removing the old element, the priority queue will get larger and extracting an element from the priority queue will take more time. However, it does not affect the correctness of the algorithm since extracting an element with the minimum key; we mark the vertex as *closed*. Extracting an element from the priority queue, we only use the information if the vertex is not *closed*. On the other hand having a longer priority queue will affect the time that we extract an item from the queue.

In order to avoid this trouble, we have introduced a new priority class that is updatable. In this class we use a Hashtable to keep track of the elements in the priority queue. The *value* in the priority queue is the *key* in the Hashtable. Before adding a (*key*, *value*) to the priority queue we check the Hashtable. If the *value* already existed in the Hashtable, we remove (old *key*, *value*) from the priority queue and add (new *key*, *value*) to the priority queue.

6.2.2 Building the TSP tour step by step

Now a detailed description is given on how to build the TSP tour using test instances. There are four phases:

1. Non-complete graph generation phase: Given a graph and list of its cities, a subset of edges will be produced using one of the six introduced edge-generating algorithms (XNN, TSXP, HVC, LHVC, MST, and RSPT). Producing a subset of

edges, in which the number of edges is *X*% of the number of edges in the complete graph, depends on which edge-generating algorithm we use.

- X-Nearest Neighbor Algorithm: We find *Y* such that the total number of edges generated by *Y*% of nearest neighbors of each city is *X*% of total number of edges in a complete graph. Since some of the nearest neighbor edges may be duplicated, there is no mathematical formulation to find *X*% of total number of edges. For each *X* in our experiments we found *Y* by trial and error.

- Top Shortest X Percentage Algorithm: Using cities as vertices, we generate the complete graph. In the next step we put all the edges in the complete graph in a min-priority queue and then we select the top *X*% of the edges.

- Hilbert Value Cluster Algorithm: Producing *Y* clusters using this algorithm, the next step is to select a random city for each cluster and generate SPT for that city. The total number of distinct edges in SPTs is *X*% of the total number of edges in the complete graph. Because of the same reason as the XNN algorithm, there is no mathematical formulation to find *Y*. For each *X* in our experiments we found *Y* by trial and error.

- SPT Leaves Hilbert Value Cluster Algorithm: The procedure is the same as the previous algorithm. This algorithm has a slightly different approach than the previous algorithm.

- Minimum Spanning Tree Algorithm: The number of edges generated by this algorithm is equal to the number of cities minus one. In order to generate *X*% of the total number of edges in the complete graph, after execution of this algorithm we add the edges generated by this algorithm to a list. We repeat the execution while not considering the edges in the list until the total number of generated edges is more than or equal to *X*% of the number of edges in the complete graph. The sources to compute MSTs are selected randomly.

- Random SPT Algorithm: We execute this algorithm until the total number of generated edges is more than or equal to *X*% of the number of edges in the complete graph. As the name of this algorithm suggested, the sources to compute SPTs are selected randomly.

2. Greedy algorithm: Subsequent to generating a subset of edges using one of the six proposed algorithm, we call the Greedy algorithm. The output is either a TSP tour or a set of disjoint simple paths. If the output is a TSP tour we go to step 4 in order to improve it and if the output is a set of disjoint simple paths we go to step 3.

3. Initial tour construction phase: Having a set of disjoint simple paths, we use them to generate a random initial TSP tour.

4. Improving the TSP tour: We improve the TSP tour using the three introduced 2-Opt improvement algorithms described in Chapter 5. The 2-Opt moves in these algorithms contain removing 2 edges from the tour add two new edges. To make these moves effortless we use Hashtable data structure. In this Hashtable cities are as *keys* and the predecessor and successor cities are as *values*. This way we can change the predecessor and successor of a city faster than using linkedList data structure. We improve the initial tour that has been constructed using Greedy algorithm (If the output of Greedy algorithm is not a tour then we use Initial tour construction algorithm to construct the initial TSP tour). The initial tour will be improved using the three introduced improvement algorithms which are Original 2-Opt algorithm, Modified 2-Opt algorithm and Saving List 2-Opt algorithm.

Chapter 7

Experiments

In this chapter we introduce our experimental methodology and summarize experimental results for the six edge-generating heuristics and three modified 2-Opt algorithms. We conduct extensive experiments on all algorithms we introduced. We test our algorithms using TSPLIB instances [64].

In our experiments, for each test instance, we compute the average result and standard deviation for each group. Using standard deviation we can measure how widely values are dispersed from the average value (the mean). The standard deviation is calculated using the following formula (*a* is the average and *n* is the number of graphs):

$$\sqrt{\ } (\sum (x - a)^2 / n)$$

In subsection 7.1, we compare the result of six non-complete graph generating algorithms. In this subsection we only complete the first three steps explained in 6.2.2. We do not improve the generated TSP tours. In subsection 7.2, we compare the three improvement tour algorithms. We complete the four steps in 6.2.2 for each improvement algorithm.

7.1 Performance of non-complete graph Generating Algorithms

In this section we summarize the results obtained by our six non-complete graph generating heuristics on Euclidean instances. Every TSP heuristic can be evaluated in terms of two key parameters which are its running time and the quality of the tours that it produces. In general every edge is a potential candidate for membership in the tour and there are $N(N-1)/2$ edges to be considered. In each of the six non-complete graph generating algorithms we generate a subset of edges out of $N(N-1)/2$ number of edges. In order to be able to evaluate these six algorithms, we generate a subset of edges having 100%, 75%, 50%, 35%, 25% and 10% of the total edges.

7.1.1 X Nearest Neighbor (XNN) Algorithm

Using this algorithm, in order to generate the desired percentages of the edges we estimate the percentage of nearest neighbors that we have to produce for each city. As shown in Table 7.1, for example to generate 50% of the edges in the complete graph, we have to compute about 45% of the nearest neighbors for each city.

Table 7.1: Desired Percentage for XNN Algorithm

% of edges	% of nearest neighbors
100	100
75	65
50	45
35	30
25	22
10	10

Figure 7.1 illustrates the average quality of the TSP tour generated, for each group of instances, using the subset of edges generated by XNN Algorithm. We have compared the distance of the tour with the optimal (or the best known) tour distance. The *variance from the optimal* in axis *y* is calculated using the following simple formula:

(generated tour distance) - (optimal tour distance) / (optimal tour distance)

Looking at the results in Figure 7.1 we can conclude that generating 35% of the total edges (about 30% of nearest neighbors) with this algorithm produce a TSP tour with 20% variance from the optimal TSP tour. The TSP tour quality is almost the same for all the subset of edges having more than 35% of the total edges. This suggests that generating 100% or 35% of the total edges using XNN algorithm, results in almost the same TSP tours (that are 20% variance from the optimal TSP tour).

To have some idea on how the results in each group are dispersed from the average result in Figure 7.1, we show the standard deviation of the results in Figure 7.2. It shows that in average the results vary about ±10% from the average result. The 2 and 2.5 groups have a higher value than 3 and 3.5 groups. This implies that in the 2 and 2.5 groups, there are a number of results that have outranged values compare to the average value.

To illustrate the worst variance and how bad the tour could be in the test result, we show the deviation of test result in each group in Figure 7.3. The *x*-axis shows the variance of tour quality compare to the optimal tour. It has been repeated for all subset of edges' sizes (100%, 75%, 50%, 35%, 25% and 10%). The *y*-axis in this figure is the percentage of instances within certain quality out of total number of instances in each group (2, 2.5, 3 and 3.5). It is computed from the following formula:

(number of instances that are within certain tour quality (shown in x-axis) in a group) / (total number of instances in that group)

For each certain tour quality and for each group of instances the percentage is equal to the high number of percentage of the group minus the low number of percentage of the group.

For example in 100% for tour quality within 10%-20% the approximate percentage of instances in group 2 = 75 - 0 = 75 (75%)

in group 2.5 = 135 - 75 = 60 (60%)

in group 3 = 225 - 135 = 90 (90%)

in group 3.5 = 295 - 235 = 60 (60%)

Since we have divided the number of instances within certain tour quality to the total number of instances in that group, the sum of the percentages of instances for “each group” should be 100% and the sum of percentages for all groups is 400%.

For example in Figure 7.3 for subset of 100% for graph size of 2 we have:

0 + 77.7777778 + 11.1111111 + 11.1111111 = 100

For subset of 100% for graph size of 2.5 we have:

3.3333333 + 63.3333333 + 36.6666667 + 0 = 100

For subset of 100% for graph size of 3 we have:

0 + 85.7142857 + 14.2857143 + 0 = 100

For subset of 100% for graph size of 3.5 we have:

0 + 63.1578947 + 36.8421053 + 0 = 100

As shown in Figure 7.3, the worst case tour quality belongs to subset size 10% in group 2.5 for about 0.05% of instances.

Another aspect of TSP tour generation that we capture in our experimental results is the time to generate subset of edges using the proposed non-complete graph generation algorithm. The average time to generate subset of edges using XNN algorithm is shown in

Figure 7.4. It shows that for the graphs having less than $10^{2.5}$ numbers of cities, the time to generate a subset of edges is always less than 5 seconds. However the grows fast when the size of the graph grows. For graphs having less than $10^{3.5}$ cities, computing the complete graph takes more than 70 seconds. As shown, in

Figure 7.4, computing the 35% of the edges takes about 1/3 time less than computing all the edges. In addition we know from the tour qualities in Figure 7.1 that computing 35% of the total edges, we can produce TSP tours having almost the same tour quality as generating the complete graph. Therefore using XNN algorithm and generating about 35% of the total edges, we are able to produce good quality TSP tours in about 1/3 times less time.

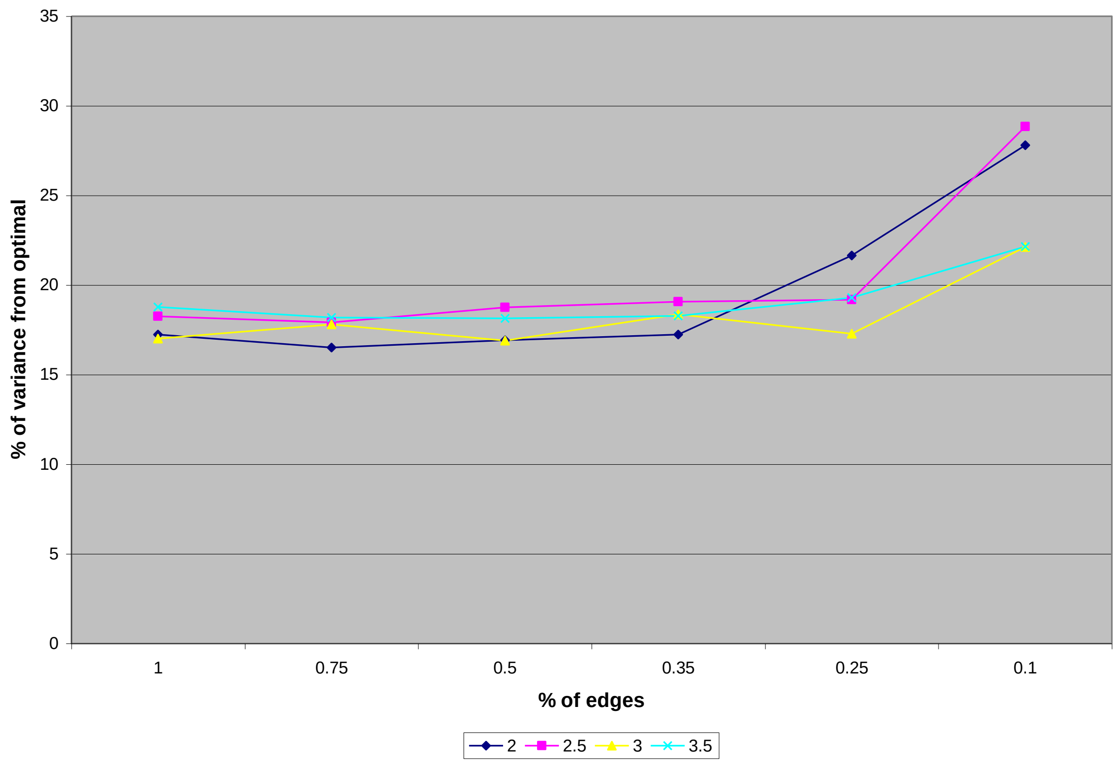


Figure 7.1: Average TSP tour quality for XNN Algorithm

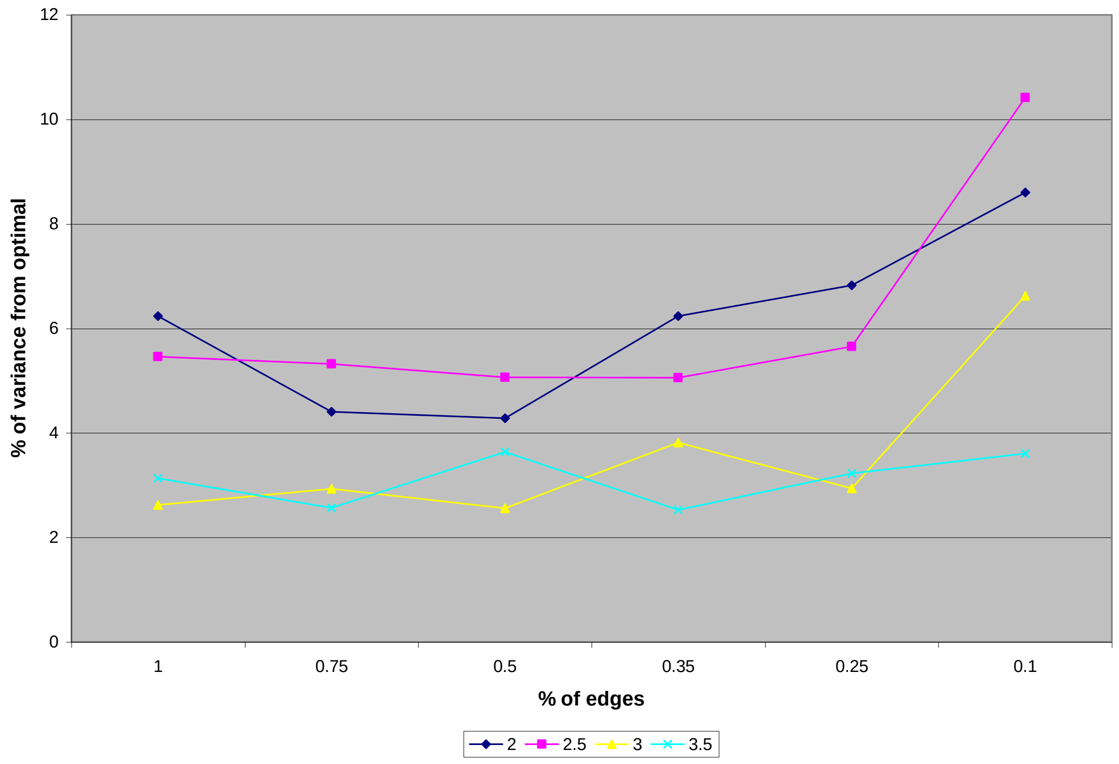


Figure 7.2: Standard Deviation TSP tour quality for XNN Algorithm

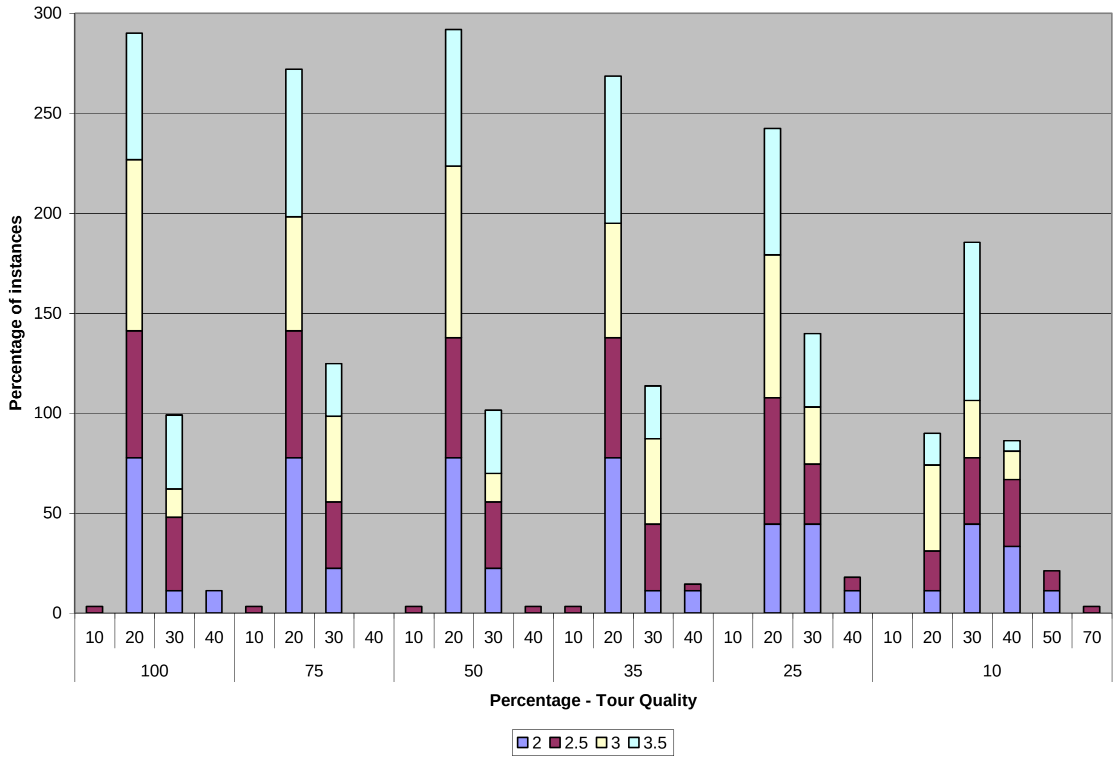


Figure 7.3: Percentage of instances within certain tour quality in each group using XNN

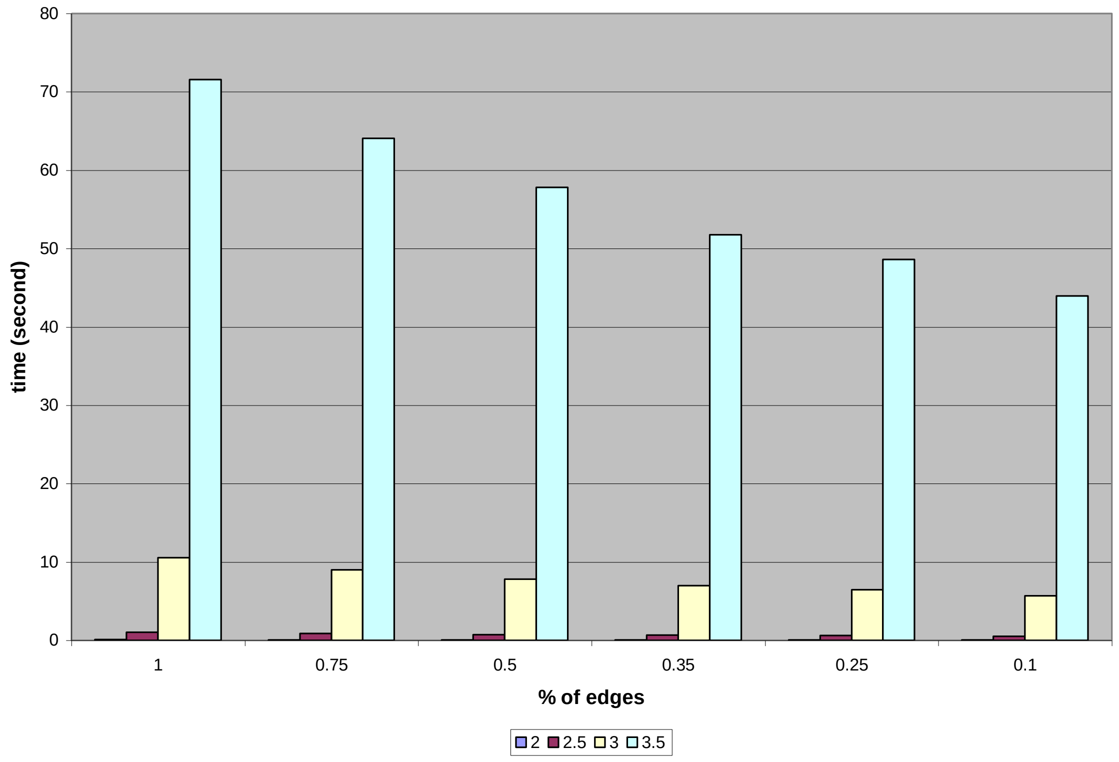


Figure 7.4: Average time for generating subset of edges using XNN algorithm

7.1.2 Top X Percentage (TSXP) Algorithm

Using Top Shortest X Percentage algorithm to generate the subset of edges, the resulted TSP tour quality is illustrated in

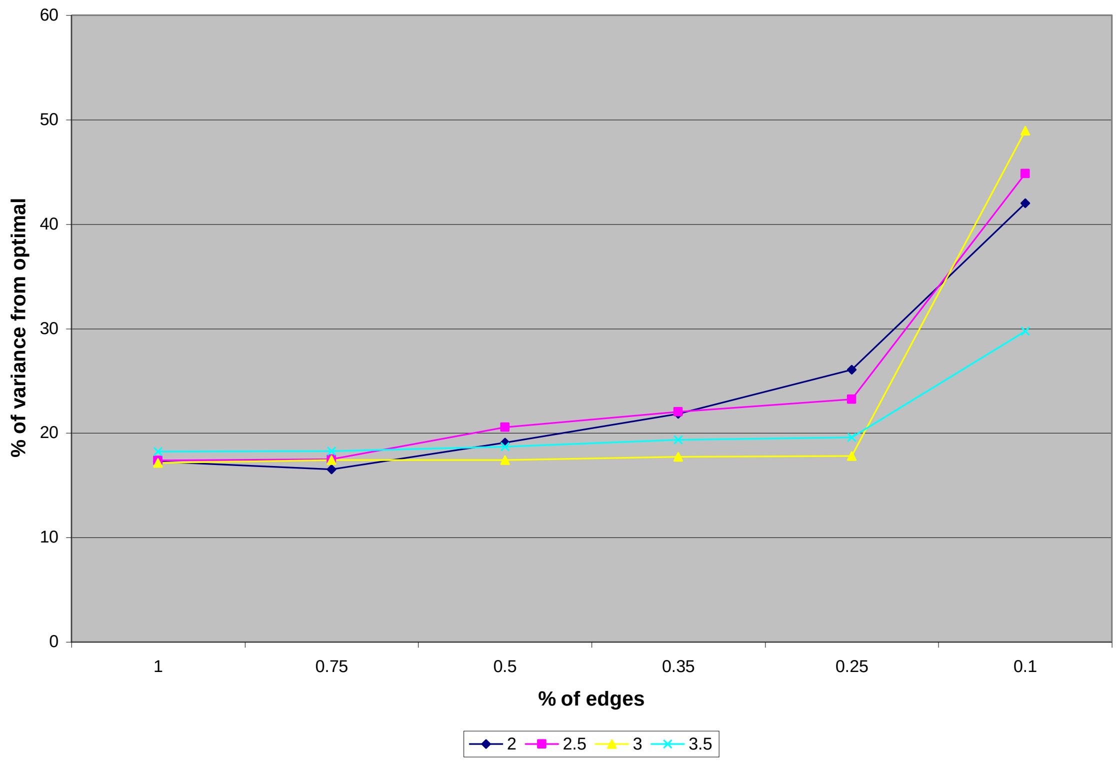


Figure 7.5. In TSXP algorithm we select the X% top shortest edges out of all the edges in the complete graph. The result shows that generating only 75% of the total edges in the complete graph will result in almost the same TSP tour distances as having a complete graph. For 3 and 3.5 groups this is true for up to 25% of the edges. For 2 and 2.5 groups, the variance of the TSP tour quality compare to the optimal tour rises by reducing the size of the subset of edges. Generally for all the groups, generating only 25% of the total edges we can produce TSP tours with 30% variance from the optimal TSP tour. Generating 0.1% of the total edges the quality of the resulted TSP tours will decrease to 50% variance from the optimal TSP tour. The result of using TSXP algorithm implies the

fact that edges used in generating a TSP tour are mostly among the shortest edges. Figure 7.6 illustrates the standard deviation of the results. This helps us to observe differentiations among the results and their variances from the average result. As it is shown, for groups 3 and 3.5 the standard deviation does not change with the size of the subset up to 25%. This means that the graph instances in these two groups result in TSP tours within the same range of quality and there is not any instance that results in an outrange TSP tour quality. Since the average tour quality for these two groups is less than 20% and the standard deviation is less than 5%, we can conclude that TSP tour quality for all the graph instances are within (20 ± 5)% variance from the optimal TSP tour. However, this is not true for subset of edges having 10% of the total edges. The reason is that the quality of TSP tours decrease a lot compare to other subsets and this results in TSP tour that are more variance from the average.

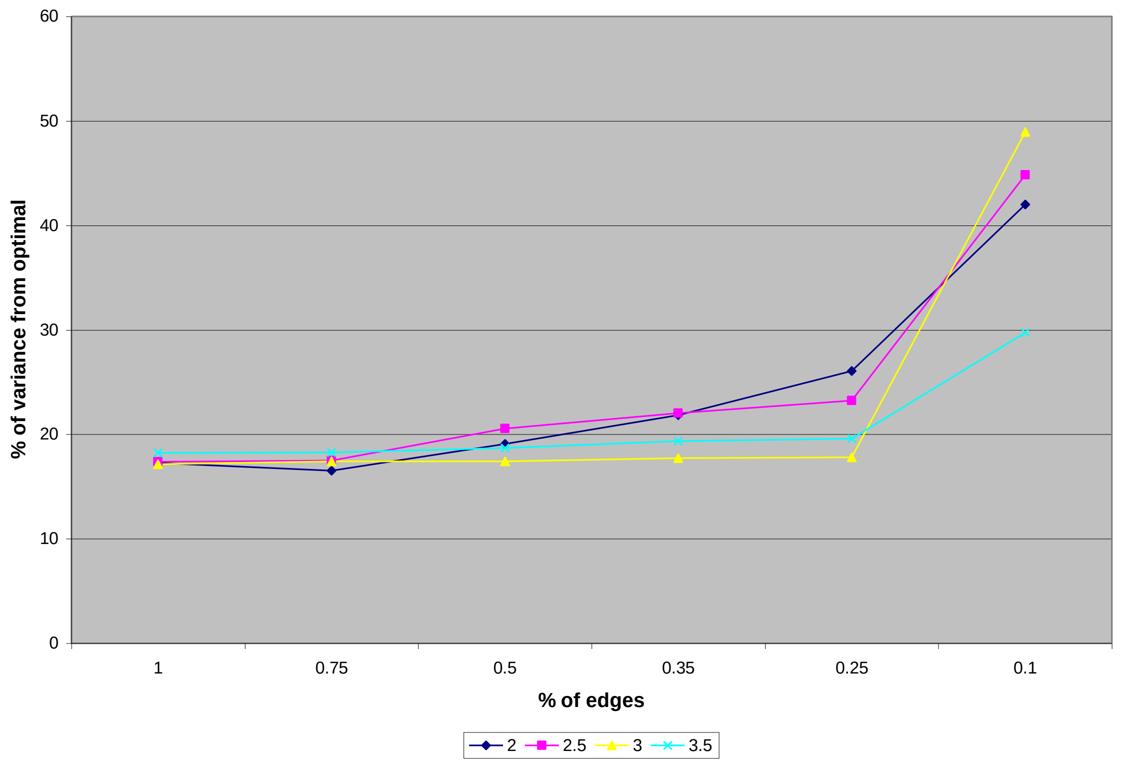


Figure 7.5: Average TSP tour quality for TSXP Algorithm

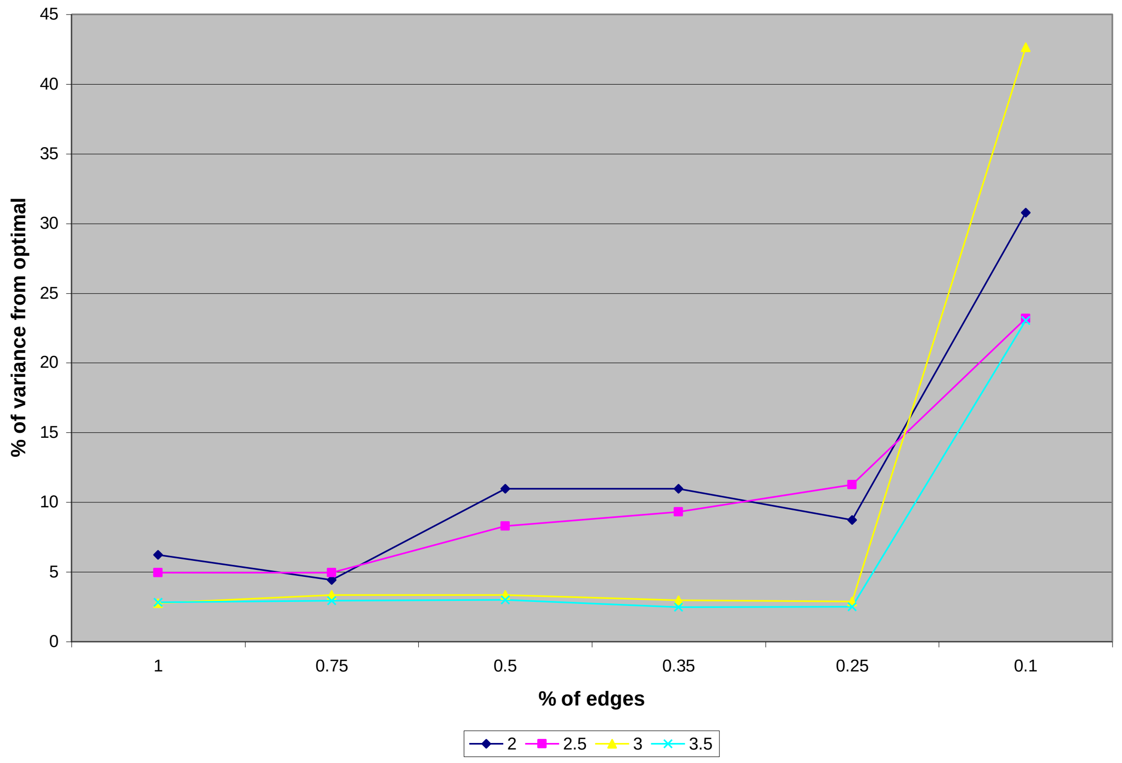


Figure 7.6: Standard Deviation TSP tour quality for TSXP Algorithm

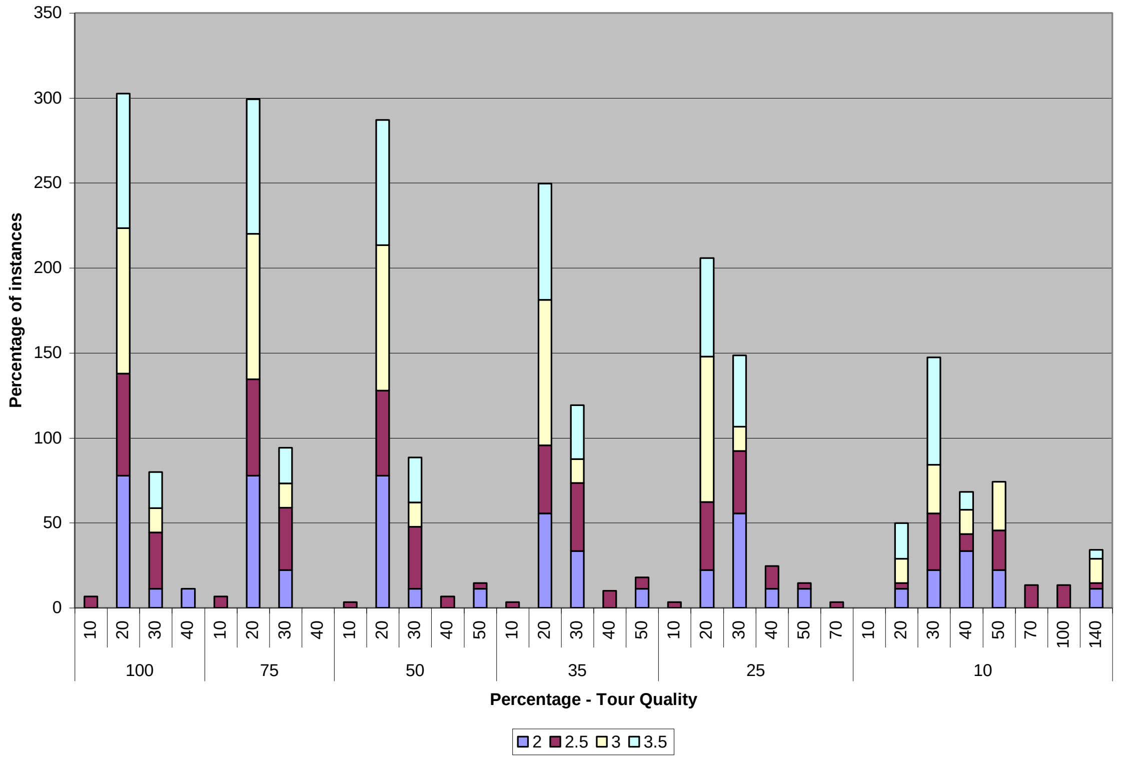


Figure 7.7: Percentage of instances within certain tour quality in each group using TSXP

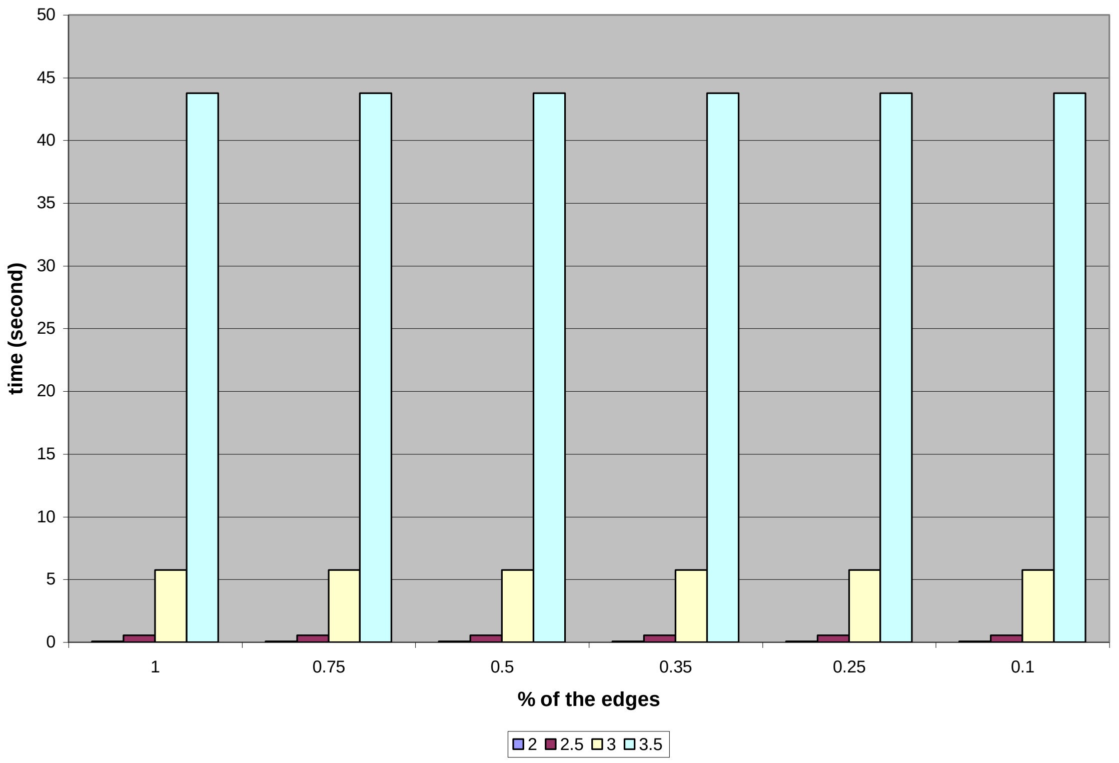


Figure 7.8: Average time for generating subset of edges using TSXP algorithm

Figure 7.7 shows the percentage of instances that are within a certain tour quality for each group of test instances. Comparing this figure with the average tour quality gives a good idea regarding the outrange instances in each group. The worst case tour quality belongs to subset size 10% in all four groups within 10%-140% of the optimal tour. As you can see by reducing the size of the subset of edges, the number of outliers increases a bit and their tour quality decreases.

Figure 7.8 demonstrates the time required to generate the subset of edges for different sizes of the subset. In this algorithm we first generate a complete graph and then sort the edges and choose the top x% of the shortest edges. That is why the time to generate the subset of edges is equal for all the percentages. Computing the complete graph is the most time-consuming step of generating TSP tour. Increasing the size of the graph, the time to generate the complete graph will raise fast.

7.1.3 Minimum Spanning Tree (MST) Algorithm

In this subsection, we demonstrate the TSP tour quality that has been produced using MST algorithm to generate the non-complete graph.

Figure 7.9 shows the quality of the TSP tour compared to the optimal tour. The result is similar to the X nearest neighbor algorithm. Generating only 50% of the edges, the distance of the generated TSP tour is almost the same as generating a TSP tour having a complete graph. For subset of edges with more than 50% of the total edges, the TSP tour is only about 20% higher than the optimal tour. Reducing the size of the subset, the variance of generated TSP tour compare to the optimal tour will grow up to 35%.

Interestingly the trend in TSP tour quality, that has been produced using MST algorithm to generate the non-complete graph, is similar to using XNN algorithm to generate the non-complete graph. For both of them having a subset of edges with more than 25% of the total edges, the quality of the generated TSP tour is less than 0.2% variance from the optimal tour. For the 10% subset of edges, the TSP tour quality is about 30% more than the optimal tour.

Figure 7.10 illustrates the variance of the tour distances from the average distance. It shows that the difference of the result compare to the average was at most 10%. This result is also similar to the result of the SD in XNN algorithm. The SD for smaller graphs with less than $10^{2.5}$ numbers of cities is higher than the graphs having more than $10^{2.5}$

numbers of cities.

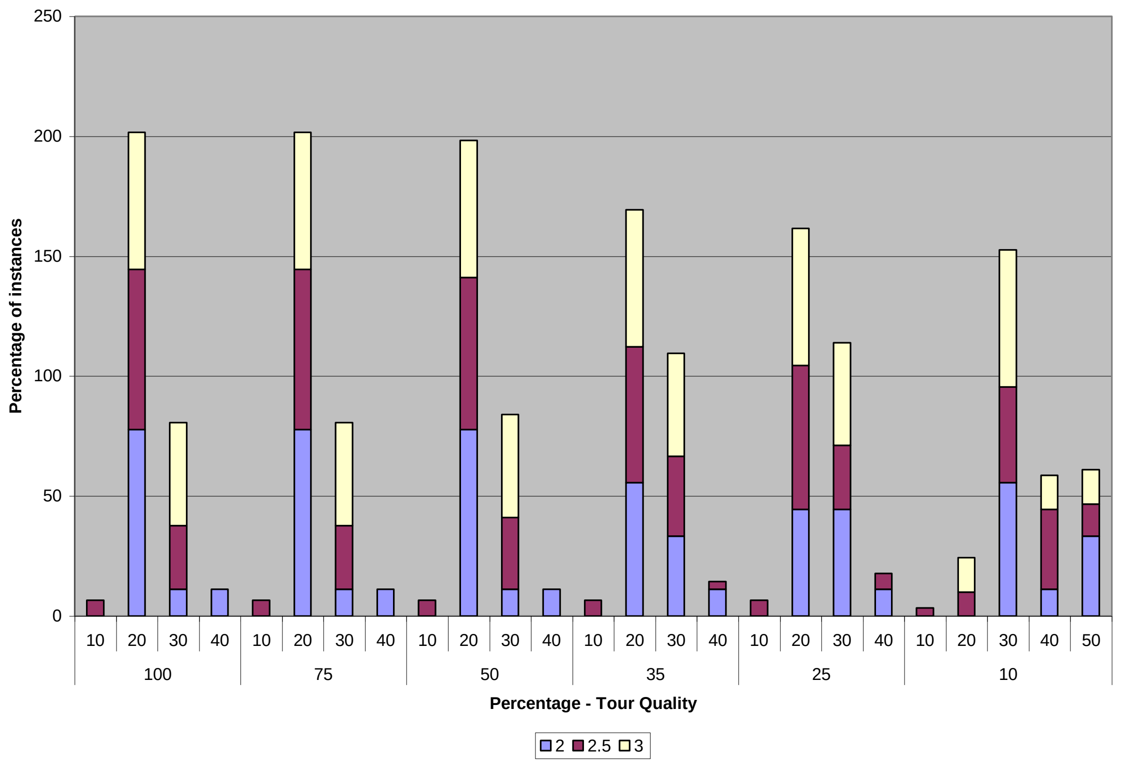


Figure 7.11 shows the percentage of instances that are within a certain tour quality for each group of test instances. The worst case tour quality belongs to subset size 10% for all three groups and their tour quality is within 40%-50%.

One thing that we notices in using MST algorithm, is that producing MSTs with the subset of edges having 100% of the edges we may not use the whole 100% of the total edges and that is why the result for 100% and 75% are the same in terms of tour quality.

Finally the time consumption in order to generate the subset of edges using MST algorithm is shown in Figure 7.12. The time rises rapidly with the size of the graph. For the large graphs we can save about half of time by computing 25% of the total edges and the quality of the resulted TSP tour will only decreases about 0.05% compare to the optimal TSP tour.

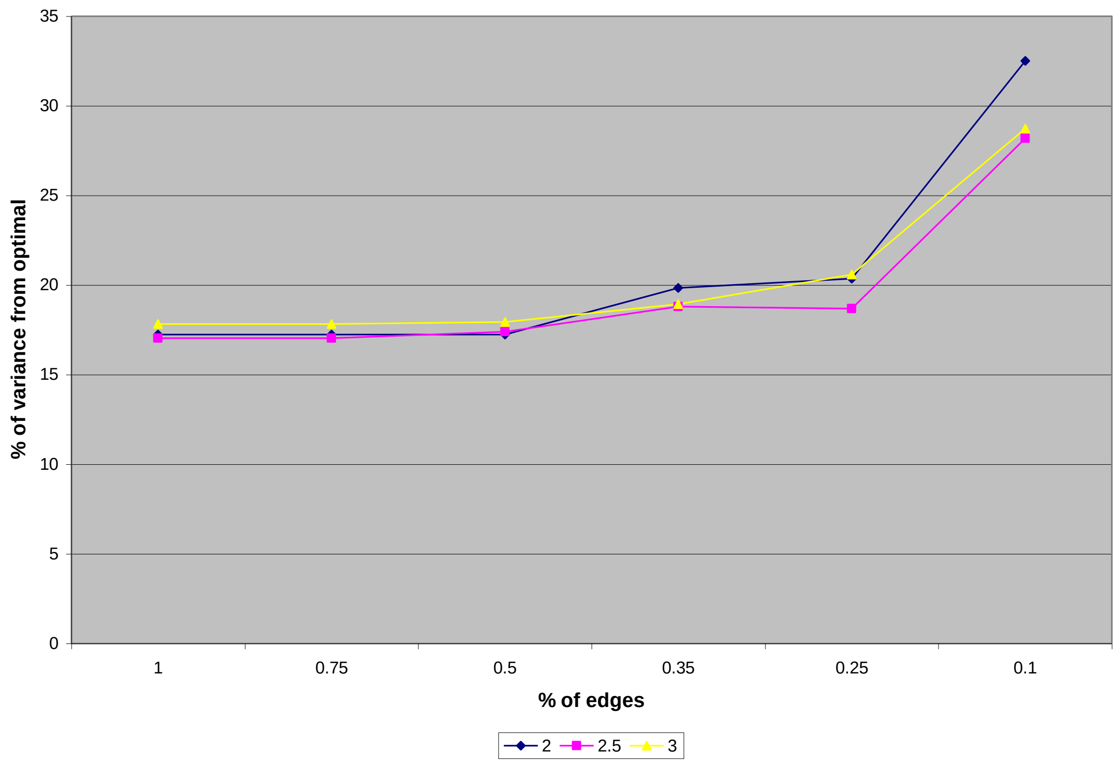


Figure 7.9: Average TSP tour quality for MST Algorithm

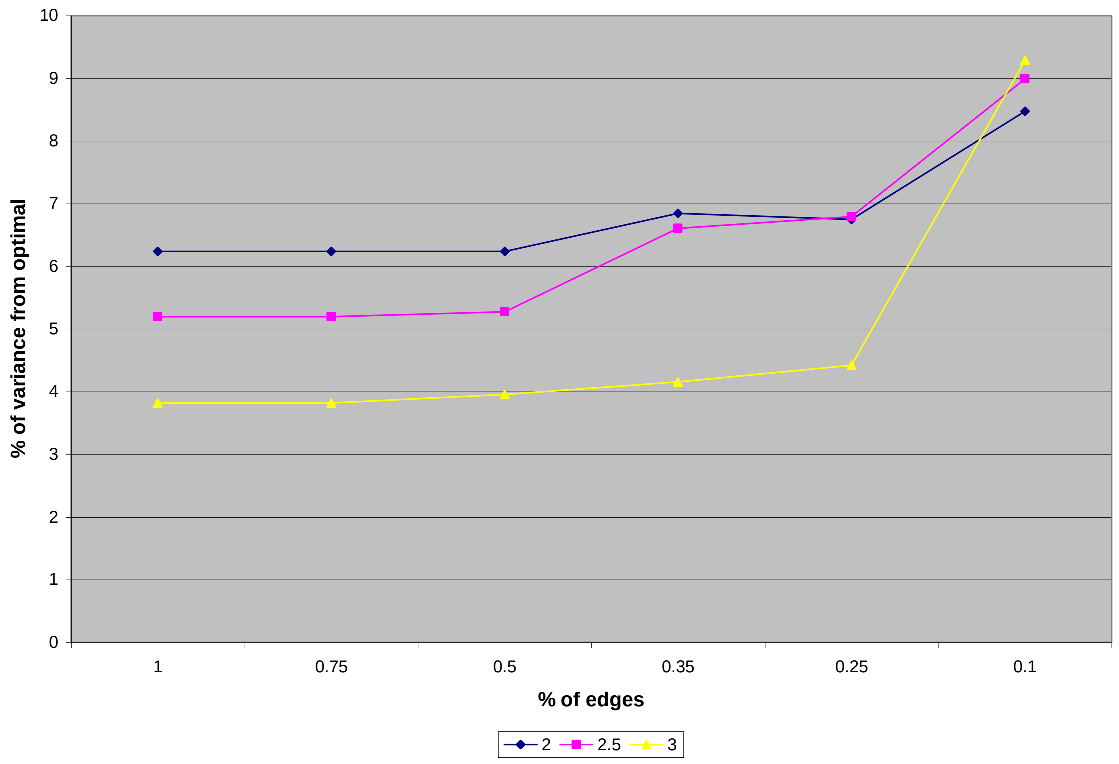


Figure 7.10: Standard Deviation TSP tour quality for MST Algorithm

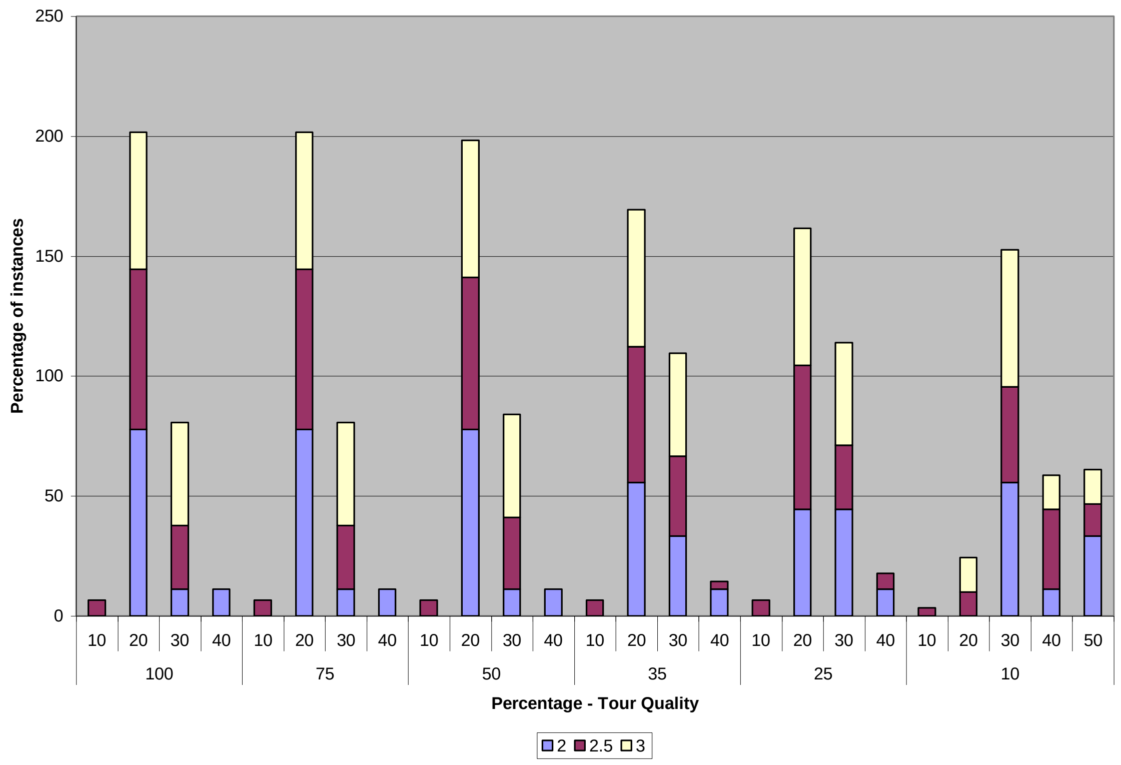


Figure 7.11: Percentage of instances within certain tour quality in each group using MST

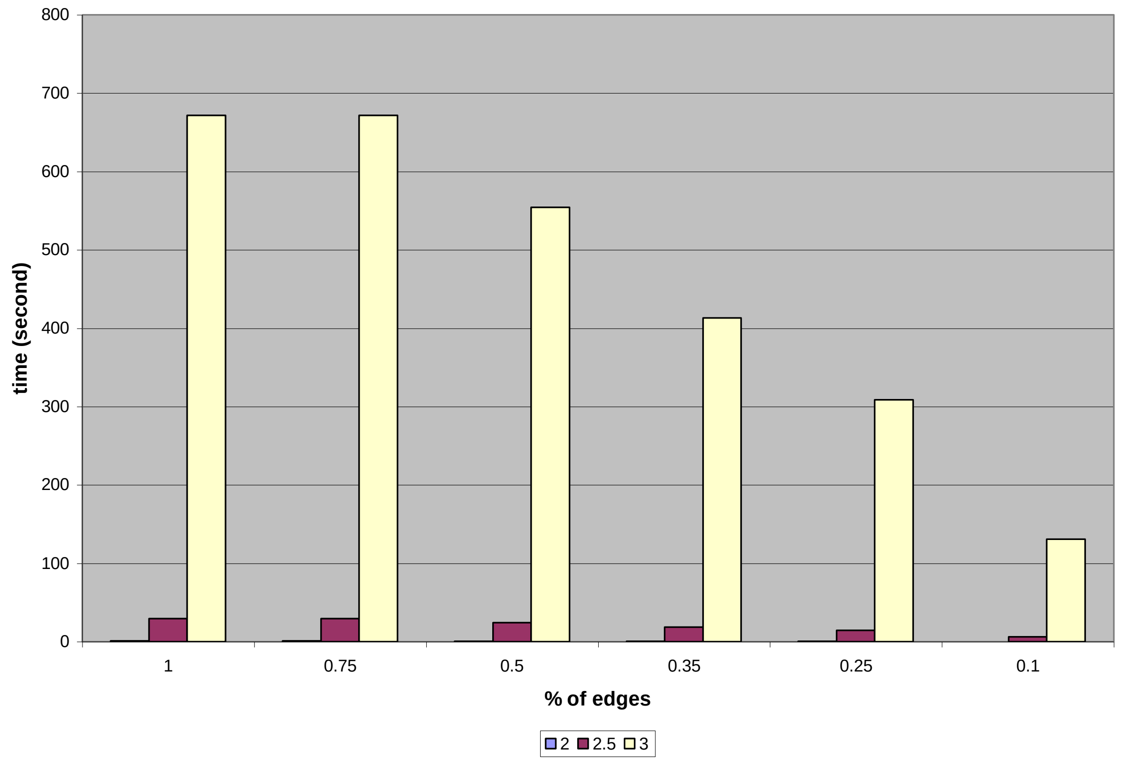


Figure 7.12: Average time for generating subset of edges using MST algorithm

7.1.4 Hilbert Value Clustering (HVC) Algorithm

The quality of the TSP tour generated using subset of edges produced by Hilbert Value Clustering has been shown in Figure 7.13. For the subset of edges with less than 75% of edges, the TSP tour distance grows fast. The less the number of edges, the larger the tour distance. For graphs with more than 10^3 cities, the TSP tour distance is 10 times longer than the optimal tour distance. This obviously is not a reasonable tour. This means that the edges generated by HVC algorithm were not useful in producing the TSP tour.

Figure 7.14 shows the standard deviation of the results compare to the average. To have a better view on the deviation of results in each group, in Figure 7.15 we show the percentage of instances that are within a certain tour quality for each group of test instances. The worst-case tour quality belongs to subset of edges with 10% of total edges for groups 3 and 3.5 and is about 75 times more than optimal tour.

Following that Figure 7.16 shows the time consumption of generating the subset of edges. Computing small subset of edges consumes much less time than computing the complete graph however the resulted tour is much poorer in terms of quality.

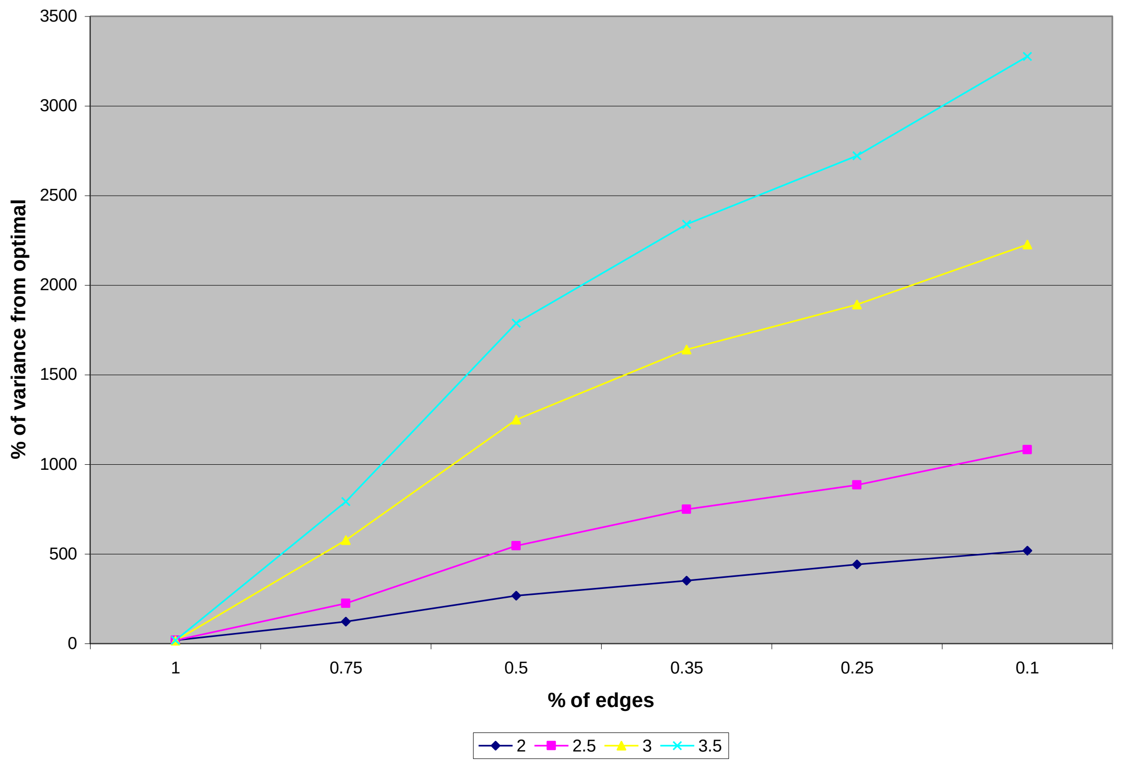


Figure 7.13: Average TSP tour quality for HVC Algorithm

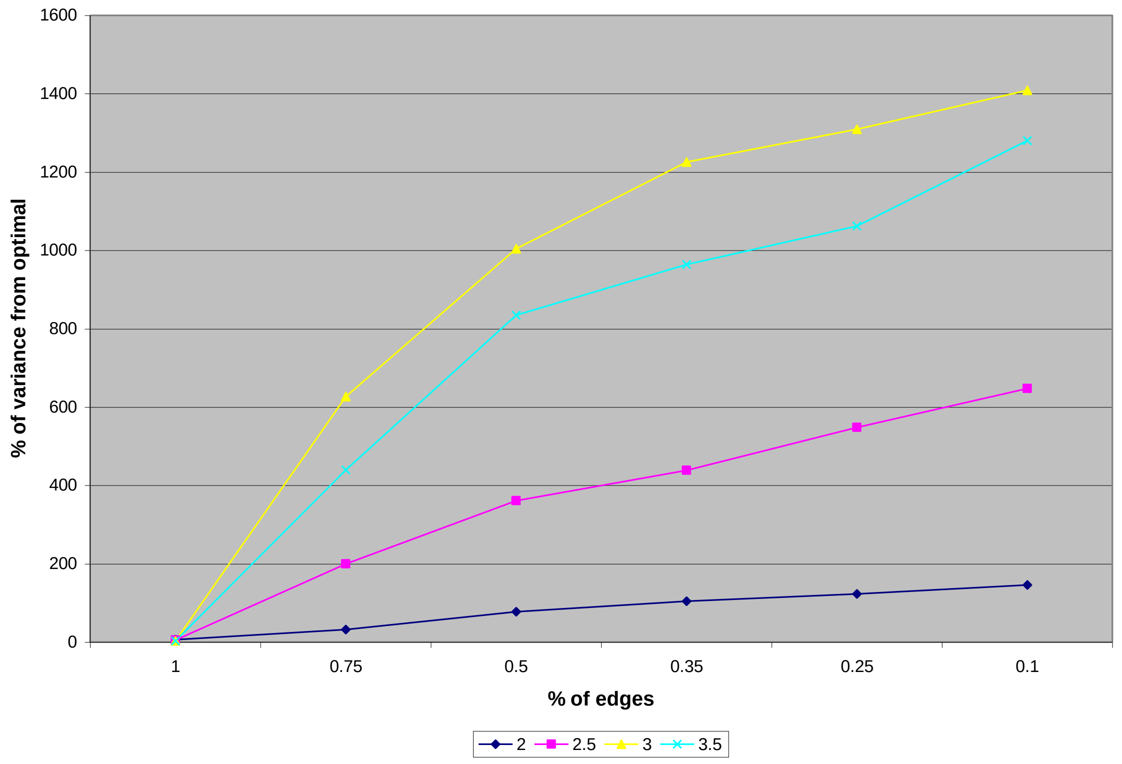


Figure 7.14: Standard Deviation TSP tour quality for HVC Algorithm

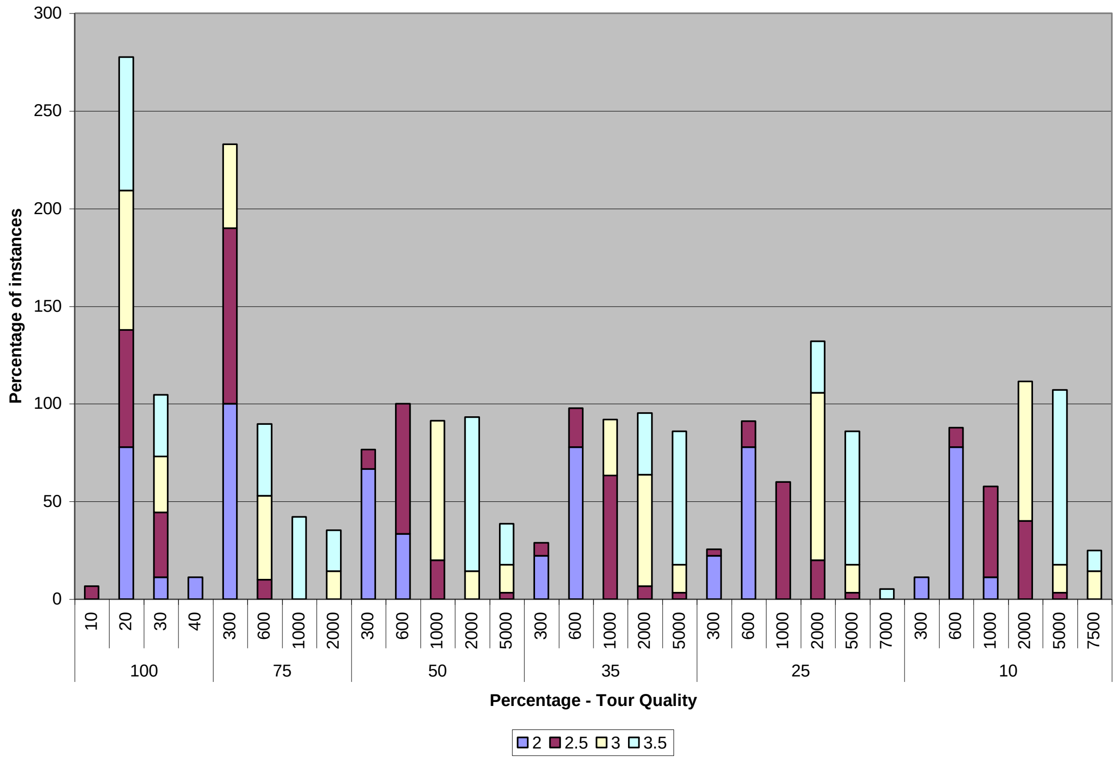


Figure 7.15: Percentage of instances within certain tour quality in each group using HVC

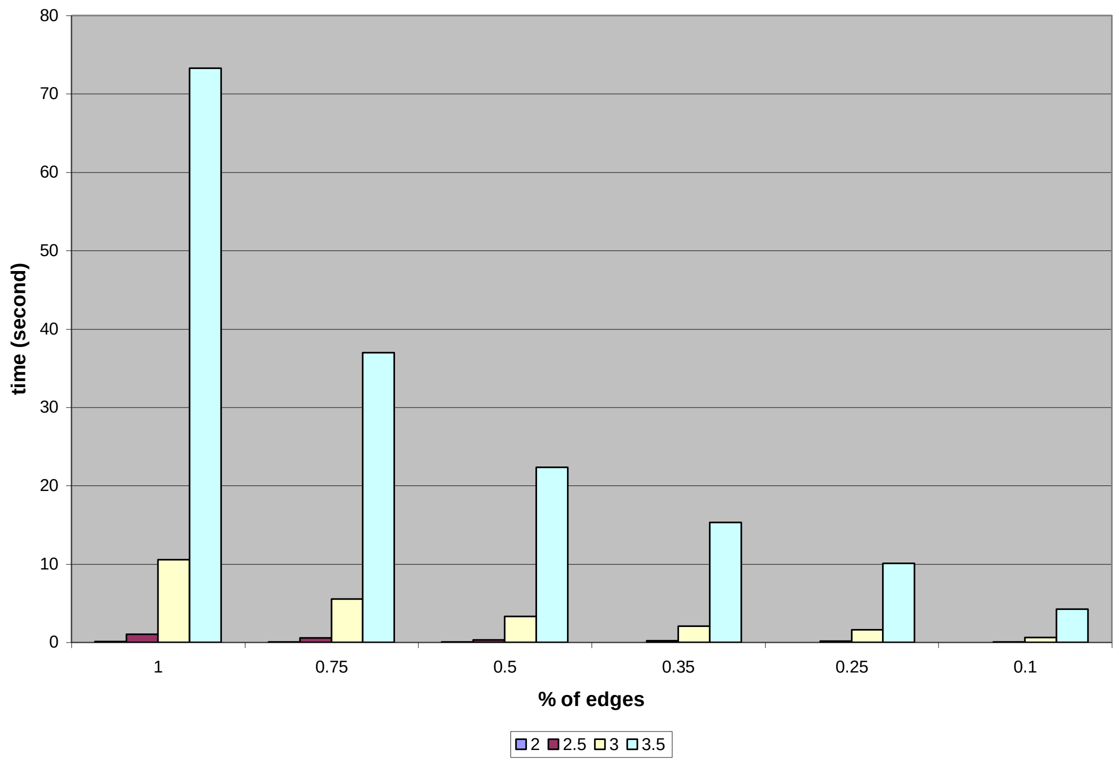


Figure 7.16: Average time for generating subset of edges using HVC algorithm

7.1.5 Shortest Path Tree Leaves Hilbert Value Clustering (LHVC) Algorithm

The experimental results on using LHVC algorithm, has been shown in figures below. Figure 7.17 illustrates the variance of the tour distances for each group of instances compare to the optimal tour. The quality of generated tours is about the same as the quality of the generated tours using HVC algorithm. This means that using SPT leaves to generate clusters is not much helpful in generating good quality clusters. The standard deviation of the results compare to the average is illustrated in Figure 7.18. It shows that in groups 3 and 3.5 tour quality result can be at most 1400% more or less than the average result. Figure 7.19 shows the percentage of instances that are within a certain tour quality for each group of test instances. The time to generate the subset of edges is shown in Figure 7.20. Since the clustering part was not time-consuming, the trend is the same as in HVC algorithm.

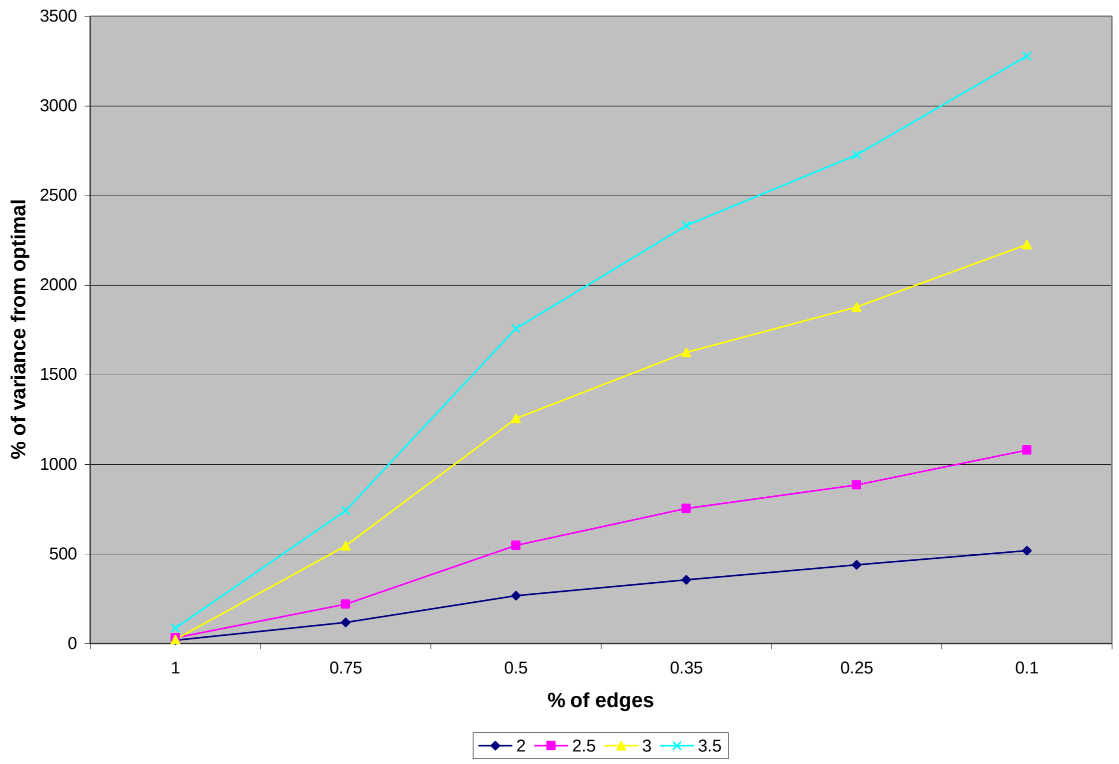


Figure 7.17: Average TSP tour quality for LHVC Algorithm

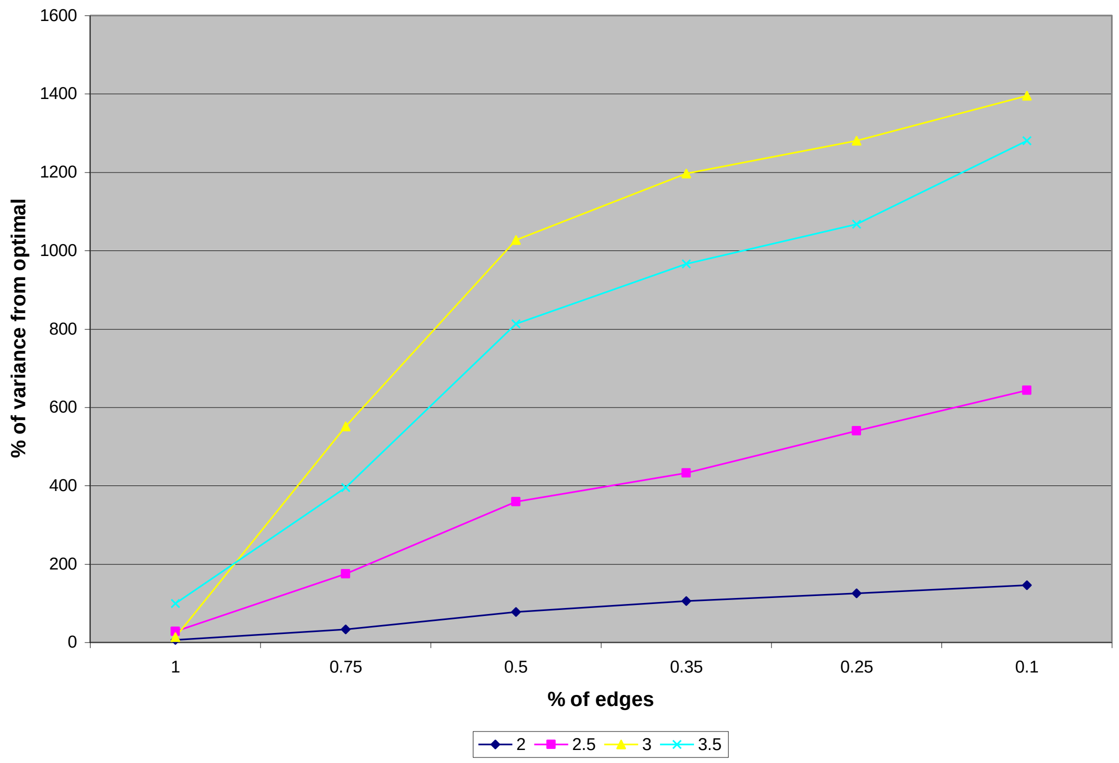


Figure 7.18: Standard Deviation TSP tour quality for LHVC Algorithm

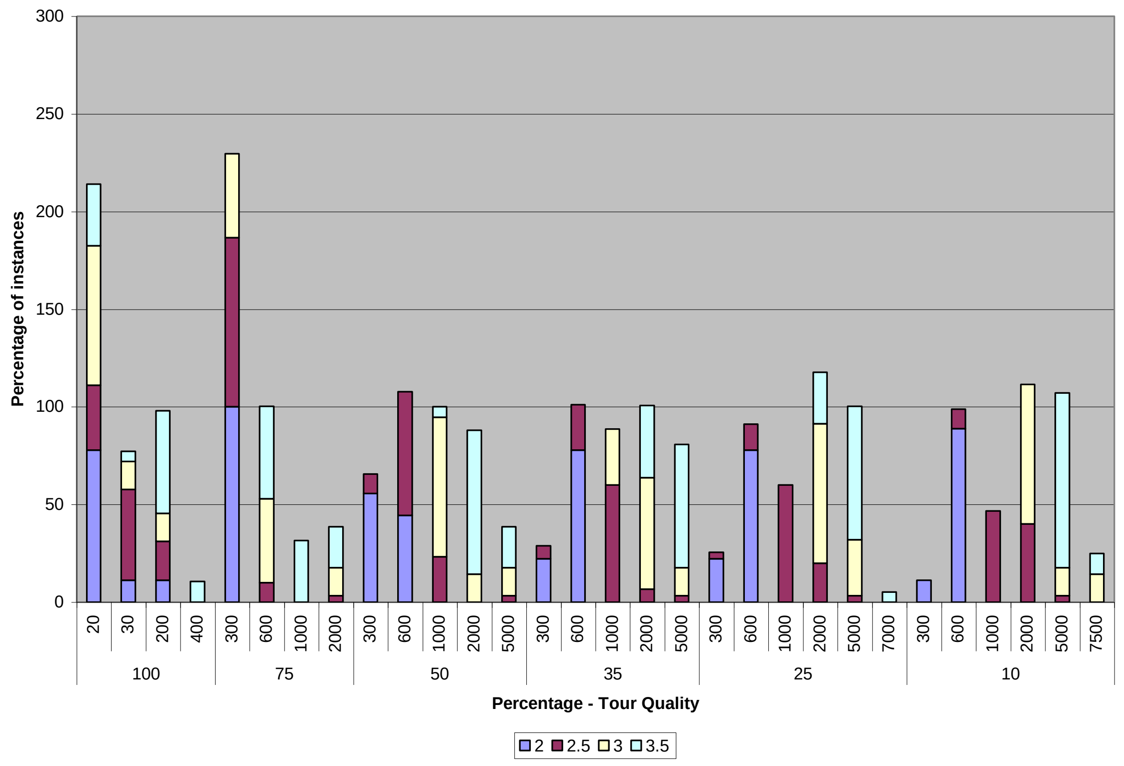


Figure 7.19: Percentage of instances within certain tour quality in each group using LHVC

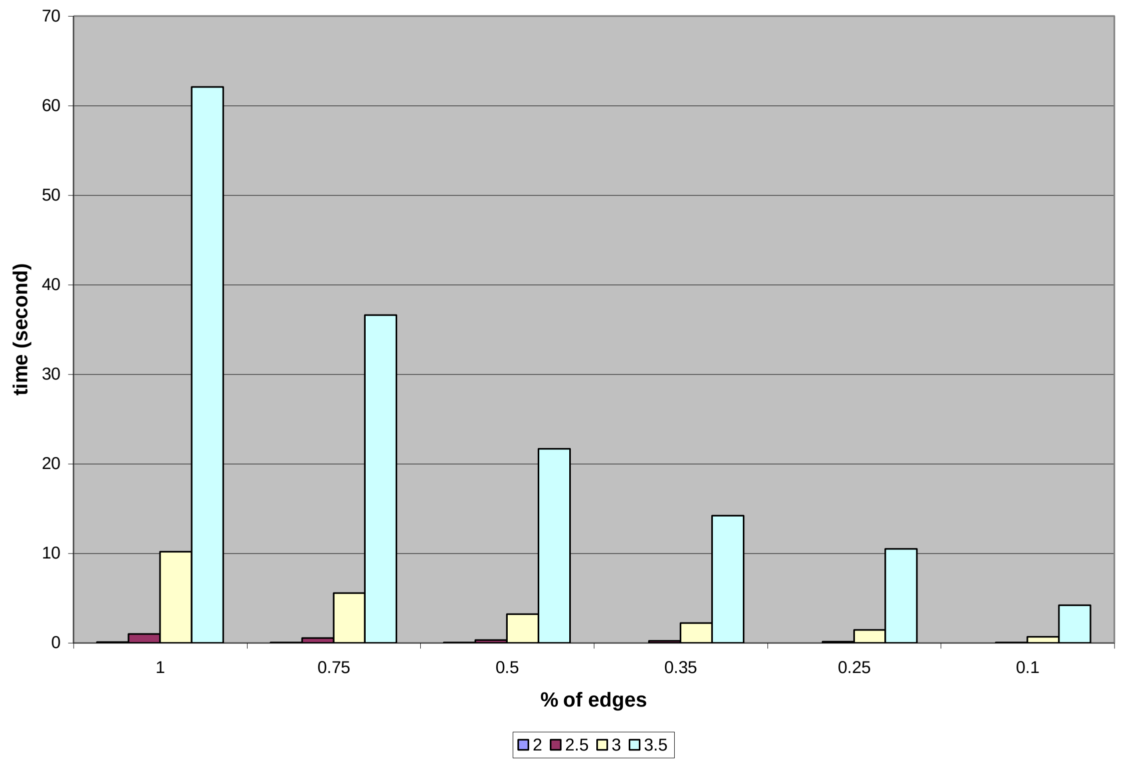


Figure 7.20: Average time for generating subset of edges using LHVC algorithm

7.1.6 Random SPT (RSPT) Algorithm

We generate random subset of edges having 100%, 75%, 50%, 35%, 25% and 10% of the total edges in the complete graph, by generating random SPTs. The quality of TSP tours produced using this algorithm (Figure 7.21) is almost the same as quality of tours produced using HVC and LHVC. The standard deviation results in Figure 7.22 shows that the distance of the generated tours can be about 1400% longer or shorter than the average. For the smaller graphs this number is about 600%. It means that using RSPT algorithm for small graphs (groups 2 and 2.5); the resulting TSP tours have less variance from the average result. Figure 7.23 shows the percentage of instances that are within a certain tour quality for each group of test instances. The average time for generating subset of edges using RSPT algorithm has been shown in Figure 7.24. Similar to the tour quality result, the time is also the same as using HVC and LHVC algorithm.

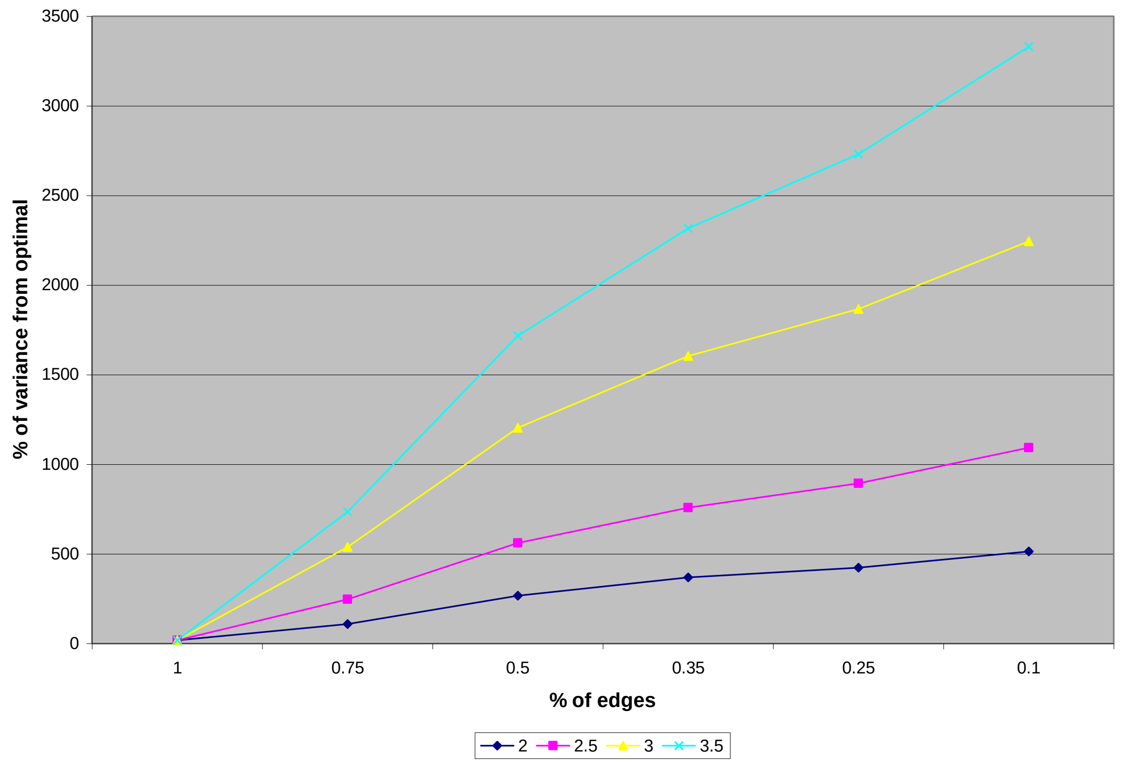


Figure 7.21: Average TSP tour quality for RSPT Algorithm

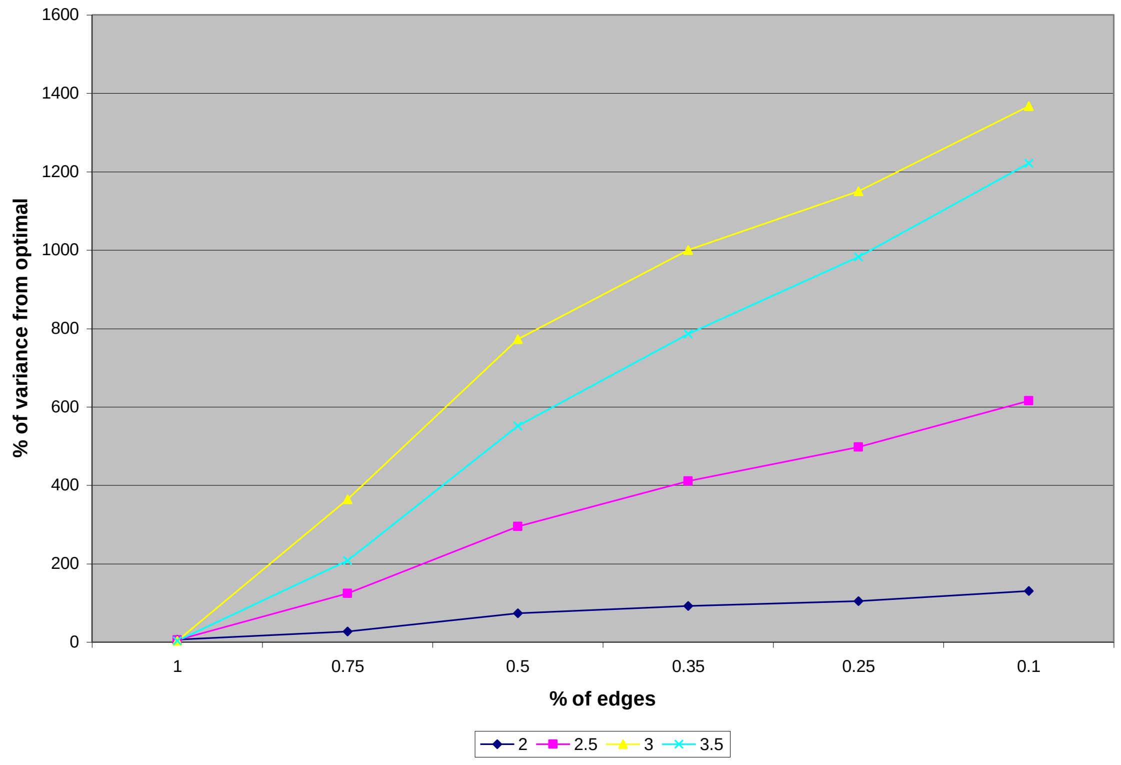


Figure 7.22: Standard Deviation TSP tour quality for RSPT Algorithm

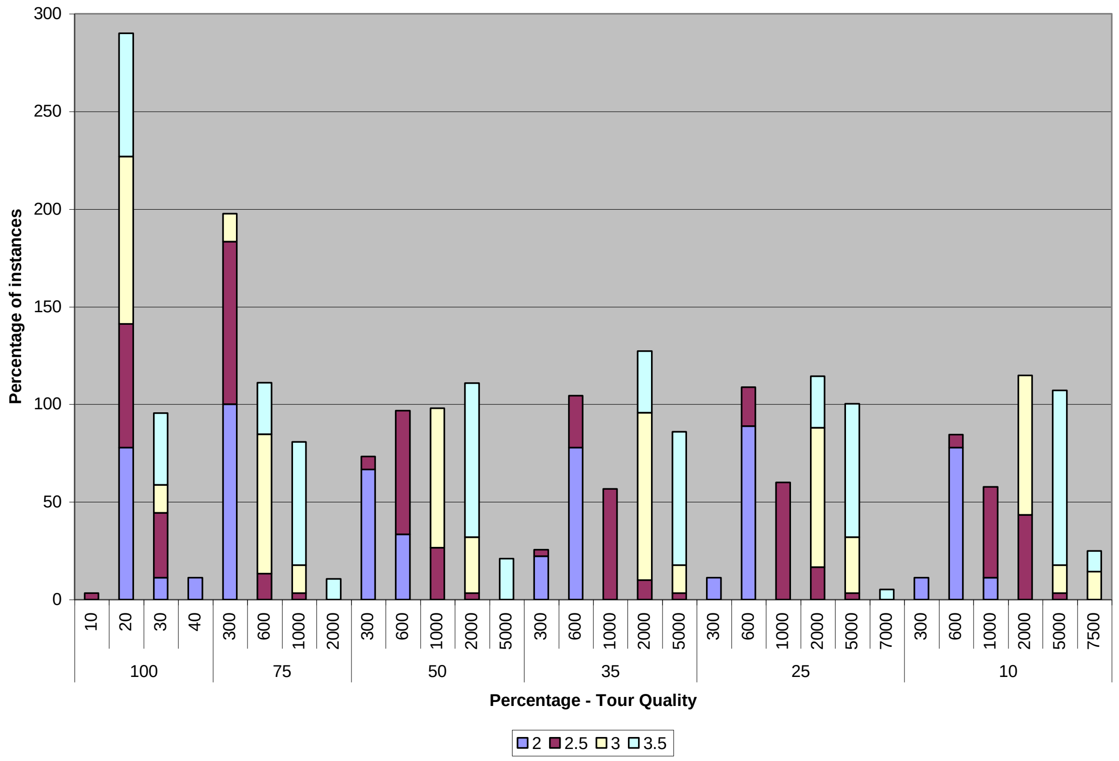


Figure 7.23: Percentage of instances within certain tour quality in each group using RSPT

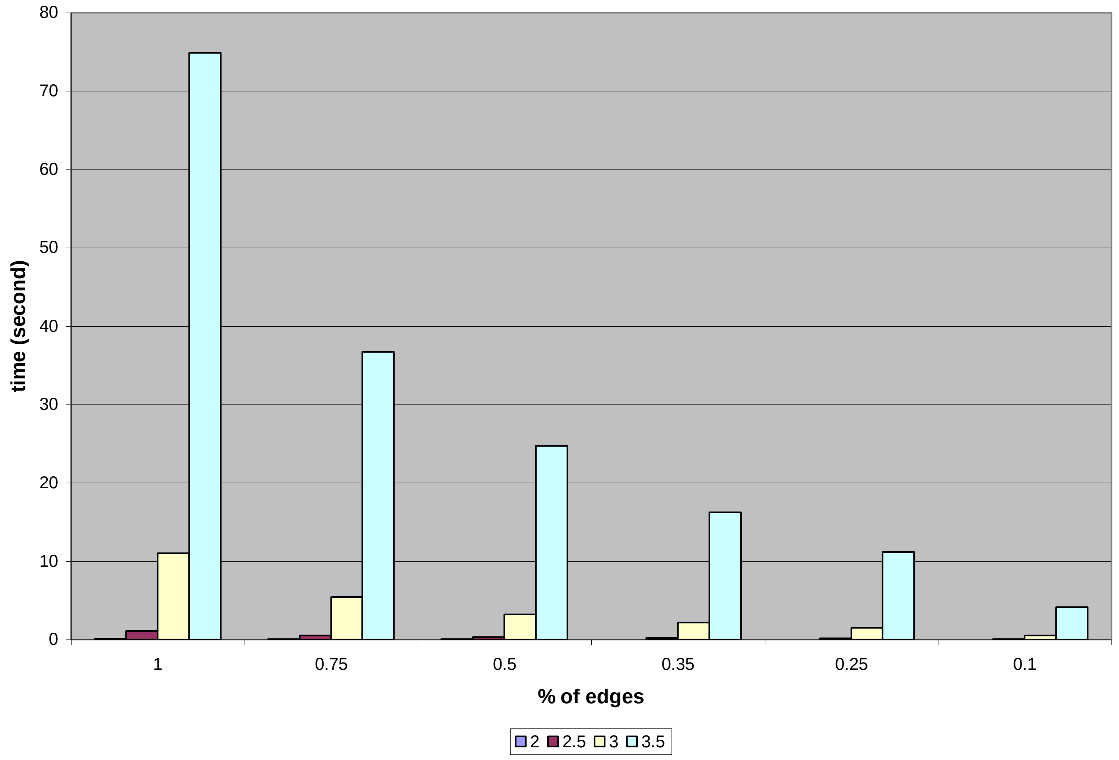


Figure 7.24: Average time for generating subset of edges using RSPT algorithm

7.2 Performance of Modified 2-opt Improvement Algorithms

In this subsection, we present the experimental results of the three introduced improvement algorithms. For all the improvement algorithms, the initial tour is the tour that has been constructed using the Greedy algorithm introduced in Section 4.1. However if the output of the Greedy algorithm is not a TSP tour, we have used Initial-tour-construction algorithm introduced in Section 4.2. Moreover for each improvement algorithm, we have used the same subset of edges that we have used in constructing the initial tour. This subset of edges can be produced using one of the six introduced non-complete graph generating algorithms.

Figure 7.25 to Figure 7.30 show the results of using six non-complete graph generating algorithms to solve the TSP. In each figure we present the quality of the generated TSP tour for six different subsets of edges (100%, 75%, 50%, 35%, 25% and 10% of the total edges) and four algorithms. Because of the lack of space in the charts we abbreviated the name of the used algorithm. The fist algorithm is called “initial” which is the TSP tour generated using the Greedy algorithm and if the output of the Greedy algorithm is not a TSP tour, we construct the tour using Initial-tour-construction algorithm. The second algorithm is called “original” which represent the Original 2-Opt algorithm introduced in Section 5.1. The third algorithm here is called “modified” which stands for the Modified 2-Opt algorithm introduced in Section 5.2 The fourth and last algorithm is called “savinglist” which is the Saving List 2-Opt algorithm that we introduced in Section 5.3.

In Figure 7.25 to Figure 7.30, the y-axis shows the variance of the quality of the TSP tours compared to the optimal TSP tour distance. In this section x% improvement of an initial tour is defined as below:

(variance of the improved tour from the optimal tour) – (variance of the initial tour from the optimal tour)

Figure 7.25 illustrates the experimental results of exploiting XNN edge-generating algorithm in order to generate a non-complete graph and using this graph; we make TSP tours by initial tour construction and improve it by three improvement algorithms. The initial TSP tour has been improved about 10% (the difference between the initial TSP tour and its improved tour is about 10%). Comparing the three improvement algorithms, we conclude that for subset of edges having 100%, 75%, 50% and 35% of the total edges, the Saving List 2-Opt improvement algorithm results in about 5% better tour quality than the other two improvement algorithms (the tour quality is based on comparison of the tour and the optimal tour). However, as shown in Figure 7.25, when the number of edges in the non-complete graph is less than 25% of the total edges, the Original 2-Opt algorithm outperforms the other two improvement algorithms with about 5% better tour quality (the difference of the tour quality between the Original 2-Opt and the other two algorithms' tour quality is less than 5%). This suggests that when the subset of edges contains the smallest edges, this subset results in more improvement in Original 2-Opt algorithm than Saving List 2-Opt improvement algorithm. Another issue that we were eager to know was the impact of the graph size on the improvement algorithms. As it has been shown in Figure 7.25, the above conclusion on using XNN edge-generating algorithm in order to generate a non-complete graph is true for all the graph group sizes.

The above result is also true for TSXP edge-generating algorithm shown in Figure 7.26. Except that for subset of edges having 35% of the total edges, the best improvement algorithm is the Original 2-Opt algorithm. The other differentiation from the XNN algorithms is that for subset of edges having less than 10% of edges the TSP tour distance is about 10% poorer in quality than the same size subset generated by XNN or MST algorithm. This implies that although the subset of edges contains the shortest edges, they are not good enough to generate the TSP tour compare to the same size subset of edges generated by XNN or MST. In terms of graph sizes we can say that different sizes of graphs result into the same described trend.

Experimental result for MST graph generating algorithm is shown in Figure 7.27. Similar to TSXP, in the case that the size of the subset of edges is less than 35% of the total edges, the Original 2-Opt algorithm result in about 5% to 10% less TSP tour distances compare to the other two improvement algorithms. For the subset of edges having more than 35% of the total edges, the Saving List 2-Opt improvement algorithm outperforms the other two improvement algorithms by less than 5%. The overall result in terms of tour quality is much like the result of using XNN algorithm to generate the non-complete graph.

Figure 7.28, Figure 7.29 and Figure 7.30 respectively show the experimental results of using HVC, LHVC and RSPT algorithms in order to generate the non-complete graph. The overall results for all these three algorithms are similar to each other. This means that clustering using HVC and LHVC do not benefit us in generating the non-complete graph. Comparing the three improvement algorithms, we can say that the Original 2-Opt algorithm outperforms the other two for the subset of edges with less than 75% of the total edges. However the tour quality is poor for both initial tour and improvement tour.

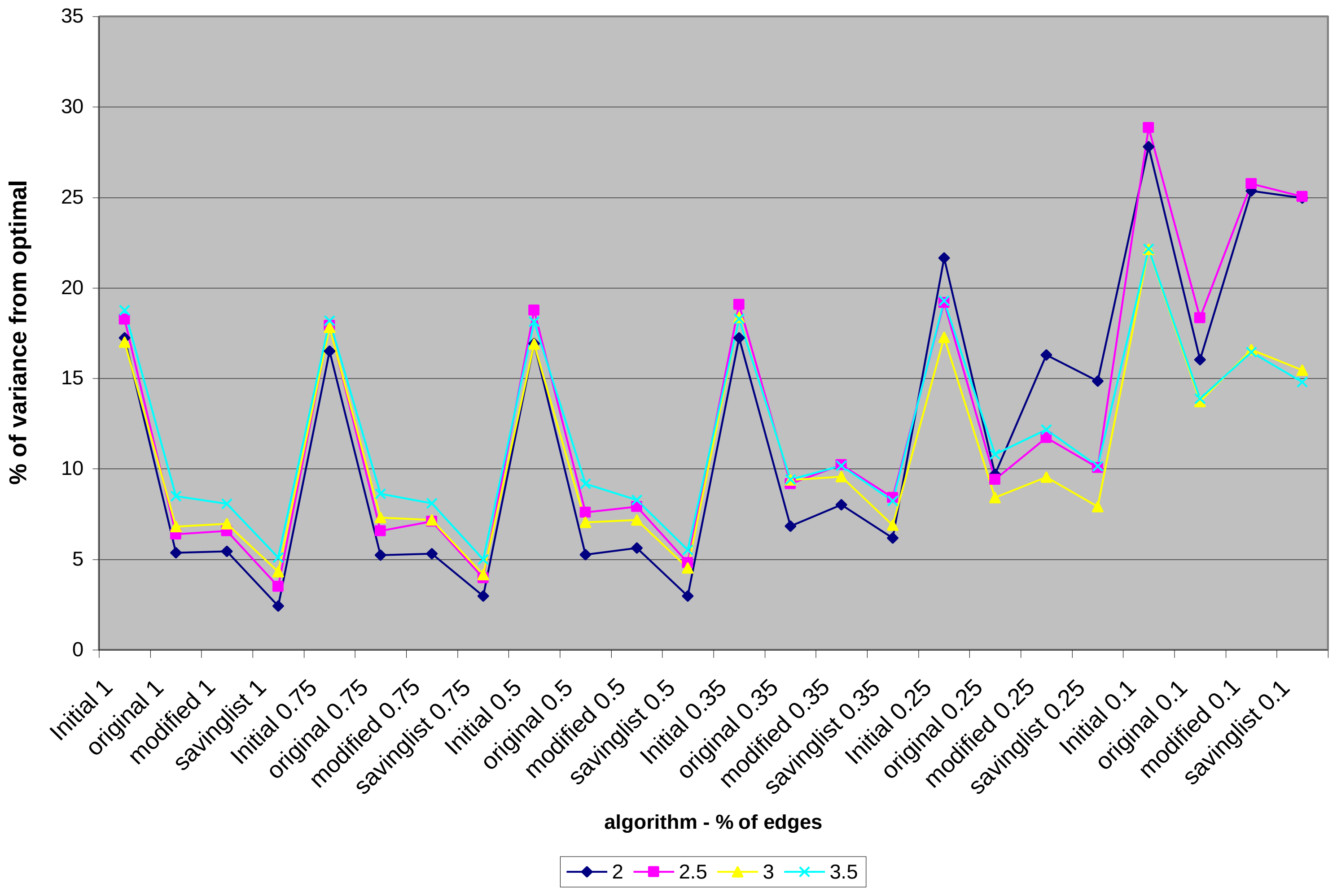


Figure 7.25: improved TSP tour quality generated using XNN algorithm

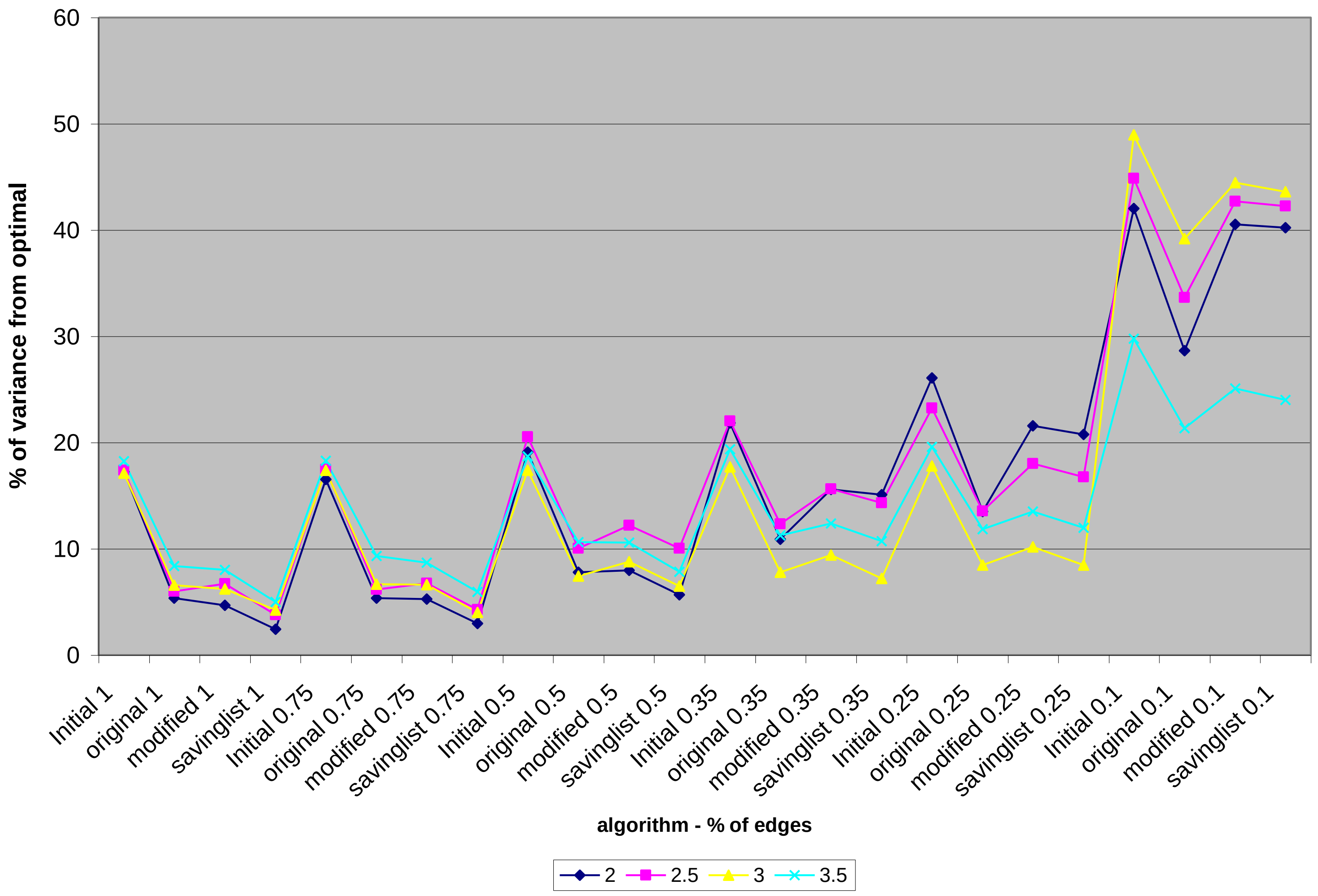


Figure 7.26: improved TSP tour quality generated using TSXP algorithm

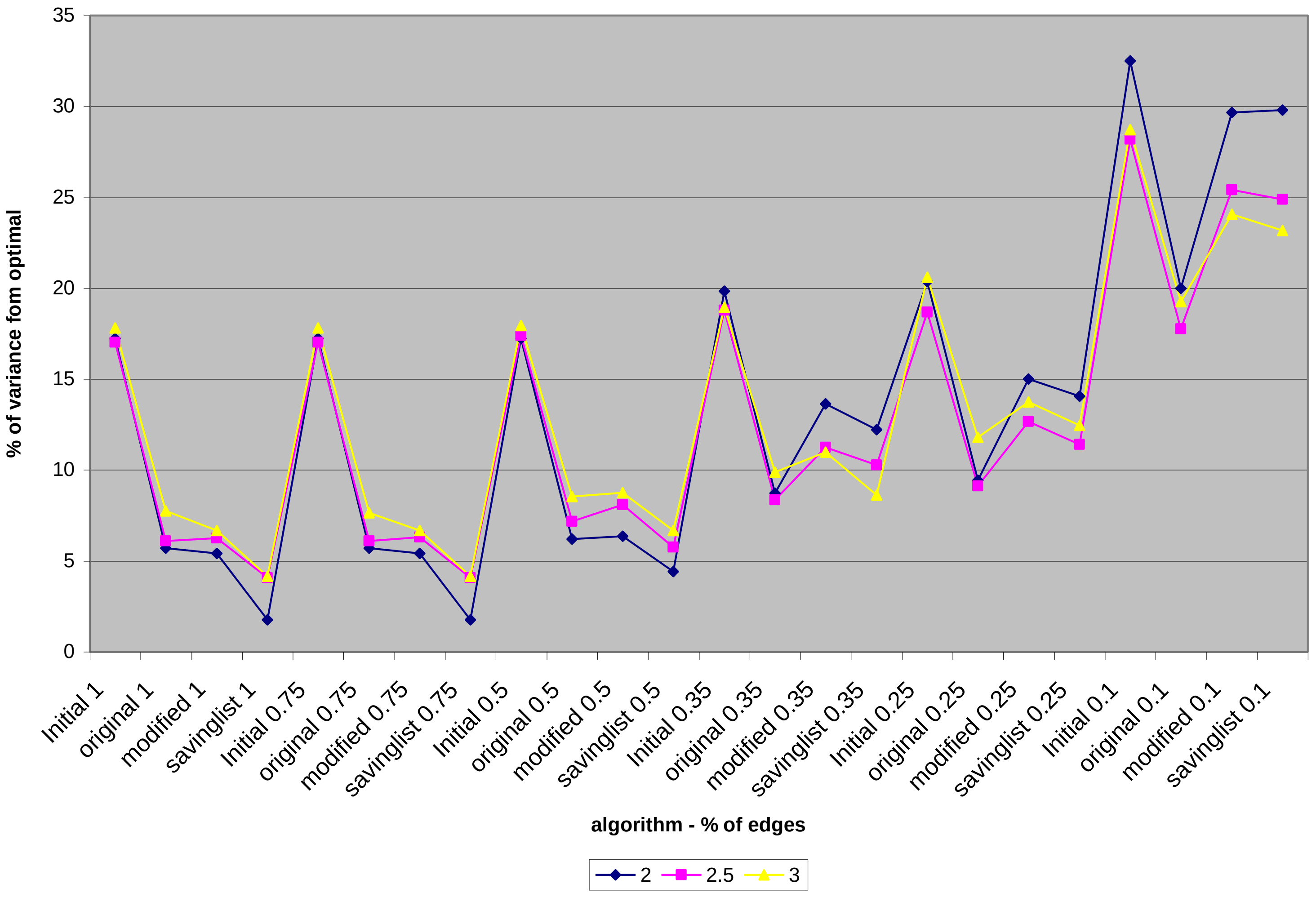


Figure 7.27: improved TSP tour quality generated using MST algorithm

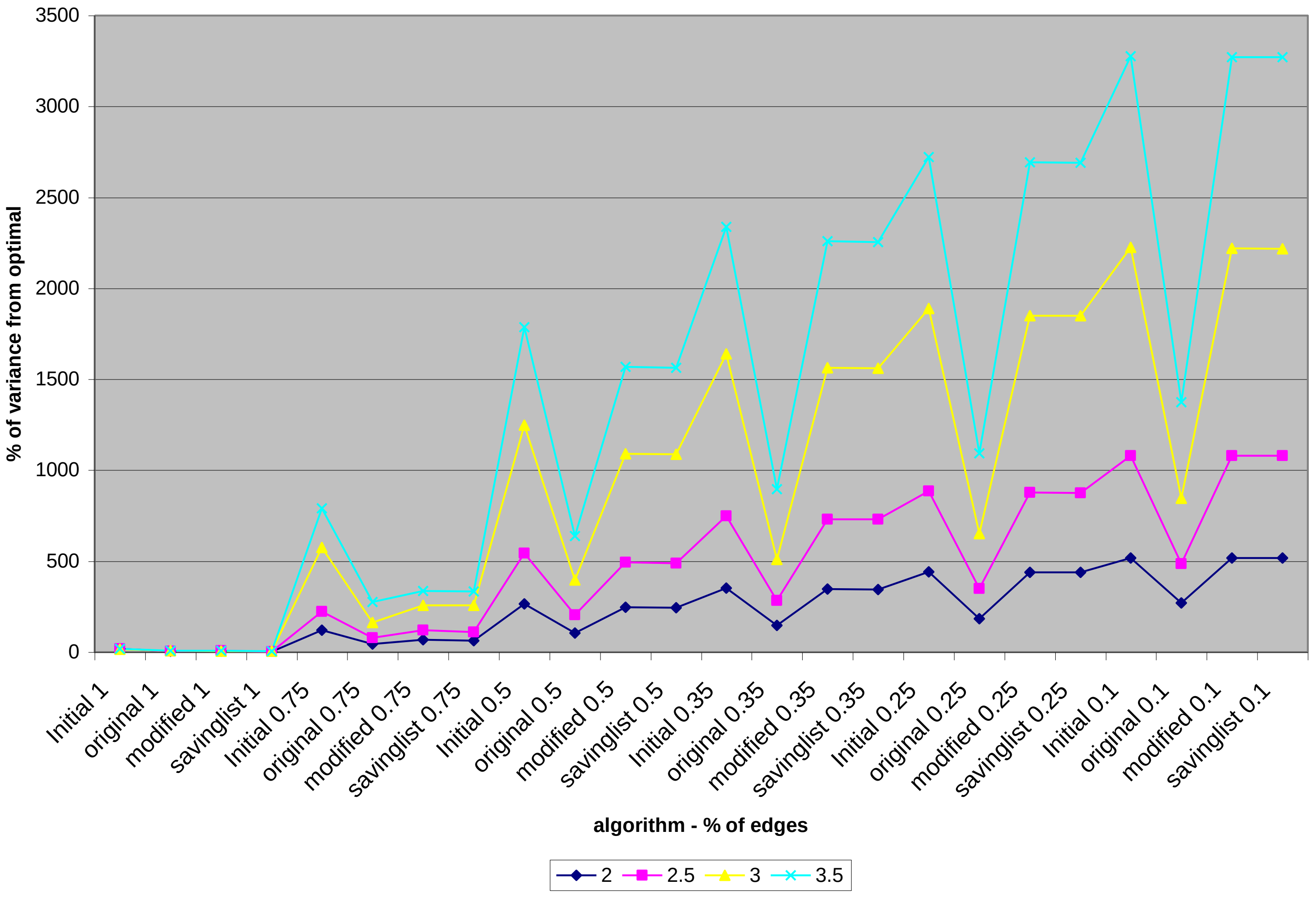


Figure 7.28: improved TSP tour quality generated using HVC algorithm

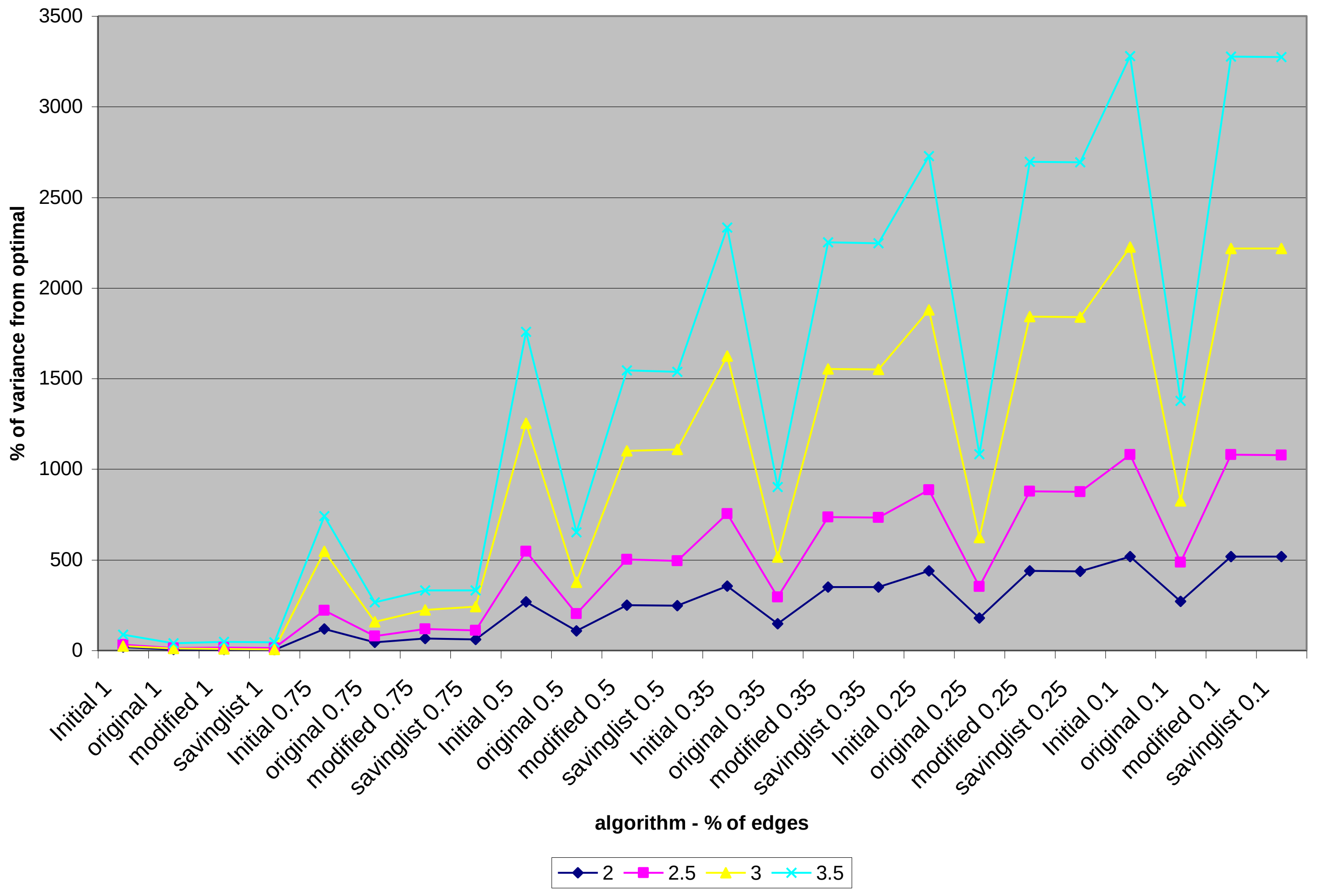


Figure 7.29: improved TSP tour quality generated using LHVC algorithm

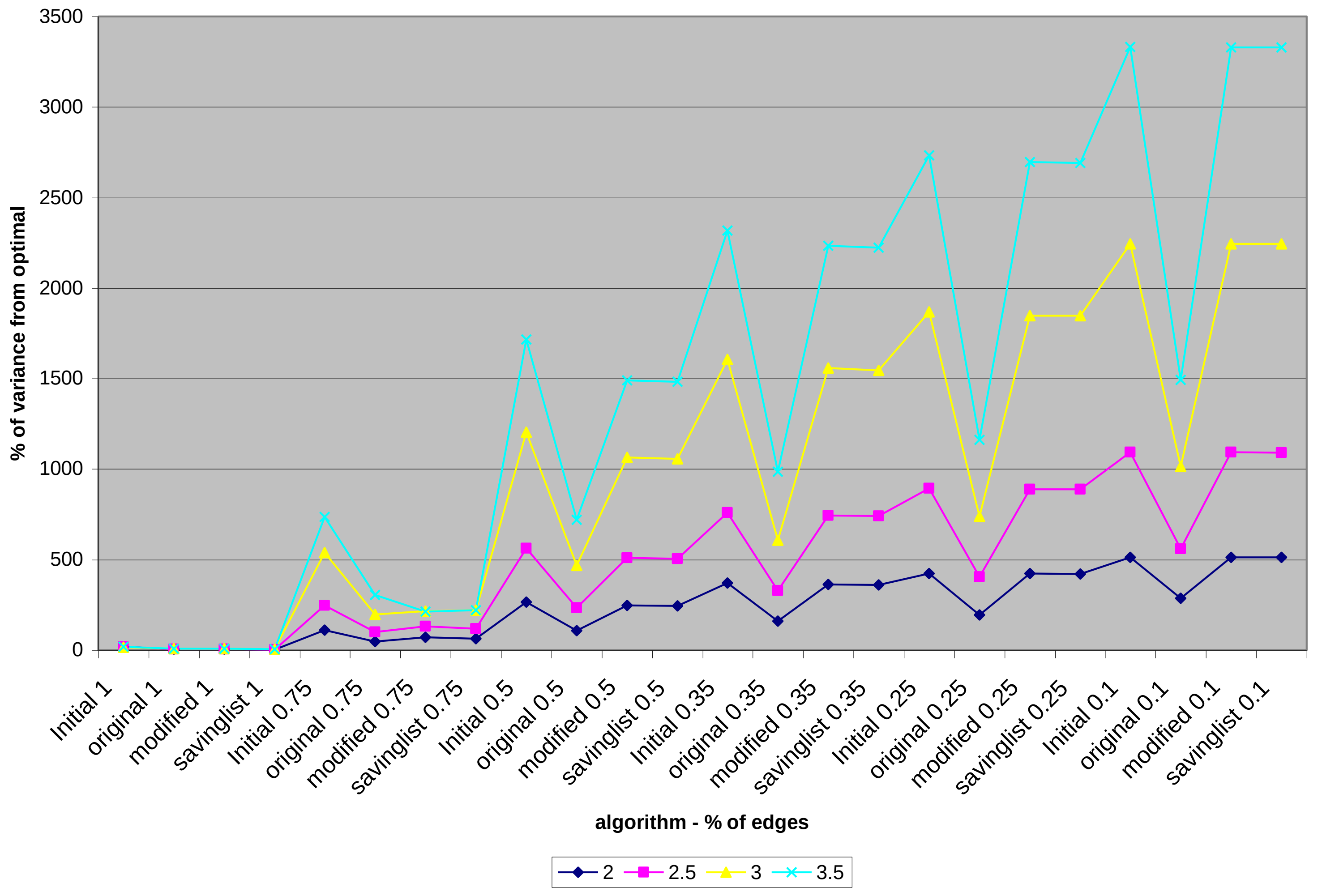


Figure 7.30: improved TSP tour quality generated using RSPT algorithm

Figure 7.31 to Figure 7.36 illustrate the time consumption for three introduced improvement algorithms using each non-complete graph generating algorithm. The approaches based on XNN and TSXP algorithms require almost the same amount of time (Figure 7.31 and Figure 7.32). Using MST algorithm to generate the subset of edges is much more time consuming. That is why we were not able to consider the group of graphs with 10^3-$10^{3.5}$ cities in our experimental results. As shown in Figure 7.33 the behavior of the diagram is different than in Figure 7.31 and Figure 7.32 in a way that for subset of edges having 100% of the edges, the time consumption is much greater from 75% subset of the edges; while the tour quality is the same for both 100% and 75% of total edges.

Figure 7.34, Figure 7.35 and Figure 7.36 respectively show the time consumption of using HVC, SPTLHVC and RSPT algorithms in order to generate the non-complete graph. The overall time consumption for all these three algorithms is similar to each other. This might be because of the fact that the subsets of edges generated by these algorithms have one commonality which is containing random SPT edges.

The two properties that are the same for all the six non-complete graph generating algorithms are:

- The Saving List 2-Opt improvement algorithm is the most time consuming algorithm out of three 2-Opt algorithms.
- The Saving List 2-Opt improvement algorithm time consumption has much more increase than the other 2-Opt algorithm when the size of the graph grows.

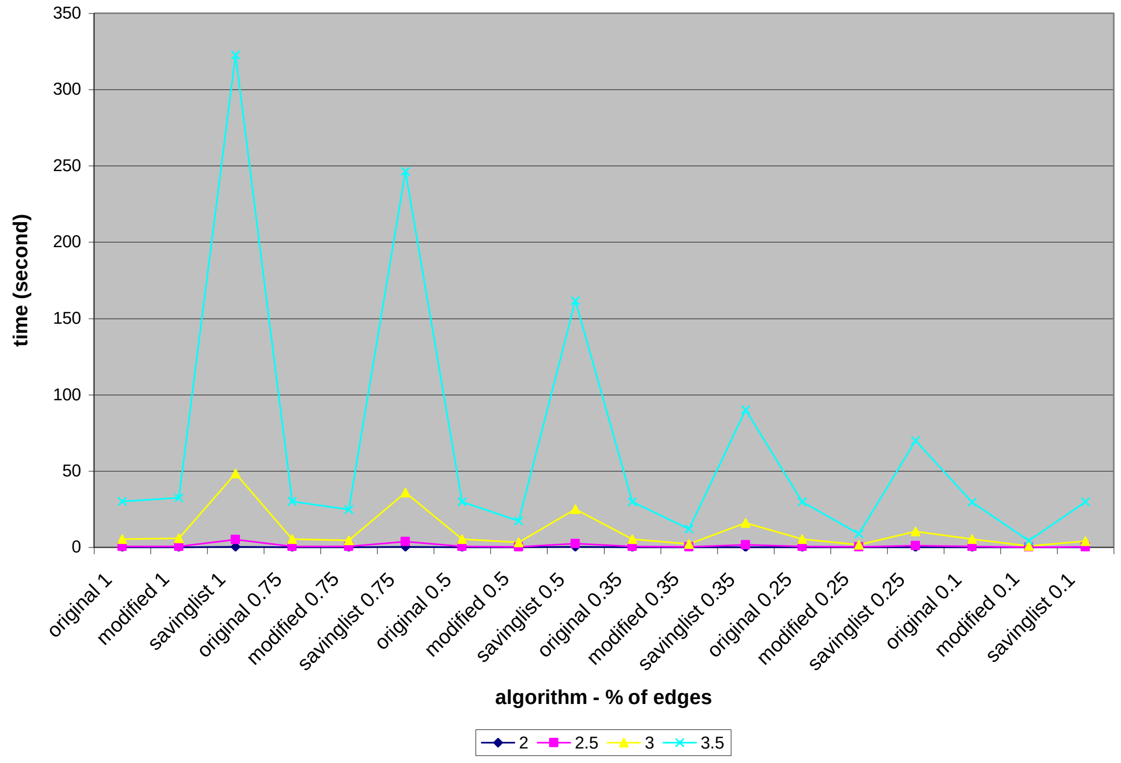


Figure 7.31: improved TSP tour time generated using XNN algorithm

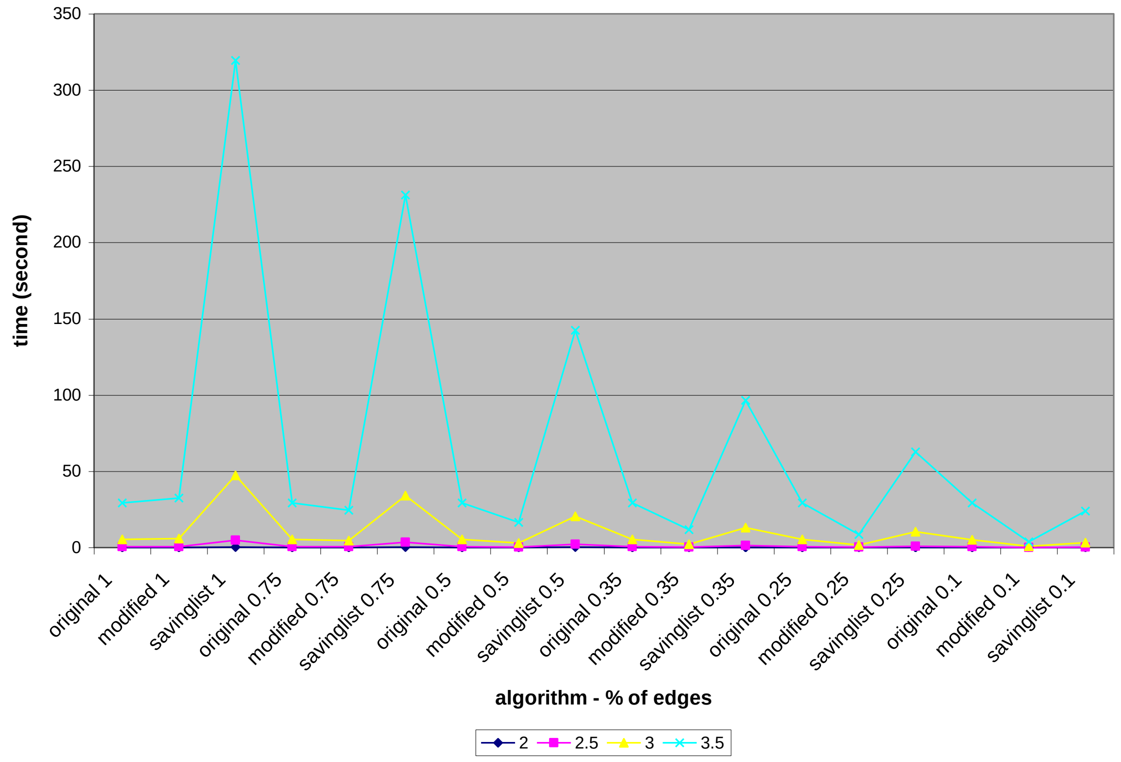


Figure 7.32: improved TSP tour time generated using TSXP algorithm

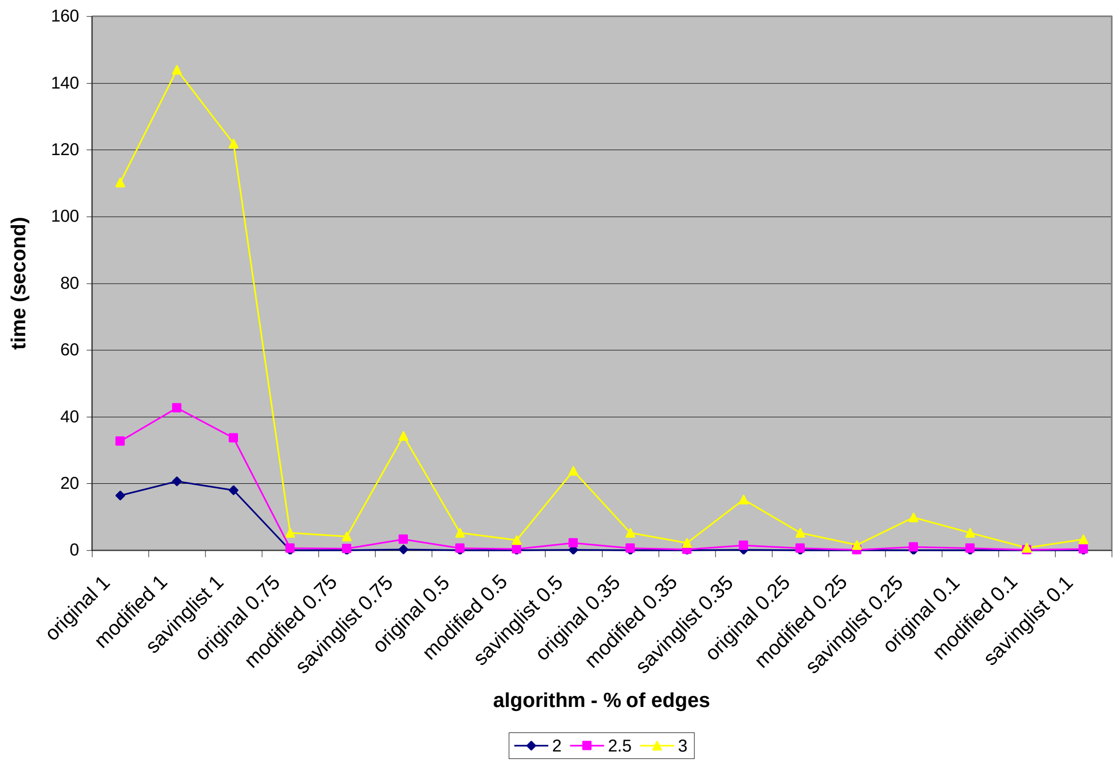


Figure 7.33: improved TSP tour time generated using MST algorithm

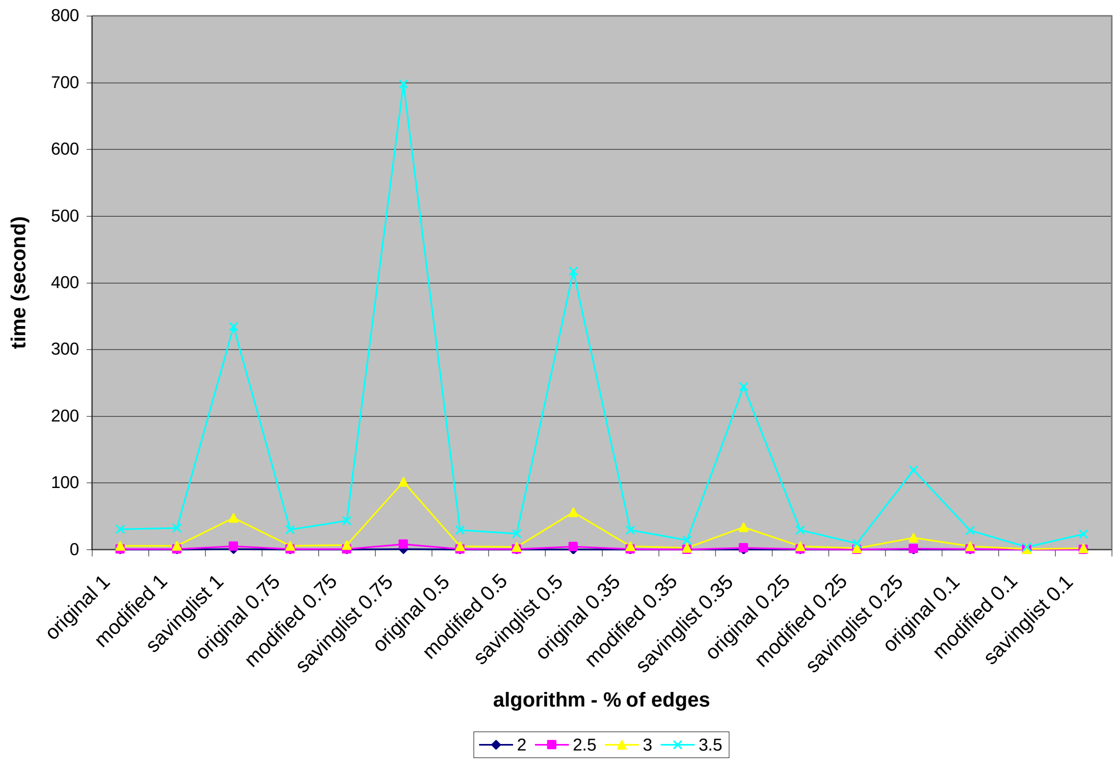


Figure 7.34: improved TSP tour time generated using HVC algorithm

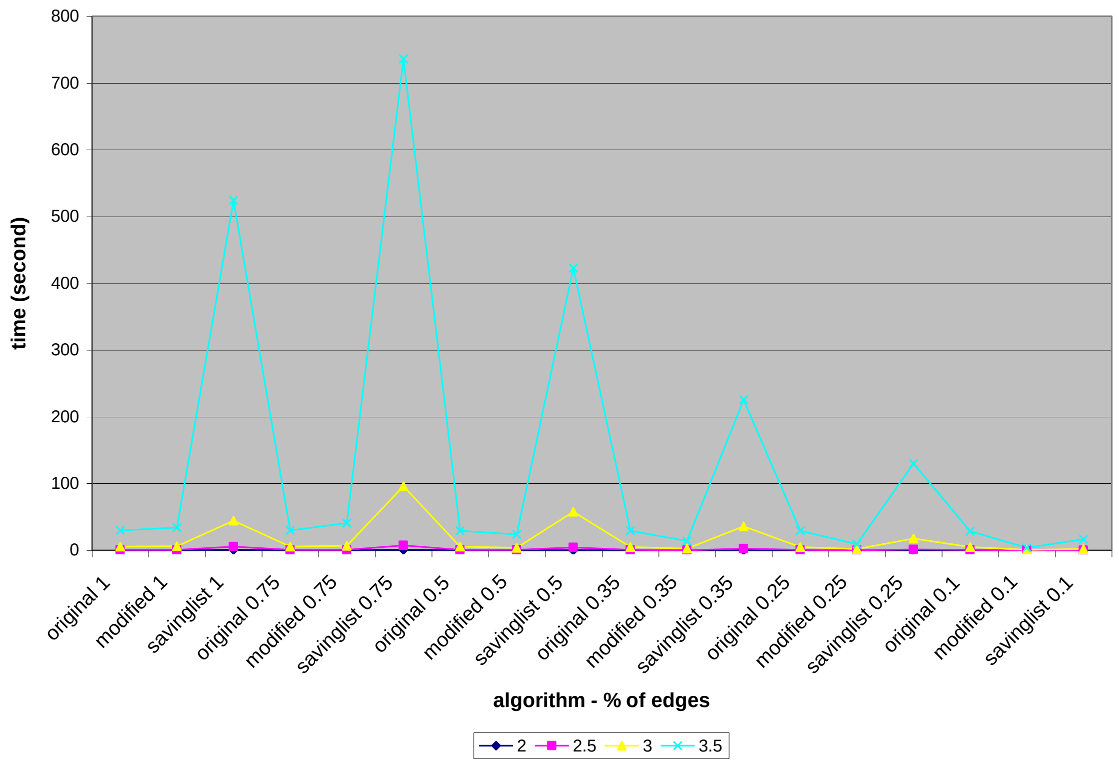


Figure 7.35: improved TSP tour time generated using LHVC algorithm

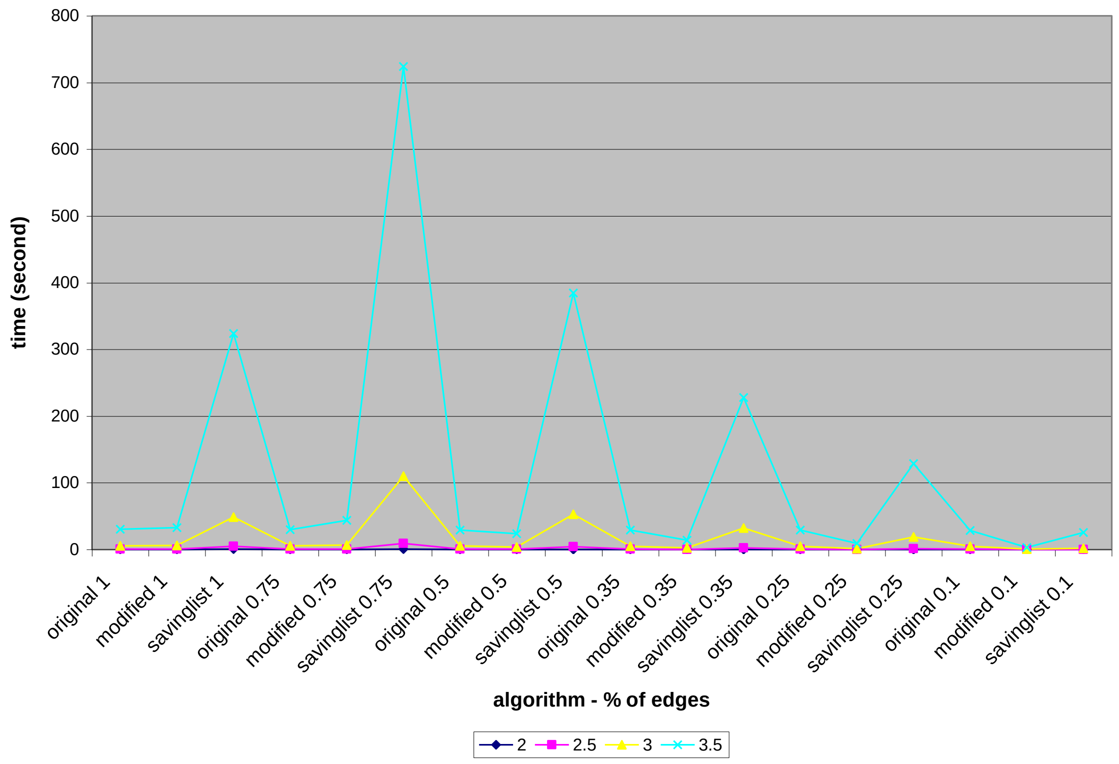


Figure 7.36: improved TSP tour time generated using RSPT algorithm

7.3 Summery of the Experimental Results

In our experiments, for all the non-complete graph generating algorithms, the time to run the Greedy and Initial Tour Construction algorithms was negligible compare to the time to generate the non-complete graph or improving the tour. That is why we have not included the time to run these two algorithms in the experimental results.

Among the six introduced graph generating algorithms, XNN, TSXP and MST have the best tour quality with at most 60% variance from the optimal tour. HVC, LHVC and RSPT have almost the same and also much poorer tour quality compare to the other three algorithms. Based on this reason, in this subsection we mostly focus on comparing the results of XNN, TSP and MST algorithms and we overlook the results of HVC, LHVC and RSPT.

Using XNN algorithm to generate a non-complete graph, the resulting TSP tours are between 15%-30% variance from the optimal TSP tour. In terms of time, XNN consumes at most 70 seconds for graphs in group 3.5 and this time reduces a lot by decreasing the size of the subset; while the tour quality changes only about 10%. Computing 25% of the total edges using this algorithm, the time is about 1/3 less than computing the complete graph while the cost is less than 10% decrease in tour quality. The standard deviation results demonstrate that the tour quality could be mostly about ±10% different from the average result. Exploring the outliers for each group, we find out that the worst case tour quality belongs to subset of size 10% in group 2.5 for only about 5% of instances. The worst case tour quality is less than 70% variance from the optimal tour quality and majority of instances are between 10%-30% variance from the optimal tour.

Using TSXP algorithm, the tour quality is a bit poorer compare to XNN algorithm however it is still between 10%-50%. The drawback of using this algorithm is that we have to generate the complete graph first and then select the top x%. So the time consumption is the same for all 4 groups of graphs (2, 2.5, 3 and 3.5).

Using MST algorithm to generate a subset of edges, the tour quality is similar to using XNN algorithm. The produced TSP tours are between 15%-35% variance from the optimal TSP tour. However with regard to time, MST is more time-consuming than XNN.

We have compared our result with the Greedy algorithm in the state-of-the-art. It shows that the TSP tour quality resulted from the Greedy algorithm is in between 14-19.9 percent over the optimal result. Our experimental results show that the TSP tours that have been generated using Greedy algorithm are between 15-20 percent over the optimal result. It means that our Greedy algorithm implementation is comparable to the state-of-the-art. The more important result in our research is that we can find a TSP tour with almost the same tour quality using only 30% of the total edges. This way one does not need to compute the complete graph and can save a lot of time.

Comparing the three introduced 2-Opt improvement algorithms, the interesting result was that for non-complete graphs having more than 30% of the total edges the best improvement algorithm was the Saving List 2-opt algorithm. However this iterative algorithm is more time-consuming and the tour quality is only less than 5% better than the other two improvement algorithms. For the non-complete graphs having less than 30% of the total edges, the best improvement algorithm was the Original 2-Opt algorithm with about less than 5% better than the other two algorithms. For all XNN, TSXP and

MST algorithms the commonality of the subset of edges having less than 30% of total edges, is that all of these subsets mostly contain the shortest edges in the compete graph. We assume that this might be the reason that Original 2-Opt shows better results. For subset of edges having 10% or fewer edges, the tour quality decreases for about 0.1% in XNN algorithm. In MST algorithm the decrease is about 15% and in TSXP it is about 20%. This result suggests that having 10% of the top shortest edges (using TSXP algorithm) is not good enough to construct TSP tour compare to XNN algorithm.

Chapter 8

Conclusions and Future Work

In finding a TSP tour, one of the main assumptions in the literature is having a complete graph. However, generating a complete graph for large number of cities is a time-consuming step. In this thesis, we introduced six non-complete graph generating algorithms (XNN, TSXP, MST, HVC, LHVC, and RSPT). Our experimental results suggest the following. Out of the six algorithms, XNN is the best in terms of both tour quality and time-consumption. The resulting tours were between 15%-30% variance from the optimal TSP tour. MST algorithm has almost the same performance as XNN. However, it consumes more time. TSXP requires the same amount of time as computing a complete graph. On the other hand, since the tour quality was less than 60% variance from the optimal result, it suggests that using only the top x% shortest edges, one can produce a near-optimal TSP tours. HVC, LHVC and RSPT hadn't shown a good performance. However, there is a lot of room to enhance the clustering and produce useful subsets of edges.

We have modified the 2-Opt improvement algorithm in a way that we only use a subset of edges that has been produced using one of the six non-complete graph generating algorithms. The experimental results show that for small subsets of edges the Original 2-Opt algorithm has a better performance when compare to the other two improvement algorithms. For subsets of edges having more than 30% of total edges, the Saving List 2-Opt algorithm has a better performance. However, Saving List 2-Opt algorithm is too time consuming when compare to the other two algorithms and having just about 5% better tour quality. The future on improvement algorithms could be trying other than 2-Opt algorithms such as 3-Opt, *k*-Opt and variable-Opt improvement algorithms and compare how the changes on number of Opt moves can change the tour quality. But using 3-Opt or *k*-Opt; it has a drawback of having more than two matching cases when adding an edge to a tour.

Finally the impressive development in solution methods for generating non-complete graphs to produce the TSP gives hope of significant improvements in solution methods for generating non-complete graph for other classes of the VRP.

Bibliography

[1] A E.H.L Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley, Chichester, UK, 1997.

[2] N. Agatz, A.M. Campbell, M. Fleischmann, and M. Savelsbergh. *Challenges and opportunities in attended home delivery*. In B. Golden, S. Raghavan, and E. Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*. 2007.

[3] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, *private communication* (1994).

[4] L.D. Aroson, Algorithms for vehicle routing – A survey. 1995

[5] R. Baldacci, M. Battarra and D. Vigo. Routing a Heterogeneous Fleet of Vehicles. Technical report. 2007.

[6] T. Bektas. *The multiple traveling salesman problem: an overview of formulations and solution procedures*. Omega, 34:209–219, 2006.

[7] J. L. Bentley, *Experiments on traveling salesman heuristics,* in Proc. 1st Ann. ACMSIAM Symposium. On Discrete Algorithms, SIAM, Philadelphia, PA, 1990, 91-99.

[8] J. L. Bentley, *Fast algorithms for geometric traveling salesman problems,* ORSA J. Comput.4 (1992), 387-411.

[9] L. Bertazzi, M.G. Speranza, M.W.P. Savelsbergh, (2008), *Inventory Routing, in The Vehicle routing Problem: Latest Advances and New Chalenges,* B. Golden, R. Raghavan, E. Wasil, (eds.), Operations Research/Computer Science Interfaces Series, 2008.

[10] R. G. Bland and D. F. Shallcross, *Large traveling salesman problems arising from experiments in X-ray crystallography: A preliminary report on computation,* Operations Research. Lett. 8 (1989), 125-128.

[11] L.D. Bodin, B.L. Golden, A.A. Assad, and M. Ball. Routing and scheduling of vehicles and crews, the state of the art. *Computers and Operations Research*, 10(2):63-212, 1983.

[12] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Elsevier North-Holland, 1976.

[13] A. R. Butz, Convergence with Hilbert's space filling curve, J. Comput. Sys. Sci., vol. 3, May 1969, pp 128-146.

[14] E.P.F. Chan, and H. Lim, Optimization and Evaluation of Shortest Path Queries,VLDB Journal (16:3) July 2007, pp.343-369.

[15] E.P.F. Chan, and J. Zhang, , A Fast Unified Optimal Route Query Evaluation Algorithm, Proceedings of ACM 16th Conference on Information and Knowledge Management (CIKM 07), pp. 371-380, Lisboa, Portugal, Nov. 2007.

[16] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem,* Report No. 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[17] N. Christofides. Vehicle routing. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Traveling Salesman Problem*, Wiley, Chichester, UK, 1985, pp. 431-448.

[18] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N. Christofides, A. Mingozzi, P. Toth and C. Sandi, editors, *Combinatorial Optimization*, Wiley, Chichester, UK, 1979, pp. 315-338

[19] G. Clarke and J.V. Wright. *Scheduling of vehicle from a central depot to a number of delivery points*. Operations Research, 12:568-581, 1964.

[20] J-F Cordeau, M Gendreau, G Laporte, J-Y Potvin and F Semet. *A guide to vehicle routing heuristics*. Journal of Operational Research Society. (2002) 53, 512-522.

[21] T.H. Cormen, C.E. Leiserson, R.L.Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, 2001.

[22] G. A. Croes, *A method for solving traveling salesman problems*, Operations Research 6 (1958), 791-812.

[23] H. Crowder and M. Padberg, *Solving large-scale symmetric travelling salesman problems to optimality*, Mgmt. Sci. 26 (1980), 495-509.

[24] B. Chandra, H. Karloff, and C. Tovey, *New results on the old k-opt algorithm for the TSP*, in ‘‘Proceedings 5th ACM-SIAM Symposium on Discrete Algorithms,’’ Society for Industrial and Applied Mathematics, Philadelphia, 1994, 150-159.

[25] G.B. Dantizg and J.H. Ramser. *The truck dispatching problem*. Management Science, 6:80, 1959.

[26] M. Desrochers, J.K. Lenstra, and M.W.P. Savelsbergh. A classification scheme for vehicle routing and scheduling problems. Journal of Operational Research Society, 46:322-332, 1990.

[27] M.L. Fisher. Vehicle routing. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network Routing, Handbooks in Operations Research and Management Science 8*, North-Holland, Amsterdam, 1995, pp. 1-33.

[28] A. M. Frieze, *Worst-case analysis of algorithms for traveling salesman problems*, Methods of Operations Research 32 (1979), 97-112.

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

[30] M. Gendreau, A. Hertz, and G. Laporte, *New insertion and post-optimization procedures for the traveling salesman problem*, Operations Research 40 (1992), 1086-1094.

[31] M. Gendreau, G. Laporte, and J.-Y. Potvin. *Vehicle routing: Modern heuristics.* In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*. Siam, Bologna, Italy, 1997, pp. 129-154.

[32] B.L. Golden, E.A. Wasil, J.P. Kelly, and I.M. Chao. Metaheuristics for the capacitated VRP. In T.G Crainic and G. Laporte, editors, *Fleet Management and Logistics*, Kluwer, Boston, MA, 1998, pp. 33-56.

[33] M. Held and R. M. Karp, *The traveling-salesman problem and minimum spanning trees,* Operations Research 18 (1970), 1138-1162.

[34] M. Held and R. M. Karp, *The traveling-salesman problem and minimum spanning trees: Part II,* Math. Programming 1 (1971), 6-25.

[35] D. Hilbert, "Ueber die stetige Abbildung einer Line auf ein Flächenstück", *Mathematische Annalen* **38**: (1891) 459–460

[36] D. S. Johnson. *A Theoretician's Guide to the Experimental Analysis of Algorithms* To appear in *Proceedings of the 5th and 6th DIMACS Implementation Challenges*, M. Goldwasser, D. S. Johnson, and C. C. McGeoch, Editors, American Mathematical Society, Providence, 2002.

[37] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, and A. Zverovich, *Experimental Analysis of Heuristics for the ATSP* in *The Traveling Salesman Problem and its Variations*, G. Gutin and A. Punnen, Editors, Kluwer Academic Publishers, 2002, Boston, 445-487

[38] D.S. Johnson and L.A. McGeoch. *The traveling salesman problem: A case study*. In E.H.L Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley, Chichester, UK, 1997, pp. 215-310.

[39] D. S. Johnson and L. A. McGeoch, *Experimental Analysis of Heuristics for the STSP* in *The Traveling Salesman Problem and its Variations*, G. Gutin and A. Punnen, Editors, Kluwer Academic Publishers, 2002, Boston, 369-443

[40] M. Junger, G. Reinelt, and G. Rinaldi, *The Traveling Salesman Problem*, Report No. 92.113, Angewandte Mathematik und Informatik, Universit.a. t zu K.o. ln, Cologne, Germany,1994.

[41] B. Korte, *Applications of combinatorial optimization*, talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.

[42] G. Laporte: *The Traveling Salesman Problem: An overview of exact and approximate algorithm*. European Journal of Operations Research 59 (1992), pp. 231-247.

[43] G. Laporte and Y. Nobert. *Exact algorithms for the vehicle routing problem*. Annals of Discrete Mathematics, 31:147-184, 1987.

[44] S. Lin, *Computer solutions of the traveling salesman problem*, Bell Syst. Tech. J. 44 (1965), 2245-2269.

[45] S. Lin and B. W. Kernighan, *An Effective Heuristic Algorithm for the Traveling Salesman Problem*, Operations Research 21 (1973), 498-516.

[46] B.G. Madsen, A. Larsen, M. M. Solomon, *Dynamic Vehicle Routing Systems – Survey and Classification*, Proceeding of the Tristan IV Conference (2007)

[47] H. L. Ong and J. B. Moore, *Worst-case analysis of two traveling salesman heuristics*, Operations Research Letter 2 (1984), 273-277.

[48] I. OR, *Traveling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Regional Blood Banking*, Ph.D. Thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, 1976.

[49] M. Padberg and G. Rinaldi, *Optimization of a 532-city symmetric traveling salesman problem by branch and cut*, Operations Research Letter 6 (1987), 1-7.

[50] C. H. Papadiamitriou and K. Steiglitz, *On the complexity of local search for the traveling salesman problem,* SIAM J. Comput. 6 (1977), 76-83.

[51] C. H. Papadiamitriou and K. Steiglitz, *Some examples of difficult traveling salesman problems*, Operations Research 26 (1978), 434-443.

[52] C. Potts and S. Van de Velde, *Dynasearch—Iterative local improvement by dynamic programming: Part I, The traveling salesman problem,* manuscript (1995).

[53] G. Reinelt, *The Traveling Salesman Problem: Computational Solutions for TSP Applications,* Lecture Notes in Computer Science 840, Springer-Verlag, Berlin, 1994.

[54] H. G. Rinnooy Kan E. L. Lawler, J. K. Lenstra and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985.

[55] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, *An analysis of several heuristics for the traveling salesman problem,* SIAM J. Comput. 6 (1977), 563-581.

[56] S. Sahni and T. Gonzalez, *P-complete approximation problems,* J. Assoc. Comput. Mach. 23 (1976), 555-565.

[57] Schrijver. *On the history of combinatorial optimization (till 1960).* In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, Discrete Optimization, volume 12 of Handbooks in Operations Research and Management Science, chapter 1. Elsevier, 2005.

[58] D. B. Shmoys and D. P. Williamson, *Analyzing the Held-Karp TSP bound: A monotonicity property with applications,* Inform. Process. Lett. 35 (1990), 281-285.

[59] E.D. Tailard, L.M. Gambardella, M. Gendreau, and J.-Y. Potvin. *Adaptive memory programming: A unified view of metaheuristics.* Research Report IDSIA/19-98, IDSIA, Lugano, Switazerland, 1998.

[60] P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*. Siam, Bologna, Italy.

[61] P.Toth and D.Vigo. Exact algorithms for vehicle routing. In T.G Crainic and G. Laporte, editors, *Fleet Management and Logistics*, Kluwer, Boston, MA, 1998, pp. 1-31.

[62] P.Toth and D.Vigo. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*.

[63] L. Wolsey, *Heuristic analysis, linear programming, and branch and bound*, Math. Prog. Stud. 13 (1980), 121-134.

[64] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[65] http://www.tsp.gatech.edu/sweden/index.html

[66] http://www.tsp.gatech.edu/world/index.html

[67] http://www.research.att.com/~dsj/chtsp/