Construction Heuristics for Hard Combinatorial Optimisation Problems

Alexei E. Zverovitch

Draft – September 2, 2002
Table of Contents

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Tour Construction Heuristics for the Asymmetric TSP

1.1 Introduction ................................................. 1
  1.1.1 History of the Travelling Salesman Problem ........ 2
  1.1.2 Types of the TSP ........................................ 2
  1.1.3 Formulations of the TSP ............................... 3
  1.1.4 Complexity ............................................. 4
  1.1.5 Classification of TSP algorithms .................... 4
  1.1.6 Tour construction heuristics ......................... 6

1.2 Terminology and notation .................................... 7

1.3 Algorithms under consideration .............................. 10
  1.3.1 The Greedy algorithm ................................. 10
  1.3.2 Random Insertion ...................................... 14
  1.3.3 Karp-Steele Patching ................................. 15
  1.3.4 Recursive Path Contraction ........................... 17
  1.3.5 Extended Karp-Steele Patching ...................... 20
  1.3.6 Contract-or-Patch heuristic ......................... 22

1.4 Theoretical evaluation of heuristics ......................... 28
## TABLE OF CONTENTS

1.4.1 Approximation analysis ........................................ 28
1.4.2 Domination analysis ........................................... 29

1.5 Computational study ............................................. 31
1.5.1 Computing environment ....................................... 32
1.5.2 Test instances .................................................. 32
1.5.3 Lower bounds ................................................... 33
1.5.4 Implementing Extended Karp-Steele Patching ............... 34
1.5.5 Choice of Contract-or-Patch variant .......................... 36
1.5.6 Computational results .......................................... 41

1.6 Reference implementation of Contract-or-Patch ............... 58

1.7 Conclusions ....................................................... 59

2 A Heuristic for the Resource-Constrained TSP .......................... 61
2.1 Introduction ....................................................... 61
2.2 Heuristic ........................................................ 62
2.3 Lower Bound ..................................................... 63
2.4 Computational results ............................................ 67
2.5 Conclusion ......................................................... 69

3 Detecting Embedded Networks in Linear Programmes ............. 74
3.1 Introduction ....................................................... 74
3.2 Terminology and notation ........................................ 75
3.3 Embedded networks and signed graphs .......................... 76
3.4 Network extraction algorithm .................................... 80
3.5 Implementation details .......................................... 81
3.6 Algorithm complexity ............................................ 84
3.7 Computational study ............................................. 85
3.7.1 Overview ...................................................... 85
3.7.2 Row-Scanning Deletion algorithm ............................ 89
3.7.3 GUB-based algorithms ....................................... 91
3.7.4 Results and discussion ...................................... 94
3.7.5 Choice of spanning forest ................................... 98
3.8 Conclusions ....................................................... 102

A Reference implementation of Algorithm COP ..................... 104
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Reference implementation of Algorithm SGA</td>
<td>111</td>
</tr>
<tr>
<td>Bibliography</td>
<td>120</td>
</tr>
</tbody>
</table>
List of Figures

1.1 COP variants: tour quality ................................. 38
1.2 COP variants: execution time ............................. 39
1.3 Random Asymmetric instances: tour quality and running time 49
1.4 Random Asymmetric instances \((i \times j\)-bounded costs): tour quality and running time ................................. 51
1.5 Random Symmetric instances: tour quality and running time 53
1.6 Random Symmetric instances \((i \times j\)-bounded costs): tour quality and running time ................................. 54
1.7 Sloped Plane instances: tour quality and running time 56

3.1 Average increase in the number of network rows detected by SGA if several spanning forests are built ................................. 102
List of Tables

1.1 Running times of two implementations of Extended Karp-Steele Patching .......................... 35
1.2 Summary of COP variants ............................................. 36
1.3 Asymmetric TSPLIB instances: Excess over optimum ................................. 43
1.4 Asymmetric TSPLIB instances: Running time ................................. 44
1.5 Euclidean TSPLIB instances: Excess over optimum ................................. 45
1.6 Euclidean TSPLIB instances: Running time ................................. 47
1.7 Summary of tour quality results for ATSP construction heuristics 57

2.1 Effectiveness of the algorithm ............................................. 70
2.2 Algorithm running time ............................................. 71
2.3 Adjusted algorithm running time ............................................. 72

3.1 Running times of the row-scan and column-wise scan procedures 84
3.2 Characteristics of test problems and performance of four network detection algorithms ................................. 86
3.3 Distribution of relative performance of the algorithms compared to SGA on test problems with not fewer than 100 rows in the $(0, \pm 1)$-matrix ............................................. 95
3.4 Distribution of running times of the tested algorithms on test problems with not fewer than 100 rows in the $(0, \pm 1)$-matrix 96
3.5 Total running time (in seconds) of each of four algorithms on a set of 100 Netlib problems ................................. 97
3.6 Problem statistics and the number of network rows for supply chain models ................................. 97
3.7 Statistics of 100 runs of SGA on random spanning forests .... 100
List of Algorithms

1.1 Greedy Algorithm .............................................. 11
1.2 Random Insertion .............................................. 14
1.3 Karp-Steele Patching .......................................... 16
1.4 Recursive Path Contraction ................................. 19
1.5 Extended Karp-Steele Patching ....................... 23
1.6 Contract-or-Patch, variant 1 (COP/xxx) ............... 26
1.7 Contract-or-Patch, variant 2 (xxx/COP) ............... 26
2.1 Heuristic for RCTSP .......................................... 64
2.2 Lower bound for RCTSP ...................................... 66
3.1 Algorithm SGA ................................................. 80
3.2 Step 1 of SGA expanded .................................. 81
3.3 Minimum Degree Greedy algorithm ................. 88
3.4 Row-Scanning Deletion algorithm (RSD) .......... 90
3.5 GUB-based algorithm for network detection ....... 92
Chapter 1

Tour Construction Heuristics for the Asymmetric TSP

1.1 Introduction

In this chapter we study construction heuristics for the Asymmetric Travelling Salesman Problem. We develop two new construction algorithms. We also implement a number of well-known construction heuristics, as well a recently published algorithm which possesses some interesting theoretical properties but whose practical performance has never been studied in the literature before. We compare computation performance of the algorithms on a diverse testbed comprising seven different families of problem instances. Computational evidence presented in this chapter suggests that the well-known (and widely used) algorithms may not be a very good choice for the Asymmetric Travelling Salesman Problem. One of the new algorithms clearly outperforms all other tested algorithms in terms of robustness (ability to produce high-quality solutions on a wide variety of instances of the problem). While each of the other algorithms fails very badly on at least one of the seven families of instances, the new algorithm remains competitive on each of the seven instance families. We recommend this new algorithm as a good first choice for a general-purpose construction heuristic for the Asymmetric Travelling Salesman Problem.
1.1.1 History of the Travelling Salesman Problem

The Travelling Salesman Problem (or TSP for short) is among the most studied and well-known problems in the field of combinatorial optimisation. The origins of the problem are somewhat obscure. According to [67], the earliest known reference to the problem goes back to a manual for travelling salesmen published in Germany in 1832. The manual was entitled "Der Handlungsreisende — wie er sien soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein — von einem alten Commiss-Voyageur" ("The travelling salesman — how he should be and what he has to do, to obtain orders and to be sure of a happy success in his business — by an old travelling salesman" [67]). The last chapter of the manual has the earliest known formulation of the Travelling Salesman Problem (the original formulation along with an English translation can be found in [67]). It is not known who was the first mathematician to study the problem; K. Menger is regarded as being the first mathematician to write about it. Menger wrote a number of papers on a closely related subject in late 1920s and early 1930s, and gave a mathematical formulation of the TSP (albeit under a different name — Menger called it the "messenger problem"). Several scientists have studied the problem in early 1930s and this is when the term "Travelling Salesman Problem" was coined, although it is not known by whom. However, it was not until 1954 when a major breakthrough was made. In their paper [24], Dantzig, Fulkerson and Johnson described a method for solving the Travelling Salesman Problem and have successfully applied the method to a 49-city instance of the TSP.

A detailed historical account of the Travelling Salesman Problem can be found in [67]. Over the years, hundreds of research papers and a number of books (such as [36, 55, 66]) have been published studying various aspects of the problem and solution methods.

1.1.2 Types of the TSP

In its simplest form, the Travelling Salesman Problem can be stated as follows: given a set of cities and distances between them, the task is to find the shortest route that visits all cities exactly once and returns to the starting city.
The most frequently studied in the literature type of the TSP is that with Euclidean distances, or "Euclidean TSP" (also known as "Geometric TSP", see [52]). In Euclidean TSP the cities are represented by their coordinates on the plane and distances are Euclidean distances between points on the plane. This is a special case of a more general case when distances are symmetric (that is, distance from city $A$ to city $B$ is the same as distance from $B$ to $A$) but are not necessarily Euclidean. This is known as "Symmetric TSP". In this chapter we study the most general case of the TSP, in which distances can be arbitrary and distance from city $A$ to city $B$ is not necessarily the same as distance from $B$ to $A$. This variation of the TSP is frequently called "Asymmetric TSP" in the literature to distinguish it from other types, such as Symmetric or Euclidean TSP. Note that despite the word "asymmetric" in its name, this type of the Travelling Salesman Problem includes all other types (including symmetric) as special cases.

Other special cases of the Travelling Salesman Problem include those with distances satisfying the triangle inequality (these can be both asymmetric and symmetric, [52]; the latter is also known as "Metric TSP", see for example [71]). A large number of special cases are described in the literature, some of which can be solved in polynomial time; for a survey on the subject, see [16].

In this chapter we study the most general case of Asymmetric TSP. Throughout the chapter, we use terms "TSP", "ATSP" and "Asymmetric TSP" interchangeably. When we need to refer to some special case, we explicitly say so (e.g. "Euclidean TSP").

In order to avoid ambiguity in terminology later in this chapter, in what follows we use the notion of cost instead of distance. Thus the problem becomes that of finding the cheapest rather than the shortest route; the problem is defined by specifying the costs of travelling between each pair of cities instead of distances between cities (note than in the asymmetric case that we study, cost of travel from $A$ to $B$ is not necessarily the same as cost of travel from $B$ to $A$).

### 1.1.3 Formulations of the TSP

Mathematically, the Travelling Salesman Problem can be formulated in a number of equivalent ways. Various formulations of the TSP include:
CHAPTER 1. TOUR CONSTRUCTION HEURISTICS FOR THE ATSP

Combinatorial formulation:
The aim is to find a cyclic permutation $\pi$ of cities $1, 2, \ldots, n$ which minimising $\sum_{i=1}^{n} c(i, \pi(i))$, where $c(i, \pi(i))$ is the cost of travelling from city $i$ to city $\pi(i)$ (see, for example, [68]).

Integer programming formulation:
A number of formulations of the TSP as an integer programming problem exist. The first such formulation is attributed to Dantzig, Fulkerson and Johnson and was published in their 1954 paper [24]. See [72] for a survey of seven different integer programming formulations of the TSP.

Graph-theoretic formulation:
The aim is to find a minimum-weight Hamiltonian cycle in a complete weighted digraph where arc weights represent the costs of travelling between cities.

In this chapter we will exclusively use the graph-theoretic formulation.

1.1.4 Complexity
The Travelling Salesman Problem is NP-complete (see, for example, [23,30]); even Euclidean TSP is NP-complete (see [29,62]). Therefore it is highly unlikely that a polynomial-time algorithm exists than can solve TSP to optimality.

1.1.5 Classification of TSP algorithms
Algorithms for the Travelling Salesman Problem (and other Combinatorial Optimisation problems) can be divided into following categories:

Exact algorithms:
Exact algorithms always find an optimal solution. However, since TSP is NP-hard, in the worst case exact methods may take exponential time (unless P=NP). Examples of exact algorithms include Cutting Planes method (see [24]) and Branch-and-Bound (see [9]). To date, the most successful exact method of solving the Travelling Salesman Problem is Branch-and-Cut (implemented, for example, in [4]), which is a hybrid of the Cutting Planes and Branch-and-Bound methods.
Non-exact algorithms:

As the name implies, non-exact algorithms not always find an optimal solution. Algorithms in this category are frequently called ”approximation algorithms” or ”heuristics”. Sometimes the term ”approximation algorithm” is used to refer to an algorithm with a known solution quality guarantee (see Section 1.4 for more details), while the term ”heuristic” is used to refer to an algorithm without such guarantee. Sometimes the two terms are used interchangeably. In this chapter we use the term ”heuristic” to refer to a non-exact algorithm which may or may not guarantee solutions of particular quality.

TSP heuristics usually run in polynomial time. Although frequently no solution quality guarantee is provided, some heuristics perform very well in practice, producing high-quality solutions in reasonable time.

Heuristic algorithms for the Travelling Salesman Problem can be further subdivided into two broad classes: tour construction and tour improvement algorithms.

Tour construction algorithms:

Tour construction algorithms build a TSP tour from scratch. They usually employ some heuristic rule for incrementally constructing a tour, and stop once a complete tour has been built. Ways in which tours are built can vary: a tour can be constructed one vertex at a time (for example, Nearest Neighbour and various vertex insertion heuristics use this approach), one arc at a time (the Greedy algorithm for the TSP is an example of such an algorithm), or can even be constructed from a cycle cover by eliminating subtours (Karp-Steele Patching of [53, 54]). The latter group of algorithms utilise the fact that the Assignment Problem relaxation of the Travelling Salesman Problem can be solved in \( O(n^3) \) time. The solution of the Assignment Problem can be interpreted as a vertex-disjoint cycle factor of minimum cost, which such algorithms then transform into a (sub-optimal) solution of the TSP. Several algorithms discussed in this study are based on this method.

Tour improvement algorithms:

Tour improvement algorithm are usually iterative procedures which
start with an initial tour, and iteratively try to improve it. Examples of tour improvement algorithms include various local search methods, such as $k$-Opt, Lin-Kernighan [56] and Kanellakis-Papadimitriou [52] algorithms. Applications of various meta-heuristics (for example, simulated annealing, tabu search, evolutionary methods, iterated local search) to the local search framework also fits into this category.

Improvement algorithms usually utilise tour construction algorithms in order to build the initial tour.

In this chapter we study tour construction algorithms, which are discussed in the next subsection.

1.1.6 Tour construction heuristics

A construction heuristic for the Travelling Salesman Problem builds a tour without an attempt to improve the tour once it is constructed. Most of the construction heuristics for the TSP [49, 66] are very fast; they can be used to produce near-optimal solutions for the TSP when the time is restricted, to provide good initial solutions for tour improvement heuristics, to obtain upper bounds for exact algorithms (such as branch-and-bound), etc.

Extensive research has been devoted to construction heuristics for the Euclidean TSP (see, for example, [66]). Construction heuristics for non-Euclidean TSP have been investigated much less. Quite often, the greedy algorithm is chosen as a construction heuristic for non-Euclidean TSP (see, for example, [49]). Computational evidence presented in this chapter shows that this heuristic is far from being the best choice for ATSP in terms of solution quality and robustness. Various insertion algorithms [66] which perform very well for the Euclidean TSP produce poor quality solutions for instances that are not Euclidean (and are not close to Euclidean, such as, for example, Sloped Plane instances defined in Subsection 1.5.2).

In this chapter we study construction heuristics for the Asymmetric TSP. We develop two new construction heuristics for the Asymmetric TSP. We also present results of a detailed computational study of the new algorithms as well as a number of well-known heuristics. The study is performed on a diverse set of seven different families of TSP instances. These results show
that overall one of the new algorithms clearly outperforms the well-known
construction heuristics for the Asymmetric Travelling Salesman Problem.
While other heuristics produce good quality tours for some families of TSP
instances and fail (sometimes badly) on some other families, the new algo-
rithm appears to be much more robust. On the testbed considered, it usually
produces tours of quality comparable with the winner of each respective type
of instance (it is the winner in 4 out of 7 cases), and never fails as badly as
the other algorithms tested in this study.

For the reader interested in solving instances from a particular family
of non-Euclidean TSP instances, this chapter may suggest a construction
algorithm which is appropriate for the type of instances under consideration.

1.2 Terminology and notation

A directed graph (or just digraph) \( D \) consists of a non-empty finite set \( V(D) \)
of elements called vertices or nodes and a finite set \( A(D) \) of ordered pairs of
distinct vertices called arcs. We call \( V(D) \) the vertex set and \( A(D) \) the arc
set of \( D \). We also write \( D = (V, A) \) which means that \( V \) and \( A \) are the vertex
set and arc set of \( D \) respectively. The first vertex of an arc is its initial vertex
tail, the second vertex is the arc’s terminal vertex head. An arc \( a \) with
tail \( x \) and head \( y \) is written as \( xy \) or \( (x, y) \); we say that \( x \) is an in-neighbour
of \( y \), and \( y \) is an out-neighbour of \( x \), and that \( x \) and \( y \) are adjacent.

A directed graph \( D \) is complete if every for pair of distinct vertices \( x \) and
\( y \) of \( D \) both \( xy \) and \( yx \) are arcs of \( D \). A complete directed graph on \( n \) vertices
is denoted by \( \vec{K}_n \).

A path or cycle \( W \) in directed graph \( D \) is a Hamiltonian path or cycle if
\( V(W) = V(D) \). A tour is a Hamiltonian cycle and a subtour is an arbitrary
cycle.

A weighted digraph is a digraph \( D = (V, A) \) along with a mapping \( c : A \to \mathbb{R} \),
where \( \mathbb{R} \) is the set of real numbers. Thus, a weighted digraph is a
triple \( D = (V, A, c) \). If \( a \) is an arc of a weighted digraph \( D = (V, A, c) \), then
\( c(a) \) is called the weight (or cost) of \( a \). If the vertices of \( D = (V, A, c) \) are
labelled, say \( x_1, x_2, \ldots, x_n \), it is convenient to determine the cost of the arcs
by the cost matrix with entries \( c_{ij} = c(x_i, x_j) \) for \( i \neq j \) and \( c_{ii} = 0 \).
The weight or cost \( c(A') \) of a subset \( A' \subseteq A \) is the sum of the costs of arcs of \( A' \). Thus, the cost of a cycle or path \( W \) is the sum of the costs of the arcs of \( W \).

The length of a cycle or path \( W \) is the number of arcs in \( W \). The first vertex of a path is called its initial vertex (tail), the last vertex is its terminal vertex (head). Thus, \( v_1 \) is the initial vertex (tail) and \( v_m \) is the terminal vertex (head) of path \( P = v_1v_2 \ldots v_m \).

The Asymmetric Travelling Salesman Problem (ATSP) is defined as follows: given a weighted complete digraph \( \overrightarrow{K}_n \) on \( n \) vertices, find a Hamiltonian cycle (tour) \( T \) of \( \overrightarrow{K}_n \) of minimum cost. In the ATSP context, we will frequently label graph vertices 1, 2, \ldots, \( n \) and define the instance of the problem by giving the \( n \times n \) cost matrix with rows corresponding to arc tails and columns corresponding to arc heads; elements of the matrix represent costs of the corresponding arcs.

The domination number of tour \( T \) in \( \overrightarrow{K}_n \) is the number of tours in \( \overrightarrow{K}_n \) which are more expensive or have the same cost as \( T \) (including \( T \) itself). The domination number \( \text{domn}(H, n) \) of heuristic \( H \) for the ATSP is the maximum integer \( k = k(n) \), such that for every ATSP instance on \( n \) vertices, \( H \) produces a tour whose domination number is at least \( k \).

A cycle factor of \( \overrightarrow{K}_n \) is a collection of vertex-disjoint cycles in \( \overrightarrow{K}_n \) covering all vertices of \( \overrightarrow{K}_n \). The cost of a cycle factor \( F \) is the sum of the cost of the individual cycles comprising \( F \). A cycle factor in \( \overrightarrow{K}_n \) of minimum cost can be found in time \( O(n^3) \) using algorithms for solving the Assignment Problem defined on the corresponding weighted complete bipartite graph (see \([21, 53, 54]\)). Since each tour in \( \overrightarrow{K}_n \) is a valid cycle factor in \( \overrightarrow{K}_n \), the cost of the cheapest cycle factor of \( \overrightarrow{K}_n \) provides a lower bound to the solution of the ATSP. We call this lower bound the Assignment Problem (AP) lower bound.

Let \( C = x_1x_2\ldots x_kx_1 \) be a cycle in \( \overrightarrow{K}_n \). The operation of removal of a vertex \( x_i \) (\( 1 \leq i \leq k \)) results in the cycle \( x_1x_2\ldots x_{i-1}x_{i+1}\ldots x_kx_1 \). Thus, removal of \( x_i \) is not deletion of \( x_i \) from \( C \); deletion of \( x_i \) gives the path \( x_i+1x_{i+2}\ldots x_kx_1x_2\ldots x_{i-1}x_i \). Similarly, deletion of arc \( x_i, x_{i+1} \) from \( C \) gives the path \( x_i+1x_{i+2}\ldots x_kx_1x_2\ldots x_{i-1}x_i \).

Let \( y \) be a vertex of \( \overrightarrow{K}_n \) not in \( C \). The operation of insertion of \( y \) into an arc \( (x_i, x_{i+1}) \) results in the cycle \( x_1x_2\ldots x_iyx_{i+1}\ldots x_kx_1 \). The cost of the inser-
CHAPTER 1. TOUR CONSTRUCTION HEURISTICS FOR THE ATSP

tion is defined as \( c(x_i, y) + c(y, x_{i+1}) - c(x_i, x_{i+1}) \). For a set \( Z = \{z_1, \ldots, z_s\} \) (\( s \leq k \)) of vertices not in \( C \), an insertion of \( Z \) into \( C \) results in the tour obtained by inserting the nodes of \( Z \) into different arcs of the cycle. In particular, insertion of \( y \) into \( C \) involves insertion of \( y \) into one of the arcs of \( C \).

For a path \( P = x_1x_2...x_m \) in \( (\vec{K}_n, c) \), the contraction of \( P \) in \( (\vec{K}_n, c) \), \( (\vec{K}_n / P, c') \), is a complete digraph with vertex set

\[
V(\vec{K}_n / P) = V(\vec{K}_n) \cup \{v_P\} - V(P),
\]

where \( v_P \) is a new vertex, \( v_P \notin V(\vec{K}_n) \). The cost \( c'(u, w) \) of arc \( uw \), for \( u, w \in V(\vec{K}_n / P) \), is defined as follows:

\[
c'(u, w) = \begin{cases} 
  c(u, x_1) & \text{if } w = v_P \\
  c(x_m, w) & \text{if } u = v_P \\
  c(u, w) & \text{otherwise.}
\end{cases}
\] (1.1)

We can consider an arc \( a = (x, y) \) as the path \( xy \) of length one; this allows us to look at \( \vec{K}_n / a \) as a special case of the above definition. The above definition has an obvious extension to a set of vertex-disjoint paths.

We also use the operation of patching of two cycles \( C_1 \) and \( C_2 \), which is defined as follows: fixed arcs \( x_1y_1 \) from \( C_1 \) and \( x_2y_2 \) from \( C_2 \) are deleted and two arcs joining the cycles together \( (x_1y_2 \text{ and } x_2y_1) \) are added. The patching is said to be at arcs \( x_1y_1 \) and \( x_2y_2 \). The cost of patching of \( C_1 \) and \( C_2 \) at \( x_1y_1 \) and \( x_2y_2 \) is defined as \( c(x_1, y_2) + c(x_2, y_1) - c(x_1, y_1) - c(x_2, y_2) \). This cost is the difference between the sum of the costs of the inserted arcs and the sum of the costs of the deleted arcs.

Clearly, two cycles \( C_1 \) and \( C_2 \) of lengths \( l_1 \) and \( l_2 \) can be patched together in \( l_1l_2 \) different ways. We sometimes refer to the cheapest patching of two cycles, which is simply the patching that has minimum cost among all \( l_1l_2 \) possible patching operations.
1.3 Algorithms under consideration

This section provides detailed descriptions of the tour construction algorithms considered in this study. The algorithms we test include three well-known tour construction heuristics for the ATSP: the Greedy algorithm, Random Insertion and Karp-Steele Patching. These algorithms are described in detail in Subsections 1.3.1, 1.3.2 and 1.3.3 respectively. In this work we also test the heuristic of [74], the practical performance of which has never been studied in the literature before. Subsection 1.3.4 provides details of this algorithm, which we call Recursive Path Contraction. Finally, we develop two new tour construction algorithms, Extended Karp-Steele Patching and Contract-or-Patch. These two algorithms are detailed in Subsections 1.3.5 and 1.3.6 respectively.

1.3.1 The Greedy algorithm

The Greedy algorithm (denoted by GR) is one of the most well-known tour construction heuristics for the Travelling Salesman Problem.

The Greedy algorithm for the ATSP builds a tour by repeatedly choosing the cheapest eligible arc of $(\mathcal{K}_n, c)$ until the chosen arcs form a Hamiltonian cycle. An arc $a = uv$ is eligible if:

1. none of the previously selected arcs have the same initial vertex $u$ or the same terminal vertex $v$; and
2. the arc can be added to the set of already selected arcs without creating a non-Hamiltonian cycle.

The Greedy algorithm can also be formulated in terms of arc contraction as follows. The algorithm starts with the initial complete digraph $(K, \overline{\pi}) := (\mathcal{K}_n, c)$. The algorithm selects the cheapest arc $a$ in $(K, \overline{\pi})$, and contracts it. The same procedure is recursively applied to the contracted digraph $(K, \overline{\pi}) := (K/a, (\overline{\pi})')$ till $|V(K)| = 2$ (that is, $K$ consists of two vertices connected by two arcs). The remaining two arcs together with the arcs contracted during the previous steps of the algorithm form the resulting (“greedy”) tour.
Algorithm 1.1 Greedy Algorithm

**Arguments:**
- \( n \): number of vertices in the ATSP instance;
- \( C = [c(uv)] \): \( n \times n \) matrix of inter-city distances (arc costs);

**Result:**
- \( T \): Greedy tour;

**Variables:**
- \( Selected \): set of selected arcs;
- \( Initial, Terminal \): Boolean arrays of size \( n \);
- \( Colour \): integer array of size \( n \);
- \( Arcs \): array of size \( n(n - 1) \) containing all arcs in the graph;

**Invariants:**
- \( Initial[u] = \text{true} \) iff \( uv \in Selected \) for some \( v \);
- \( Terminal[v] = \text{true} \) iff \( uv \in Selected \) for some \( u \);
- \( Colour[u] = Colour[v] \) iff there is a path from \( u \) to \( v \) or from \( v \) to \( u \) in \( Selected \);

**begin**

```
Selected ← ∅
for \( i ← 1 \) to \( n \) do
    Initial[i] ← Terminal[i] ← false
    Colour[i] ← i
end for

Arcs ← \{uv : 1 ≤ u, v ≤ n, u ≠ v\} (* All arcs in the graph *)
Sort Arcs: \( c(Arcs[i]) ≤ c(Arcs[j]) \) for all \( 1 ≤ i < j ≤ n \)

for each \( a \) in \( Arcs \) do
    Let \( u \) and \( v \) be the initial and terminal vertices of \( a \) respectively: \( a = uv \)
    if \( Initial[u] = \text{true} \) or \( Terminal[v] = \text{true} \), then continue
    if \( Colour[u] = Colour[v] \) and \( |Selected| < n - 1 \), then continue
    Selected ← Selected ∪ \{a\}
    if \( |Selected| = n \), then stop: arcs in \( Selected \) form the greedy tour \( T \)
    Initial[u] ← Terminal[v] ← true
    Colour[w] ← Colour[u] for all \( w \) such that \( Colour[w] = Colour[v] \)
end for
```
end
Our implementation of the Greedy algorithm is based on the first formulation of the algorithm. The code builds a list of all arcs of $\vec{K}_n$ sorted by non-decreasing costs, and then sequentially scans this list choosing all eligible arcs until they form a Hamiltonian cycle, at which point the algorithm stops.

Running time of such an implementation is dominated by the sorting step. An implementation of GR based on the Quicksort routine of [46] has $O(n^2 \log n)$ average time complexity and $O(n^4)$ worst-case complexity. If the sorting routine used to sort arcs guarantees $O(n \log n)$ worst-case complexity (for example, Heapsort of [73] and Introsort of [60] provide such guarantee), the worst-case time complexity of the Greedy algorithm becomes $\Theta(n^2 \log n)$. The algorithm requires $\Theta(n^2)$ memory, which is necessary to store and sort the list of all arcs of $\vec{K}_n$.

The Greedy algorithm is formally presented in Algorithm 1.1. The pseudo-code uses a vertex colouring technique to detect non-Hamiltonian cycles. At each stage of the algorithm, the set of currently selected arcs defines a set of directed vertex-disjoint paths in the original graph. All vertices along the same path are assigned the same colour, and different paths have different colours. Isolated vertices that do not belong to any path are treated as paths of length zero, and therefore each of such vertices has a unique colour assigned to it. Whenever an arc is considered, it is easy to see whether selecting this arc will result in the formation of a cycle. A cycle will be formed if and only if the arc’s initial and terminal vertices are of the same colour. Thus non-Hamiltonian cycles can be easily avoided.

Vertex colours need to be updated whenever an additional arc is selected. The arc joins two paths together and vertices need to be re-coloured so that all vertices in the new combined path are of the same colour (for example, by changing the colour of all vertices in either of the two original paths to match the colour of the other original path). A straightforward procedure requires $\Theta(n^2)$ time to perform all $n-1$ path re-colourings. Thus the overall time complexity of the Greedy algorithm is clearly dominated by the sorting step, which requires $\Theta(n^2 \log n)$ time in the worst case.

The above procedure for detecting non-Hamiltonian cycles can be further improved by noting that only path initial and terminal vertices need to be coloured accurately in order for the algorithm to function correctly; colours of vertices other than initial and terminal path vertices are never used by the
algorithm. Having made this observation, it becomes possible to dispense
with the idea of vertex colours, and only track path initial and terminal
vertices (path endpoints) and vertices at the other end of the respective paths,
as illustrated below.

Assume \( n = 6 \) and we have directed paths \((1, 3, 6), (2), \text{ and } (4, 5)\). The
data structure described above (denoted by `other_endpoint` here) will contain
the following values:

- \( \text{other_endpoint}[1] = 6 \) and \( \text{other_endpoint}[6] = 1 \);
- \( \text{other_endpoint}[2] = 2 \);
- \( \text{other_endpoint}[4] = 5 \) and \( \text{other_endpoint}[5] = 4 \);
- \( \text{other_endpoint}[3] \) is undefined as vertex 3 is not a path endpoint.

Given an arc \( a = uv \) that satisfies the first eligibility criterion defined
on page 10, it is easy to see whether adding this arc will cause a cycle
to be created. This will happen if and only if \( \text{other_endpoint}[v] = u \) (or
\( \text{other_endpoint}[u] = v \)). If the addition of the arc will not create any non-
Hamiltonian cycles, the arc \( a = uv \) can be added to the list of selected arcs.
The `other_endpoint` data structure can be updated in constant time to reflect
this by using the following constant-time procedure:

\[
\begin{align*}
u' & \leftarrow \text{other_endpoint}[u] \\
v' & \leftarrow \text{other_endpoint}[v] \\
\text{other_endpoint}[u'] & \leftarrow v' \\
\text{other_endpoint}[v'] & \leftarrow u'
\end{align*}
\]

Note that the vertices \( u \) and \( v \) are no longer path endpoints and therefore
the values of \( \text{other_endpoint}[u] \) and \( \text{other_endpoint}[v] \) need not be updated.

Overall, this data structure requires \( \Theta(n) \) operations in order to allow
detecting non-Hamiltonian cycles at all stages of the Greedy algorithm. Thus,
it represents an improvement over the \( \Theta(n^2) \) time required by the original
procedure based on vertex colouring. However, the overall complexity of the
Greedy algorithm is dominated by the sorting step, so improving complexity
of detecting non-Hamiltonian cycles is unlikely to have significant impact
on the overall performance of the algorithm. Therefore, we chose to not
implement this improvement and use the colouring-based procedure we have already implemented.

1.3.2 Random Insertion

The second well-known tour construction heuristic considered in this study is Random Insertion (denoted by RI). Both Greedy and Random Insertion heuristics are extensively used, especially for the Euclidean TSP (see, for example, [49,66]), and they are known to produce good results for the Euclidean problem. In this work we investigate their performance for the Asymmetric TSP.

Algorithm 1.2 Random Insertion

Arguments:
\( n \): number of vertices in the ATSP instance;
\( C = [c(\text{uv})] \): \( n \times n \) matrix of inter-city distances (arc costs);

Result:
\( T \): tour;

begin
\( (v_1, v_2, \ldots, v_n) \leftarrow \) random permutation of \( (1, 2, \ldots, n) \)
\( T \leftarrow (v_1, v_2, v_1) \) (* Below, \( T_i \) denotes \( i \)-th vertex in \( T \) starting from \( v_1 \) *)

for \( k \leftarrow 3 \) to \( n \) do
  \( c_0 \leftarrow +\infty \)
  for \( j \leftarrow 1 \) to \( k - 1 \) do
    \( c' \leftarrow c(T_jv_k) + c(v_kT_{j+1}) - c(T_jT_{j+1}) \)
    if \( c' < c_0 \) then
      \( c_0 \leftarrow c' \)
      \( j_0 \leftarrow j \)
    end if
  end for
  Insert \( v_k \) into \( T \) after \( j_0 \)
end for
return \( T \)
end
The Random Insertion algorithm starts by constructing a random ordering \( v_1, \ldots, v_n \) of all vertices of \((\overrightarrow{K}_n, c)\) and forming the cycle \( v_1 v_2 v_1 \). The vertices \( v_3, \ldots, v_n \) are then inserted into the cycle one by one in this order, each inserted in the cheapest possible way (that is, minimising the cost of the cycle after the insertion). More formally, for \( k = 3, \ldots, n \) the vertex \( v_k \) is inserted into the current \((k-1)\)-cycle \( v_{\pi(1)} v_{\pi(2)} \ldots v_{\pi(k-1)} v_{\pi(k)=\pi(1)} \) between the vertices \( v_{\pi(j_0)} \) and \( v_{\pi(j_0+1)} \), where \( j_0 \) minimises the expression

\[
c(v_{\pi(j)} v_k) + c(v_k v_{\pi(j+1)}) - c(v_{\pi(j)} v_{\pi(j+1)})
\]

over all \( j, 1 \leq j \leq (k-1) \). Clearly, when all vertices have been inserted into the cycle, a tour is formed.

Random Insertion is formally presented in Algorithm 1.2.

Random Insertion is the fastest among the algorithms considered in this study, and has \( \Theta(n^2) \) time and \( \Theta(n) \) space complexity.

### 1.3.3 Karp-Steele Patching

The Karp-Steele Patching algorithm (denoted by KSP) originates from [53, 54]. It is first of several algorithms considered in this study which are based on solving the Assignment Problem and then transforming the resulting cycle factor into a Hamiltonian cycle.

The Karp-Steele Patching algorithm first constructs a minimum cost cycle factor by solving the Assignment Problem. If the cycle factor consists of a single cycle, it is an optimal solution for the problem. Otherwise, the algorithm repeats the following procedure until a single (Hamiltonian) cycle is obtained: select the two longest cycles and combine them into one by performing the cheapest patching of the two cycles.

Assume that the minimum cost cycle factor \( \mathcal{F} \) in \((\overrightarrow{K}_n, c)\) consists of \( m \) cycles: \( \mathcal{F} = \{C_1, C_2, \ldots, C_m\} \). The length (that is, the number of vertices) of \( C_i \) is denoted by \( k_i \). Clearly, \( \sum_{i=1}^{m} k_i = n \), where \( n \) is the number of vertices in \( \overrightarrow{K}_n \). Assume that cycles \( C_1, \ldots, C_m \) are ordered from the longest to the shortest: \( k_1 \geq k_2 \geq \ldots \geq k_m \). At each step of the algorithm, the two currently longest cycles are combined into one, which becomes the new longest cycle in \( \mathcal{F} \). At the \( i \)-th patching step the currently longest cycle
Algorithm 1.3 Karp-Steele Patching

Arguments:
\((K_n, c)\): ATSP instance;

Result:
\(T\): tour;

Notation:
\(A(C)\): set of arcs of cycle \(C\);

\(|C|\): length of cycle \(C\);

\(patching\_cost(C_1, a_1, C_2, a_2)\): cost of patching \(C_1\) and \(C_2\) at arcs \(a_1\) and \(a_2\);

\(patch(C_1, a_1, C_2, a_2)\): patch \(C_1\) and \(C_2\) at \(a_1\) and \(a_2\) and return the resulting cycle;

begin
Solve the Assignment Problem and obtain cycle factor \(F = \{C_1, \ldots, C_m\}\).
Sort cycles in \(F\) by their lengths: \(|C_1| \geq |C_2| \geq \ldots \geq |C_m|\).
\(C \leftarrow C_1\) (* The longest cycle *)
for \(i \leftarrow 2\) to \(m\) do
\((*\ \text{Find cheapest patching of } C \text{ and } C_i\ *))\n\(c_0 \leftarrow +\infty\)
for \(x\ in\ A(C)\) do
\(\text{for } y \ in \ A(C_i) \ do\)
\(c' \leftarrow patching\_cost(C, x, C_i, y)\)
if \(c' < c_0\) then
\(c_0 \leftarrow c'\)
\(x_0 \leftarrow x\)
\(y_0 \leftarrow y\)
end if
end for
end for
\((*\ \text{Perform the cheapest patching } *)\)
\(C \leftarrow patch(C, x_0, C_i, y_0)\)
end for
\(T \leftarrow C\)
return \(T\)
end
(which is the result of the previous \((i - 1)\) patching operations) has length \(\sum_{j=1}^{i} k_j\). The second longest cycle has length \(k_{i+1}\). Thus the number of operations required on the \(i\)-th step to find the cheapest patching between the two longest cycles is \(O(k_{i+1} \sum_{j=1}^{i} k_j)\). Since the number of such steps is \((m - 1)\), the overall number of patching operations considered is

\[
O\left(\sum_{i=1}^{m-1} (k_{i+1} \sum_{j=1}^{i} k_j)\right) = O(n^2).
\]

The rest of the patching procedure (including initialisation and actually carrying out the \((m - 1)\) patching operations) can be implemented to run in \(o(n^2)\) time. Thus, the overall complexity of the patching procedure is \(O(n^2)\). In practice, the running time of the algorithm is usually dominated by that of solving the Assignment Problem, which may potentially require \(\Theta(n^3)\) time.

The algorithm has no significant memory overhead and a straightforward implementation should not require more than \(O(n)\) intermediate storage.

The Karp-Steele Patching algorithm is formally presented in Algorithm 1.3.

Our implementation of the heuristic uses a \(C++\) implementation of the shortest augmenting path algorithm of [50] for solving the Assignment Problem.

1.3.4 Recursive Path Contraction

The Recursive Path Contraction algorithm (denoted by RPC) originates from [74]. Similarly to the Karp-Steele Patching heuristic studied in the previous section, this algorithm is based on building a cycle factor of minimum cost and transforming the cycle factor into a tour. Unlike the previous three algorithms, computational performance of Recursive Path Contraction has never been studied in the literature before.

As the name implies, the algorithm is based on a recursive procedure. The algorithm first solves the Assignment Problem in order to construct a minimum cost cycle factor \(F = \{C_1, C_2, \ldots, C_m\}\). If \(F\) consists of a single cycle, the algorithm terminates and returns this cycle as an optimal solution to the problem. If the number of cycles \(m\) is greater than one, the most expensive arc of every cycle is deleted, thus converting each cycle \(C_i\) to path
CHAPTER 1. TOUR CONSTRUCTION HEURISTICS FOR THE ATSP

Paths $P_1, P_2, \ldots, P_m$ of $(\vec{K}_n, c)$ are contracted one by one. The resulting digraph is denoted by $(\vec{K}_m, c')$; note that every arc of $(\vec{K}_m, c')$ has a corresponding arc in $(\vec{K}_n, c)$. A Hamiltonian cycle $H$ in $(\vec{K}_m, c')$ is then obtained by recursively applying the algorithm to $(\vec{K}_m, c')$. Paths $P_1, P_2, \ldots, P_m$ of $(\vec{K}_n, c)$ together with the arcs of the digraph $(\vec{K}_n, c)$ that correspond to the arcs of $H$ form a Hamiltonian cycle in $(\vec{K}_n, c)$. This Hamiltonian cycle is returned as the heuristic solution to the problem.

The main feature of this algorithm is the fact that it produces solutions with large domination numbers. TSP heuristics yielding tours with exponential yet much smaller domination numbers were introduced in the literature on so-called exponential neighbourhoods (for a comprehensive survey of the topic, see [25]). Exponential neighbourhood local search [32, 35, 37] has already shown some computational potential for finding near-optimal solutions to the Travelling Salesman Problem (see, for example, [8, 17]). Domination analysis is discussed in Subsection 1.4.2.

Let $t$ be the number of levels of recursion of RPC that are necessary to build a Hamiltonian cycle in some $(\vec{K}_n, c)$. We denote the cycle factors built at each level of recursion as $F_1, F_2, \ldots, F_t$. The number of cycles in $F_i$ is denoted by $m_i$. It can be shown that the domination number of the tour constructed by RPC is at least $(n - m_1 - 1)! (m_1 - m_2 - 1)! \ldots (m_{t-1} - m_t - 1)!$ (see [74]). This number is quite large when the number of cycles in cycle factors is small, which is often the case for pure asymmetric instances of the TSP (that is, instances for which $c(u, v) \neq c(v, u)$ for all or almost all pairs of vertices $u \neq v$).

The Recursive Path Contraction algorithm is formally presented in Algorithm 1.4.

We now show that the algorithm can be implemented to run in $O(n^3)$ time. Solving the Assignment Problem requires $O(n^3)$ time. The number of cycles in the cycle factor $F$ cannot exceed $\lfloor n/2 \rfloor$. This means that the algorithm is applied recursively to a problem of size of at most half of the


\textbf{Algorithm 1.4 Recursive Path Contraction}

\begin{algorithm}
procedure $RPC$;
\begin{itemize}
\item \textbf{Arguments:}
\begin{itemize}
\item $(\vec{K}_n, c)$: ATSP instance;
\end{itemize}
\item \textbf{Result:}
\begin{itemize}
\item $T$: tour;
\end{itemize}
\end{itemize}
\begin{algorithmic}
\begin{itemize}
\itembegin
\item Solve the Assignment Problem and obtain cycle factor $\mathcal{F} = \{C_1, \ldots, C_m\}$.
\item $\mathcal{P} \leftarrow \emptyset$ (* Set of paths *)
\item for $i \leftarrow 1$ to $m$ do
\begin{itemize}
\item Delete most expensive arc from $C_i$, store result in $\mathcal{P}_i$
\item $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{P}_i\}$
\end{itemize}
\itemend for
\item $(\vec{K}_m, c') \leftarrow (\vec{K}_n, c)/\mathcal{P}$ (* Contract paths in $\mathcal{P}$ *)
\item $T' \leftarrow RPC((\vec{K}_m, c'))$ (* Apply the procedure recursively to $(\vec{K}_m, c')$ *)
\item Expand vertices of $T'$ to corresponding arcs of $\mathcal{P}$ to obtain $T$
\item return $T$
\end{itemize}
\end{algorithmic}
\end{algorithm}

\end{algorithm}
size of the original problem. We have
\[
O(n^3 + (n/2)^3 + (n/4)^3 + (n/8)^3 + \ldots + 1) \leq \\
O(n^3 \sum_{k=0}^{\infty} (2^{-3})^k) = \\
O(n^3 \frac{1}{1 - 2^{-3}}) = \\
O(n^3).
\]
Thus the time complexity of the algorithm is \(O(n^3)\).

Space complexity depends greatly on data structures employed, but a cubic-time implementation is possible that only requires \(O(n)\) space. The first level of recursion can be implemented to require \(\Theta(n)\) storage. The second level of recursion requires \(\Theta(n/2)\) memory to define the contracted instance (this can be done by simply storing the mapping between contracted and original vertices). Additionally, the second level of recursion requires \(\Theta(n/2)\) storage to solve the assignment problem. Similarly, the third level requires \(\Theta(n/4) + \Theta(n/4)\) storage, and so on. The overall space complexity is
\[
O(n + (n/2 + n/2) + (n/4 + n/4) + \ldots + 1) = \\
O(n(1 + (1 + 1/2 + 1/4 + 1/8 + \ldots + 1/n))) \leq \\
O(n(1 + \frac{1}{1 - 1/2})) = \\
O(n).
\]

Our implementation of the heuristic uses a C++ implementation of the shortest augmenting path algorithm of [50] for solving the Assignment Problem.

1.3.5 Extended Karp-Steele Patching

Extended Karp-Steele Patching (EKSP) algorithm also starts by solving the Assignment Problem and constructing a minimum-cost cycle factor. This is a new algorithms which is based on Karp-Steele Patching but considers a larger set of patching operations. The patching routine is different from
that of the original KSP routine discussed in Subsection 1.3.3: instead of patching together the two longest cycles it performs a patching operation that has minimum cost and involves any two cycles in the current cycle factor. The above procedure is repeated until all cycles have been combined together to form a tour.

Clearly, this procedure considers a larger set of patching operations compared to original Karp-Steele Patching procedure at the cost of higher computational complexity. In the worst case (when the number of cycles in the minimum cost cycle factor is $\lfloor n/2 \rfloor$), the number of pairs of cycles that are considered for patching during the whole patching procedure is

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \frac{i(i-1)}{2} = \frac{1}{6}((\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1) = \Theta(n^3),$$

each taking at least $\Theta(1)$ time. Thus, a simple implementation of the patching procedure has $\Omega(n^3)$ worst-case complexity. In what follows, let $u$ and $v$ be a pair of vertices that belong to different cycles; the number of such pairs is not greater than $n(n - 1)/2$. We say that a patching is at $u$ and $v$ if it deletes the two arcs originating from $u$ and $v$ in the current cycle factor and appends the two arcs that join the cycles together.

Clearly, patching at the two given vertices $u$ and $v$ does not need to be considered more than $(\lfloor n/2 \rfloor - 1)$ times, which is the total number of patching operations performed by the algorithm. Recalling that there are at most $n(n - 1)/2$ choices of $u$ and $v$ and every patching operation can be considered in constant time, we have that the patching procedure does not require more than $O(\frac{1}{2}n(n - 1)((\lfloor n/2 \rfloor - 1)) = O(n^3)$ time. We have shown that the worst-case complexity of the patching procedure is both $\Omega(n^3)$ and $O(n^3)$, hence it is $\Theta(n^3)$. Thus the overall time complexity of the algorithm is $O(n^3)$. It is possible to implement the algorithm to require as little as $\Theta(n)$ space, although such an implementation may not be a very efficient one in practice.

As we will show in Subsection 1.5.4, a straightforward implementation of this algorithm would be very inefficient in terms of execution time. To partly overcome this obstacle, we propose an implementation of the algorithm that employs some additional data structures in order to make finding the
cheapest available patching faster. In particular, we use an $n \times n$ matrix $M = [m(i, j)]$ with rows and columns corresponding to vertices of $(\overline{K}_n, c)$. Each element $m(i, j)$ of $M$ represents the cost of patching at the vertices $i$ and $j$, or is set to $+\infty$ if patching at $i$ and $j$ is not available (that is, when $i$ and $j$ are in the same cycle). Clearly, finding the cheapest patching in the current cycle factor is equivalent to finding the smallest element in $M$. To facilitate the search for the smallest element in $M$, we introduce an additional $n$-vector $B = [b(i)]$ with elements corresponding to rows of $M$. Each entry $b(i)$ caches the column index of the smallest element of the $i$-th row of $M$ if such element is known, or is set to "unknown" otherwise.

The smallest element of $M$ can be found by scanning $B$ once and taking the smallest over all rows of $M$; entries of $B$ that are set to "unknown" are re-calculated along the way by scanning the corresponding rows of $M$ and locating the smallest row elements. Whenever an element of $M$ changes (this happens after a patching has been performed), the corresponding entry of $B$ is updated incrementally in constant time if possible, or set to "unknown" otherwise. The algorithm is based on the observation that most of the time entries of $B$ can be kept up-to-date incrementally, and a full re-scan of the row of $M$ in order to re-calculate an element of $B$ is needed relatively seldom. Thus, the cheapest patching in the current cycle factor can often be found in $O(n)$ time.

The algorithm does not improve on the worst-case time complexity, which is $O(n^3)$, but computational evidence presented in Subsection 1.5.4 on page 34 suggests that this version of the algorithm offers a significant reduction in execution times compared to a straightforward implementation. Pseudo-code for this implementation of Extended Karp-Steele Patching is presented in Algorithm 1.5.

1.3.6 Contract-or-Patch heuristic

It is perhaps intuitively clear that performance of the Karp-Steele Patching algorithm (and, to a lesser extent, of Extended Karp-Steele Patching) suffers when the initial cycle factor contains a large number of short cycles, especially those of length two. Computational evidence confirms this.

On the other hand, the Recursive Path Contraction algorithm performs
Algorithm 1.5 Extended Karp-Steele Patching

Arguments:
\( (K_n, c) \): ATSP instance;

Result:
\( T \): tour;

Notation:
\( \text{patching cost}(\mathcal{F}, v_1, v_2) \): cost of patching \( C_1 \) and \( C_2 \) at vertices \( v_1 \) and \( v_2 \);
\( \text{patching cost}(\mathcal{F}, v_1, v_2) \) returns +\( \infty \) if \( v_1 \) and \( v_2 \) belong to the same cycle in \( \mathcal{F} \);
\( \text{patch}(\mathcal{F}, v_1, v_2) \): modify \( \mathcal{F} \) by patching together two cycles at \( v_1 \) and \( v_2 \);

Variables:
\( M \): symmetric \( n \times n \) matrix of patching costs, only the lower triangle is stored;
\( B \): \( n \)-vector of column indices in \( M \);

procedure update\_cost\( (\text{row, col, cost}) \); (* Update \( M \) and \( B \) *)
begin
if (row < col) swap\( (\text{row, col}) \) (* \( M \) is symmetric: ensure that row \( \geq \) col *)
\( M[\text{row, col}] \leftarrow \text{cost} \)
if (\( B[\text{row}] \neq -1 \) and \( \text{cost} < M[\text{row, B[\text{row}]}] \)) then
(* New value is smaller than the current row minimum *)
\( B[\text{row}] \leftarrow \text{col} \)
else if (\( B[\text{row}] = \text{col} \) and \( \text{cost} > M[\text{row, B[\text{row}]}] \)) then
(* Current row minimum has been updated to a greater value *)
\( B[\text{row}] \leftarrow -1 \) (* ”-1” stands for ”unknown” *)
end if
end;

continued on the next page...
begin
Solve the Assignment Problem and obtain cycle factor $F = \{C_1, \ldots, C_m\}$.

for $u \leftarrow 1$ to $n$ do (* vertices are numbered from 1 to $n$ *)
  for $v \leftarrow 1$ to $u$ do
    $M[u,v] \leftarrow patching\_cost(F, u, v)$
  end for
  $B[u] \leftarrow \text{argmin}(M[u,v] | 1 \leq v < u)$
end for

while $F$ consists of more than one cycle do
  $best_u \leftarrow \text{argmin}(B[u] | 1 \leq u \leq n)$
  $best_v \leftarrow B[best_u]$
  (* ($best_u, best_v$) is the cheapest patching *)
  for all $u$ which are in the same cycle as $best_u$ do
    for all $v$ which are in the same cycle as $best_v$ do
      update\_cost($u, v, +\infty$)
    end do
  end do

  end do

  for all $w$ which do not belong to the same cycle as $best_u$ or $best_v$ do
    update\_cost($best_u, w, patching\_cost(F, best_u, w)$)
    update\_cost($w, best_v, patching\_cost(F, w, best_v)$)
  end do

  patch($F, best_u, best_v$)
  (* Recompute parts of $B$ if necessary *)

  for $u \leftarrow 1$ to $n$ do
    if $B[u] = -1$ then (* $B[u]$ needs updating *)
      $B[u] \leftarrow \text{argmin}(M[u,v] | 1 \leq v < u)$
    end if
  end for

  end while

$T \leftarrow F$ (* $F$ is a Hamiltonian cycle *)
return $T$
end
quite well in these circumstances. In our experience, on instances where the Assignment Problem solution consists of a large number of short cycles, RPC performs significantly better than the three well-known heuristics tested in this study.

In this subsection we describe a new tour construction heuristic, called Contract-or-Patch (or COP for short). The idea of this algorithm is to combine the desirable properties of both Karp-Steele Patching (or Extended Karp-Steele Patching) and Recursive Path Contraction. As we shall see, in addition to being a robust construction heuristic, the combined method frequently outperforms not just one of the algorithms it is based on, but both of them at the same time.

The main idea of the algorithm is to apply an RPC-like procedure to the initial cycle factor in order to eliminate short cycles; cycle factors that consist of a large number of short cycles are known to present difficulties to the patching heuristics, KSP and EKSP. We evaluate two variants of the technique for eliminating short cycles. One is more generic in that it can be combined with any tour construction method. Another is less generic as it can only be combined with some construction algorithms, such as KSP and EKSP, but it may offer improved tour quality compared with the other variant.

Both methods have a parameter, \textit{threshold}, that governs which cycles are considered "short" and are subject to the elimination procedure.

The first variant of Contract-or-Patch is presented in Algorithm 1.6. In essence, the algorithm repeatedly contracts short cycles until none are left, and then applies another tour construction algorithm to the contracted problem. Expanding the contracted paths gives an approximate solution to the original problem. We combine this technique with Karp-Steele Patching and Extended Karp-Steele Patching heuristics, denoting the combined algorithms by COP/KSP and COP/EKSP respectively. A computational evaluation of these two algorithms is presented in Section 1.5.

The second variant of Contract-or-Patch is based on the notion of cycle patching algorithm. A \textit{cycle patching} algorithm accepts an arbitrary cycle factor for the given TSP instance; it then uses some method to eliminate subtours and transform the cycle factor into a valid TSP tour. Both KSP and EKSP can be easily modified into cycle patching algorithms by removing
Algorithm 1.6 Contract-or-Patch, variant 1 (COP/xxx)

Step 1. Fix threshold \( t \).

Step 2. Find a minimum cost cycle factor \( \mathcal{F} \) by solving the Assignment Problem.

Step 3. If there is a cycle in \( \mathcal{F} \) of length less than \( t \) (a short cycle), delete the most expensive arc from every short cycle, contract the obtained paths (the vertices that belong to long cycles are left intact), and go to Step 2.

Step 4. Apply a tour construction algorithm to the contracted instance to obtain a "contracted" tour \( T \).

Step 5. Expand the arcs of \( T \) there were contracted during applications of Step 3. The result is an approximate solution to the original problem.

Algorithm 1.7 Contract-or-Patch, variant 2 (xxx/COP)

Step 1. Fix threshold \( t \).

Step 2. Find a minimum cost cycle factor \( \mathcal{F} \) by solving the Assignment Problem.

Step 3. If there is a cycle in \( \mathcal{F} \) of length less than \( t \) (a short cycle), delete the most expensive arc from every short cycle, contract the obtained paths (the vertices that belong to long cycles are left intact), and go to Step 2.

Step 4. Expand the arcs of \( \mathcal{F} \) there were contracted during applications of Step 3. This transforms \( \mathcal{F} \) into a cycle factor for the original graph.

Step 5. Apply a cycle patching algorithm (for example, KSP or EKSP) to transform \( \mathcal{F} \) into a tour \( T \). Use \( T \) as an approximate solution for the original problem.
their first step ("Solve the Assignment Problem"). Instead of obtaining a cycle factor on their own by solving the AP, the modified algorithms accept a cycle factor as input and transform it into a valid tour.

The second variant of Contract-or-Patch is presented in Algorithm 1.7. The procedure can be viewed as a preprocessing technique that can be combined with any cycle patching algorithm to produce a complete heuristic for the TSP. The algorithm first solves the Assignment Problem, and then applies an RPC-like method to try and eliminate short cycles from the cycle factor it obtained from the AP. The result is another cycle factor which contains no or few short cycles. This cycle factor is then passed to a cycle patching method (such as the variants of KSP or EKSP described above). The cycle patching method transforms the cycle factor into a valid tour for the original problem.

The procedure used by this variant of COP in order to eliminate short cycles is as follows. First of all, the most expensive arc is deleted from each short cycle in the cycle factor. Then, the set of resulting paths is contracted. The same procedure is applied recursively to the contracted TSP instance until either the cycle factor contains no short cycles, or an optimal solution to the current contracted instance has been obtained. When this happens, the algorithm transforms the cycle factor that it has for the current contracted problem into a cycle factor for the original problem. This is done by expanding the arcs that have been contracted during all previous steps of the algorithm. Note that the resulting cycle factor is guaranteed to contain no short cycles. As it has been noted previously, the algorithm can then apply a cycle patching method to transform the cycle factor into a tour for the original problem.

In this study we consider the combinations of this algorithm with Karp-Steele Patching and Extended Karp-Steele Patching. The corresponding complete construction heuristics are denoted by KSP/COP and EKSP/COP respectively (not to be confused with COP/KSP and COP/EKSP, which denote the first variant of COP).

Thus, there are four different COP-based methods considered in this study. Each of these methods can be fine-tuned by adjusting the threshold parameter. In Subsection 1.5.5 we perform a comparative study of the four COP-based construction heuristics. There we also study the impact of the
value of the threshold parameter on the quality of tours produced by all four algorithms.

1.4 Theoretical evaluation of heuristics

The primary focus of this chapter is to provide a computation study of construction heuristics and develop new algorithms that perform well in practice. In this section, we will discuss approaches to evaluating heuristics theoretically. Two such approaches are discussed in Subsections 1.4.1 and 1.4.2.

1.4.1 Approximation analysis

Approximation analysis is a traditional technique for studying quality of non-exact algorithms theoretically.

It is said that an approximation algorithm has a ratio bound $\rho(n)$ if for any instance of size $n$ the algorithm produces a solution of cost at most $\rho(n) \cdot c^*$ where $c^*$ is the cost of the optimal solution. An algorithm with ratio bound $\rho(n)$ is often called $\rho(n)$-approximation algorithm.

While approximation algorithms with constant ratio bounds are known for the Euclidean TSP (and for the general Symmetric TSP with cost function satisfying the triangle inequality), the situation with Asymmetric TSP is much worse in this respect.

For nearly 20 years, the best approximation known for the Euclidean TSP (due to Christofides [19]) achieved an approximation factor of $3/2$. In 1996, it has been shown by Arora in [5] that Euclidean TSP in fixed number of dimensions admits a Polynomial-Time Approximation Scheme (PTAS); that is, for any constant $\epsilon > 0$ it is possible to build a polynomial-time $(1 + \epsilon)$-approximation method. This is, however, not true if the number of dimensions is $\log n$ rather than constant, as shown in [71].

The best currently known approximation algorithm for the Asymmetric TSP (obeying the triangle inequality) is the Repeated Assignment algorithm of [27]. The algorithm has a proven ratio bound of $O(\log n)$ (that is, solutions produced by the algorithm are guaranteed to be at most $O(\log n)$ times more expensive than the optimum). This guarantee is clearly not as impressive as the approximation schemes that are known for the Euclidean TSP.
1.4.2 Domination analysis

Domination analysis is a relatively new approach to studying heuristics theoretically. It was first considered by Glover and Punnen in [32]. In domination analysis, the quality of a solution is assessed by finding out how many of the finite set of feasible solutions have the same, or higher, cost. To achieve this, the notion of domination number is introduced.

Recall that the domination number of tour $T$ in $K_n$ is the number of tours in $K_n$ which are more expensive or have the same cost as $T$ (including $T$ itself). The domination number $\text{domn}(H, n)$ of heuristic $H$ for the ATSP is the maximum integer $k = k(n)$, such that for every ATSP instance on $n$ vertices, $H$ produces a tour whose domination number is at least $k$. Observe that $\text{domn}(A, n) = (n - 1)!$ for any exact ATSP algorithm $A$.

In the rest of this subsection we summarise known results on the domination number of heuristics considered in this study. We also include some new results.

**Greedy Algorithm**

The following theorem shows that the Greedy algorithm for the TSP has the domination number of one (that is, when applied to some instances, the algorithm finds the unique most-expensive tour).

**Theorem 1.1.** The domination number of GR for the TSP is 1.

**Proof:** This proof holds for both ATSP and STSP, but for simplicity we assume that we deal with the STSP. We will consider tours of STSP as sets of their edges. For a set $S = \{e_1, \ldots, e_s\}$ of edges forming a partial tour in $K_n$ (that is, this set of edges can be extended to a tour), $Z(e_1, \ldots, e_s)$ denotes the set of edges not in $S$ such that each edge from $Z(e_1, \ldots, e_s)$ can be added to $S$ to form a (larger) partial tour.

Let $T' = \{e'_1, e'_2, \ldots, e'_n\}$ be an arbitrary fixed tour and let $T$ be an arbitrary tour distinct from $T'$. It is easy to see that

$$
\sum_{j=0}^{n-1} |Z(e'_1, e'_2, \ldots, e'_j) \cap T| < n(n + 1)/2. \quad (1.2)
$$
Let \( M \gg n \), let \( c(e_i') = iM \) for each \( e_i' \in T' \) and, for \( e \notin T' \), let \( c(e) = 1 + jM \) if \( e \in Z(e_1', e_2', \ldots, e_{j-1}') \) but \( e \notin Z(e_1', e_2', \ldots, e_j') \). Clearly, GR constructs \( T' \) and \( c(T') = M(n + 1)/2. \)

Let \( T = \{e_1, e_2, \ldots, e_k\} \). Assume that \( c(e_i) \in \{aM, aM + 1\} \). Then clearly
\[
e_i \in Z(e_1', e_2', \ldots, e_{a-1}'),
\]
but \( e_i \notin Z(e_1', e_2', \ldots, e_a') \), so \( e_i \) lies in \( Z(e_1', e_2', \ldots, e_j') \cap T \), provided \( j \leq a - 1 \). Thus, \( e_i \) is counted \( a \) times in the sum in (1.2). Hence,
\[
c(T) = \sum_{i=1}^{n} c(e_i) \leq n + M \sum_{j=0}^{n-1} |Z(\{e_1', e_2', \ldots, e_j\}) \cap T| \\
\leq n + M(n(n + 1)/2 - 1) = n - M + c(T'),
\]
which is less than the cost of \( T' \) as \( M > n \). Since GR finds \( T' \), and \( T \) is arbitrary, we see that GR finds the unique most expensive tour. \( \square \)

**Near Neighbour**

The Nearest Neighbour heuristic (or NN for short) starts from a pre-determined vertex. The algorithm then visits all other vertices in the graph by repeatedly moving to the nearest yet-unvisited vertex. Once all vertices have been visited, the resulting Hamilton path is closed to form a tour.

**Theorem 1.2.** The domination number of the Nearest Neighbour algorithm for the TSP is 1.

**Proof:** The proof of this theorem is the same as the proof of Theorem 1.1. Observe that, when applied to the graph constructed in the proof of Theorem 1.1, both GR and NN find the same tour (without loss of generality we assume that NN starts from vertex 1). \( \square \)

**Repetitive Nearest Neighbour**

Repetitive Nearest Neighbour (or RNN for short) is a variant of the Nearest Neighbour algorithm which starts NN from each vertex in turn, and then chooses the best of the \( n \) tours.
**Theorem 1.3.** Let \( n \geq 4 \). The domination number of RNN for the ATSP is at least \( n/2 \) and at most \( n - 1 \).

Proof of this theorem can be found in [38].

**Theorem 1.4.** Let \( n \geq 4 \). The domination number of RNN for the STSP is at least \( n/4 \) and at most \( 2^{n-3} \).

**Proof:** An instance of the STSP can be transformed into an instance of the ATSP by replacing every undirected edge \( xy \) of \( K_n \) by a pair \( xy, yx \) of directed arcs of the same cost as edge \( xy \). Observe that a heuristic \( H \) for the ATSP with domination number \( d(n) \) has domination number at least \( d(n)/2 \) for the STSP. The factor \( 1/2 \) is due to the fact that STSP tours are non-directional: two tours \( T \) and \( \overrightarrow{T} \) that traverse vertices of \( \overrightarrow{K_n} \) in opposite directions are indistinguishable in \( K_n \). It follows from this observation and the lower bound in Theorem 1.3 that the domination number of RNN for the STSP is at least \( n/4 \).

A proof of the upper bound can be found in [38].

**Recursive Path Contraction**

The Recursive Path Contraction algorithm for the ATSP originates from [74]. It is claimed in the paper that the algorithm has the domination number of at least

\[
\frac{c \times n!}{4^n + \frac{\log(n)n^2}{4}},
\]

where \( c \) is some constant.

**1.5 Computational study**

We have implemented the three well-known heuristics discussed in Subsections 1.3.1-1.3.3: Greedy, Random Insertion and Karp-Steele Patching. We have also implemented Recursive Path Contraction discussed in Subsection 1.3.4. Two different implementation of the Extended Karp-Steele Patching algorithm (Subsection 1.3.5) have been tested; the difference between the two implementations and relative performance are studied in Subsection 1.5.4. Finally, all four variants of the Contract-or-Patch heuristic of Subsection 1.3.6 have been implemented.

In this section we study computational performance of this selection of algorithms on a wide variety of instances of the ATSP comprising seven different families of instances.
1.5.1 Computing environment

We implemented all algorithms in the C++ programming language [70]. All computational tests presented in this section were conducted on an Intel Celeron 550MHz computer with 384MB of RAM. All times reported in this section are elapsed CPU times.

TODO: Reference and link to the AP code that we’ve used.

1.5.2 Test instances

We have tested the algorithms considered in this study on a set of seven families of instances of the Travelling Salesman Problem. Families include both real-world and randomly generated instances with various properties. Two families of real-world instances (Euclidean and asymmetric) are taken from TSPLIB of [65] - a library of benchmark instances of the TSP collected from various sources. TSPLIB instances are available for download from http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. The five randomly-generated families of instances include both symmetric and asymmetric instances, as well as a family of asymmetric yet close to Euclidean instances (so-called ”Sloped Plane” instances).

The seven families of instances used in this study are as follows:

1. TSPLIB(A)
   This family includes all asymmetric TSP instances from TSPLIB;

2. TSPLIB(E)
   This family includes all Euclidean TSP instances from TSPLIB with the number of vertices not exceeding 3000;

3. Random Asymmetric \((10^5)\)
   This family consists of asymmetric TSP instances with cost matrix \(C = [c(i,j)]\), where \(c(i,j)\) are independently and uniformly chosen random numbers from \(\{0, 1, 2, \ldots , 10^5\}\);

4. Random Asymmetric \((i \times j)\)
   This family consists of asymmetric TSP instances with cost matrix \(C = [c(i,j)]\), where \(c(i,j)\) are independently and uniformly chosen random numbers from \(\{0, 1, 2, \ldots , i \times j\}\);
5. Random Symmetric \((10^5)\)
This family consists of symmetric TSP instances with cost matrix \(C = [c(i, j)]\), where \(c(i, j)\) are independently and uniformly chosen random numbers from \(\{0, 1, 2, \ldots, 10^5\}\) \((i < j)\);

6. Random Symmetric \((i \times j)\)
This family consists of symmetric TSP instances with cost matrix \(C = [c(i, j)]\), where \(c(i, j)\) are independently and uniformly chosen random numbers from \(\{0, 1, 2, \ldots, i \times j\}\) \((i < j)\);

7. Sloped Plane
Sloped Plane instances originate from [48]. They are defined as follows.
For a given pair of vertices \(p_i\) and \(p_j\), defined by their planar coordinates \(p_i = (x_i, y_i)\) and \(p_j = (x_j, y_j)\), the cost of the corresponding arc is
\[
c(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - \max(0, y_i - y_j) + 2 \times \max(0, y_j - y_i).
\]
We have tested the algorithms on Sloped Plane instances with independently and uniformly chosen random coordinates from \(\{0, 1, 2, \ldots, 10^5\}\).

For randomly-generated instances (families 3-7), the number of vertices \(n\) was varied from 100 to 3000. We generated instance of the following sizes: 100, 500, 1000, 1500, 2000, 2500 and 3000. For 100 \(\leq n \leq 1000\), the results were averaged over 10 randomly-generated instances of size \(n\). For 1000 \(< n \leq 3000\), the results are average over three instances of size \(n\).

The number and variety of the families used allows us to check robustness of tested algorithms. We use instances produced in both random and deterministic manner. It should be noted that in study we exclusively consider tour construction heuristics, which due to their nature cannot be expected to consistently produce near-optimal solutions.

1.5.3 Lower bounds
In order to assess the quality of tours produced by a heuristic, whenever possible we compare the cost of the tour found by the heuristic with the cost of the optimal tour. Since optimal tour costs are only known for TSPLIB
For families 3, 4 and 7 we use the Assignment Problem (AP) lower bound, the cost of the cheapest cycle factor. The AP lower bound is known to be of high quality for the pure asymmetric TSP [9].

For the symmetric families 5 and 6, we exploited the Held-Karp (HK) lower bound [43, 44], which is known to be very effective for this type of TSP instances [49].

### 1.5.4 Implementing Extended Karp-Steele Patching

In this subsection we study relative computational performance of two implementations of the Extended Karp-Steele Patching algorithm. We call one implementation "optimised" and the other "simple". The algorithm itself and the optimised implementation were presented in Subsection 1.3.5 on page 20. The simple implementation is described below.

The simple implementation works in perhaps the most natural and straightforward fashion. At each stage, it computes the cost of all available patching operations and then performs the cheapest patching, repeating until a Hamiltonian cycle is obtained. It makes no attempt to speed up its operations by introducing any additional data structures beyond what is absolutely necessary.

The main advantages of the of the simple implementation are its compactness, and therefore ease of implementation and maintenance. Our implementation required approximately 70 lines of relatively sparse C++ code, which even includes some basic error checking. This compares favourably with several hundred lines of code that were necessary for the optimised implementation.

There is no difference in quality of solution obtained by the two implementations: if care is taken to ensure that tie-breaking rules are the same, both implementations can be made to produce identical solutions.

In order to compare execution times of the two implementations, we have run both codes on all five families of randomly generated instances described in Subsection 1.5.2. For each of the families, we run both codes on instances of each of the following sizes: 100, 316, 1000 and 3162. The results were


<table>
<thead>
<tr>
<th></th>
<th>Optimised</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100 nodes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random asymmetric ($10^5$)</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>Random asymmetric ($i \times j$)</td>
<td>0.01s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Random symmetric ($10^5$)</td>
<td>0.02s</td>
<td>0.07s</td>
</tr>
<tr>
<td>Random symmetric ($i \times j$)</td>
<td>0.01s</td>
<td>0.07s</td>
</tr>
<tr>
<td>Sloped plane</td>
<td>0.01s</td>
<td>0.06s</td>
</tr>
<tr>
<td><strong>316 nodes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random asymmetric ($10^5$)</td>
<td>0.05s</td>
<td>0.06s</td>
</tr>
<tr>
<td>Random asymmetric ($i \times j$)</td>
<td>0.07s</td>
<td>0.08s</td>
</tr>
<tr>
<td>Random symmetric ($10^5$)</td>
<td>0.08s</td>
<td>2.72s</td>
</tr>
<tr>
<td>Random symmetric ($i \times j$)</td>
<td>0.10s</td>
<td>2.79s</td>
</tr>
<tr>
<td>Sloped plane</td>
<td>0.16s</td>
<td>3.42s</td>
</tr>
<tr>
<td><strong>1000 nodes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random asymmetric ($10^5$)</td>
<td>0.58s</td>
<td>0.73s</td>
</tr>
<tr>
<td>Random asymmetric ($i \times j$)</td>
<td>1.09s</td>
<td>1.55s</td>
</tr>
<tr>
<td>Random symmetric ($10^5$)</td>
<td>1.15s</td>
<td>120.21s</td>
</tr>
<tr>
<td>Random symmetric ($i \times j$)</td>
<td>1.75s</td>
<td>115.39s</td>
</tr>
<tr>
<td>Sloped plane</td>
<td>2.79s</td>
<td>142.03s</td>
</tr>
<tr>
<td><strong>3162 nodes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random asymmetric ($10^5$)</td>
<td>10.7s</td>
<td>17.0s</td>
</tr>
<tr>
<td>Random asymmetric ($i \times j$)</td>
<td>17.9s</td>
<td>21.4s</td>
</tr>
<tr>
<td>Random symmetric ($10^5$)</td>
<td>18.2s</td>
<td>—*</td>
</tr>
<tr>
<td>Random symmetric ($i \times j$)</td>
<td>26.7s</td>
<td>—*</td>
</tr>
<tr>
<td>Sloped plane</td>
<td>48.3s</td>
<td>—*</td>
</tr>
</tbody>
</table>

*“—*” entries indicate that the code has exceeded its time allocation of 10 minutes, and has been terminated

Table 1.1: Running times of two implementations of Extended Karp-Steele Patching
averaged over three trials and are presented in Table 1.1.

It can be seen from Table 1.1 that the optimised implementation is consistently faster, in some cases by as much as a factor of over 100. The simple implementation is faster only on two instances, both times marginally: the difference is approximately one hundredth of a second and may well be attributed to the inaccuracy of the timer used to measure these timings.

We were unable to obtain timings for the simple implementation on instances of size 3162 from three of the five families. In each of these cases the code has exceeded its allocation of 10 minutes of CPU time and has been terminated (on one occasion, we waited for about 40 minutes for the code to terminate, but even 40 minutes turned out to be insufficient in that instance). Huge execution times make the simple implementation of the algorithm impractical for large instances. On the other hand, the optimised implementation appears perfectly capable of dealing with larger instances, solving each of the 3162-node instances in under 50 seconds (most of them in about half of that time).

Therefore in the rest of this chapter we exclusively deal with the optimised implementation of the Extended Karp-Steele Patching heuristic.

### 1.5.5 Choice of Contract-or-Patch variant

In this subsection we study computational performance of four variants of the Contract-or-Patch heuristic (or COP for short) and the effect the choice of the algorithms’ parameter (threshold) has on their computational performance.

We have implemented all four variants of the COP algorithm: COP/KSP, COP/EKSP, KSP/COP and EKSP/COP discussed in Subsection 1.3.6. A summary of the four variants of COP is presented in Table 1.2. Each algorithm has been tested with the value of threshold varied from 3 to 15 (note that when the threshold value is less than 3, the algorithms effectively degen-
erate into the corresponding cycle patching algorithms — KSP or EKSP). All four algorithms have been tested on the seven families of TSP instances described in Subsection 1.5.2 on page 32.

For the randomly generated instances (families 3-7), we have performed a preliminary study of all algorithms on instances of sizes 500, 1000 and 2000, and have found the results to be quite similar for each of these sizes. Therefore we have decided to perform our detailed evaluation on instances of a fixed size. We have chosen 1000 as instance size, since it represents instances of moderate size and at the same time allows us to run a large number of trials in reasonable computational time. We varied the threshold value between 3 and 15, and for the families 3-7 the results presented below represent an average of 50 trials for every algorithm with each of the threshold values.

For the TSPLIB instances — families 1 and 2 — solutions obtained by each algorithm were compared to the optima. For the asymmetric instance families 3, 4 and 7 we used the Assignment Problem lower bound, the cost of the cheapest cycle factor. For the symmetric families 5 and 6, we exploited the Held-Karp lower bound [43, 44], which is known to be very effective for this type of TSP instances [49].

For the TSPLIB families 1 and 2, the tests were performed on all asymmetric instances in TSPLIB, and all Euclidean instance of not more than 3000 cities. Each algorithm was run once with every parameter value on each of the tested instances. The average excess over optimum was then calculated in each of the families, and the results were plotted on the top two graphs of Figure 1.1 on page 38. The overall time taken by every algorithm to solve all problems in each of the families 1 and 2 is presented on top two graphs of Figure 1.1.

For each of the families 3-7, fifty problem instances of size 1000 were generated randomly. All four algorithms were executed on each of these instances with every parameter value. The excess over the corresponding lower bound and the running time were then averaged over fifty trials in each family, and plotted in Figures 1.1 and 1.2 respectively.

The rest of this subsection is organised as follows. First, we compare the results produced by the heuristics based on the Extended Karp-Steele Patching algorithm to heuristics based on the original Karp-Steele Patching
Figure 1.1: COP variants: tour quality
### Figure 1.2: COP variants: execution time

#### TSPLIB: asymmetric

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.50</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>5</td>
<td>2.50</td>
</tr>
<tr>
<td>6</td>
<td>3.00</td>
</tr>
<tr>
<td>7</td>
<td>3.50</td>
</tr>
<tr>
<td>8</td>
<td>4.00</td>
</tr>
<tr>
<td>9</td>
<td>4.50</td>
</tr>
<tr>
<td>10</td>
<td>5.00</td>
</tr>
<tr>
<td>11</td>
<td>5.50</td>
</tr>
<tr>
<td>12</td>
<td>6.00</td>
</tr>
</tbody>
</table>

#### TSPLIB: Euclidean

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.50</td>
</tr>
<tr>
<td>5</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>2.50</td>
</tr>
<tr>
<td>7</td>
<td>3.00</td>
</tr>
<tr>
<td>8</td>
<td>3.50</td>
</tr>
<tr>
<td>9</td>
<td>4.00</td>
</tr>
<tr>
<td>10</td>
<td>4.50</td>
</tr>
<tr>
<td>11</td>
<td>5.00</td>
</tr>
<tr>
<td>12</td>
<td>5.50</td>
</tr>
<tr>
<td>13</td>
<td>6.00</td>
</tr>
</tbody>
</table>

#### Random Symmetric (10^5)

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.85</td>
</tr>
<tr>
<td>4</td>
<td>0.90</td>
</tr>
<tr>
<td>5</td>
<td>0.95</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
</tr>
<tr>
<td>7</td>
<td>1.05</td>
</tr>
<tr>
<td>8</td>
<td>1.10</td>
</tr>
<tr>
<td>9</td>
<td>1.15</td>
</tr>
<tr>
<td>10</td>
<td>1.20</td>
</tr>
<tr>
<td>11</td>
<td>1.25</td>
</tr>
<tr>
<td>12</td>
<td>1.30</td>
</tr>
<tr>
<td>13</td>
<td>1.35</td>
</tr>
</tbody>
</table>

#### Random Symmetric (i*j)

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.50</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>5</td>
<td>2.50</td>
</tr>
<tr>
<td>6</td>
<td>3.00</td>
</tr>
<tr>
<td>7</td>
<td>3.50</td>
</tr>
<tr>
<td>8</td>
<td>4.00</td>
</tr>
<tr>
<td>9</td>
<td>4.50</td>
</tr>
<tr>
<td>10</td>
<td>5.00</td>
</tr>
<tr>
<td>11</td>
<td>5.50</td>
</tr>
<tr>
<td>12</td>
<td>6.00</td>
</tr>
</tbody>
</table>

#### Random Asymmetric (10^5)

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.10</td>
</tr>
<tr>
<td>4</td>
<td>1.20</td>
</tr>
<tr>
<td>5</td>
<td>1.30</td>
</tr>
<tr>
<td>6</td>
<td>1.40</td>
</tr>
<tr>
<td>7</td>
<td>1.50</td>
</tr>
<tr>
<td>8</td>
<td>1.60</td>
</tr>
<tr>
<td>9</td>
<td>1.70</td>
</tr>
<tr>
<td>10</td>
<td>1.80</td>
</tr>
<tr>
<td>11</td>
<td>1.90</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

#### Random Asymmetric (i*j)

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.50</td>
</tr>
<tr>
<td>4</td>
<td>3.00</td>
</tr>
<tr>
<td>5</td>
<td>3.50</td>
</tr>
<tr>
<td>6</td>
<td>4.00</td>
</tr>
<tr>
<td>7</td>
<td>4.50</td>
</tr>
<tr>
<td>8</td>
<td>5.00</td>
</tr>
<tr>
<td>9</td>
<td>5.50</td>
</tr>
<tr>
<td>10</td>
<td>6.00</td>
</tr>
<tr>
<td>11</td>
<td>6.50</td>
</tr>
<tr>
<td>12</td>
<td>7.00</td>
</tr>
</tbody>
</table>

#### Sloped Plane

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6.00</td>
</tr>
<tr>
<td>4</td>
<td>5.50</td>
</tr>
<tr>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>6</td>
<td>4.50</td>
</tr>
<tr>
<td>7</td>
<td>4.00</td>
</tr>
<tr>
<td>8</td>
<td>3.50</td>
</tr>
<tr>
<td>9</td>
<td>3.00</td>
</tr>
<tr>
<td>10</td>
<td>2.50</td>
</tr>
<tr>
<td>11</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>1.50</td>
</tr>
<tr>
<td>13</td>
<td>1.00</td>
</tr>
<tr>
<td>14</td>
<td>0.50</td>
</tr>
<tr>
<td>15</td>
<td>0.00</td>
</tr>
</tbody>
</table>
routine. Then we compare the variants that apply patching to the original instance of the problem with those that apply patching to the contracted instance (see Subsection 1.3.6 for details). Finally, we examine the impact of the threshold value on the quality of tours and execution times of all four algorithms, and make some recommendations for choosing the value of the threshold.

As it can be seen in Figure 1.1, the quality of tours produced by the EKSP-based variants of COP is very close to the tour quality of their KSP-based counterparts. Figure 1.2 shows that EKSP-based algorithms are usually somewhat slower than those based on KSP. Although in some cases algorithms based on EKSP do offer a minor improvement in tour quality, the overall complexity of EKSP may not always justify its use. We suggest that in cases when small degradation of in the quality of the tours produced by a construction heuristic can be tolerated, algorithms based on KSP can be used in favour of their EKSP-based counterparts.

We now compare the COP variants that apply patching to the contracted instance (COP/KSP and COP/EKSP) with their counterparts that apply patching to the original instance (KSP/COP and EKSP/COP). The data presented in Figures 1.1 and 1.2 suggests that the latter variants generally perform slightly better, at the expense of increased execution time. Since the difference is small, in the detailed computational study later in the chapter we will exclusively concentrate on the variants of COP which apply patching to the original (uncontracted) instance.

Based on the above observations, and bearing in mind simplicity of the KSP-based algorithms, we recommend KSP/COP as the first choice of COP algorithm.

It can be seen in the Figures 1.1 and 1.2 that running times and quality of tours produced by the algorithms vary substantially depending on the value of the threshold. In most cases increase in running time leads to better tours found by the algorithms. Thus, the parameter provides means for controlling the trade-off between tour quality and running time. Unfortunately, the behaviour is not consistent across different families of instances: in some cases increasing the value of threshold leads to better tours (and increased running times), in other cases the behaviour is opposite. Note that when threshold is set to a value < 3, COP effectively degenerates into the cycle patching
algorithm on which it is based; when threshold is $\geq n$, COP degenerates into Recursive Path Contraction (see Subsection 1.3.4).

In most cases the quality of tours deteriorates, sometimes quickly, when the threshold is increased. The only cases when increasing the threshold clearly results in better tours are the random asymmetric families 3 and 4. For these families the tours produced even with small values of the threshold are already very close to the lower bound (within 2.2% and 1.3% of the lower bound). This suggests that if a fixed value of the parameter is to be used, a small value of the threshold will result in better all-round performance. We therefore recommend that the threshold value of three is used as a good universal choice.

A more elaborate implementation could use information on the number and lengths of the cycles in the initial cycle factor in order to determine the value of threshold to use. In our experience, instances on which the assignment problem generates a large number of short cycles tend to benefit from small threshold, and vice versa. In this study, we do not pursue this avenue of research further.

1.5.6 Computational results

We tested the algorithms on an Intel Celeron 550MHz computer with 384 megabytes of RAM. All results for TSPLIB instances are compared to optima. For the asymmetric instance families 3, 4 and 7 we used Assignment Problem lower bound, the cost of the cheapest cycle factor. The AP lower bound is known to be of high quality for the pure asymmetric TSP [9]. For the symmetric instance families 5 and 6, we exploited the Held-Karp lower bound [43, 44], which is known to be very effective for this type of TSP instances [49].

Family 1: TSPLIB Asymmetric

Tables 1.3 and 1.4 starting on page 43 show tour quality and running times of the algorithms on asymmetric TSPLIB instances, family 1. It can be seen from Table 1.3 that there is no clear winner in terms of tour quality. The algorithms can be divided into two classes, one performing substantially better than the other. KSP, EKSP and COP perform rather well (with EKSP
being perhaps slightly ahead of the other two). The three remaining algorithms (GR, RI and RPC) perform substantially worse than the leaders. In terms of running time, all algorithms except one require little computational time. The exception is GR whose execution time explodes as the instance size increases. Among the remaining algorithms RI is the fastest (due to its $\Theta(n^2)$ time complexity).

Family 2: TSPLIB Euclidean

Results for the second family, Euclidean TSPLIB instances, are presented in Tables 1.5 and 1.6 on pages 45–48. In order to reduce the amount of data presented in the tables, only results for instances with at least 200 nodes are shown; smaller instances are omitted from Tables 1.5 and 1.6. The best performing algorithm for this family is Random Insertion with 11.2% average excess over optimum. The worst performer is RPC with 36.8% average excess. The remaining four algorithms perform fairly close to one another, all producing results which are on average within approximately 15%-18% over the optimum. When running times are considered, the situation is fairly similar to the previous family, Asymmetric TSPLIB instances. RI is the fastest algorithm and GR is by far the slowest (overall at least one order of magnitude slower than the next slowest algorithm). One interesting irregularity is the relatively high amount of time taken by KSP/COP to produce the tour for instance u2319. This appears to be related to the structure of the instance. After the short cycles have been broken and the resulting paths contracted, the optimal cycle factor in the contracted instance again contains one or more short cycles; the process is repeated a large number of times (nearly 20). This results in large execution times. Possible ways to address this problem include introducing a limit on the number of levels of recursion, or allowing a certain small number of short cycles to remain in the cycle factor instead of insisting that all are eliminated.

Families 3-7

Figures 1.3–1.7 on pages 49–56 show quality of solutions and running time of each of the heuristics on families 3-7. Quality of tours produced by the tested algorithms varies widely: from under 1% above the lower bound to
Table 1.3: Asymmetric TSPLIB instances: Excess over optimum

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (%)</th>
<th>RI (%)</th>
<th>KSP (%)</th>
<th>EKSP (%)</th>
<th>RPC (%)</th>
<th>KSP/COP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>br17</td>
<td>17</td>
<td>148.72</td>
<td>5.13</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>12.82</td>
</tr>
<tr>
<td>p43</td>
<td>43</td>
<td>0.94</td>
<td>0.52</td>
<td>0.11</td>
<td>0.32</td>
<td>0.66</td>
<td>0.34</td>
</tr>
<tr>
<td>ry48p</td>
<td>48</td>
<td>32.55</td>
<td>6.98</td>
<td>7.23</td>
<td>4.52</td>
<td>29.50</td>
<td>11.79</td>
</tr>
<tr>
<td>ft53</td>
<td>53</td>
<td>80.84</td>
<td>23.04</td>
<td>12.99</td>
<td>12.31</td>
<td>18.64</td>
<td>13.05</td>
</tr>
<tr>
<td>ft70</td>
<td>70</td>
<td>14.84</td>
<td>9.30</td>
<td>1.88</td>
<td>2.84</td>
<td>5.89</td>
<td>1.47</td>
</tr>
<tr>
<td>ftv35</td>
<td>36</td>
<td>30.75</td>
<td>12.15</td>
<td>1.56</td>
<td>1.09</td>
<td>21.18</td>
<td>1.36</td>
</tr>
<tr>
<td>ftv38</td>
<td>39</td>
<td>31.44</td>
<td>10.13</td>
<td>1.50</td>
<td>1.05</td>
<td>25.69</td>
<td>1.37</td>
</tr>
<tr>
<td>ftv44</td>
<td>45</td>
<td>25.05</td>
<td>16.49</td>
<td>7.69</td>
<td>5.33</td>
<td>22.26</td>
<td>4.90</td>
</tr>
<tr>
<td>ftv47</td>
<td>48</td>
<td>25.45</td>
<td>16.44</td>
<td>3.04</td>
<td>1.69</td>
<td>28.72</td>
<td>4.00</td>
</tr>
<tr>
<td>ftv55</td>
<td>56</td>
<td>39.74</td>
<td>11.94</td>
<td>3.05</td>
<td>3.05</td>
<td>33.27</td>
<td>3.73</td>
</tr>
<tr>
<td>ftv64</td>
<td>65</td>
<td>25.45</td>
<td>19.90</td>
<td>3.61</td>
<td>2.61</td>
<td>29.09</td>
<td>3.97</td>
</tr>
<tr>
<td>ftv70</td>
<td>71</td>
<td>29.85</td>
<td>15.79</td>
<td>3.33</td>
<td>2.87</td>
<td>22.77</td>
<td>0.92</td>
</tr>
<tr>
<td>ftv100</td>
<td>101</td>
<td>62.08</td>
<td>9.51</td>
<td>3.52</td>
<td>5.31</td>
<td>29.75</td>
<td>6.88</td>
</tr>
<tr>
<td>ftv110</td>
<td>111</td>
<td>42.44</td>
<td>22.37</td>
<td>5.41</td>
<td>5.67</td>
<td>21.35</td>
<td>5.72</td>
</tr>
<tr>
<td>ftv120</td>
<td>121</td>
<td>41.23</td>
<td>9.74</td>
<td>7.62</td>
<td>5.12</td>
<td>19.25</td>
<td>5.72</td>
</tr>
<tr>
<td>ftv130</td>
<td>131</td>
<td>54.36</td>
<td>7.67</td>
<td>5.46</td>
<td>4.90</td>
<td>17.34</td>
<td>4.85</td>
</tr>
<tr>
<td>ftv140</td>
<td>141</td>
<td>37.52</td>
<td>21.78</td>
<td>4.46</td>
<td>4.67</td>
<td>23.84</td>
<td>4.75</td>
</tr>
<tr>
<td>ftv150</td>
<td>151</td>
<td>39.87</td>
<td>22.71</td>
<td>4.75</td>
<td>4.33</td>
<td>22.56</td>
<td>4.75</td>
</tr>
<tr>
<td>ftv160</td>
<td>161</td>
<td>41.82</td>
<td>25.34</td>
<td>1.71</td>
<td>1.49</td>
<td>22.77</td>
<td>2.91</td>
</tr>
<tr>
<td>ftv170</td>
<td>171</td>
<td>42.11</td>
<td>21.45</td>
<td>2.40</td>
<td>1.38</td>
<td>25.66</td>
<td>1.71</td>
</tr>
<tr>
<td>kro124p</td>
<td>100</td>
<td>21.01</td>
<td>11.80</td>
<td>16.95</td>
<td>8.69</td>
<td>23.06</td>
<td>7.83</td>
</tr>
<tr>
<td>rbg323</td>
<td>323</td>
<td>7.39</td>
<td>30.32</td>
<td>0.00</td>
<td>0.00</td>
<td>0.53</td>
<td>0.00</td>
</tr>
<tr>
<td>rbg358</td>
<td>358</td>
<td>8.08</td>
<td>40.33</td>
<td>0.00</td>
<td>0.00</td>
<td>2.32</td>
<td>0.00</td>
</tr>
<tr>
<td>rbg403</td>
<td>403</td>
<td>0.93</td>
<td>8.36</td>
<td>0.00</td>
<td>0.00</td>
<td>0.69</td>
<td>0.20</td>
</tr>
<tr>
<td>rbg443</td>
<td>443</td>
<td>0.74</td>
<td>9.82</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

KSP/COP is tested with the threshold parameter set to 3;
Items in **bold** indicate the best solution found for the given problem.
Table 1.4: Asymmetric TSPLIB instances: Running time

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (s)</th>
<th>RI (s)</th>
<th>KSP (s)</th>
<th>EKSP (s)</th>
<th>RPC (s)</th>
<th>KSP/COP (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>br17</td>
<td>17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>p43</td>
<td>43</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ry48p</td>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>ft53</td>
<td>53</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ft70</td>
<td>70</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv33</td>
<td>34</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv35</td>
<td>36</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv38</td>
<td>39</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv44</td>
<td>45</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv47</td>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv55</td>
<td>56</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv64</td>
<td>65</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv70</td>
<td>71</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ftv100</td>
<td>101</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv110</td>
<td>111</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv120</td>
<td>121</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv130</td>
<td>131</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv140</td>
<td>141</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv150</td>
<td>151</td>
<td>0.09</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv160</td>
<td>161</td>
<td>0.13</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>ftv170</td>
<td>171</td>
<td>0.16</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>kro124p</td>
<td>100</td>
<td>0.02</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>rbg323</td>
<td>323</td>
<td>16.69</td>
<td>0.02</td>
<td>0.08</td>
<td>0.08</td>
<td>0.06</td>
<td>0.08</td>
</tr>
<tr>
<td>rbg358</td>
<td>358</td>
<td>27.83</td>
<td>0.02</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
<td>0.39</td>
</tr>
<tr>
<td>rbg403</td>
<td>403</td>
<td>26.14</td>
<td>0.02</td>
<td>0.09</td>
<td>0.09</td>
<td>0.08</td>
<td>0.75</td>
</tr>
<tr>
<td>rbg443</td>
<td>443</td>
<td>70.08</td>
<td>0.02</td>
<td>0.13</td>
<td>0.14</td>
<td>0.11</td>
<td>0.39</td>
</tr>
</tbody>
</table>

KSP/COP is tested with the threshold parameter set to 3;
Items in bold indicate the smallest running time for the given problem.
Table 1.5: Euclidean TSPLIB instances: Excess over optimum

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (%)</th>
<th>RI (%)</th>
<th>KSP (%)</th>
<th>EKSP (%)</th>
<th>RPC (%)</th>
<th>KSP/COP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a280</td>
<td>280</td>
<td>25.40</td>
<td>14.54</td>
<td>15.94</td>
<td>15.74</td>
<td>43.74</td>
<td>18.57</td>
</tr>
<tr>
<td>d493</td>
<td>493</td>
<td>17.70</td>
<td>10.76</td>
<td>17.33</td>
<td>16.16</td>
<td>32.14</td>
<td>16.46</td>
</tr>
<tr>
<td>d657</td>
<td>657</td>
<td>14.34</td>
<td>11.40</td>
<td>18.78</td>
<td>17.94</td>
<td>35.60</td>
<td>19.53</td>
</tr>
<tr>
<td>d1291</td>
<td>1291</td>
<td>23.41</td>
<td>20.63</td>
<td>19.38</td>
<td><strong>17.94</strong></td>
<td>42.52</td>
<td>29.30</td>
</tr>
<tr>
<td>d1655</td>
<td>1655</td>
<td>13.84</td>
<td>16.62</td>
<td>20.58</td>
<td>18.58</td>
<td>38.52</td>
<td>20.62</td>
</tr>
<tr>
<td>d2103</td>
<td>2103</td>
<td>13.52</td>
<td>22.73</td>
<td>12.55</td>
<td><strong>6.12</strong></td>
<td>26.52</td>
<td>10.01</td>
</tr>
<tr>
<td>fl417</td>
<td>417</td>
<td>11.91</td>
<td>7.72</td>
<td>7.82</td>
<td>16.19</td>
<td>50.77</td>
<td>22.06</td>
</tr>
<tr>
<td>fl1400</td>
<td>1400</td>
<td>17.14</td>
<td>5.89</td>
<td>9.81</td>
<td>14.40</td>
<td>32.73</td>
<td>12.47</td>
</tr>
<tr>
<td>fl1577</td>
<td>1577</td>
<td>18.60</td>
<td>21.12</td>
<td>19.00</td>
<td>26.41</td>
<td>34.69</td>
<td>21.01</td>
</tr>
<tr>
<td>gil262</td>
<td>262</td>
<td>16.32</td>
<td>11.94</td>
<td>14.80</td>
<td>18.76</td>
<td>39.70</td>
<td>18.88</td>
</tr>
<tr>
<td>kroa200</td>
<td>200</td>
<td>17.82</td>
<td>11.77</td>
<td><strong>11.67</strong></td>
<td>21.53</td>
<td>38.32</td>
<td>22.68</td>
</tr>
<tr>
<td>krob200</td>
<td>200</td>
<td>22.21</td>
<td>10.55</td>
<td><strong>9.09</strong></td>
<td>17.79</td>
<td>49.15</td>
<td>16.78</td>
</tr>
<tr>
<td>lin318</td>
<td>318</td>
<td>18.72</td>
<td>9.72</td>
<td>17.25</td>
<td>13.53</td>
<td>44.96</td>
<td>20.57</td>
</tr>
<tr>
<td>linhp318</td>
<td>318</td>
<td>20.69</td>
<td>9.89</td>
<td>19.19</td>
<td>15.40</td>
<td>47.36</td>
<td>22.56</td>
</tr>
<tr>
<td>nrw1379</td>
<td>1379</td>
<td>16.72</td>
<td>12.40</td>
<td>16.91</td>
<td>14.36</td>
<td>35.45</td>
<td>16.61</td>
</tr>
<tr>
<td>p654</td>
<td>654</td>
<td>14.18</td>
<td>6.62</td>
<td>11.34</td>
<td>15.86</td>
<td>48.54</td>
<td>18.42</td>
</tr>
<tr>
<td>pcb442</td>
<td>442</td>
<td>20.73</td>
<td>13.95</td>
<td>19.87</td>
<td>16.03</td>
<td>42.79</td>
<td>18.68</td>
</tr>
<tr>
<td>pcb1173</td>
<td>1173</td>
<td>20.09</td>
<td>16.54</td>
<td>21.08</td>
<td>20.47</td>
<td>39.70</td>
<td>23.32</td>
</tr>
<tr>
<td>pr226</td>
<td>226</td>
<td>20.76</td>
<td>11.11</td>
<td><strong>10.87</strong></td>
<td>17.29</td>
<td>22.27</td>
<td>14.12</td>
</tr>
<tr>
<td>pr264</td>
<td>264</td>
<td>13.70</td>
<td><strong>10.28</strong></td>
<td>11.56</td>
<td>18.99</td>
<td>33.31</td>
<td>16.86</td>
</tr>
</tbody>
</table>

*continued on the next page...*
Table 1.5  ...continued from the previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (%)</th>
<th>RI (%)</th>
<th>KSP (%)</th>
<th>EKSP (%)</th>
<th>RPC (%)</th>
<th>KSP/COP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr299</td>
<td>299</td>
<td>20.51</td>
<td>10.18</td>
<td>14.27</td>
<td>21.41</td>
<td>42.06</td>
<td>18.95</td>
</tr>
<tr>
<td>pr439</td>
<td>439</td>
<td>20.08</td>
<td>14.38</td>
<td>24.48</td>
<td>21.55</td>
<td>35.91</td>
<td>19.59</td>
</tr>
<tr>
<td>pr1002</td>
<td>1002</td>
<td>18.78</td>
<td>12.83</td>
<td>16.90</td>
<td>17.82</td>
<td>31.34</td>
<td>16.73</td>
</tr>
<tr>
<td>pr2392</td>
<td>2392</td>
<td>18.43</td>
<td>16.37</td>
<td>20.08</td>
<td>22.26</td>
<td>27.76</td>
<td>2.18</td>
</tr>
<tr>
<td>rat575</td>
<td>575</td>
<td>19.95</td>
<td>11.25</td>
<td>16.20</td>
<td>16.79</td>
<td>38.80</td>
<td>19.47</td>
</tr>
<tr>
<td>rat783</td>
<td>783</td>
<td>14.27</td>
<td>13.34</td>
<td>20.89</td>
<td>16.49</td>
<td>37.80</td>
<td>21.07</td>
</tr>
<tr>
<td>rd400</td>
<td>400</td>
<td>15.01</td>
<td>11.14</td>
<td>14.45</td>
<td>15.55</td>
<td>33.99</td>
<td>19.55</td>
</tr>
<tr>
<td>rl1304</td>
<td>1304</td>
<td>19.22</td>
<td>21.47</td>
<td>23.16</td>
<td>31.30</td>
<td>50.06</td>
<td>33.26</td>
</tr>
<tr>
<td>rl1323</td>
<td>1323</td>
<td>15.98</td>
<td>20.25</td>
<td>21.30</td>
<td>30.29</td>
<td>50.65</td>
<td>35.12</td>
</tr>
<tr>
<td>rl1889</td>
<td>1889</td>
<td>18.63</td>
<td>19.02</td>
<td>21.12</td>
<td>27.82</td>
<td>52.25</td>
<td>27.46</td>
</tr>
<tr>
<td>tsp225</td>
<td>225</td>
<td>22.71</td>
<td>12.15</td>
<td>15.51</td>
<td>13.65</td>
<td>29.37</td>
<td>9.19</td>
</tr>
<tr>
<td>u574</td>
<td>574</td>
<td>26.91</td>
<td>13.21</td>
<td>16.82</td>
<td>20.05</td>
<td>36.90</td>
<td>15.84</td>
</tr>
<tr>
<td>u724</td>
<td>724</td>
<td>19.91</td>
<td>11.46</td>
<td>15.02</td>
<td>18.31</td>
<td>43.70</td>
<td>18.84</td>
</tr>
<tr>
<td>u1060</td>
<td>1060</td>
<td>18.19</td>
<td>9.54</td>
<td>18.45</td>
<td>20.30</td>
<td>36.42</td>
<td>19.34</td>
</tr>
<tr>
<td>u1432</td>
<td>1432</td>
<td>23.42</td>
<td>12.97</td>
<td>13.48</td>
<td>10.15</td>
<td>31.37</td>
<td>12.63</td>
</tr>
<tr>
<td>u1817</td>
<td>1817</td>
<td>14.84</td>
<td>19.48</td>
<td>22.91</td>
<td>15.19</td>
<td>38.15</td>
<td>17.74</td>
</tr>
<tr>
<td>u2152</td>
<td>2152</td>
<td>16.79</td>
<td>19.76</td>
<td>22.71</td>
<td>15.82</td>
<td>36.69</td>
<td>21.78</td>
</tr>
<tr>
<td>u2319</td>
<td>2319</td>
<td>14.51</td>
<td>7.74</td>
<td>4.99</td>
<td>3.81</td>
<td>22.63</td>
<td>6.07</td>
</tr>
<tr>
<td>vm1084</td>
<td>1084</td>
<td>23.05</td>
<td>10.41</td>
<td>18.56</td>
<td>22.53</td>
<td>46.68</td>
<td>23.50</td>
</tr>
<tr>
<td>vm1748</td>
<td>1748</td>
<td>19.37</td>
<td>11.33</td>
<td>18.02</td>
<td>21.50</td>
<td>44.80</td>
<td>20.48</td>
</tr>
</tbody>
</table>

KSP/COP is tested with the threshold parameter set to 3;  
Items in **bold** indicate the best solution found for the given problem.
Table 1.6: Euclidean TSPLIB instances: Running time

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (s)</th>
<th>RI (s)</th>
<th>KSP (s)</th>
<th>EKSP (s)</th>
<th>RPC (s)</th>
<th>KSP/COP (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a280</td>
<td>280</td>
<td>0.30</td>
<td>0.00</td>
<td>0.02</td>
<td>0.05</td>
<td>0.03</td>
<td>0.06</td>
</tr>
<tr>
<td>d493</td>
<td>493</td>
<td>0.67</td>
<td>0.03</td>
<td>0.09</td>
<td>0.17</td>
<td>0.08</td>
<td>0.24</td>
</tr>
<tr>
<td>d657</td>
<td>657</td>
<td>1.44</td>
<td>0.05</td>
<td>0.14</td>
<td>0.34</td>
<td>0.14</td>
<td>0.39</td>
</tr>
<tr>
<td>d1291</td>
<td>1291</td>
<td>13.42</td>
<td>0.22</td>
<td>0.55</td>
<td>1.42</td>
<td>0.59</td>
<td>1.86</td>
</tr>
<tr>
<td>d1655</td>
<td>1655</td>
<td>35.03</td>
<td>0.38</td>
<td>0.95</td>
<td>2.50</td>
<td>1.05</td>
<td>3.02</td>
</tr>
<tr>
<td>d2103</td>
<td>2103</td>
<td>77.16</td>
<td>0.69</td>
<td>1.64</td>
<td>3.66</td>
<td>1.33</td>
<td>4.56</td>
</tr>
<tr>
<td>fl417</td>
<td>417</td>
<td>0.50</td>
<td>0.02</td>
<td>0.05</td>
<td>0.13</td>
<td>0.06</td>
<td>0.09</td>
</tr>
<tr>
<td>fl1400</td>
<td>1400</td>
<td>38.58</td>
<td>0.27</td>
<td>0.74</td>
<td>1.63</td>
<td>0.59</td>
<td>5.13</td>
</tr>
<tr>
<td>fl1577</td>
<td>1577</td>
<td>40.00</td>
<td>0.34</td>
<td>0.81</td>
<td>2.19</td>
<td>0.80</td>
<td>2.74</td>
</tr>
<tr>
<td>gil262</td>
<td>262</td>
<td>0.28</td>
<td>0.00</td>
<td>0.03</td>
<td>0.05</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>kroa200</td>
<td>200</td>
<td>0.06</td>
<td>0.00</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>krob200</td>
<td>200</td>
<td>0.06</td>
<td>0.00</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>lin318</td>
<td>318</td>
<td>0.19</td>
<td>0.02</td>
<td>0.03</td>
<td>0.06</td>
<td>0.03</td>
<td>0.08</td>
</tr>
<tr>
<td>linhp318</td>
<td>318</td>
<td>0.19</td>
<td>0.02</td>
<td>0.03</td>
<td>0.06</td>
<td>0.03</td>
<td>0.06</td>
</tr>
<tr>
<td>nrw1379</td>
<td>1379</td>
<td>20.44</td>
<td>0.27</td>
<td>0.66</td>
<td>1.69</td>
<td>0.80</td>
<td>2.75</td>
</tr>
<tr>
<td>p654</td>
<td>654</td>
<td>1.77</td>
<td>0.05</td>
<td>0.16</td>
<td>0.28</td>
<td>0.14</td>
<td>0.34</td>
</tr>
<tr>
<td>pcb442</td>
<td>442</td>
<td>0.44</td>
<td>0.02</td>
<td>0.06</td>
<td>0.16</td>
<td>0.08</td>
<td>0.20</td>
</tr>
<tr>
<td>pcb1173</td>
<td>1173</td>
<td>10.19</td>
<td>0.17</td>
<td>0.47</td>
<td>1.13</td>
<td>0.55</td>
<td>1.48</td>
</tr>
<tr>
<td>pr226</td>
<td>226</td>
<td>0.09</td>
<td>0.00</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>pr264</td>
<td>264</td>
<td>0.13</td>
<td>0.00</td>
<td>0.03</td>
<td>0.05</td>
<td>0.03</td>
<td>0.05</td>
</tr>
</tbody>
</table>

*continued on the next page...*
Table 1.6

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GR (%)</th>
<th>RI (%)</th>
<th>KSP (%)</th>
<th>EKSP (%)</th>
<th>RPC (%)</th>
<th>KSP/COP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr299</td>
<td>299</td>
<td>0.16</td>
<td>0.02</td>
<td>0.03</td>
<td>0.06</td>
<td>0.03</td>
<td>0.08</td>
</tr>
<tr>
<td>pr439</td>
<td>439</td>
<td>0.38</td>
<td>0.02</td>
<td>0.06</td>
<td>0.13</td>
<td>0.06</td>
<td>0.19</td>
</tr>
<tr>
<td>pr1002</td>
<td>1002</td>
<td>2.81</td>
<td>0.13</td>
<td>0.31</td>
<td>0.78</td>
<td>0.33</td>
<td>0.73</td>
</tr>
<tr>
<td>pr2392</td>
<td>2392</td>
<td>38.81</td>
<td>0.92</td>
<td>1.94</td>
<td>4.84</td>
<td>1.70</td>
<td>4.95</td>
</tr>
<tr>
<td>rat575</td>
<td>575</td>
<td>2.84</td>
<td>0.03</td>
<td>0.11</td>
<td>0.25</td>
<td>0.13</td>
<td>0.31</td>
</tr>
<tr>
<td>rat783</td>
<td>783</td>
<td>8.36</td>
<td>0.08</td>
<td>0.20</td>
<td>0.49</td>
<td>0.24</td>
<td>0.75</td>
</tr>
<tr>
<td>rd100</td>
<td>100</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>rd400</td>
<td>400</td>
<td>0.49</td>
<td>0.02</td>
<td>0.05</td>
<td>0.13</td>
<td>0.05</td>
<td>0.11</td>
</tr>
<tr>
<td>rl1304</td>
<td>1304</td>
<td>5.80</td>
<td>0.24</td>
<td>0.53</td>
<td>1.53</td>
<td>0.55</td>
<td>1.55</td>
</tr>
<tr>
<td>rl1323</td>
<td>1323</td>
<td>6.06</td>
<td>0.25</td>
<td>0.56</td>
<td>1.61</td>
<td>0.55</td>
<td>1.36</td>
</tr>
<tr>
<td>rl1889</td>
<td>1889</td>
<td>16.61</td>
<td>0.56</td>
<td>1.22</td>
<td>3.41</td>
<td>1.17</td>
<td>3.72</td>
</tr>
<tr>
<td>ts225</td>
<td>225</td>
<td>0.16</td>
<td>0.00</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td>tsp225</td>
<td>225</td>
<td>0.13</td>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>u574</td>
<td>574</td>
<td>0.95</td>
<td>0.05</td>
<td>0.09</td>
<td>0.22</td>
<td>0.09</td>
<td>0.23</td>
</tr>
<tr>
<td>u724</td>
<td>724</td>
<td>2.03</td>
<td>0.06</td>
<td>0.17</td>
<td>0.39</td>
<td>0.16</td>
<td>0.39</td>
</tr>
<tr>
<td>u1060</td>
<td>1060</td>
<td>3.55</td>
<td>0.14</td>
<td>0.36</td>
<td>0.86</td>
<td>0.36</td>
<td>1.31</td>
</tr>
<tr>
<td>u1432</td>
<td>1432</td>
<td>66.73</td>
<td>0.28</td>
<td>0.78</td>
<td>1.59</td>
<td>0.69</td>
<td>4.75</td>
</tr>
<tr>
<td>u1817</td>
<td>1817</td>
<td>54.83</td>
<td>0.50</td>
<td>1.14</td>
<td>2.75</td>
<td>1.02</td>
<td>3.53</td>
</tr>
<tr>
<td>u2152</td>
<td>2152</td>
<td>115.30</td>
<td>0.75</td>
<td>1.66</td>
<td>4.03</td>
<td>1.50</td>
<td>6.66</td>
</tr>
<tr>
<td>u2319</td>
<td>2319</td>
<td>683.97</td>
<td>0.86</td>
<td>6.47</td>
<td>8.06</td>
<td>5.95</td>
<td>48.41</td>
</tr>
<tr>
<td>vm1084</td>
<td>1084</td>
<td>3.77</td>
<td>0.16</td>
<td>0.38</td>
<td>1.06</td>
<td>0.38</td>
<td>1.05</td>
</tr>
<tr>
<td>vm1748</td>
<td>1748</td>
<td>13.36</td>
<td>0.47</td>
<td>1.08</td>
<td>3.00</td>
<td>1.02</td>
<td>3.09</td>
</tr>
</tbody>
</table>

KSP/COP is tested with the threshold parameter set to 3;
Items in bold indicate the smallest running time for the given problem.
Figure 1.3: Random Asymmetric instances: tour quality and running time over 3000% above the LB. To accommodate such a wide range, in Figures 1.3–1.7 the excess of the LB is plotted on a logarithmic scale.

**Family 3: Random Asymmetric**

Figure 1.3 on page 49 shows the quality of tours and the running times of the algorithms on forty two Random Asymmetric instances of varying size. It can be seen from the top graph that Random Insertion (RI) fails completely on this family of ATSP instances, producing tours that are 300%–2200% worse than the lower bound. The quality of tours deteriorates as the instance
size increases. The Greedy algorithm (GR) also performs poorly, producing tours that are 160%–400% over the LB. Similarly to RI, quality of tours deteriorates with increased instance size. Recursive Path Contraction (RPC) is also a poor performer, although it does not fail as badly as RI and GR. On this family of instances RPC produces tours that are 75%–135% above the lower bound. However, there is no apparent trend that would suggest that tour quality worsens with an increase in instance size. Karp-Steele Patching (KSP) and Extended Karp-Steele Patching (EKSP) perform very similarly to each other, with EKSP being slightly ahead of KSP on four instances, and falling slightly behind on one instance. Both algorithms perform very well on Random Asymmetric instances, producing tours that are 1.3%–6.5% above the lower bound, with a clear trend towards the lower bound as the instance size increases. The clear winner on this family of instances is the Contract-or-Patch algorithm (COP), which produced tours 0.6%–5.7% above the lower bound, also with a trend to getting closer to the lower bound as instance size increases. COP is bettered by other algorithms (typically EKSP) on just 3 out of 42 instances in this family, each time by a small margin (0.01%–1%).

The gap between the lower bound and the cost of tours produced by KSP, EKSP and COP narrows as the instance size increases. However, we do not know exactly how tight the lower bound is. Therefore it is not clear whether this trend indicates that the algorithms perform better on larger instances, or that the lower bound gets tighter, or perhaps some combination of both (see [28] for a theoretical study of this phenomenon). Nonetheless, it is clear that the above three algorithms produce tours which are very close to optimum, especially for large instances. It is also clear that quality of tours produced by RI and GR worsens as the instance size increases, as the trend here cannot be explained by the lower bound getting tighter for large instances.

The bottom graph in Figure 1.3 shows the trends in running time of the algorithms as a function of instance size. GR is clearly the slowest (its runtime is dominated by the sorting step, which has $\Theta(n^2 \log n)$ average-case complexity). RI is the fastest due to its $\Theta(n^2)$ complexity. KSP, EKSP and RPC require very similar amounts of time. Since the patching procedure of KSP is guaranteed to be fast ($O(n^2)$), this indicates that the running time of the three algorithms is dominated by their common first step – solving
the Assignment Problem. Note that although solving the AP requires $O(n^3)$
time in the worst case, the AP algorithm used in this study performs very
well in practice and typically requires very little time to run. The remaining
algorithm, COP, is slower than the other three AP-based algorithms, taking
up to two times are much time as EKSP. Nevertheless, it is still fast: even
the 3000-node instances take under 20 seconds to complete.

**Family 4: Random Asymmetric with $i \times j$-bounded costs**

Results for the next family, Random Asymmetric instances with costs $c(i, j)$
bounded by $i \times j$, are presented in Figure 1.4. The tour quality results
are similar to the previous instance family, with three heuristics failing to produce good results and three heuristics performing well. Both Greedy and Random Insertion perform very poorly on this instance family, producing tours that cost 280%–1700% over the lower bound. Quality of tours produced by both algorithms deteriorates as instance size increases. Recursive Path Contraction also fails on instances of this type. It produces tours that are 150%–270% above the LB. Unlike previous family, instances of this type become harder for RPC as their size increases. KSP and EKSP again show very similar performance to one another; EKSP tends to perform slightly better than KSP. Contract-or-Patch is again the leader, producing tours that are 0.5%–3.6% above the lower bound. Only for 5 out of 42 instances another algorithm produces tours that are better than those found by COP. In all five cases, the difference in tour cost is small, ranging between 0.01% and 0.7%.

Trends in running time (Figure 1.4, bottom graph) are also similar to the previous instance family. Random Insertion is the fastest algorithm, Greedy is the slowest. KSP, EKSP and COP require very similar amount of time to run, again suggesting that solving the Assignment Problem dominates the solution time. COP is again slower than EKSP by up to a factor of two.

It is interesting to note that for some heuristics $i \times j$-bounded Random Asymmetric instances are more difficult than plain Random Asymmetric instances. GR and RPC perform substantially worse on instances of this type. Other heuristics perform better on this type of instances. However, for KSP, EKSP and COP, this comes at a price of nearly doubled execution time.

Family 5: Random Symmetric

Tour quality and execution time results for the next family – Random Symmetric instances – are presented in Figure 1.5. Except for a small number of similarities, tour quality trends a quite different here. Random Insertion is still the worst-performing algorithm on test, producing tours 220%–1800% above the lower bound. KSP and EKSP perform surprisingly poorly (110%–880% and 90%–690% respectively). Although EKSP performs better than KSP, it is still not a match even for Greedy, which performed very poorly
Figure 1.5: Random Symmetric instances: tour quality and running time on the previous two classes of instances. On instances of this class GR finds tours that are 90%–210% above the lower bound. Recursive Path Contraction performs much better on this class than it did on the previous two: its average excess over the lower bound is in the range 95%–135%. However, the clear winner is once again COP, with average excess over the LB of 36%–47%. Compared to the previous two classes of instances, this still leaves a relatively wide gap between the lower bound and the solutions produced by the construction heuristics. It is not clear how much of this gap is due to suboptimal performance of the heuristics and how much is due to looseness of the lower bound.

Trends in algorithm running times are similar to those seen in previous two families. The Greedy algorithm is the slowest, and Random Insertion
is the fastest. The slowest of the remaining algorithms this time is EKSP, followed by COP. The remaining two algorithms, KSP and RPC, take nearly identical time, again indicating that their execution time is dominated by solving the initial Assignment Problem. All algorithms except GR on average take 15 seconds or less to solve each of the 3000-node instances.

Family 6: Random Symmetric with $i \times j$-bounded costs

Figure 1.6 shows tour quality and execution time results for the next family – Random Symmetric instances with costs $c(i,j)$ bounded by $i \times j$. Once again GR and RI are the worst performers, finding tours that are 480%–
1000% and 240%–1300% above the lower bound, respectively. KSP also performs very poorly, with average excess over the lower bound of 90%–440%. EKSP improves on KSP’s performance, but insufficiently to become competitive: its average excess over LB is 65%–220%. RPC also fails on this family of instances: its tours are 165%–350% over the lower bound. COP is once again the winner, with tours of 37%–42% above the lower bound. Most algorithms (GR, RI, KSP, and EKSP) display a clear trend to produce worse tours as the instance size increases. RPC displays a similar trend, although in a less obvious fashion. No such trend is apparent for COP, whose excess over the lower bound remains very stable for all tested instance sizes.

There are no surprises in the running times. Greedy is the slowest and Random Insertion is the fastest. ESKP is the second slowest, followed by COP and then RPC and KSP, which once again require very similar running times.

**Family 7: Sloped Plane**

The last family of instances is the Sloped Plane instances, defined on page 33. Computational results for this family of instances are presented in Figure 1.7. Tour quality trends are quite different from what we have seen so far. GR displays its worst performance yet, with tour costs above 3000% of the lower bound for larger instances. All the remaining algorithm show very stable performance. The worst performer among the remaining algorithms is RPC, with tours of approximately 70% above the lower bound. Random Insertion and the remaining AP-based algorithms display very similar performance to each other, with Random Insertion slightly ahead of the rest. RI finds tours that are 40%–44% above the lower bound.

An interesting phenomenon observed in running times is that the two slowest algorithms, GR and COP, require nearly identical amounts of time to run. They are completely unrelated algorithms, and operate in totally different way. At present we do not have an explanation for this phenomenon. Random Insertion is once again the fastest with its $\Theta(n^2)$ complexity. RPC and EKSP require very similar amount of time to run, and KSP is somewhat faster than the two.
Figure 1.7: Sloped Plane instances: tour quality and running time

Summary

A overview of tour quality results of all algorithms on all tested families is presented in Table 1.7.

We say that an algorithm is competitive on instance family $\mathcal{F}$ if its average excess over the lower bound on $\mathcal{F}$ is at most twice as high as that of any other tested heuristic.

The Greedy algorithm is the slowest heuristic on test. At the same time it is only competitive on Euclidean instances, and fails on all other families. However, even for Euclidean instances it is a poor choice as Random Insertion performs both better and faster on instances of this type.

Random Insertion is only competitive on Euclidean and Sloped Plane
CHAPTER 1. TOUR CONSTRUCTION HEURISTICS FOR THE ATSP

<table>
<thead>
<tr>
<th>Family</th>
<th>GR</th>
<th>RI</th>
<th>KSP</th>
<th>EKSP</th>
<th>RPC</th>
<th>COP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: TSPLIB/A</td>
<td>34.6%</td>
<td>15.5%</td>
<td>4.3%</td>
<td><strong>3.4%</strong></td>
<td>18.0%</td>
<td>4.3%</td>
</tr>
<tr>
<td>2: TSPLIB/E</td>
<td>18.0%</td>
<td><strong>11.2%</strong></td>
<td>15.0%</td>
<td>17.3%</td>
<td>36.8%</td>
<td>18.1%</td>
</tr>
<tr>
<td>3: RA</td>
<td>278.7%</td>
<td>1107.1%</td>
<td>4.3%</td>
<td>4.3%</td>
<td>102.8%</td>
<td><strong>3.1%</strong></td>
</tr>
<tr>
<td>4: RA ((i \times j))</td>
<td>1074.6%</td>
<td>872.7%</td>
<td>2.4%</td>
<td>2.3%</td>
<td>199.4%</td>
<td><strong>1.9%</strong></td>
</tr>
<tr>
<td>5: RS</td>
<td>153.4%</td>
<td>873.5%</td>
<td>429.7%</td>
<td>339.5%</td>
<td>111.4%</td>
<td><strong>42.3%</strong></td>
</tr>
<tr>
<td>6: RS ((i \times j))</td>
<td>707.3%</td>
<td>679.9%</td>
<td>235.1%</td>
<td>137.5%</td>
<td>220.6%</td>
<td><strong>38.9%</strong></td>
</tr>
<tr>
<td>7: SP</td>
<td>1657.6%</td>
<td><strong>42.1%</strong></td>
<td>44.8%</td>
<td>46.5%</td>
<td>70.7%</td>
<td>48.0%</td>
</tr>
</tbody>
</table>

Table 1.7: Summary of tour quality results for ATSP construction heuristics

instances, which are close to Euclidean. In fact, it outperforms all other tested algorithms on these two families. It is also very fast. It may be a good candidate in cases when instances are Euclidean, or have a cost structure which is close to Euclidean. However, Random Insertion is clearly unsuitable as a general-purpose construction heuristic for the ATSP as it fails quite badly on the remaining five families.

Karp-Steele Patching and Extended Karp-Steele Patching remain competitive on five instance families. However, they both fail on symmetric instances, families 5 and 6. It is interesting to note that Extended Karp-Steele Patching is the champion among all six algorithms on asymmetric instances from TSPLIB. Compared to plain KSP, it offers a substantial improvement in tour quality at the price of increased implementation complexity and somewhat increased execution times. Despite a 30% increase in running times compared to KSP, the algorithm remains very fast: it takes just over 0.5 second to solve the entire Asymmetric TSPLIB testbed consisting of 26 instances. Perhaps this makes EKSP a worthwhile candidate for certain types of ATSP instances that arise in practical applications, and warrants further research into performance of EKSP. Nevertheless, both KSP and EKSP perform very poorly on symmetric instances, families 5 and 6, and therefore are not suitable as general-purpose algorithms that can be used on arbitrary ATSP instances with unknown structure.

Recursive Path Contraction displays disappointing performance. In our tests it never fails as badly as do some other algorithms, but it does not come close enough to the leaders. The algorithm is competitive on only one family, Sloped Plane instances, but even on this family its average excess over the
lower bound is approximately 70%, compared with 42% of the leader.

Contract-or-Patch is the only algorithm on test that remains competitive on each of the tested families of instances. It outperforms all other algorithms on four out of seven families of instances. On the remaining families it comes relatively close to the leaders. We therefore recommend it as a general-purpose ATSP construction heuristic that can be used to quickly construct near-optimal tours for arbitrary instances of the ATSP.

1.6 Reference implementation of Contract-or-Patch

We provide a reference implementation of the Contract-or-Patch algorithm (COP/KSP), which is presented in Appendix A. The latest version of the code can be downloaded in electronic form from http://www.cs.rhul.ac.uk/~zvero/thesis/cop_ref/.

The implementation is done in the C++ programming language (see [70]). The code aims to provide a concise yet complete implementation of the algorithm with less emphasis on efficiency. The goal is to make it easy to get started for anyone wishing to study or use the algorithm.

The implementation is significantly more high-level and compact than the one used in the computational evaluation of the TSP algorithms. However, it is also less efficient. One (perhaps the most significant) area where efficiency can be improved is due to constraints imposed by integration with the third-party Linear Assignment Problem code used by the COP/KSP implementation. The Assignment Problem code requires the matrix of costs to be supplied to it as a raw two-dimensional C array. This poses little problem when the Assignment Problem is solved on the original (uncontracted) TSP cost matrix, at least when the TSP cost matrix is stored anyway, rather than computed on the fly. At subsequent stages of the algorithm some paths are contracted; the entire 2D cost matrix for the "contracted" instance needs to be pre-computed into a C array before the Assignment Problem code can be called. If the AP code would be more flexible in the data structure it accepts, computing the entire "contracted" cost matrix would not necessarily be required, as we show below.
Assume the path $v_1v_2\ldots v_k$ is contracted into the new vertex $v'$. Arcs leaving $v'$ in the contracted graph correspond to arcs leaving $v_k$ in the original graph; arcs entering $v'$ in the contracted graph correspond to arcs entering $v_1$ in the original graph. All arcs in the contracted graph have the same cost as the corresponding arcs in the original graph. Therefore, instead of computing complete "contracted" cost matrices for each stage of the algorithm, an implementation of COP can rely on a single cost matrix (representing the original "uncontracted" TSP instance). Such an implementation would then maintain the association between the vertices of the current contracted graph and the original graph (in the example above, vertex $v'$ in the contracted graph would correspond to two original vertices $v_1$ and $v_k$, one for in-arcs, another for out-arcs). Given such associations and the original cost matrix, the "contracted" cost matrices can be computed on-the-fly instead of being pre-computed and stored. The cost of this optimisation is two extra indirections per access to an element of the cost matrix corresponding to the contracted graph.

This optimisation would considerably reduce memory requirements of the code as well as potentially reducing execution times, although the precise effect on execution times is less clear.

The code of the reference implementation can be used free of change for any purpose (including commercial use).

1.7 Conclusions

In this chapter we studied computational performance of a number of tour construction heuristics for the Asymmetric Travelling Salesman Problem. Algorithms tested in this study include some well-known and widely used heuristics found in the literature. One of the algorithms tested in this study (Recursive Path Contraction) is known to possess interesting theoretical properties; however, to our knowledge its practical performance has never been studied in the literature before. Finally, in this chapter we developed several new algorithms and compared their computational performance with that of the well-known algorithms. Performance of all algorithms was measured using a testbed comprising a diverse set of seven families of ATSP
Computational evidence presented in this chapter suggests that some frequently-used heuristics that are known to perform well on Symmetric TSP may not be a good choice for the asymmetric case. Each of the well-known algorithms failed to produce quality tours on at least one of the instance families. On the other hand, one of the new algorithms, Contract-or-Patch, demonstrated very robust performance by producing competitive solutions on each of the tested families. We therefore recommend Contract-or-Patch (more specifically, its KSP/COP variant) as a good universal construction heuristic for the Asymmetric Travelling Salesman Problem.

In this chapter we also studied the impact of the COP parameter, threshold, on the quality of tours produced, and recommended a good all-round choice for the value of the parameter based on our computational experience. The parameter can be used to fine-tune the algorithm to the specific TSP instance under consideration. It also provides means for controlling the trade-off between running time and quality of solutions produces.

We provide a reference implementation of the Contract-or-Patch algorithm. This implementation of the algorithm is made available for any use (including commercial) free of charge and can be of interest to anyone evaluating or applying the algorithm.
Appendix A

Reference implementation of Algorithm COP

// cop_ref.cxx
//
// Copyright 2002 Alexei E. Zverovitch
//
// Permission to use, copy, modify, distribute and sell this software
// for any purpose is hereby granted without fee, provided that the
// above copyright notice appears in all copies and that both that
// copyright notice and this permission notice appear in supporting
// documentation. The author makes no representations about the
// suitability of this software for any purpose. It is provided
// "as is" without express or implied warranty.
//
// The most up-to-date version of this file can be downloaded from
// http://www.cs.rhul.ac.uk/~zvero/thesis/cop_ref/
//
// This code requires the C++ code by Jonker and Volgenant for solving
// the Linear Assignment Problem. The code can be downloaded from:
// http://www.magiclogic.com/assignment/lap_cxx.zip
//
// This code has been tested with the version 3.0 of the g++
// compiler
//
// $Id: cop_ref.cxx,v 1.3 2002/04/23 08:14:00 alexei Exp $

#include <string>
#include <vector>
#include <fstream>
#include <stdexcept>
#include <algorithm>
#include "lap.h"

using namespace std;

// Version information
string revision("$Revision: 1.3 $");
string version_temp(revision.substr(revision.find(':' ) + 1));
string version(version_temp.substr(0, version_temp.length() - 1));

// Cost matrix
typedef vector<vector<cost>> cost_matrix_t;

// Cycle factor - cf[i] is the vertex that follows i in the cycle factor
typedef vector<int> cycle_factor_t;

// Tour - t[i] is the vertex that follows i in the tour
typedef vector<int> tour_t;

// Infinity
const int INF = 1000000;

// The threshold parameter
const int THRESHOLD = 3;

// Check that the cycle factor is valid
void validate_cycle_factor(const cycle_factor_t& cf)
{
    vector<bool> seen(cf.size(), false);
    for (int i = 0; i != cf.size(); ++i)
    {
        if (cf[i] < 0 || cf[i] >= cf.size() || seen[i])
        {
            throw runtime_error("Invalid cycle factor");
        }
        seen[i] = true;
    }
}

// Check that the tour is valid
void validate_tour(const tour_t& tour)
{
    validate_cycle_factor(tour);
    int n(0);
    for (int i = 0, first = 1; i != 0 || first; i = tour[i], first = 0)
    {
        ++n;
    }
    if (n != tour.size()) throw runtime_error("Invalid tour");
}

int get_tour_cost(int size, const cost_matrix_t& C, const tour_t& tour)
{
    int cost(0);
    for (int i = 0; i != size; ++i)
APPENDIX A. REFERENCE IMPLEMENTATION OF ALGORITHM COP

```cpp
// Eliminate short cycles

// Solve the LAP

// not-yet-visited cycle

// delete most expensive arc and contract the path
```
for (int j = i, first = 1; j != i || first; j = next[j], first = 0) {
    contracted_vertices.push_back(make_pair(j, j));
}
++ncycles;
}

if (ncycles == 1 || !contracted) return next;

int contracted_size=contracted_vertices.size();
cost_matrix_t contracted_C=contracted_size;

for (int i = 0; i != contracted_size; ++i) {
    contracted_C[i].resize(contract); // expand contracted paths
}

for (int i = 0; i != size; ++i) {
    if (new_next[i] == -1) new_next[i] = next[i];
}

validate_cycle_factor(new_next);

return new_next;

bool compare_cycle_info(const pair<int, int>& c1, const pair<int, int>& c2) {
    return c1.first > c2.first;
}

// Transforms cycle factor next into a tour (in-place)
void KSP(int size, const cost_matrix_t& C, cycle_factor_t& next) {
    
}
vector<bool> visited(size, false);

typedef vector<pair<int, int>> cycle_info_t;
cycle_info_t cycle_info;

for (int i = 0; i != size; ++i)
{
    if (!visited[i]) // not-yet-visited cycle
    {
        int len(0); // cycle length
        // find cycle length
        for (int j = i, first = 1; j != i || first; j = next[j], first = 0)
        {
            ++len;
            visited[j] = true;
        }
        cycle_info.push_back(make_pair(len, i));
    }
}

sort(cycle_info.begin(), cycle_info.end(), compare_cycle_info);

pair<int, int> c0(*cycle_info.begin());

for (cycle_info_t::iterator ci = cycle_info.begin() + 1; ci != cycle_info.end(); ++ci)
{
    int min_patching_cost(INF);
    int min_i;
    int min_j;

    for (int i = c0.second, first = 1; i != c0.second || first; i = next[i], first = 0)
    {
        for (int j = ci->second, first = 1;
            j != ci->second || first;
            j = next[j], first = 0)
        {
            int patching_cost(C[i][next[j]] + C[j][next[i]] - C[i][next[i]] - C[j][next[j]]);

            if (patching_cost < min_patching_cost)
            {
                min_i = i;
                min_j = j;
                min_patching_cost = patching_cost;
            }
        }
    }

    swap(next[min_i], next[min_j]);
}

validate_tour(next);
// Implementation of KSP/COP

tour_t COP(const string& filename)
{
    int size;
    cost_matrix_t C;

    // Read the ATSP instance
    cout << "Solving " << filename << endl;
    ifstream f(filename.c_str());
    if (!f) throw runtime_error("File not found: " + filename);
    f >> size;
    C.resize(size);
    for (int i = 0; i != size; ++i)
    {
        C[i].resize(size);
        for (int j = 0; j != size; ++j)
        {
            f >> C[i][j];
        }
        C[i][i] = INF;
    }

    // Apply KSP/COP
    cycle_factor_t next(eliminate_short_cycles(size, C));
    KSP(size, C, next);
    cout << "Tour cost: " << get_tour_cost(size, C, next) << endl;
    return next;
}

// Format of input file:
//
// num_vertices
// c(1,1) c(1,2) ... c(1,n)
// ...
// c(n,1) c(n,2) ... c(n,n)

int main(int argc, char** argv)
{
    try
    {
        cout << "cop_ref version" << version << "built on " _DATE_ " " << endl;
        if (argc < 2)
        {
            cout << "Usage: cop_ref file ..." << endl;
        }
        else
        {
            for (int i = 1; i < argc; ++i) COP(argv[i]);
        }
        return EXIT_SUCCESS;
    }
    catch (exception& e)
    { 
        // Handle exception
    }
{ 
    cerr << "Error: " << e.what() << endl;
    return EXIT_FAILURE;
};
Bibliography


BIBLIOGRAPHY


dense and sparse linear assignment problems. *Computing*, 38:325–340,
1987. The authors’ implementations of the algorithm in Pascal, Fortran,

[51] A. Juttner, B. Sziatovszki, I. Mecs, and Z. Rajko. Lagrange re-
relaxation based method for the QoS routing problem. In *IEEE IN-

[52] P.C. Kanellakis and C.H. Papadimitriou. Local search for the asymmet-
1980.

[53] R. M. Karp. A patching algorithm for the nonsymmetric traveling-
1979.

Lawler et al. [55], pages 181–205.

[55] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys, edi-
tors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial

[56] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the

multi-period strategic planning problem. In A.H. Christer, S. Osaki,
and L.C. Thomas, editors, *Stochastic Modelling in Innovative Manu-
ufacturing*, volume 445 of *Lecture Notes in Economics and Mathematical

[58] H.M. Markowitz. The elimination form of the inverse and its application

[59] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-
dimensionally equidistributed uniform pseudorandom number genera-


