Tour merging via tree decomposition

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

Mattias Beimers - 3672565

July 14, 2015

Abstract

A hybrid approach between heuristics and exact solutions for TSP and VRP using tree decompositions.

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 nescessarily 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 publishing, the solutions found by Cook and Seymour improved on the best known results for instances with almost 25000 vertices [**?**]. Since then other heuristics have improved massively and outperform the approach by Cook and Seymour [7]. 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 (TODO: OR NOT, WAIT FOR RESULTS!).

The paper 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 treedecomposition 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 [6, **?**], most notably the implementation of Helsgaun [7]. 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 [8]. We will discuss the original Lin-Kernighan heuristic in Section 2.1 and the modifications on the original algorithm by Helsgaun's implementation in Section 2.2.

For the Vehicle Routing Problem the currently best heuristics are tabu search algorithms [**?**, **?**]. 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 [**?**, **?**]. 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 **??** and Section **??**.

2.1 Lin Kernighan

The Lin-Kernighan heuristic [5] 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 criterium (e.g. the solution is good enough, the pool of initial tours is deplenished, or a time limit is reached).

The interseting 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 [2] and Lin [3], respecively in 1958 and 1965. k-opts with higher values of k have been tried as well, for example by Christofides and Eilon [4], 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 preceeded 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, untill 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 describes 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 [6]. We omit them here as they are not essential to this thesis. For the exact details we refer to the original paper [5].

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 speeds up the overall algorithm significantly. Note that in order to not bias the solution too much, we only use this restriction for $i \geq 4$.

In the second optimization 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 [6, 7] is based on the algorithm of Lin and Kernighan, but improves on several points.

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 distance to a minimum 1-tree, called *α-nearness*, is a better estimation. A 1-tree for a graph $G = (V, E)$ is a spanning tree on the

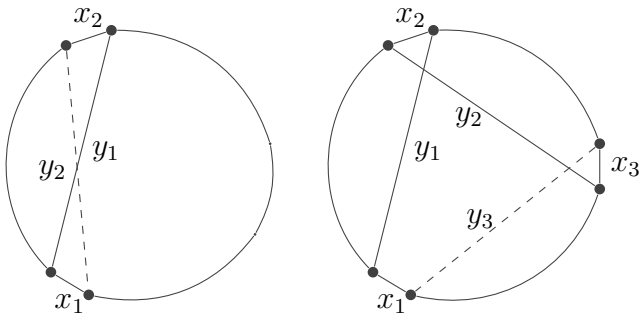

Figure 1: An example of a 3-opt move, as constructed by the Lin-Kernighan heuristic. Note that vertices are displayed in a circle in the order they appear in the original tour.

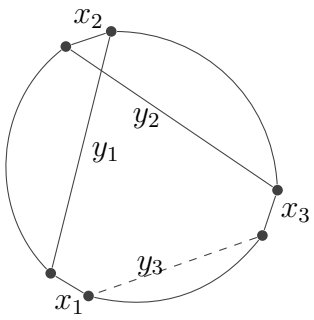

Figure 2: An example where the wrong choice for x_3 is made, resulting in two disjunct tours.

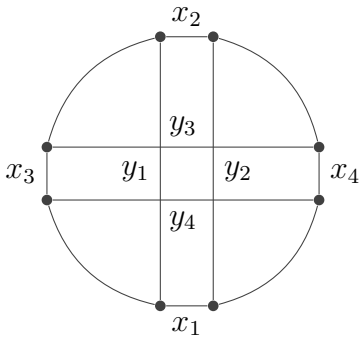

Figure 3: The double bridge move

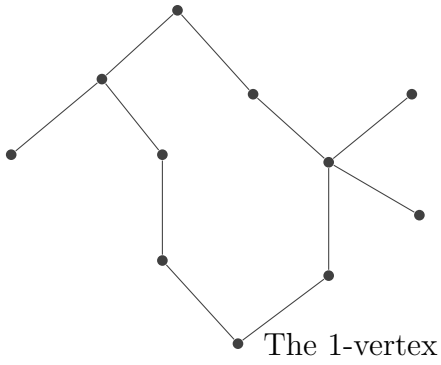

Figure 4: A 1-tree

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 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. 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 [6] 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 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 [7]. 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 unnescessary. 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, but Helsgaun doesn't go into much detail on them, so for these we refer to Helsgaun's paper [7].

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 defenition 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

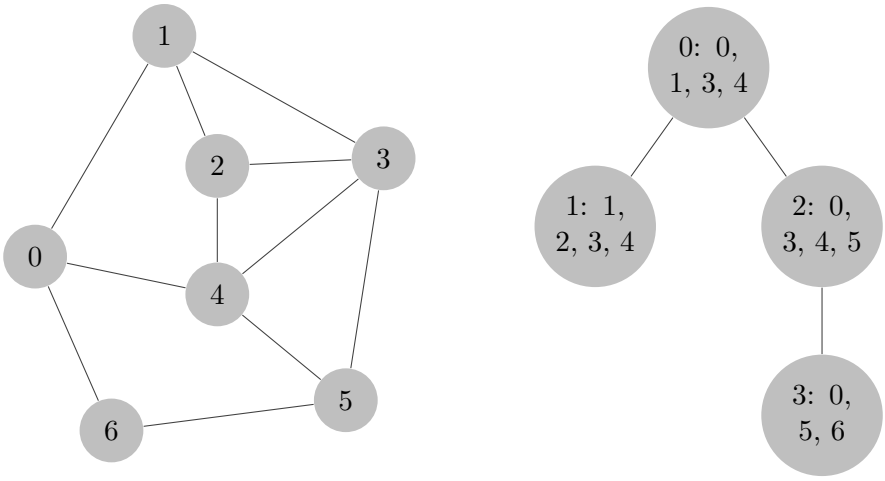

Figure 5: An example of a graph G and its tree decomposition.

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 is an NP-Hard problem [**?**], so finding an optimal decomposition in reasonable time is infeasable unless P=NP. We do not nescessarily need a tree decomposition of optimal width, we just need the width to be sufficiently small so that our DP algorithm runs fast enough. Therefore, we compute our tree decomposition with a heuristic.

Bodlaender and Koster [9] evaluated a number of construction heuristics. We chose to use the Minimum Degree Heuristic, originally designed by Markowitz [10], 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. 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 untill 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.

As a small optimization step specific to the TSP and VRP we remove the last two vertices from W. The corresponding bags only contain 1 or 2 vertices and are subsets of another bag. We can do this because the graph G is obtained by merging a set of tours, and therefore every vertex is guaranteed to have at least two neighbours. When we are solving the VRP, we also add the depot vertex to every bag.

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 define the function $F(X_i, D, M)$ to be the minimum total cost of the edges in a series of paths starting and ending in bag X_i, where D is a set of degrees for the vertices in X_i and M a matching. If there is an edge $\{u, v\} \in M$ 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).

One way of looking at this is to see the set of degrees D 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 ofa 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.

To find the different ways of dividing the requirements for a bag over it's children, we focus solely on the degrees. We also have to decide on the edges in the matching of the children, but we will find them as 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. For non-leaf bags these edges mainly glue paths from the children together in the paths as required by the D and M 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

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

for all P_i ways of dividing D and M into the $D_{p,j}$ and $M_{p,j}$ sets and the corresponding $E_p \subset Y_i$. If no valid edge set is found, $F(X_i, D, M) = \infty$.

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 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. 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, and the demand d_0 of the depot vertex v_0 is equal to 0. Note that G is not 2-connected as was the case for the TSP, as removing the depot vertex causes it to be disconnected, but it still holds that all vertices have a degree of two or more.

The main idea of the algorithm is the same as the algorithm for the TSP. We try to find a series paths that together form set of 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 one of $\{0, 1, 2\}$ like before, but the degree of the depot vertex can be any value between 0 and $2M$. Furthermore we have a restriction that the depot can only appear as an endpoint of a path. The matching-edges used to encode a path are extended with an edge-weight $c_{u,v}$, which denotes that the total demand of vertices on the path from u to v can be at most $c_{u,v}$. While they look simple, these changes complicate the algorithm quite a lot.

TODO: - Instead of considering child bags, consider all paths in all childs. - Try all different demand combinations on these paths. - Changed edge selection. - Merge paths in root bag. - Extra requirements (depot vertex only as endpoints, ...).

4.3 Speed

Although DP running time upperbounds of $O(n3^k2^{k^2}) - TODO???$ and $O(Mn3^k2^{\cdots})$ are terrible, in practice these limits are never reached. This is because edges... TODO

5 Results

Todo

6 Conclusion

Todo

References

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

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

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

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

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

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

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

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

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

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