Tour merging via tree decomposition

A hybrid approach between heuristics and exact solutions for the TSP and VRP

Master Thesis

Mattias Beimers
(ICA-3672565)

Supervisors:
dr. Johan van Rooij
dr. Hans Bodlaender

Department of Information and Computing Sciences
Utrecht University, The Netherlands

August 28, 2015

Abstract

The TSP and VRP are well known optimization problems. In this thesis we evaluate the use of a dynamic programming algorithm on a tree decomposition of a graph, with the goal to improve the solutions to these problems given by several heuristics. The used heuristics are the LKH algorithm, the savings algorithm and the sweep algorithm.

1 Introduction

For many optimization problems calculating provably optimal solutions is not feasible in practical applications, because the computation time grows exponentionally with the problem size. Hence, heuristics are used to find solutions that are good, but not necessarily optimal. To get more certainty that a solution is good, or to improve the solution even more, the heuristics are often applied multiple times and the best solution is selected. Although this works well, Cook and Seymour noted in their work on the Traveling Salesman Problem [1] that by discarding all but the best solution, possibly valuable information is lost. Hence the idea emerged to merge all the found solution tours in a single graph and calculate the optimal solution on the branch-decomposition of that graph.

At the time of writing, the solutions found by Cook and Seymour improved on the best known results, sometimes even getting optimal solutions, for instances with 15000 and 18000 vertices [1]. Since then other heuristics have improved massively and outperform the approach by Cook and Seymour [10]. In this thesis, we will try if the strategy for TSP by Cook and Seymour can still improve current heuristics even more. Furthermore, we will try to extend it to work for the Vehicle Routing Problem (VRP).

The Traveling Salesman Problem (TSP) is one of the most well studied NP-hard problems, where a merchant wants to visit a number of cities and get back at his starting point in the shortest possible amount of time. We recognize the TSP problem in many practical applications, from planning a school bus route to scheduling a machine to drill holes in a circuit board. A generalized version of this problem, where there are not one but a number of merchants (or trucks) visiting the cities from the starting point (or depot), is widely used in the transportation sector. This problem is known as the Vehicle Routing Problem (VRP).

We define the TSP, given a complete graph $G' = (V, E')$, as finding a tour, or cycle, that visits all cities exactly once with smallest total cost. For this thesis we assume the cost c_e of an edge $e = (v, w)$ is the euclidean distance between v and w. Given additionally a demand d_v for each vertex v, a maximum capacity C of goods per truck, a number M of available trucks and a special vertex v_0 that is the depot, we can define the VRP as finding a set of at most M tours with the least total cost. Each tour has to start and end at the depot and can satisfy a total demand of at most C. Each vertex has to be visited by a tour exactly once. There are many other variants of the VRP with additional constraints or freedoms, but these are out of the scope of this thesis.

To solve the TSP and VRP we apply the following stragegy: We start by calculating an initial set of solutions using heuristics. We then merge all the edges of these solutions into a set of promising edges $E \subset E'$. After that we merge the solutions into a subgraph $G = (V, E)$. On this graph we (hopefully) find a tree decomposition with small width k. With that decomposition we can calculate the optimal solution in G using a dynamic programming algorithm that has a running time exponential in k but linear in the number of vertices. This solution often improves on each of the solutions of the heuristic.

The rest of this thesis is organised as follows: in Section 2 we will discuss the heuristics used to generate the initial tours and routes. In Section 3 we will discuss how the solutions are merged and how the tree decomposition is calculated and in Section 4 we will show the dynamic programming algorithms on the computed decompositions. In Section 5 we will discuss the result and finally we conclude in Section 6.

2 Heuristics

Although many different heuristics have been tried to solve the Traveling Salesman Problem, there are few that can compete with (variants of) the Lin-Kernighan heuristic [9, 12], most notably the implementation of Helsgaun [10]. One relatively new family of algorithms are the Stem-and-cycle algorithms [3, 4, 2], which perform better than the basic Lin-kernighan implementations [2], but cannot beat the advanced implementations like the one from Helsgaun. To find initial solutions for the TSP we chose to use the Lin-Kernighan-Helsgaun heuristic because it is one of the best heuristics available and its source code is available for academic use [11]. We will discuss the original Lin-Kernighan heuristic in Section 2.1 and the modifications on the original algorithm in Helsgaun's implementation in Section 2.2.

For the Vehicle Routing Problem the currently best heuristics are tabu search algorithms [12, 13]. Unfortunately they often require to finetune a lot of parameters and are focussed on specific instances of VRP, rather than giving consistent solutions for all versions [12]. Other heuristics like the classic savings heuristic or the sweep heuristic do give good solutions for all variants of VRP, but they can't get the results one gets with the tabu heuristics. For the VRP we chose to use one run of the savings heuristic and multiple runs of the sweep heuristic, because of their fast running times, reasonable quality of solutions and ease of implementation. We will discuss these heuristics in Section 2.3 and Section 2.4.

2.1 Lin Kernighan

The Lin-Kernighan heuristic [8] is an improvement heuristic. That means that the strategy to solve the TSP consists of the following steps:

1. Generate a (random) initial tour.
2. Try to find a modification of the tour that improves it.
3. If an improved solution is found, replace the tour and repeat from step 2.
4. If no improved solutions can be found anymore, we are at a local optimum. We can either start again from step 1 or stop, depending on some stopping criteria (e.g. the solution is good enough, the pool of initial tours is deplenished, or a time limit is reached).

The interesting part of this strategy is step 2: how do we improve on the current tour. One way of doing this is to use a k-opt algorithm. In a k-opt algorithm we try to find two disjoint sets of edges X and Y, both containing k edges, such that when we remove the edges in X from the current tour and replace them with the edges from Y the tour will have a lower cost. k-opt algorithms are well known and often used heuristics because they improve a tour effectively while being easy to implement. 2-opt and 3-opt improvements were proposed for the first time by Croes [5] and Lin [6], respectively in 1958 and 1965. k-opts with higher values of k have been tried as well, for example by Christofides and Eilon [7], who tried values for k up to 5. However, using k-opt for fixed k has its limitations. It is unknown beforehand which value of k will give a good result for the running time involved, as it costs substantially more time to find the edge sets and their improvements for increasing values of k.

The Lin-Kernighan algorithm is a k-opt algorithm, but for a dynamic k. The edge sets to be replaced are equal to a sequence of 2-opt exchanges, possibly preceded by a single 3-opt. The difference between this approach and repeatedly applying 2-opt optimizations is that not every part of the sequence has to improve on the cost of the tour; it is the cost of the entire sequence that matters. Another difference with the fixed k-opt algorithms is in how we choose the sets of edges to be removed or inserted. In the 2-opt algorithm we first choose the set X of edges in the original tour to be removed (i.e. two crossing edges with the euclidean metric) and then find the corresponding set Y of new edges to be inserted in the tour. In the Lin-Kernighan algorithm, the sets X and Y are built up step by step, as the sequence grows.

In the algorithm we start by choosing an initial vertex t_1 and choose one of the two adjacent edges $x_1 = (t_1, t_2)$ in the original tour. After we have chosen the first edge x_1, we repeatedly choose edges $y_i = (t_{2i}, t_{2i+1})$ and $x_{i+1} = (t_{2i+1}, t_{2i+2})$, according to some criteria. Note that X contains one edge more than Y. Because of that, if we remove the edges in X from the current tour and add the edges in Y to it, the result will not be a tour but a path. However, if X and Y are constructed in the right way, we can close the path by adding the edge $y_{i+1} = (t_{2i+2}, t_1)$ to get a tour. We choose an edge y_i from a set of edges to the 5 nearest neighbours of t_{2i}. This edge y_i may not be in the original tour already and the current sequence must have a positive gain, i.e. $\sum_{j=0}^{i} x_j - y_j > 0$. Furthermore, y_i should be chosen such that a next edge x_{i+1} exists (i.e. x_{i+1} is not chosen already). The edge x_{i+1} is uniquely defined for all $i \geq 1$, as there are only two edges adjacent to t_{2i} in a tour, and if the wrong one is chosen the tour cannot be closed anymore (see Figure 2). An exception is made for x_2, where the wrong edge choice is allowed as it can be fixed by the choice for y_3 and x_4. If there are multiple valid choices for y_i or x_{i+1}, we initially choose the one with the highest gain. For y_i we look ahead and choose the edge where $x_{i+1} - y_i$ is largest. For x_{i+1} we choose the edge with the largest weight. Other choices are ignored at first, but may be examined via backtracking. We keep adding edges y_i and x_{i+1} to the sets, until either the complete sequence (including the last edge y_{i+1} that will close the tour) is shorter than the previous tour (in which case we succeeded) or that there is no edge y_i to find that improves the tour even if the closing edge y_{i+1} would have weight 0 (in which case we failed). If we fail, we can either stop, start again from scratch (with a different vertex for t_1), or use backtracking. The latter means that we go back a few steps and try another edge y_i (or occasionally another edge x_{i+1}). As backtracking is quite time consuming, we only allow it at the first two levels ($i \leq 2$). An example of the dynamic k-opt procedure is shown in Figure 1.

This completes the overview of the Lin-Kernighan heuristic. We will describe two more optimizations below, one to speed up the algorithm, the other to improve the solution. There are many more details which are essential for the efficiency of the algorithm however, as it is not a simple algorithm to implement [9]. We omit them here as they are not essential to this thesis. For the exact details we refer to the original paper [8].

The first optimization reduces the choices for the edges. In the first few solutions found by the algorithm there always are a lot of edges that appear in every solution. These common edges are recorded and then they are no longer allowed to be broken for the other solutions. This means they cannot be used as choice for x_{i+1} any more, which in turn limits the choices for y_i and

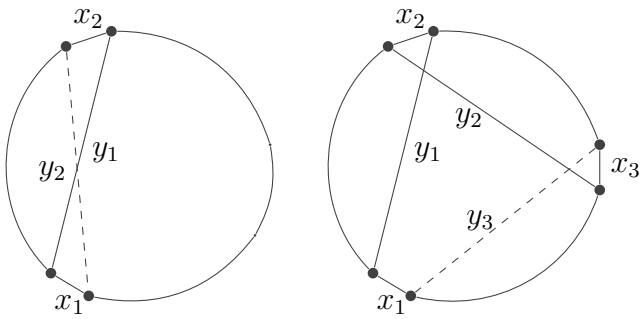

Figure 1: An example of a 3-opt move, as constructed by the Lin-Kernighan heuristic. Note that vertices are displayed in a circle in the order they appear in the original tour.

speeds up the overall algorithm significantly. Note that in order to not bias the solution too much (we do not want to find the exact same solution again after all), we only use this restriction for $i \geq 4$.

In the second optimization of the algorithm we apply the double bridge move. The dynamic k-opt procedure that we described here only allows us to find so called sequential moves: a series of connected edges, alternating inside and outside the original tour. Not all possible k-opts consist of such a sequence however, and not considering these moves can sometimes be the difference between a good solution and the optimal solution. The easiest example of a non sequential move is a 4-opt known as the the double-bridge move (see Figure 3). It is tried as a post-optimization after we finished with the entire algorithm and only for edges that are not amoungst the common edges from the previous optimization. Lin and Kernighan found that in some cases this improved the result significantly, while in other cases it did nothing. It doesn't hurt to try though, as it is a relatively cheap optimization.

2.2 Lin Kernighan Helsgaun

The Lin-Kernighan-Helsgaun algorithm [9, 10] is based on the algorithm of Lin and Kernighan, but improves it on several points.

We define a 1-tree for a graph $G = (V, E)$ as a spanning tree on the vertex set $V \setminus \{1\}$ combined with two edges from E incident to the 1-vertex. An example of a 1-tree is shown in Figure 4. Note that a 1-tree is not a tree (as it contains a cycle) and that the choice of the 1-vertex is arbitrary. A minimum 1-tree is a 1-tree of minimum length. The *α-nearness* of an edge e is defined as the difference between the length of the smallest 1-tree containing e and the

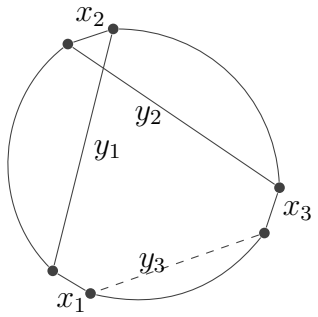

Figure 2: An example where the wrong choice for x_3 is made, resulting in two disjunct tours.

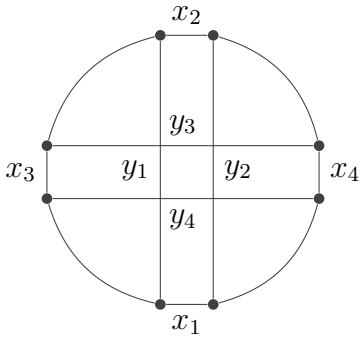

Figure 3: The double bridge move

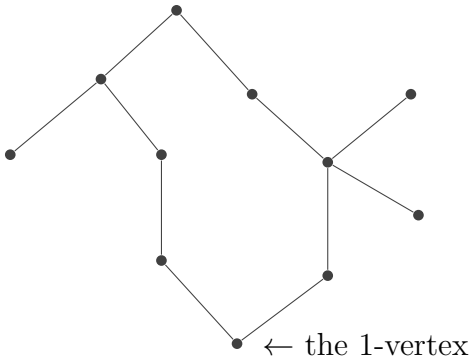

Figure 4: A 1-tree

length of the minimum 1-tree. Or in other words: α-nearness is the increase in length of the minimum 1-tree if it is required to contain the edge e. A straightforward computation of all α-nearness values of the edges takes $O(n^3)$ time, but in his paper Helsgaun shows how to compute it in $O(n^2)$ time [**?**].

The first major difference with the Lin-Kernighan algorithm is the candidate set for the edges in Y. In the original algorithm the choices for an edge y_i are limited to the edges to the first 5 nearest neighbours. The choice for this candidate set assumes that the shorter an edge is, the higher the chance is that it occurs in a tour. This is reasonable, but it is not always a good estimation. Helsgaun notices that the α-nearness, is a better estimation and uses it instead of the nearest neighbours as the candidate set. The candidate set then is further modified to always include edges that are in both of the two previous best solutions. These edges are tried first. The candidate set for the first edge to be chosen, x_1, is also changed so that no edges from the previous best tour are removed in the first level of the improvement step.

The second major change is the choice of the basic move. In the original Lin-Kernighan algorithm every move was composed as a sequence of 2-opt moves (and possibly a 3-opt). Helsgaun [9] modified it to use moves that consist of 5-opt moves (unless a k-opt move for smaller k results in an improvement already). Later [10] he modified it again to use general k-opt moves up to a certain k that we can choose ourselves as the basic step. The steps to do this are rather involved and contain a large case-analysis and won't be described here in detail. For the details we refer to the paper of Helsgaun [10]. The main idea of his approach however is that we allow edges x_i that initially break up the tour (like in Figure 2). We then look ahead to the following edges y_{i+1} and x_{i+1} to make sure that acceptable edges exist that can fix the tour. These edges in their turn do not have to be chosen; we can choose edges with higher

gain that break up the tour again, providing we look ahead and find acceptable edges that can fix the tour. This allows, amoung others, the double bridge move from Figure 3 to be included natively in the search.

The third change is not aimed at improving the quality of the solutions, but aims to speed up the algorithm. As in most improvement heuristics, the Lin-Kernighan algorithm is tried several times on different initial tours. These tours where constructed randomly because Lin and Kernighan considered construction heuristics to be unnecessary. Helsgaun notes that even though a good construction heuristic does not significantly improve the quality of the final tour, it does improve the running time of the algorithm. The heuristic used by Helsgaun tries to construct a tour greedily by choosing edges that are in the minimum 1-tree and that are in the previous best tour. If no such edge is found either a random other candidate edge is chosen or, if they are not valid either, an edge to any free vertex.

Amoungst some more additions added by Helsgaun is the option of merging tours. Similar to what we do in this thesis he merges the edges of a few solutions in a single graph and on this merged graph he solves the TSP again. Unlike what we do in this thesis he doesn't solve the merged problem to optimality, but applies the general k-opt submoves again. This time he uses larger value of k, which he can do because the graph is sparse. We disable this option in our experiments, as we perform this calculation ourselves. There are some more additions, not specifically related to the Lin-Kernighan algorithm, but more general applicable to the TSP, for which we refer to Helsgaun's paper [10].

2.3 Savings

The savings heuristic [14] is a simple but relatively effective construction heuristic for the VRP. A construction heuristic is a heuristic that it creates a set of tours without relying on anything except the data on the vertices and truck capacity. There are two versions of the savings algorithm, the parallel version and the sequential version. We implement the parallel version here, as the resulting tours are usually better than the tours generated by the sequential savings algorithm [12, 13]. Our savings algorithm consists of the following steps:

1. Initially we create $n-1$ tours, one for each non-depot vertex. Each tour starts at the depot, goes to the associated vertex and then goes back to the depot.
2. For every pair of vertices i, j we then calculate the saving $S_{i,j} = c_{(0,i)} +$

$c_{(0,j)} - c_{(i,j)}$. This represents the gain in cost we get by removing the two edges $(0, i)$ and $(0, j)$, and adding the edge (i, j).

3. We now take the saving $S_{i,j}$ with the highest gain and check if we can apply it. With applying $S_{i,j}$ we mean that we try to find two tours, one of which ends with the edge $(0, i)$ and one of which begins with the edge $(0, j)$ (or the other way around). We then connect the two tours together by removing these edges and adding the edge (i, j). This is of course only possible if there are two tours that begin or end with these edges and if the demand of both tours together does not exceed the capacity of a truck.
4. We continue to repeat step 3 until no more savings can be applied. If the total amount of tours is less or equal than the allowed amount of trucks, we are done. If it exceeds the limit, we failed.
5. If the heuristic has successfully found a set of tours, we optimize each tour separately using the Lin-Kernighan-Helsgaun heuristic as described in Section 2.2.

We decided not to make an exception for the savings that have a negative gain (i.e. savings that make the total cost worse). This means that we apply them just like we apply the positive savings. The reasoning behind this is that it's better to have a slightly worse solution than an invalid solution. Applying the negative savings means we use one less truck, which is a hard constraint on the problem and thus more important. Note that this situation occurs very little (if ever) because the negative savings are tried last, so this is not likely to have a big impact on the solution of the tours.

The characteristic behaviour for the savings algorithm is to put a lot of effort into getting a set of tours with low cost. This produces good tours, which is nice, but it also has a negative side effect. This disadvantage is that it does not pay much effort into making sure the capacity of a truck is used optimally. This means that sometimes it generates solutions that need more trucks than it is allowed to use, which makes the given set of tours an invalid solution. A possible workaround would be to allow solutions that use more tours than strictly allowed in the hope that the edges would be of use in creating the final tour after the merging step from Section 3. We decided against this because that would be cheating - after all, the goal is not to finetune heuristics for the merging process but the other way around. Another disadvantage of this heuristic is that it is deterministic, and can therefore only produce a single solution. To get more solutions we use the sweep heuristic in Section 2.4.

2.4 Sweep

The sweep heuristic is another heuristic for the VRP. It is most commonly attributed to Gillett and Miller in 1974 [17], but can be found earlier in a book from Wren [15] or a paper from Wren and Holliday [16] in 1971 and 1972 respectively. It depends on the geometry of the vertices and is only applicable for the euclidean VRP, not in general. This is not a problem for us because we aim to solve the euclidean VRP anyway. Unlike the savings heuristic, the sweep heuristic is an non-deterministic algorithm in the sense that it depends on the starting vertex (or starting angle) and can produce different sets of tours for the same instance. Note that a starting angle and a starting vertex correspond one on one with each other. If you begin with a starting angle the starting vertex is the first vertex that it will hit when rotating clockwise. If a starting vertex is given, all angles between the angle through the starting vertex and the angle through the first vertex counter clockwise of the starting vertex create the same set of tours.

The sweep heuristic is a two-phase algorithm. That means that it splits up the task of creating a set of tours by first clustering the vertices into groups that can be served by a single truck, and then deciding for each cluster of vertices separately in which order the truck should visit the vertices. The first stage of clustering the vertices is what the sweep algorithm is designed to solve. The second stage is essentially a set of TSP problems, which can be solved using a dedicated algorithm. We solve this second step using the Lin-Kernighan-Helsgaun heuristic from Section 2.2.

Because of its geometric nature, the sweep heuristic can be best explained by picturing a drawing of the vertices on a plane. The algorithm then consists of the following steps:

1. We start by drawing a ray from the depot through the starting vertex. We add these two vertices to the first tour; the tour we are currently building.
2. We then rotate the line clockwise until it hits a vertex. If the vertex can still be added to the previous tour (i.e. the total demand of the vertices in the tour does not exceed the capacity of a truck), we add it to the tour.
3. We repeat step 2 until adding the next vertex will exceed the capacity limit (or until there are no vertices left, in which case we are finished and go to step 5).
4. Once the capacity limit is reached, we add the next vertex and the depot to a new tour, and we repeat from step 2 until there are no vertices left.

5. Once we are finished with the sweep part, we solve each tour separately using LKH.

The tours created by this heuristic are not necessarily bad, but they are very clustered. This means that it will never explore a large part of the solution space. To make for a bit more of a variety of edges, we use the savings heuristic as described in Section 2.3.

One thing left to discuss is the choice of the vertex to start with. For the choice of the starting vertex, given the start vertex for the previous set of tours, there are basically two options. One option is to start with the first vertex that comes next with respect to clockwise rotation, this would create a sequence of tours that look a lot like each other. The other choice would be to spread out the start vertices evenly, causing the set of tours to be move diverse but less related. We chose the second option to choose the starting vertices uniformly. One reason is that the first option is basically a disguised version of the String Relocation heuristic, which is an improvement heuristic that removes a sequence of vertices from one tour and places it in the neighbouring tour. The second option however should add a diverse set of edges to the final tour, which should allow for broader viewpoint.

3 Tree decomposition

Once we have generated a set of good tours for our original graph $G' = (V, E')$, we merge these tours in one graph. All tours $E'_j \subset E'$, for $0 \leq j < \#tours$ as found by the heuristics are merged together into a new graph $G = (V, E)$, where $E = \bigcup E'_j$. For this graph G we will compute a tree decomposition so that we can compute the optimal tour in this reduced problem. Before we show how we compute this decomposition in Section 3.2, we first give the definition of a tree decomposition in Section!3.1.

3.1 Tree decomposition and width

A *tree decomposition* of a graph $G = (V, E)$ is a pair $(T = (W, H), X)$, where T is a tree with an arbitrary root vertex and $X = \{X_i \subset V : i \in W\}$ a set of *bags*, satisfying:

1. $\bigcup_{i \in W} X_i = V$,
2. for all $(u, v) \in E$ there is an $i \in W$ with $u, v \in X_i$ and

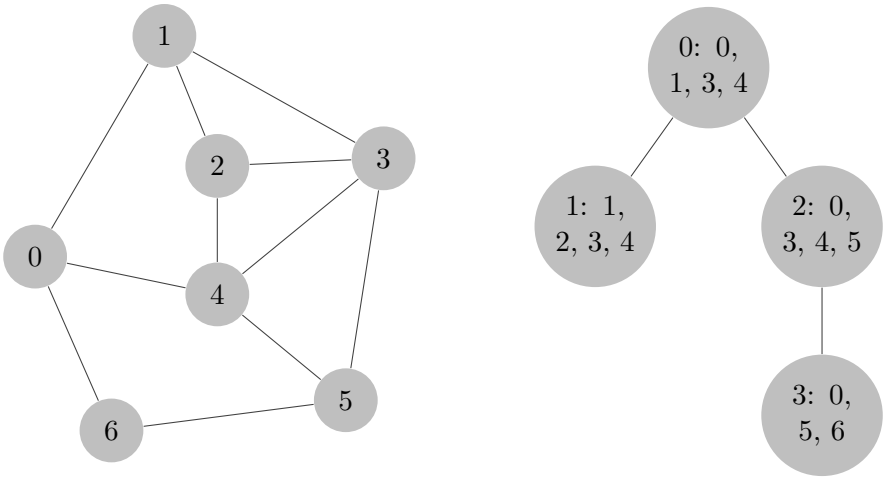

Figure 5: An example of a graph and its tree decomposition.

3. for all $v \in V$, the set $W_v = \{i \in W : v \in X_i\}$ forms a connected subtree of T.

The *width* k of the tree decomposition is $\max_{i \in W} |X_i| - 1$. The *treewidth* of a graph G, is the minimum width among all tree decompositions of G. An example of a tree decomposition is shown in Figure 5.

Throughout this thesis we often work with the edge set corresponding to the vertex $i \in W$, rather than the vertex set X_i itself. To that end we define $Y_i = \{(u, v) \in E : u, v \in X_i\}$. We say that a bag contains a vertex v if $v \in X_i$ and that it contains an edge e if $e \in Y_i$.

3.2 Minimum Degree Heuristic

Calculating the optimal treewidth or the optimal tree decomposition for unknown treewidth is an NP-Hard problem [20], so finding an optimal decomposition in reasonable time is infeasable unless P=NP. We do not necessarily need a tree decomposition of optimal width, we just need the width to be sufficiently small so that our dynamic programming algorithm runs fast enough. Therefore, we compute our tree decomposition with a heuristic.

Bodlaender and Koster [18] evaluated a number of construction heuristics. We chose to use the Minimum Degree Heuristic, originally designed by Markowitz [19], because it is a simple but effective heuristic. It is fast, obtains results close to the optimum and is easy to implement. A non-recursive version of the algorithm consists of the following steps:

1. Initially let $(T = (W, H), X)$ with W, H and X set to $\emptyset$.

2. Take the vertex $v \in V$ with minimum degree and add it to W; i.e. add a new vertex to W with the same name as v. The reason to give it the same name is that it allows us to add edges ahead of time in step 4.
3. Create a bag X_v with v and all its neighbours in G.
4. Add an edge (v, w) to H, where w is the neighbour of v in G with the smallest degree (so the first neighbour to be processed). Note that w is not yet added to W, but will be added in the future.
5. Modify G by turning all the neighbours of v into a clique and removing v from V (and its incident edges from E).
6. Repeat step 2 to 5 until all vertices are processed (i.e. $V = \emptyset$).

To complete the tree decomposition we choose the first vertex of W to be the root of the tree, and, when we are solving the VRP, we also add the depot vertex to every bag.

As a small optimization step we remove any bag that is fully contained in it's parent. This happends quite often in this algorithm, because in step 3 we add all the neighbours of a vertex v to a bag, and then connect all these neighbours to each other in step 5. Once one of these next vertices gets processed while it doesn't have any neighbours that v did not have, it will add all the neighbours of v again, but this time without v itself. Hence the duplication.

4 Dynamic programming

Provided the width is small enough, the optimal solution for the TSP or VRP on the merged graph can be computed using a dynamic programming algorithm on the tree decomposition of the graph. In the following sections we explain the details of the algorithms.

4.1 Traveling Salesman

Let $G = (V, E)$ be a simple graph with edge-weights c_e and $(T = (W, H), X)$ be the tree decomposition with width $k - 1$ and X_i and Y_i be the bags with respecitvely vertices or edges as defined in Section 3.1. We say that a bag X_j is below a bag X_i (in the tree) if i is on the path from j to the root of T. Note that because G is the result of a number of merged tours, it is 2-connected and all vertices have a degree of at least two. The main idea of the algorithm is to find a series of disjoint paths and connect them together into a Hamiltonian

tour of minimum weight. A series of these paths start and end in a bag, and visit all vertices in bags below that bag in the tree. Such a series of paths is encoded using vertex degrees and a matching. Every vertex can have degree 0, 1 or 2. Vertices with degree 2 are already *used* in a path, vertices with degree 1 are *endpoints* of a path and vertices with degree 0 are *free*, so not yet used in any of the paths. For every pair of endpoints we have an edge $\{u, v\} : u, v \in V$ in the *matching* to mark which vertices are the endpoints of a path.

We now want to use the function $F(X_i, D_i, M_i)$ as the minimum total cost of the edges in a series of paths starting and ending in bag X_i, where D_i is a set of degrees for the vertices in X_i and M_i a matching on the vertices of degree 1. If there is an edge $\{u, v\} \in M_i$ then there should be a path that starts in u and ends in v. All vertices in X_i itself should have degrees as given in the degrees parameter, and the vertices that occur only in the bags below X_i in the tree should all be used (have a degree of 2) in one of the paths.

One way of looking at this is to see the set of degrees D_i as an instruction to a specific part of the tree (the bag X_i and all bags below in the tree) to deliver a set of edges, together forming a series of disjoint paths, such that all the degrees of vertices in this bag match with the degrees in D and that all vertices that occur only in bags below X_i in the tree are used. Of course, we do not just want any set of edges, we want the edges that can do it with the minimum cost. To get the cost of a tour through the entire graph, we can now call $F(X_0, D_0 = \{(v, 2), \text{for } v \in X_0\}, \emptyset)$. The root of the tree is the special case where we allow the paths to form a (single) cycle. Therefore if we give the instruction to the root bag X_0 to give us a set of edges such that all the vertices inside X_0 itself have degree 2 (as required by the set D_0) and all vertices in bags below the root have degree 2 as well (by specification of the function F), we actually give the instruction to find the weight of a set of edges that visits all the vertices of G in a single cycle. And because this set of edges should have minimum cost, this gives us the cost of the TSP tour through G.

Of course, for a good tree decomposition not all the edges are contained in a single bag. The main problem for a non-leaf bag X_i now is not how to find a subset of edges in Y_i that satisfy all the requirements for the degrees (and the matching), but how to divide the degrees over it's children so that they (recursively) can find the right edge sets that satisfy their part of the requirements. Selecting some edges from Y_i is mostly used to stitch the different paths from the child bags together so that the complete series of paths meets the requirements of D and M. An example of a path through a part of a tree decomposition is depicted in Figure 6.

To find the different ways of dividing the requirements for a bag over it's

$F(X_{top},\ \{(v_0,1),(v_1,1),(v_2,2),(v_3,2),(v_7,0)\},\ \{\{v_0,v_1\}\}),$
$F(X_{left},\ \{(v_0,1),(v_3,1),(v_6,2),(v_7,0)\},\ \{\{v_0,v_3\}\}),$
$F(X_{right},\ \{(v_2,1),(v_1,1),(v_{10},2)\},\ \{\{v_2,v_1\}\})$

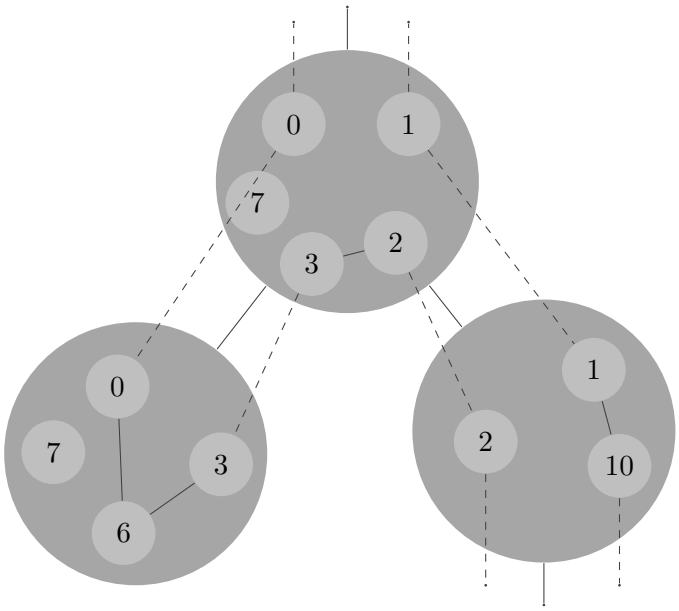

Figure 6: An example of a path through a part of the tree decomposition, together with the instructions per bag.

children, we have to go through these three steps:

1. Find the degrees
2. Find the matchings
3. Find the edge sets

Step 1 and 2 are done simultaniously. We first focus on the degrees and at the same time have to decide on the edges in the matching of the children. We will find them while we set the requirements on the degrees. We try all possible combinations of dividing the degrees per vertex. For a given vertex v, assuming we are required to give it a degree of 2, we first try to give it to each of the children. So for possibility p and child bag j we try to set the degree of v in $D_{p,j}$ to 2. Afterwards, assuming we have to give v a degree of at least 1, we try to use it as an endpoint coupled with each of the vertices in all of the child bags. So the degrees of two vertices, v and some other vertex u in $D_{p,j}$ are set to 1. We try this for all (valid) combinations of u and j. At this step we also add the edge $\{u, v\}$ to $M_{p,j}$. Finally we try not to assign v to any of the child bags, so that we can give its degree with one of Y_i's edges. Of course, vertices are only given to a child bag if it contains the vertex.

For the remaining degrees that are not handled by any of the child bags we calculate a subset E_p of Y_i. This is the third step. For non-leaf bags these edges mainly glue paths from the children together in the paths as required by the D_i and M_i parameters. For the leaf bags there are of course no child bags to delegate the degrees to so it all has to be solved using the bags own edges. Note that the edge set is not allowed to introduce cycles, so in particular two endpoints in an edge of a matching are not allowed to be connected. This is the reason why we need to give the matching as parameter to F, without it we do not have sufficient information to determine which vertices can and which vertices cannot be connected. The root bag is of course an exception, because there all paths are merged in a single cycle.

In summary, a vertex' degree can be satisfied by passing it on to one (or two) of the child bags or in the bag itself by choosing an edge from Y_i. Formally this becomes

$$F(X_i, D_i, M_i) = \min_{1 \leq p \leq P_i} \Big(\sum_{j \in W:\ \text{Parent}(j)=i} F(X_j, D_{p,j}, M_{p,j}) + \sum_{e \in E_p} c_e \Big)$$

for all P_i ways of dividing D_i and M_i into the $D_{p,j}$ and $M_{p,j}$ sets and the corresponding $E_p \subset Y_i$. The edges are not allowed to form a cycle. If no valid edge set is found, $F(X_i, D_i, M_i) = \infty$.

The overall algorithm then consists of a top down approach where we tabulate all entries for the function F, starting at the root and then recursively work downwards in the tree. Then the value of each table entry is finished bottem up as the recursion returns the values for the child entries.

4.2 Vehicle Routing

Let $G = (V, E)$ again be a simple graph with edge-weights c_e and $(T = (W, H), X)$ be the tree decomposition with width $k - 1$ and X_i and Y_i be the bags with respectively vertices or edges as defined in Section 3.1. The definition of the instruction-function $F(X_i, D_i, M_i, W_i)$ remains the same except that the edges $e \in M_i$ now have an edge weight $w_e \in W_i$ as well. Furthermore let M be the number of trucks that we have to use (and therefore te number of tours that we should find) and let C be the capacity of each truck. We denote the demand of all vertices $v_i \in V$ by d_i, where the demand d_0 of the depot vertex v_0 is equal to 0. Note that G, the resulting graph after merging the solutions of the heuristics, is not 2-connected as was the case for the TSP. This is because removing the depot vertex causes it to be disconnected. In our algorithm we still assume that all vertices have a degree of two or more. This is not always true, as there can be a tour from the depot to a single vertex v and then back, but this case is trivial, as the cost of that tour is $2 \cdot c_{(0,v)}$, and it can be excluded from the input to the function F. With the demand of a path, we mean the total demand of all the vertices on this path. When the matching edge of a path is in the instruction function F for the current bag X_i, we call this path a *main path*. Whenever two vertices in a main path are connected with a matching edge, rather than a normal edge from Y_i, this edge of the matching encodes the start and end vertices of a *subpath* through a child bag of X_i. For example, the bag X_{top} in Figure 6 has one main path and two subpaths.

The main idea of the algorithm is similar to that of the algorithm for the TSP. We try to find a series of paths that together form not one, but a set of M tours. The instruction of how these paths should be delivered by a bag is again done by specifying the degrees of the vertices and a matching. The degrees of all non-depot vertices can be between 0 and 2 like before, but the degree of the depot vertex can be any value between 0 and $2M$. Furthermore we add the restriction that the depot can only appear as an endpoint of a path. This restriction makes the final merging of the paths easier and effectively ensures that the depot is part of every tour. The greatest difference however lies in the fact that we need to restrict the total demand of the vertices in each of the tours to be at most C, the capacity of a truck. To do this we need to be able to

enforce this restriction on each of the paths separately in the parameters of the function F. As an edge in the matching relates one on one to a path, it comes natural to extended the edge with an edge-weight $w_{\{u,v\}}$, which denotes that the demand of path from u to v can be at most $w_{\{u,v\}}$. To get the solution for a VRP instance we call $F(X_0, D_0, M_0, W_0)$, where D_0 contains the maximum degree for all vertices, and M_0 contains M times the edge $\{0, 0\}$ with weight $w_{\{0,0\}}$ is equal to the capacity C of a truck.

The difficulty lies in how to divide the demand from the main paths in the current bag X_i, over the demands for the subpaths through the child bags. Note here that the focus has shifted from the bags to the individual paths through the bags. For the VRP we have the three steps from the TSP, but with an additional fourth step.

1. Find the degrees
2. Find the matchings
3. Find the edge sets
4. Find the demands per path

In the dynamic programming algorithm for the TSP the main difficulty lay at dividing the degrees over the bags, but we didn't care which vertex was included in which path. We only had to ensure that there was some path satisfying the degree of a vertex as was instructed; for as long as its endpoints coincided with the matching of course. Likewise, for the VRP algorithm, we first try to divide the degrees over the bags while initially ignoring the individual paths. As before we find the edges in the matchings as a side effect of finding the degrees, but at this point we don't know the weight of the edges yet. Once we divided both the degrees and found the matchings, we try, like before, to find an edge selection to support these choices of D_i and M_i. Only after we are done with the first three steps we try to find the weight of the matching. If a main path only has one subpath, this is quite simple. The demand of the subpath is the demand of the main path minus the demand of the vertices already visited inside the current bag. Each main path can have multiple subpaths however, and these subpaths are independend of the bags. Multiple subpaths from the same main path can be spread over multiple bags (but don't have to), and there can be subpaths from different main paths in the same bag. Because we don't know how much capacity is needed for the subpaths yet, we simply have to try all possibilities. In the example of Figure 6 the bag X_{top} might be allowed to have a demand of at most 6 (if all vertices have a demand of 1 and the child bag of X_{right} has no additional vertices not

in X_{right} itself this is just enough). We now need to try to give one bag a demand of 6 and the other 0 (and the other way around), then try to give one 5 and the other 1, then 4 and 2 and finally we need to try giving both bags a demand of 3.

Another effect of the capacity constraint is the choice of the edge sets $E_p \subset Y_i$, which are constructed after the degrees and matching are chosen. For the TSP algorithm we just have to find a path that uses all vertices that we instructed it to use, but it doesn't matter whether a vertex is used in one path or another. For the VRP placing a vertex in one path rather than in another might mean that the demand of that path is more than the capacity of a truck, leaving the solution invalid. As a consequence we do not have to evaluate just the cheepest edge selection, but all of them.

5 Results

We implemented the algorithms described in this thesis in C++ using the GNU GCC compiler on both Linux and Windows (using Cygwin in the latter case). We used both the source code and the binary for the Lin-Kernighan-Helsgaun algorithm, which is available for academic use on the website of Helsgaun [11]. All experiments ran on a laptop with an Intel i5-4210U CPU, with a clock rate of 1.70GHz.

5.1 Traveling Salesman Problem

In our test we considered some of the TSPLIB instances provided by the 8th DIMACS Implementation challange [21]. Specifically we tested the ones that have between 1000 and 2500 vertices. The LKH algorithm was limited to 100 trials and 10 runs each. This means that every tour was generated by evaluating 100 different starting vertices t_0 (see also Section 2.1), and that we ran the algorithm 10 times.

The results of our experiments with the LKH heuristic are displayed in Table 1. The name of the tour represents, apart from the TSPLIB name, also the number of vertices in the input graph. In the width column we give the width of the decomposition on the merged graph. The #tours column lists the number of LKH runs that added edges that where not used in any of the previous runs. The best and average columns represent the best and average values found by the LKH tours. The time displays the time it took for the heuristic to run. Note that the minimum degree heuristic is always completed in a few milliseconds, and does not influence the results. Note that in two

name	width	#tours	best	avg.	time
d1291	7	7	50801	50884	45s
d1655	5	8	62128	62130	48s
d2103	8	10	80460	80493	1m 58s
dsj1000	4	3	18660188	18667688	1m 31s
fl1400	14	10	20164	20167	50m 19s
fl1577	6	10	22263	22266	20m 24s
nrw1379	-	-	-	-	-
pcb1173	5	4	56892	56898	10s
pr1002	7	5	259045	259152	10s
pr2392	3	8	378032	378032	20s
rl1304	7	5	252948	253227	29s
rl1323	10	5	270199	270552	26s
rl1889	10	8	316549	316747	1m 20s
si1032	-	-	-	-	-
u1060	8	10	224094	224136	1m 33s
u1432	16	10	152970	152970	47s
u1817	12	10	57259	57277	59s
u2152	11	10	64253	64308	1m
u2318	66	10	234256	234256	3m 21s
vm1084	8	5	239303	239407	36s
vm1748	12	8	336739	336758	35s

Table 1: The results of the LKH algorithm

cases our program had a problem reading the input graph and aborted with an exception.

The results of the experiments with the dynamic program algorithm are displayed in Table 2. The name, width and #tours column are the same as before. The cost column displays the cost of the optimal tour through the merged graph. The time displays the time it took for the dynamic programming algorithm to complete. Note that in the cases where the width of the decomposition was larger than 10, we did not continue with the dynamic programming algorithm, as it would exceed the available running time. Another thing to note is that even though our program has double precision for the coordinates of the vertices, it truncates the cost of an edge to an integer value. There are different conventions about what type should be used for the computations [12]. In this case the LKH algorithm uses double precision for the edge costs. This causes the differences between the cost found by LKH and the

name	width	#tours	cost	time
d1291	7	7	50658	6s
d1655	5	8	61748	0s
d2103	8	10	79310	2m 39s
dsj1000	4	3	18659188	0s
fl1400	14	10	-	-
fl1577	6	10	21298	0s
nrw1379	-	-	-	-
pcb1173	5	4	56699	0s
pr1002	7	5	258829	0s
pr2392	3	8	377553	0s
rl1304	7	5	252774	1s
rl1323	10	5	270012	30m 18s
rl1889	10	8	316240	1m 25s
si1032	-	-	-	-
u1060	8	10	223581	26s
u1432	16	10	-	-
u1817	12	10	-	-
u2152	11	10	-	-
u2318	66	10	-	-
vm1084	8	5	239019	27s
vm1748	12	8	-	-

Table 2: The results of the dynaming algorithm

cost as calculated by the dynamic programming to be larger than it actually is. The new tours found by the dynamic programming do improve the LKH tours in most cases, but the difference is somewhat less as that it seems.

Interesting to see is that there is no correlation between the running times of the heuristic and the running times of the dynamic programming algorithm. The exponential behaviour of the decomposition width is clearly visible however. Usually the time to run LKH dominates the time to run the dynamic programming algorithm, until the width of the decomposition reaches 10.

5.2 Vehicle Routing Problem

To test how the tour merging approach works for the VRP, we downloaded several instances for the Capacitated VRP from the website of the NEO research group [22]. The instances vary from having 32 vertices to 80 vertices in total. The number of trucks vary from 5 to 10 trucks. The capacity of a

name	n	width	#savings	#sweep	best	avg.
A-n32-k5	32	6	1	9	832	973
A-n32-k5	19	6	0	6	535	656
A-n32-k5	12	4	0	8	460	478
A-n33-k5	33	5	0	9	715	757
A-n33-k5	13	4	0	6	413	435
A-n33-k6	14	5	1	4	330	450
A-n32-k5(c)	13	4	0	8	444	474
A-n32-k5(c)	14	5	0	8	448	485
A-n32-k5(c)	15	5	0	7	448	500

Table 3: The results of the savings and sweep algorithms

truck is equal for all the instances and is set to 100. Even though these where amoung the smallest instances available, even the smallest ones were aborted due to too long running times.

Because of the limited timeframe and computing power we decided to modify the instances, in search of what our algorithm can handle. First we chose to limit the number of vertices (and accordingly the number of trucks necessary for the demand of the cities). We did this by simply removing the vertices after the n-th vertex. This showed us that our algorithm was able to handle instances with a limit of approximately 13 vertices. Apart from that we decided to change the capacity parameter from 100 to 10 for some of the instances, and round down the demands of the vertices accordingly. This allowed our program to handle a few more vertices, but not very much. The cases where we changed the capacity are labelled with a (c) behind the name.

The results of the computations concerning the heuristics are displayed in Table 3. In the name column we display the name of the instance, which includes the original number of vertices and the number of trucks used. In the next column we display the number of vertices that we limited the computation to. The width column shows the width of the decomposition again. The #savings and #sweep columns show the number of runs in which the algorithms added new edges to the set of tours. The best and average columns show the best and average costs of set of tours found by the heuristics. Noteworthy is that in all the cases where the savings algorithm found an acceptable tour it had the best result. Running times are not displayed as all heuristics, including the minimum degree heuristic used to compute the tree decomposition, ran in less than a second.

The results of the computations by the dynamic programming algorithm

name	n	width	#tours	best	avg.	cost	time
A-n32-k5	32	6	10	832	973	-	8h (aborted)
A-n32-k5	19	6	6	535	757	-	4h (aborted)
A-n32-k5	12	4	8	460	478	406	2m 26s
A-n33-k5	33	5	9	715	656	-	8h (aborted)
A-n33-k5	13	4	6	413	435	406	19m 4s
A-n33-k6	14	5	5	330	450	-	8h (aborted)
A-n32-k5(c)	13	4	8	444	474	436	2s
A-n32-k5(c)	14	5	8	448	485	441	6s
A-n32-k5(c)	15	5	7	448	500	441	19m 56s

Table 4: The results of the dynamic programming algorithm

for the VRP are displayed in Table 4. The name, amount of vertices, width, best and average columns are the same as before. The #tours column contains the combined number of merged sets of tours by the savings algorithm and the sweep algorithm. The cost column contains the cost of the tour as calculated by the dynamic programming algorithm. The time column contains the running time of the dynamic programming algorithm. Overall the cost of the tours given by heuristics is improved, but at a huge cost in complexity of the code and running time. Note that all of the larger instances where aborted after a given number of hours.

6 Conclusion

In this thesis we have looked at the tour merging technique introduced by Cook and Seymour [1], for the Traveling Salesman and Vehicle Routing problems. We have seen that merging a set tours created by heuristics improves the overall solutions. In the case of the TSP this is already known, as it was shown by Cook and Seymour before. In the case of the VRP this is new, but it is not very helpful. There are better heuristics available than the ones we used, but these are often rejected because of the complexity of the code and running times (and also because they are not generally applicable). Our algorithm is both very complex in code as slow in running time and not applicable for large instances, and thus is not an effective alternative for more complicated heuristics.

References

[1] Cook, W., & Seymour, P. (2003). *Tour merging via branch-decomposition.* INFORMS Journal on Computing, 15(3), 233-248.

[2] Rego, C., Gamboa, D., Glover, F., & Osterman, C. (2011). *Traveling salesman problem heuristics: leading methods, implementations and latest advances.* European Journal of Operational Research, 211(3), 427-441.

[3] Pesch E., Glover, F. (1997). *TSP ejection chains.* Discrete Applied Mathematics, 76(1), 165-181.

[4] Rego, C. (1998). *Relaxed tours and path ejections for the traveling salesman problem.* European Journal of Operational Research, 106(2), 522-538.

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

[6] Lin, S. (1965). *Computer solutions of the traveling salesman problem.* Bell System Technical Journal, The, 44(10), 2245-2269.

[7] Christofides, N., & Eilon, S. (1972). *Algorithms for large-scale travelling salesman problems.* Operational Research Quarterly, 511-518.

[8] Lin, S., & Kernighan, B. W. (1973). *An effective heuristic algorithm for the traveling-salesman problem.* Operations research, 21(2), 498-516.

[9] Helsgaun, K. (2000). *An effective implementation of the Lin-Kernighan traveling salesman heuristic.* European Journal of Operational Research, 126(1), 106-130.

[10] Helsgaun, K. (2009). *General k-opt submoves for the Lin-Kernighan TSP heuristic.* Mathematical Programming Computation, 1(2-3), 119-163.

[11] Helsgaun, K. *LKH* `http://www.akira.ruc.dk/~keld/research/LKH/`

[12] Laporte, G., Gendreau, M., Potvin, J. Y., & Semet, F. (2000). *Classical and modern heuristics for the vehicle routing problem.* International transactions in operational research, 7(4-5), 285-300.

[13] Cordeau, J. F., Gendreau, M., Laporte, G., Potvin, J. Y., & Semet, F. (2002). *A guide to vehicle routing heuristics.* Journal of the Operational Research society, 512-522.

[14] Clarke, G. U., & Wright, J. W. (1964). *Scheduling of vehicles from a central depot to a number of delivery points.* Operations research, 12(4), 568-581.

[15] Wren, A., (1971). *Computers in Transport Planning and Operation.* Ian Allan, London.

[16] Wren, A., Holliday, A., (1972). *Computer scheduling of vehicles form one or more depots to a number of delivery points.* Operational Research Quarterly 23, 333-344.

[17] Gillett, B. E., & Miller, L. R. (1974). *A heuristic algorithm for the vehicle-dispatch problem.* Operations research, 22(2), 340-349.

[18] Bodlaender, H. L., & Koster, A. M. (2010). *Treewidth computations I. Upper bounds.* Information and Computation, 208(3), 259-275.

[19] Markowitz, H. M. (1957). *The elimination form of the inverse and its application to linear programming.* Management Science, 3(3), 255-269.

[20] Arnborg, S., Corneil, D. G., & Proskurowski, A. (1987). *Complexity of finding embeddings in ak-tree.* SIAM Journal on Algebraic Discrete Methods, 8(2), 277-284.

[21] Johnson, D., McGeoch, L., Glover, F. & Rego, C. *8th DIMACS Implementation Challenge: The Traveling Salesman Problem* `http://dimacs.rutgers.edu/Challenges/TSP/download.html`

[22] NEO research group *Vehicle Routing Problem* `http://neo.lcc.uma.es/vrp/solution-methods/`