Universiteit Utrecht

Faculteit Bètawetenschappen

Algorithms for the travelling salesman problem

Autor:

Isabel Droste
5492335

Supervisor:

Prof. Dr. R.H. Bisseling

Bachelor Thesis

Mathematics

June 2017

Contents

1 Introduction

Imagine a salesman visiting a number of cities. He begins in his hometown and next wants to visits each city on a list exactly once. Finally, he returns to his home town. He wants to do this in such a way that the total distance covered is minimised. This is known as the travelling salesman problem (TSP).

The problem sounds very simple, yet the solution is much more difficult. For almost 100 years mathematicians have tried to solve it. The beauty of the problem lies both in its simple formulation and the visualisation of the solutions. We will be mainly interested in instances of TSP that represent actual cities and the distances between them as travelled by car. We will illustrate our results using several maps.

One might think the problem can be solved by simply calculating the length of each possible tour and then selecting the shortest. But this is only possible for small instances since the number of different tours is very large. For only 14 cities there are more than 3 billion possible tours. For most instances, checking all possibilities cannot be done in a reasonable amount of time. Therefore, we need algorithms to solve the problem.

TSP is one of the most intensely studied problems in combinatorial optimisation and it is proven to be $\mathcal{NP}$-hard. Travelling salesmen in the 19th century were already known with the problem. In the 1930s, mathematicians became interested in it. Since then, better and better algorithms were developed. In 1954, the largest solved instance consisted of 49 cities in the USA. This was a great achievement at that time. In 2004, the shortest tour passing through all 24,978 cities in Sweden was found (Figure 2). The current record is an instance of 85,900 locations on a computer chip. It was solved in 2009 by Appelgate et al.[1].

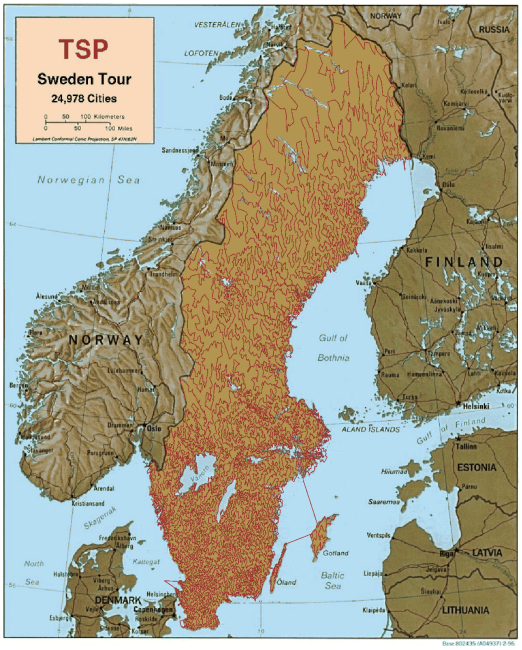

Figure 2: *The optimal tour through all 24,978 cities of Sweden. It has a length of approximately 72,500 km.*
Source: http://www.math.uwaterloo.ca/tsp/sweden/index.html

The travelling salesman problem has many applications. Some examples are package delivery, picking up children with a school bus, order picking in a warehouse and drilling holes in a printed circuit board.

In this thesis we will study algorithms for TSP. These algorithms can be divided into exact and heuristic algorithms. An exact algorithm guarantees to find the shortest tour. A heuristic algorithm will find a good tour but it is not guaranteed that this will be the best tour. The advantage of an heuristic algorithm is the shorter running time which makes it more suitable for large instances.

We will treat one exact algorithm and one heuristic algorithm. The exact algorithm is called branch and bound. It is based on the idea of dividing the set of all tours into groups of tours that have common properties. In this way we do not need to consider each individual tour. We will study the heuristic algorithm Ant Colony Optimisation. This algorithm is based on the behaviour of an ant colony when finding the shortest route to food sources. We will present an implementation of both algorithms in Python.

The thesis is structured as follows. In Chapter 2 we will give a formal definition of the problem and prove that it is $\mathcal{NP}$-hard. Chapter 3 and 4 are devoted to the exact algorithm. Chapter 5 is about the heuristic algorithm. Finally, chapter 6 contains the conclusion.

2 The travelling salesman problem

We are interested in instances of the travelling salesman problem that represent cities and the distances between them as travelled by car. We assume that the distance between two cities is the same when travelled in the opposite direction. In reality there might be a small difference due to for example one-way roads but we assume these differences are small enough and assume that the distance matrix is symmetric. The variant of TSP we will study can then be defined as follows.

Definition 2.1. *Let $G(V, E)$ be a complete, simple graph. The set V contains the vertices $1, 2, ..., n$ and the set of edges E contains all the unordered pairs (i, j) with $i, j \in V$. Each edge (i, j) has length $w_{ij} \geq 0$. The problem is to find a set $T \subset E$ such that the edges in T form a tour of minimal length. This length is defined as $\sum_{(i,j)\in T} w_{ij}$. The edges form a tour when for every $s, t \in V$ there exists a path from s to t using edges only from T and every $i = 1, .., n$ is present in exactly two edges from T.*

The number of possible tours is very large. Assume there are n cities. We can choose the starting city arbitrarily. For the next city there are $n - 1$ options. Once this city has been chosen there are $n - 2$ unvisited cities left to choose from, and so forth. This means the number of different tours is $\frac{1}{2}(n-1)!$, the factor $\frac{1}{2}$ coming from the fact that it does not matter in what direction we travel a tour.

Proof of $\mathcal{NP}$-hardness
We will now prove that TSP is $\mathcal{NP}$-hard given that the problem Hamiltonian cycle is $\mathcal{NP}$-hard. This problem is in the list of $\mathcal{NP}$-hard problems of Gary & Johnson [5] and is defined as follows

Hamiltonian cycle problem
Given a graph $H(V, E)$. Does this graph contain a tour $T \subset E$ that visits every vertex $v \in V$ exactly once?

The decision variant of TSP is defined as follows:

Decision variant of TSP
Given a complete graph $G(V, E)$, the lengths $w_{ij} \geq 0$ for every edge $(i, j) \in E$ and a value $M \in \mathbb{R}$. Does the graph contain a tour of length L such that $L \leq M$?

Suppose we have a solution to the decision variant. This solution consists of a set T of edges that are in the tour. This set can be stored in polynomial space. To check the solution we have to check that each vertex is visited exactly once and that there are no subtours. Next we have to sum w_{ij} for all $(i, j) \in T$ and compare the result to M. This can be done in polynomial time, so the decision variant belongs to $\mathcal{NP}$.

Next we prove that the decision variant is $\mathcal{NP}$-complete by reduction from the HAMILTONIAN CYCLE PROBLEM. Let $H(V, E)$ be an instance of HAMILTONIAN CYCLE. We construct the particular instance of TSP with the full graph $G(V, E')$ where E' contains the unordered pairs (i, j) for all $i, j \in V$ and the lengths

$$w_{ij} = \begin{cases} 0 & (i,j) \in E \\ 1 & \text{otherwise} \end{cases}$$

Further we choose $M = 0$. This reduction can be done in polynomial time. We now show that this instance of TSP is true if and only if HAMILTONIAN CYCLE is true. Suppose there exists a tour in G of length at most $M = 0$. Let $T \subset E'$ be the set of all edges that are in the tour. Because all lengths are non-negative if follows that $w_{ij} = 0$ for all edges $(i, j) \in T$. This means that $T \subset E$. The edges in T now form a Hamiltonian cycle in $H(V, E)$. Now suppose there exists a Hamiltonian cycle in H. Let T be the set of edges in the cycle. Then $w_{ij} = 0$ for all $(i, j) \in T$. Then the edges in T form a tour of length 0. So the decision variant is $\mathcal{NP}$-complete and therefore TSP is $\mathcal{NP}$-hard. □

Self-intersection

When the lengths w_{ij} form a metric, it can be proven that a tour that self-intersects, cannot be optimal. Consider Figure 3. Left we see a tour where edges AD and BC intersect. The point of intersection is called M. Right these edges have been replaced by AC and BD. From the triangle inequality it follows that $|AC| \leq |AM| + |CM|$ and $|BD| \leq |BM| + |DM|$ and so the right tour is shorter than the left tour. The lengths that we will use, distances as travelled by car, satisfy the definition of a metric (if we assume the distances to be symmetric) and so this condition holds.

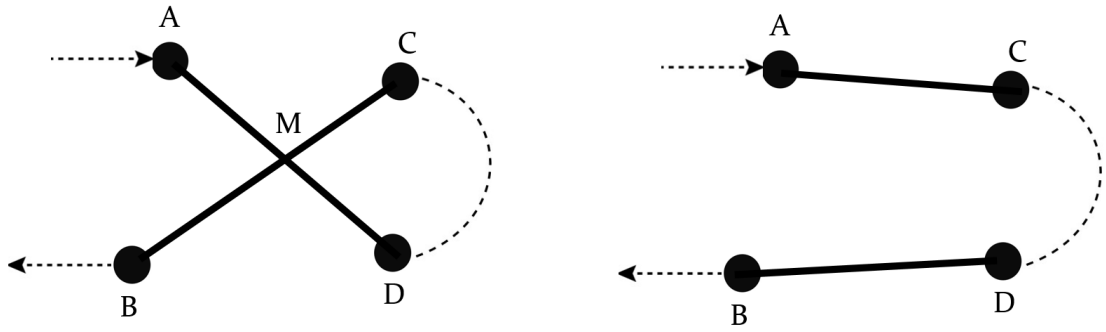

Figure 3: *Self-intersecting tour (left) and non-self-intersecting tour (right)*

3 Branch and bound algorithms: the theory

In this section we will discuss the theory behind branch and bound (B&B) algorithms. The term branch and bound does not refer to one single algorithm, it is a framework that is used to design algorithms for problems in combinatorial optimisation. It was first described by Land & Doig in [9]. We will explain B&B in a general context but we keep TSP in the back of our mind and use it to make things more specific.

Suppose we have a set of feasible solutions to a certain problem in combinatorial optimisation. Each solution has an objective value and we want to find the solution that minimises that objective value. For TSP this objective value is the length of the tour. The idea is to split up the set of all solutions into two or more disjoint subsets. Each of these subsets can also be divided into subsets and so forth. The splitting is based on a constraint that the solutions in the subset have to satisfy. This process is called branching. In this way we create a tree in which each node contains a subset of all solutions. The root node has no constraints and contains all solutions. Each child node inherits the constraints of their parent and also has an additional constraint. In the context of TSP the constraints could be the presence of a certain edge in the tour. A node then has two children: the left child represents all tours that do contain a certain edge and the right child represents all tours that do not contain the edge. At a certain point in the tree the constraints define one single solution. These are the leaves of the tree: no further constraints can be imposed.

For each node we can, in a way that depends on the problem, calculate a lower bound on all solutions present in the node. When a lower bound is higher than the value of some known solution, we do not need to search further in that part of the tree. We can be sure the optimal solution will not be found in this node and the node can be pruned. When a new solution is found at a leaf of the tree, we compare this to our current best solution and keep the best one. In this way we keep finding better solutions and more nodes can be pruned.

The process ends when there are no more nodes to explore. We can then be sure our current best solution is the optimal solution. This can be shown as follows. Consider the part of the tree that has been generated when the algorithm ends. Suppose that somewhere in the tree there is a better solution than our current best solution. This solution must be in any of the nodes at the end of a branch of the explored part of the tree. These nodes are either leaf nodes or pruned nodes. When they are a leaf node, they have at some point been compared to the best solution at that time but have been discarded so they cannot be smaller than the current best solution. When they are a pruned node, the lower bound for this node must be higher than our current best solution. This means we can be sure no better solution exist in the tree.

A B&B algorithm has three main components: branching, bounding and the selection of the next node. In these components, important decisions have to be made that influence the performance of the algorithm. We now briefly discuss these three components.

1. **Branching**: We have to decide which constraints we impose on the children of a node.

2. **Bounding**: We have to find a way to determine a lower bound for all solutions in a node. The quality of this lower bound is essential for the performance of the algorithm. On the one hand we want the lower bound to be as high as possible. In the ideal situation the bound would equal to the best solution in the node. In that way, more nodes will be pruned and the tree can remain small. On the other hand, the computational costs of a tight lower bound can be high.

3. **Selection of the next node**: After creating the children and computing their lower bound, we have to choose which of these children is the next node to process. This determines in what order we traverse the tree. This choice could for example depend on the lower bound of the children.

4 Exact solutions to TSP

We will now apply our theory of branch and bound to find exact solutions to small instances of the travelling salesman problem. The constraints that we impose on the nodes consist of edges that have to be present in the tour and edges that are not allowed to be present. We will call these the included and excluded edges, respectively. After two children are generated, the next node to process will be the child with the smallest lower bound because it is more likely to find better tours in that node which will result in more nodes to be pruned. For both the branching and the bounding step we will discuss two different strategies. These can be combined to four different algorithms. We present an implementation of these four algorithms in Python. We apply these to small instances of TSP and compare the performance of the four algorithms.

4.1 Strategies for branching and bounding

4.1.1 Branching

Lexicographic order
The simplest branching strategy is to impose the constraints in lexicographic order. This means we start with edge $(1, 2)$, next $(1, 3)$, $(1, 4)$ en so on. We will refer to this branching strategy as LG.

Increasing length
Another way to branch is in the order of increasing length of the edges. This means the next constraint is always the smallest edge that is not already included or excluded. The idea behind this is that the smaller edges are more likely to be present in good tours. In this way we will earlier find short tours which will result in more nodes to be pruned. We will refer to this branching strategy as IL.

After imposing a new constraint we have to check if other edges also have to be included or excluded. We do this according to the following rules as described by Wiener in [11]:

- When all but two edges adjacent to a vertex are excluded, those two edges have to be included as otherwise it would be impossible for a tour to exist.
- When two edges adjacent to a vertex are included, all other edges adjacent to this vertex have to be excluded.
- When including an edge would complete a subtour with other included edges, this edge has to be excluded

4.1.2 Bounding

Simple lower bound
One way to determine a lower bound is described in [11]. Suppose we have a tour that satisfies certain constraints. For each vertex i, let (a_i, i) and (b_i, i) be the shortest edges adjacent to this vertex that can be present in the tour given the constraints. Let (k_i, i) and (l_i, i) be the edges that are present in the tour. Then the length L of the tour is equal to

$$L = \frac{1}{2}\sum_{i=1}^{n}(w_{k_i i} + w_{l_i i})$$

and this is no less than summing over the smallest possible edges:

$$\frac{1}{2}\sum_{i=1}^{n}(w_{a_i i} + w_{b_i i}) \leq \frac{1}{2}\sum_{i=1}^{n}(w_{k_i i} + w_{l_i i}) = L$$

This gives us a lower bound on any tour. We will refer to this lower bound as SB.

Lower bound using a 1-tree
Another way of calculating the lower bound is by using a 1-tree. This was first described by Held and Karp in [7]. A 1-tree is a variant of a minimum spanning tree.

Definition 4.1. *Let $G(V, E)$ be a connected, undirected, weighted graph. Then a spanning tree is a subset $E' \subset E$ such that for every $s, t \in V$ there exists a path from s to t using edges only in E' and such that the edges in E' form no cycles. A minimum spanning tree (MST) is a spanning tree such that the sum of the weigths of the edges in E' is minimized.*

When the weights of all edges are distinct, the MST is unique. When n is the number of vertices in the graph, the number of edges in the MST is equal to $n - 1$. To find the minimum spanning tree, we can use Kruskal's algorithm [8].

Kruskal's Algorithm
Let $G(V, E)$ be a graph with $|V| = n$. Include the edges in order of increasing length unless the edge forms a cycle with the already included edges. Stop when a spanning tree is achieved. This is, after adding $n - 1$ edges.

Now a 1-tree is defined as follows

Definition 4.2. *Let $G(V, E)$ be a connected, undirected, weighted graph with $V = \{1, 2, ..., n\}$. Then a 1-tree is a spanning tree of the vertices $\{2, ..., n\}$ together with two edges adjacent to vertex 1. A minimum 1-tree is a 1-tree of minimal length.*

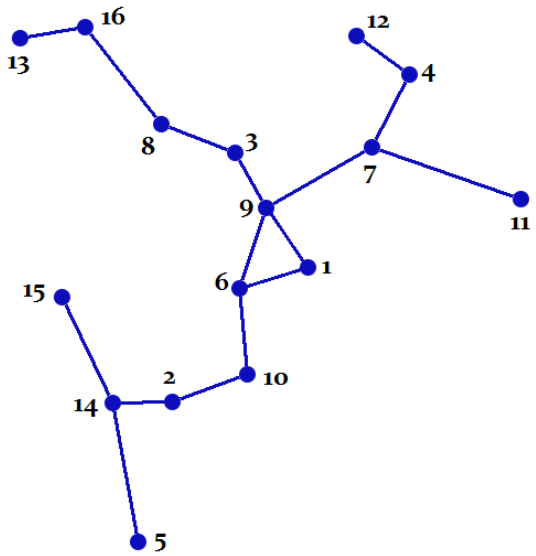

Figure 4: *Minimum 1-tree for a graph with 16 vertices.*

A minimum 1-tree can be found by adding the two shortest edges adjacent to vertex 1 to the MST of the edges $\{2, ..., n\}$. An example of a minimum 1-tree is shown in Figure 4.

A minimum 1-tree can be used as a lower bound for TSP as follows. Let $G(V, E)$ be a graph as in definition 4.2 and let $T \subset E$ be a tour in G. Remove the edges adjacent to vertex 1. Because the remaining edges form a spanning tree of the vertices $\{2, ..., n\}$, it follows that T is a 1-tree. This means that the set of all tours is a subset of the set of 1-trees. Then the length of the minimum 1-tree is a lower bound on any tour. We will refer to this lower bound as OT (1-tree).

4.2 Implemention in Python

We just discussed two strategies for branching (LG and IL) and two strategies for bounding (SB and OT). Combining these, the following four different algorithms can be made: LG-SB, LG-OT, IL-SB and IL-OT. We implemented these algorithms in Python. The code is listed in Appendix A. Note that in this text the first city has number 1, but in the code we start counting at 0. The code consists of two classes: `Node` and `TSP`. The former represents a node in the tree and the latter represents an instance of TSP. The four algorithms are very similar and they only differ in one or two methods. We first show how the code can be used. Next we present the general structure of the two classes and then discuss the points at which the algorithms differ.

4.2.1 Usage of the code

To use the code we need a distance matrix. The (i, j)th element of this matrix is the distance between city i and city j. The distance matrix has to be symmetric. To create an instance of TSP we also have to give the number of cities. This number n has to be smaller than or equal to the size of the distance matrix. When it is smaller, we only consider the first n cities. In this way it is easy to change the size of the instance without changing the distance matrix. When the algorithm is running and it finds a better tour, this will be printed. Below you see an example of the use of the code with 7 cities.

```
>>> distances =
    [[0,141,118,171,126,69,158],
    [141,0,226,34,212,208,82],
    [118,226,0,232,56,107,194],
    [171,34,232,0,200,233,63],
    [126,212,56,200,0,105,145],
    [69,208,107,233,105,0,212],
    [158,82,194,63,145,212,0]]
>>> T = TSP(7,distances)
>>> T.findSolution()

Found better tour:  0-4-2-6-1-3-5-0 of length 794 km
Found better tour:  0-4-2-6-3-1-5-0 of length 750 km
.
Found better tour:  0-5-2-4-3-1-6-0 of length 706 km
Found better tour:  0-5-2-4-1-3-6-0 of length 699 km
Found better tour:  0-3-1-6-4-2-5-0 of length 664 km
Found better tour:  0-1-3-6-4-2-5-0 of length 615 km
------------------------------------------------------------
The shortest tour is:  0-1-3-6-4-2-5-0
It has a length of 615 km
Found in 0.19409145514509593 seconds
Best tour was found after:  0.19140656751036353 seconds
Number of nodes created:  135
Number of nodes pruned:  62
```

4.2.2 The class `Node`

The class `Node` respresents a node in the search tree. Its most important variables are `self.lowerbound` and `self.constraints`. The former has the value of the lower bound for the node and is determined by the method `computeLowerBound`. The variable `self.constraints` is a matrix that contains the information about the constraints of that node. Let c_{ij} be the element of this matrix in the i-th row and the j-th column, then

$$c_{ij} = \begin{cases} 0 & \text{if edge } (i,j) \text{ is excluded} \\ 1 & \text{if edge } (i,j) \text{ is included} \\ 2 & \text{if edge } (i,j) \text{ is neither included nor excluded} \end{cases}$$

For the root node there are no constraints except for the exclusion of loops so $c_{ij} = 2$ for all $i \neq j$ and $c_{ij} = 0$ for all $i = j$. When a child node is created, the next constraint is determined by the method `next_constraint`. Then the constraint matrix is determined by the method `determine_constr` based on the constraint of its parent and the extra constraint. This is done according to the rules described on page 7. These rules are implemented in the methods `removeEdges` and `addEdges`. The method `isTour` determines whether a node represents a complete tour. This is done by checking if from every vertex there are exactly two included edges.

4.2.3 The class `TSP`

The most important method in `TSP` is `BranchAndBound`. This method creates and prunes the nodes. This happens in a recursive way. The method can be summarized as follows

1. First we check whether the current node represents a tour. If so, we compare the length of the tour to the current best tour and keep the shortest.
2. When the node is not a tour, we create two children. When the lower bound of a child is greater than the length of the current best tour, the child is pruned.
3. When one child is pruned and the other is not, we call `BranchAndBound` with the unpruned child
4. When both children are unpruned we call `BranchAndBound` with the child with the smallest lower bound. Next we check again if the other child has to be pruned since a smaller tour may have been found. If the other child does not have to be pruned, we call `BranchAndBound` with this child.

4.2.4 Differences between the four algorithms

The main code as shown in Appendix A represents algorithm LG-SB. In the two algorithms that use OT as a lower bound, the method `computeLowerBound` is different. Here Kruskal's Algorithm is used to construct a minimum spanning tree of the vertices $2, .., n$. The method `next_constraint` is different in the two algorithms that

use IL as branching strategy. They use the variable `self.allSortedEdges`. This is a list that contains all edges in order of increasing length. The alternative versions of `computeLowerBound` and `next_constraint` are also shown in appendix A below the main code.

4.2.5 Brute force algorithm

We also implemented a brute force algorithm which we will use to check our solutions. This algorithm computes the lengths of all possible tours and selects the shortest. We use the standard library function `itertools.permutations` to generate all permutations of the vertices. The code is also listed in appendix A.

4.3 Results

We applied our algorithms to small instances of TSP. We used the following 14 cities in the Netherlands.

1. Arnhem
2. Assen
3. Den Haag
4. Groningen
5. Haarlem
6. 's Hertogenbosch
7. Leeuwarden
8. Lelystad
9. Maastricht
10. Middelburg
11. Utrecht
12. Zwolle
13. Amsterdam
14. Enschede

We define the distance between two cities as the smallest distance in km from central station to central station as travelled by car. We used Google Maps to determine this. We rounded the distances to whole kilometres. Sometimes the distance between two cities was different when travelled in the opposite direction. These differences were small, at most a few kilometres. When this was the case, we took the smallest of the two. The distances can be found in Appendix B.

We tested the algorithms on the first n cities for $4 \leq n \leq 14$. The best tours can be found in Table 1 and the best tour for all 14 cities is shown in Figure 5. The maps that we use are modified versions of blank maps from *http://www.d-maps.com*.

Table 1: The shortest tour for the first n cities

n	shortest tour	length (km)
4	1-2-4-3-1	525
5	1-2-4-5-3-1	549
6	1-2-4-5-3-6-1	607
7	1-2-4-7-5-3-6-1	615
8	1-2-4-7-8-5-3-6-1	658
9	1-2-4-7-8-5-3-6-9-1	878
10	1-2-4-7-8-5-3-10-9-6-1	983
11	1-2-4-7-8-11-5-3-10-9-6-1	1019
12	1-6-9-10-3-5-11-8-7-4-2-12-1	1020
13	1-11-6-9-10-3-5-13-8-7-4-2-12-1	1027
14	1-11-6-9-10-3-5-13-8-7-4-2-12-14-1	1130

We measured the running time, the number of generated nodes and the number of pruned nodes. At a certain point in the calculation, the best tour is found but it is not yet known that this the best tour. We also measured at what time this happens. For the time measurements, we took the average of 5 runs. The results can be found in Table 2 till 5. We used the brute force algorithm to check our answers. For all instances up to 12 cities it gave the same answer. For bigger instances the brute force algorithm took too long to use it.

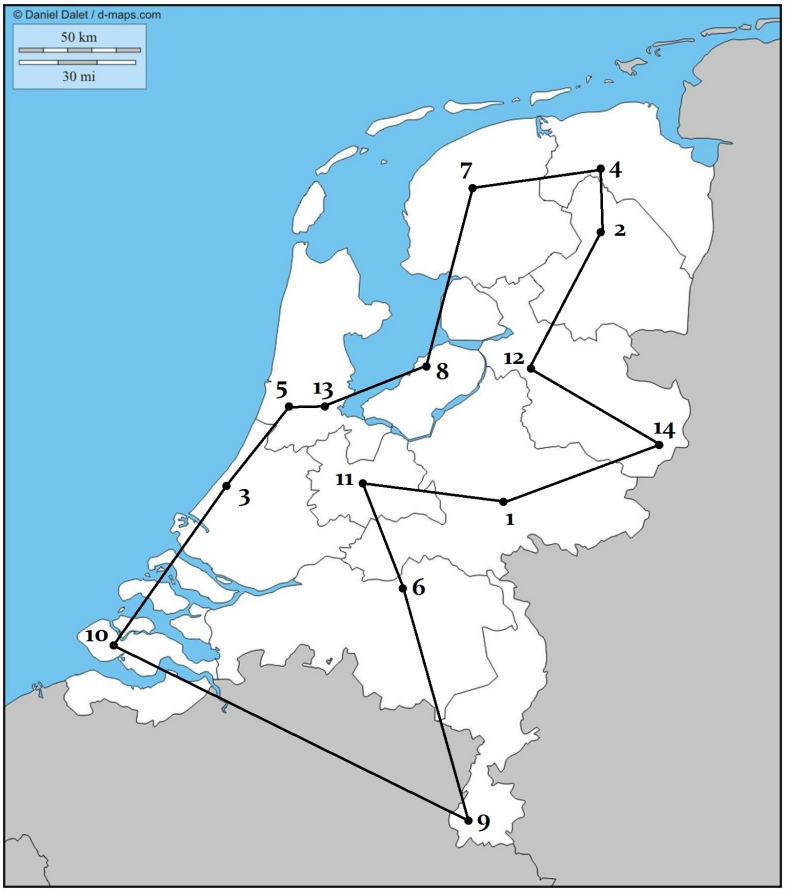

Figure 5: *The shortest tour for 14 cities: 1130 km.*

In Figure 6 we see that both the running time and the number of nodes increases exponentially with the number of cities. The two algorithms that use lower bound OT have a longer running time and also create more nodes than the algorithms that use SB. From this we can conclude that SB is a better lower bound than OT. We also see that IL-OT is faster and creates less nodes than LG-OT. The same holds for IL-SB versus LG-SB. From this it follows that IL is a better branching strategy than LG. The best algorithm is IL-SB which combines the best choices for both branching and bounding.

We also measured at what time the best tour was found. For every algorithm we calculated the average ratio between this time and the running time. This is shown in Figure 7. The use of OT for bounding and IL for branching both result in the best tour to be found relatively earlier, but the differences are not very big.

Table 2: The results of the algorithm with branching in lexicographic order and simple lower bound (LG-SB)

n	time (s)	found after (s)	created nodes	pruned nodes
4	0.0015	0.00143	5	2
5	0.00545	0.00377	11	5
6	0.0274	0.0094	41	20
7	0.132	0.13018	135	62
8	0.19	0.18352	153	73
9	1.05	1.0064	619	307
10	0.921	0.8703	475	236
11	12.7	11.16	4759	2375
12	15.1	0.252	4817	2406
13	45.1	19.6	11755	5872
14	158	79.8	34253	17115

Table 3: The results of the algorithm with branching in order of increasing length and simple lower bound (IL-SB)

n	time (s)	found after (s)	created nodes	pruned nodes
4	0.00231	0.00224	7	3
5	0.00396	0.00387	9	4
6	0.0157	0.00863	23	11
7	0.0152	0.01354	17	8
8	0.0614	0.0325	49	23
9	0.658	0.296	369	181
10	0.406	0.109	199	98
11	3.02	1.51	1153	572
12	7.49	6.19	2279	1135
13	15.9	8.07	4013	2004
14	31	12.4	6885	3440

Table 4: The results of the algorithm with branching in lexicographic order and 1-tree lower bound (LG-OT)

n	time (s)	found after	created nodes	pruned nodes
4	0.00213	0.00207	5	2
5	0.0013	0.00123	13	5
6	0.103	0.00103	51	22
7	0.385	0.384881	111	52
8	0.833	0.83287	141	67
9	9.93	9.417	1099	545
10	16.3	13.77	1153	572
11	218	204.6	11879	5935
12	504	187	19007	9501
13	1752	3.51	48367	24183
14	9318	601	197753	98872

Table 5: The results of the algorithms with branching in order of increasing length and 1-tree lower bound (IL-OT)

n	time (s)	found after (s)	created nodes	pruned nodes
4	0.00292	0.00286	7	3
5	0.00844	0.00831	9	4
6	0.0229	0.0228	13	6
7	0.0737	0.0472	23	11
8	0.424	0.14	73	35
9	4.22	0.79	513	252
10	5.52	0.56	421	209
11	46.5	14.5	2583	1287
12	147	95.4	5801	2894
13	423	160	12639	6314
14	994	313	22901	11441

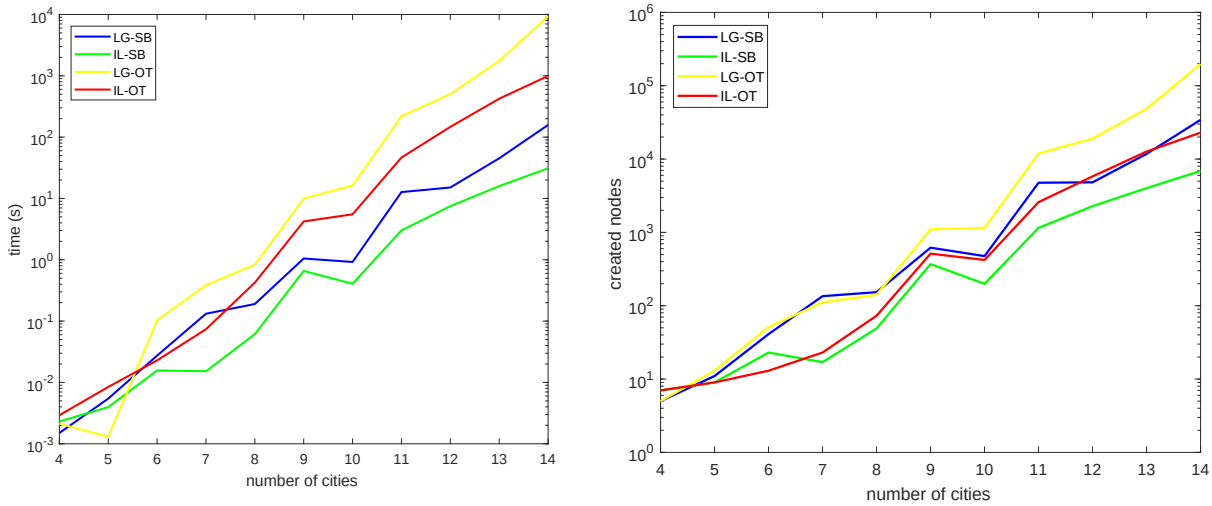

Figure 6: *The relation between the number of cities and the running time (left) and the number of created nodes (right). On the vertical axis a logarithmic scale is used.*

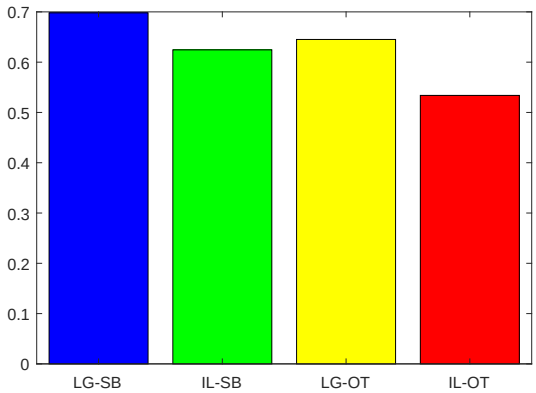

Figure 7: *The bars show for each algorithm the average ratio between the time at which the best tour was found and the total running time.*

Best result

We tested the best algorithm, IL-SB, on bigger instances. The best result we reached is the 23-city instance shown in Figure 8. For this instance we added the cities Nijmegen, Den Helder, Apeldoorn, Rotterdam, Breda, Eindhoven, Leiden, Amersfoort and Almere to the 14 previous cities. It took $22,263$ seconds to calculate the best tour, which is more than 6 hours. The total number of nodes created was 965,223. This is very small compared to the 562,000,363,888,803,840,000 tours that are possible for 23 cities.

Time complexity

Adding one city resulted in the running time of IL-SB to become approximately 2 or 3 times as large. Figure 9 shows the running time of IL-SB for all n up to 23. The steepness on the line is comparable to 3^n. From this we predict the complexity of IL-SB to be $\mathcal{O}(3^n)$.

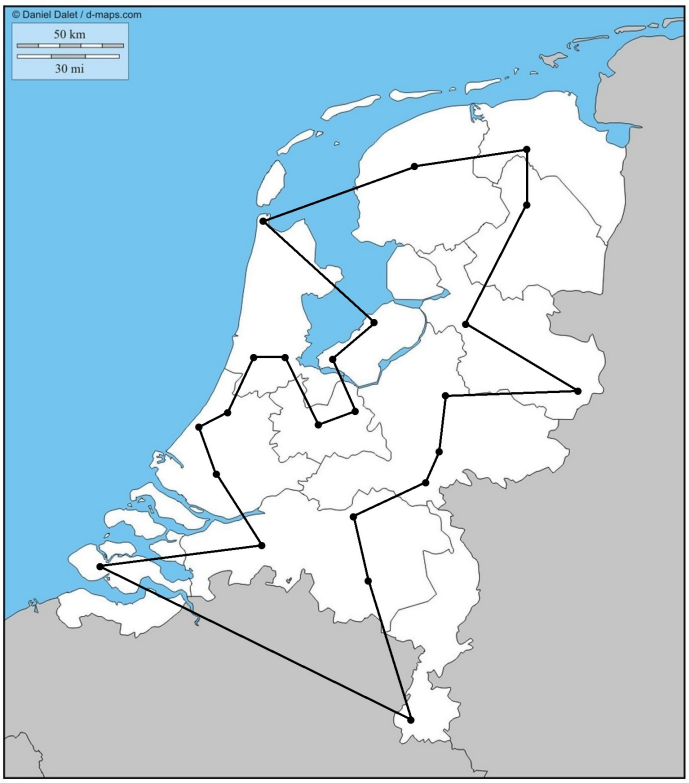

Figure 8: *The shortest tour for 23 cities. It has a length of 1295 km and it took 22,263 seconds to calculate it with IL-SB.*

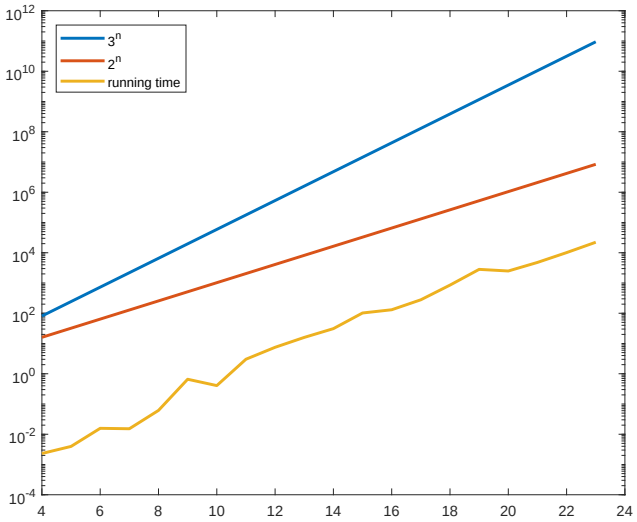

Figure 9: *The running time of IL-SB for* $4 \leq n \leq 23$*. Also* 2^n *and* 3^n *are shown.*

Ratio between pruned nodes and generated nodes

In Tables 2 till 5 we observe that the number of pruned nodes is almost half the number of created nodes. This can be explained as follows. Consider the part of the tree that has been generated when the algorithm ends. This is a binary tree in which each internal node has exactly two children. Figure 10 shows an example of such a tree. This tree can be generated by starting with the root node and step by step adding two children to an existing leaf node. Let n be the number of steps needed to generate the tree in this way. Let L_i be the number of leaf nodes, T_i be the number of internal nodes and G_i be the total number of nodes after i steps. We start with $L_0 = 1$, $T_0 = 0$ and $G_0 = 1$. In every step we add two children to a leaf node. This means that one leaf node becomes an internal node and two new leaf nodes are created. This results in a net increase of both L_i and T_i by one. In other words

$$\begin{aligned} L_{i+1} &= L_i + 1 \\ T_{i+1} &= T_i + 1 \\ G_{i+1} &= L_{i+1} + T_{i+1} \end{aligned}$$

When the tree is complete, we have that $L_n = n + 1$, $T_n = n$ and $G_n = 2n + 1$. The leaf nodes are either pruned nodes or complete tours. Let P be the number of pruned nodes and t be the number of tours found, then

$$G_n = T_n + L_n = T_n + P + t$$

The number of tours found is small compared to the number of pruned nodes. From this follows that P will be slightly smaller than $\frac{1}{2}G_n$ and this fits the observations.

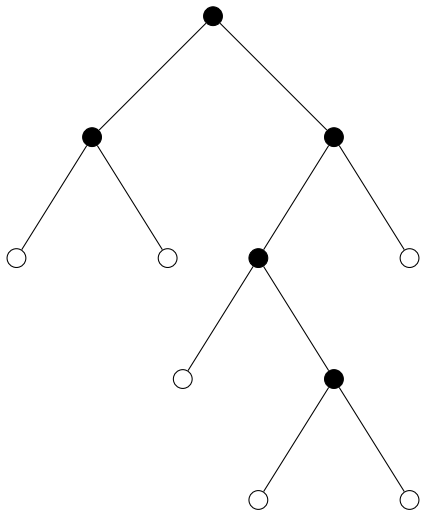

Figure 10: *An example of a binary tree in which each internal node has exactly two children. The closed circles are the internal nodes and the open circles are the leaf nodes. There are 5 internal nodes and 6 leaf nodes.*

5 Approximate solutions to TSP

In the previous section we discussed exact algorithms for TSP. The running time of exact algorithms increases exponentially with the number of cities. The best result we reached was a 23-city instance and it took more than 6 hours to calculate it. In most practical applications, we don't necessarily need to know the best tour. A tour with a length that comes close to the optimal length also suffices. In that case, it is better to use a heuristic algorithm. This is an algorithm that finds an approximate solution and has the advantage that it is much faster. This means it can be used for bigger instances as compared to exact algorithms.

There exist many different heuristic algorithms for TSP. First we shortly discuss the greedy algorithm 'nearest neighbour'. Next we discuss how the exact algorithm branch and bound could be used as a heuristic. The rest of of this chapter will be devoted to Ant Colony Optimisation (ACO). This algorithm is based on communication between ants by an odorous chemical substance called pheromone. First we present the theory behind ACO. Next we present an implementation in Python and test it on instances of TSP.

Nearest neighbour
The nearest neighbour algorithm is one of the simplest heuristic algorithms for TSP. It is defined as follows:

Select a random starting city. Repeatedly visit the nearest unvisited city until all cities have been visited. Finally, return to the starting city.

The complexity of this algorithm is $\mathcal{O}(n^2)$ because for each city we have to run through the list of unvisited cities once. The advantages of this algorithm are the easy implementation and the short running time. A disadvantage is that always choosing the nearest city may result in some cities to be 'missed'. When the tour is almost completed, we have to go back to visit the missed cities which results in a great increase in the tour length. In figure 11 we see 16 points with a nearest neighbour tour (above) and the best tour (below). The images were generated by the program Concorde. The top left city could better have been visited earlier in the tour. The best tour does not always consist of the smallest edges. Sometimes it is better to make a little detour to improve the overall solution.

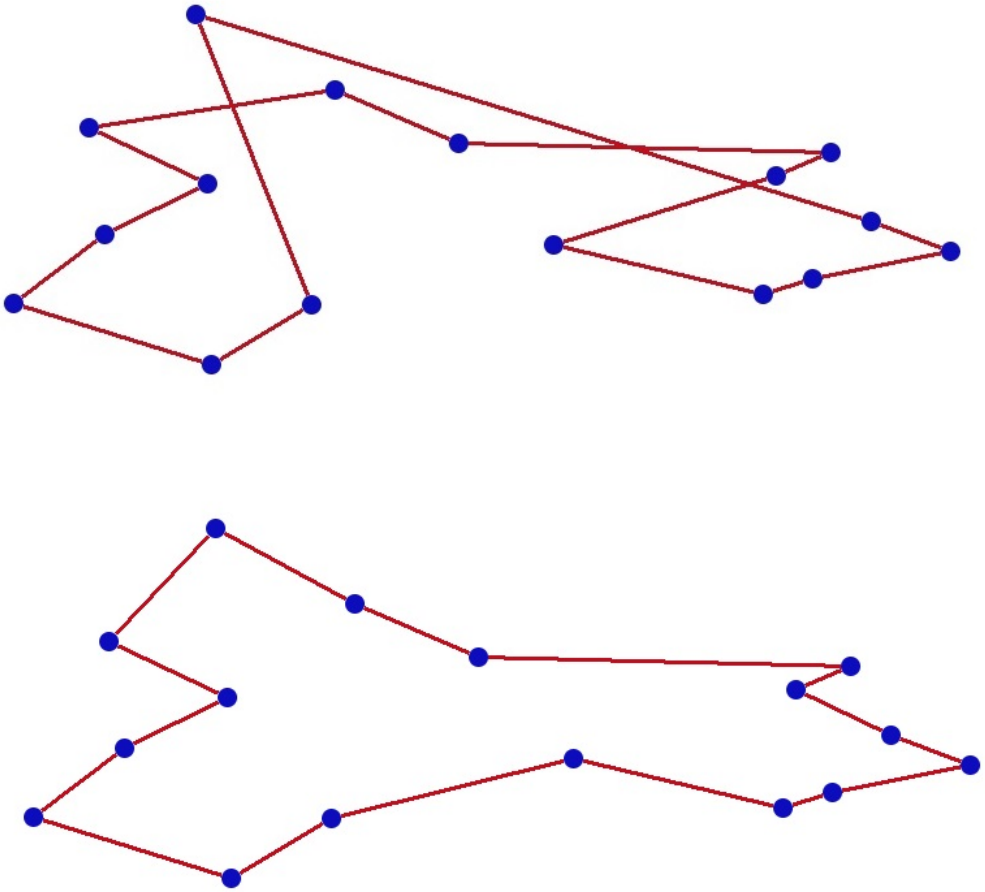

Figure 11: *A TSP-instance with 16 points and Euclidean distance. Above: the tour generated by nearest neighbour with a length of 179. Below: the optimal tour of length 131.*

Branch and bound as heuristic algorithm

In Chapter 3 and 4 we discussed branch and bound. We observed that the best solution was found on average at 63% of the running time. Even before that, other good tours were found. Based on this, we expect branch and bound can be used as an heuristic algorithm. In that case, we have to stop after a certain amount of time and take the best tour found so far. In figure 7 we see that for IL-OT the optimal solution was relatively found the earliest. But the total running time of IL-SB is much smaller than IL-OT and therefore we expect IL-SB to be the best choice to use as an heuristic algorithm. We did not test how branch and bound performs as a heuristic. This is something that could be investigated further.

5.1 Ant Colony Optimisation: the theory

Animals cannot speak like humans, but this does not mean they don't have intelligent ways of communication. For example, bees perform a dance that tells their congeners the direction and distance to nectar sources. Ants work together in large colonies and have developed a smart way to find the shortest route.

In this section we will discuss an algorithm for which the foraging behaviour of ants was the source of inspiration. This algorithm is called Ant Colony Optimisation (ACO) and can be applied to many combinatorial optimisation problems. The first ant-inspired algorithm was described in 1992 in [2]. After that, there has been more research on the subject. This research consisted both of improving the algorithm and building a theory of why the algorithm works. In 2004, Dorgio & Stützle have written the book [4] which contains a lot of this research. The theory we discuss is based on this book. We will treat the theory in a general context but also show how it can be applied to TSP.

Communication between ants

Ants are social insects that live together in colonies that can contain millions of ants. These colonies form a highly structured organisation that can accomplish much more than one individual ant. Each ant has its own task like searching for food, feeding, protecting the nest or caring for the offspring. The ants that search for food are called the foragers. The visual abilities of ants are poorly developed and some species are completely blind. To find their way to food sources, they make use of indirect communication by pheromones. These are chemical substances deposited by ants while walking. The substances can be smelled by other ants and influences the path they choose. An ant is more likely to follow a path with a high pheromone concentration.

Figure 12: *Iridomyrmex humilis.*
Source: http://www.uniprot.org/taxonomy/83485

In 1989, Goss et al. have conducted an experiment with the ant species *Iridomyrmex humilis* [6]. In this experiment the nest was connected to a food source by a double bridge (Figure 13). They varied the relative length of the branches of the bridge. In the first experiment the branches had the same length. In the beginning, the number of ants in both branches was similar but after a while almost all ants used the same branch. The experiment was conducted several times and the left and right branch occurred

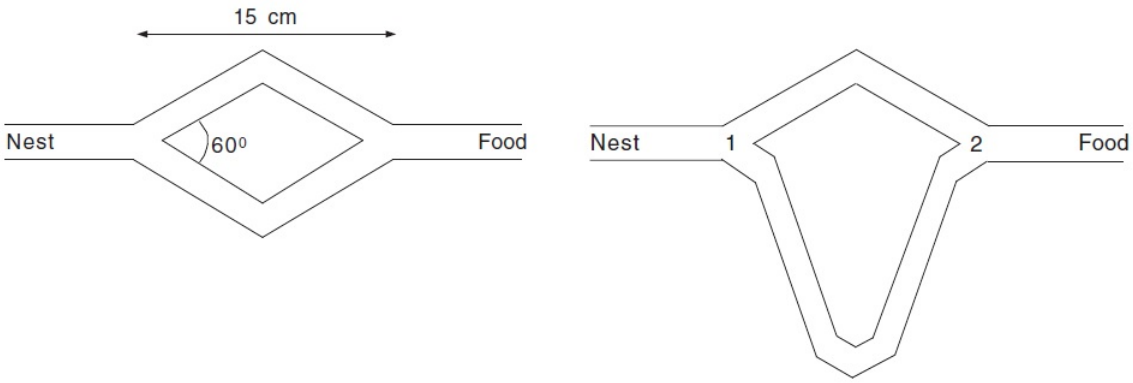

Figure 13: *The setup of the experment of Goss et al. (1989). Left: the first experiment, right: the second experiment.*

equally often as the branch chosen by all ants. This result can be explained as follows. In the beginning no pheromone is yet deposited and the ants choose randomly between the two branches. But due to random fluctuations the number of ants in one branch will be slightly larger than in the other branch. This will lead to more pheromone in this branch which will cause more ants to choose this branch which leads to even more pheromone in this branch. This positive feedback mechanism eventually results in almost all ants choosing the same branch. In the second experiment one branch was twice as long as the other. The result was that after a while, almost all of the ants chose the short branch. In the beginning they chose randomly between the two branches but the ants in the short branch reached the food earlier. When they returned to the nest, the pheromone concentration in the short branch is higher because the ants in the long branch have not yet reached the food. This will cause more ants to choose the short branch. In this way, pheromone accumulates faster on the short branch and eventually the ants will choose this branch. These experiments show that the length of the path has a greater influence on the behaviour of the ants than initial random fluctuations. What's interesting is that a small percentage of the ants kept choosing the long branch. This means the ants do not always choose the branch with the highest pheromone concentration; there is a small chance they explore other paths. This random exploration is important in finding the shortest route. When all ants keep taking the known path, there might exist a shorter path that is not yet discovered. When a small percentage of the ants deviates from the paths with highest pheromone concentrations, shorter paths can be found.

From ants to algorithm

The behaviour of ants can be used to solve combinatorial optimisation problems. A problem in combinatorial optimisation can be characterised as follows. First there is a set of n discrete variables X_i to which a value v_i can be assigned. A pair (X_i, v_i) is called a solution component. Next there is a set of constraints Ω. The set S contains all the feasible solutions. That is, every $s \in S$ is a complete assignment of values v_i to the variables X_i such that the constraints in Ω are satisfied. Finally we have the objective function $f : S \to \mathbb{R}^+$ that assigns a positive value to each feasible solution. Solving the problem means that we have to find an $s^* \in S$ such that $f(s^*) \leq f(s)$ for all $s \in S$.

In the context of TSP the variables X_i are the edges of the full graph with the cities as vertices. Further $v_i \in \{0, 1\}$ where $X_i = 1$ means that edge i is included in the tour and $X_i = 0$ means that edge i is not in the tour. The constraints in Ω are such that only assignments are allowed that form a tour. The objective function f gives the length of a tour.

Next we introduce the pheromone trail parameter T_i that has a value τ for each solution component (X_i, v_i). Now the idea is to create a set of artificial ants that are going to construct feasible solutions. The choices made by these ants are influenced by the pheromone values. After all ants have constructed a solution, the pheromone values are adapted. This is done in such a way that solution components that are present in solutions with a low objective value are increased. In that way, in the next iteration, the choices made by the ants will be biased towards better solutions.

For TSP, τ has a certain positive value for the solution components $(X_i, 1)$ and $\tau = 0$ for all solution components $(X_i, 0)$. We randomly distribute the ants over the cities. Next we let each ant choose the values of the variables X_i corresponding to the edges connected to the current city. In order to make feasible solutions, all but one of these values have to be zero. This process can also be regarded as the ant choosing the next city to visit. Thereby, all other edges connected to the current city are excluded from the tour.

The choices of the ants are not purely based on the pheromone values. It turned out to improve the algorithm to add heuristic information about the length of the edges. Also, each ant a has a memory M_a that contains the already visited cities. Let $\tau(k, r)$ be the pheromone value of edge (k, r). Let $\eta(k, r)$ be a heuristic function that we choose to be the inverse of the distance between cities k and r. When ant a is in city k, the next city r is determined by the formula:

$$r = \begin{cases} \arg\max_{u \notin M_a} \{[\tau(k,u)] \cdot [\eta(k,u)]^\beta\} & \text{if } q \leq q_0 \\ R & \text{otherwise} \end{cases} \tag{1}$$

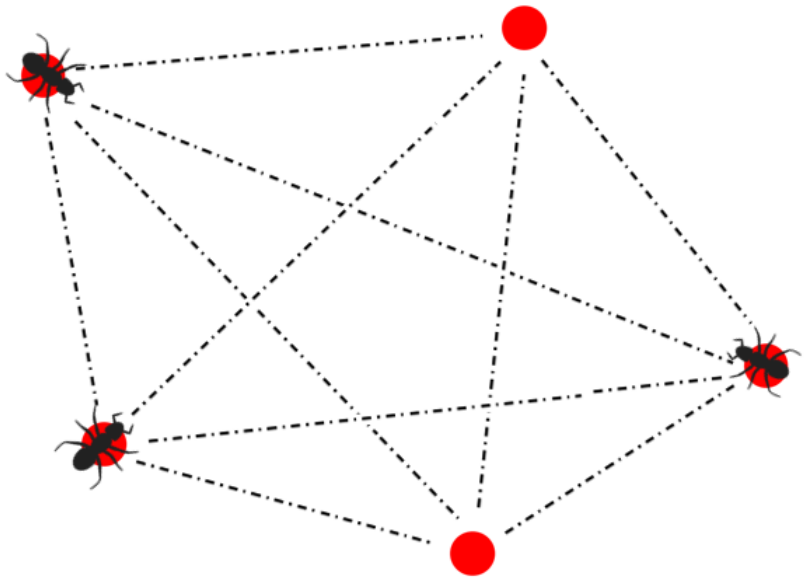

Figure 14: *We randomly distribute a group of ants over the cities and let them construct tours.*

where q is randomly chosen with uniform probability in $[0, 1]$, $q_0 \in [0, 1]$ is a parameter, β is a parameter that weights the relative importance of the pheromone values and the length of the edges and R is a randomly selected city. The probability of each edge to be selected depends on both its pheromone value and its length. The probability that city r is selected is given by

$$p_a(k,r) = \begin{cases} \frac{[\tau(k,r)]\cdot[\eta(k,r)]^\beta}{\sum_{u \notin M_a} [\tau(k,u)] \cdot [\eta(k,u)]^\beta} & \text{if } r \notin M_a \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

So with a probability q_0, the ant uses the knowledge of the colony to choose the best city and with a probability of $(1-q_0)$ chooses to explore new edges where shorter edges with higher pheromone values are more likely to be explored. This reflects the exploration behaviour we saw in the experiments.

When an ant crosses an edge, the pheromone value of this edge is changed. This is called local updating and is done according to the following rule:

$$\tau(k,r) \leftarrow (1-\rho) \cdot \tau(k,r) + \rho \cdot \tau_0 \tag{3}$$

where $\rho \in (0, 1)$ is a parameter and τ_0 is the initial pheromone value.

After each ant has completed a tour, the pheromone values are also adapted. This is called global updating. This global updating consists of both evaporation of pheromone on all edges and deposition of new pheromone on the edges belonging to the best tour so far. The amount of deposited pheromone depends on the length of the best tour. The following global updating rule is used:

$$\tau(k,r) \leftarrow (1-\alpha)\cdot\tau(k,r) + \alpha\cdot\Delta\tau(r,s) \tag{4}$$

where $\alpha \in (0,1)$ is the pheromone decay parameter and

$$\Delta\tau(k,r) = \begin{cases} \frac{1}{L_{bt}} & \text{if } (k,r) \in \text{best tour so far} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where L_{bt} is the length of the best tour so far. We let each ant walk a certain number of tours. The result of the algorithm will be the best overall tour.

There exist several ACO algorithms. The one we just described is called Ant Colony System and is described in [3]. In other versions of the algorithm, all edges are selected randomly according to formula (2). This corresponds to setting q_0 equal to 0. Also, in some versions, there is only global updating and no local updating. Other difference between ACS and other versions are the choices made in the global updating. In ACS we update the edges belonging to the best tour so far, but it is also possible to do this for the best tour found in the current iteration. We see that a lot of choices can be made when filling in the details of an ACO algorithm. Also, the above formulas are just a choice and these are the formulas that traditionally have been used in ACO. Nevertheless, other formulas that have a similar effect are suitable.

Improvement by local search

Heuristic algorithms can be subdivided into tour constructive heuristics and tour improvement heuristics. A well known tour improvement heuristic is 2-opt [10]. This algorithm removes two edges from an existing tour and reconnects the two resulting path in a different way. The 2-opt move that decreases the tour length the most is chosen. This is repeated until the tour cannot be improved by a 2-opt move. Instead of removing 2 edges, there also exist heuristics that remove a higher number of edges. These are examples of local search algorithms: they search for solutions close to a given solution by making small changes to it. In [3], the ACS algorithm is improved by adding 3-opt to it. After each iteration, the tours of all the ants are subject to 3-opt moves.

5.2 Implementation in Python

We implemented ACS as described above in Python. We chose for the version without 3-opt. The code is listed in Appendix C. The code consists of one class: `AntColonySystem`. This class represents an instance of TSP. To create an instance of `AntColonySystem`, we need to give the number of cities and a list that contains the coordinates (x_i, y_i) of each city i. The distance matrix `self.costs` will then be calculated according to this list where the distance between city i and city j is the Euclidean distance

$$d((x_i, y_i), (x_j, y_j)) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

An important variable is `self.pheromone`. This is a matrix that contains the pheromone values of all the edges. Another important variable is `self.bestTour`. This variable contains the best tour so far represented as a matrix. The (i, j)th element of this matrix is equal to 1 if edge $(i.j)$ is in the tour, otherwise it is equal to 0. The reason we chose to represent a tour as a matrix is that this makes the global trail updating easy since we need to know for each edge whether it is in the tour.

The most important method is `findSolution`. This method moves the ants between the cities. The variable `location` is a list that contains the current city of each ant. Further, `visited` is matrix of booleans that tells whether ant i has visited city j in the current iteration. The variable `tours` is a list of matrices. Matrix i keeps track of the tour of ant i. The variable `distances` is a list that contains the distance covered by each ant in the current iteration. The function used in formula (1) and (2) has been named `attraction`. The rest of the code speaks for itself. Below you see an example of the usage of the code with 10 random cities in $[0, 10]^2$.

```
>>> n = 10
>>> c = [[10*random.random(),10*random.random()] for i in range(n)]
>>> A = AntColonySystem(n,c)
>>> A.findSolution()

Best tour found:  [0, 5, 8, 3, 9, 4, 2, 1, 7, 6, 0]
It has a length of:  28.3481396696
Found in 0.8881832303006371 seconds
```

5.3 Results

We chose the following values for the parameters: $\beta = 2$, $q_0 = 0.9$ and $\rho = \alpha = 0.1$. Further we chose the number of ants equal to 10 and an initial pheromone value on all edges of $\tau_0 = \frac{1}{n \cdot L_{nn}}$ where n is the number of cities and L_{nn} is the length of a nearest neighbour tour. This corresponds to the choices made in [3].

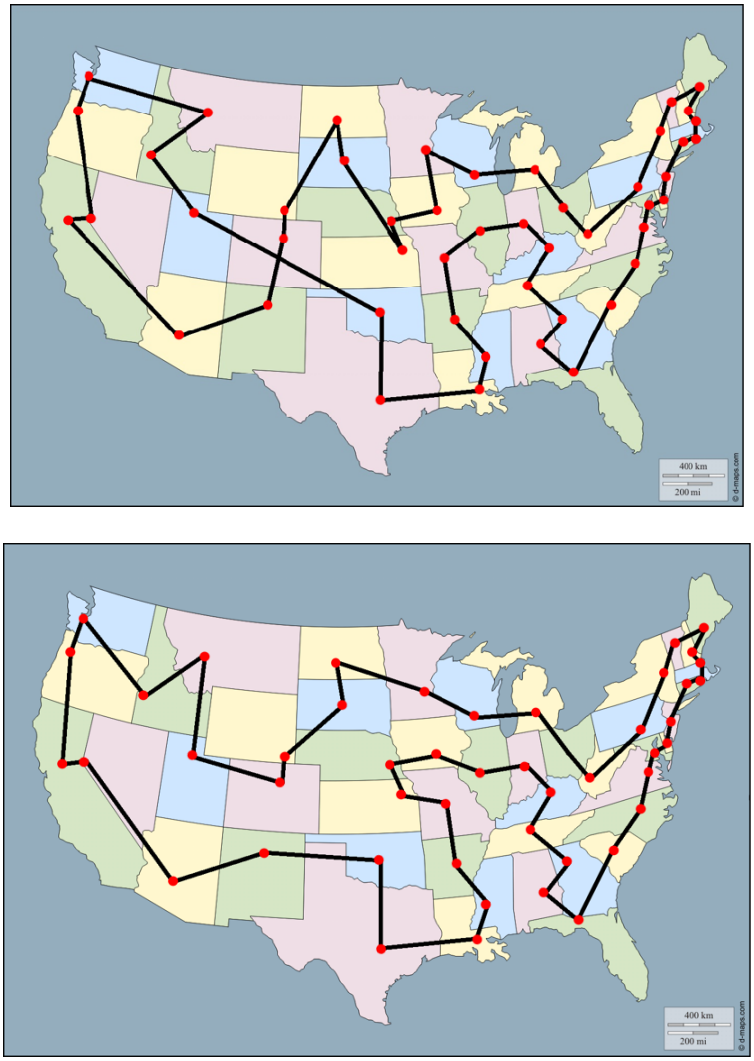

Figure 15: *The best tour found using ACS (above) and the optimal tour (below) for the 48 capitals of the contiguous United States.*

We tested the algorithm on 4 datasets. The first is the 23-city instance as on page 18. The second and the third are att48 and kroA100 from the online library TSPLIB. The fourth is a 634-city instance consisting of all populated locations in Luxembourg. This dataset is a subset of the World TSP and is available at *http://www.math.uwaterloo.ca/tsp/world/countries.html*. We chose to carry out a 100 iterations which means that each ant walks a 100 tours. We ran the algorithm 10 times for each data set. The results are shown in Table 6. Figure 15 and 16 show both the best tour found and the optimal tour for instances 2 and 4. For small instances, ACS performs well. In some cases it finds the optimal tour and in other cases it will find a tour close to optimal. For bigger instances, the error becomes much larger. Also, the resulting tour crosses itself quite often. These crossings will be solved by tour improvement heuristics. Therefore we expect that adding 2-opt or 3-opt to the algorithm will make it better suitable for large instances.

Time complexity

We see that approximately, the running time increases quadratically with the number of cities. This seems logical considering the following. The total number of tours is independent of the number of cities. When an ant carries out a tour of n cities, there are n places on which the ant has to make a choice. For each choice, we have to run once through the list of at most $(n-1)$ unvisited cities to select the next city. The results in a complexity of $\mathcal{O}(n^2)$.

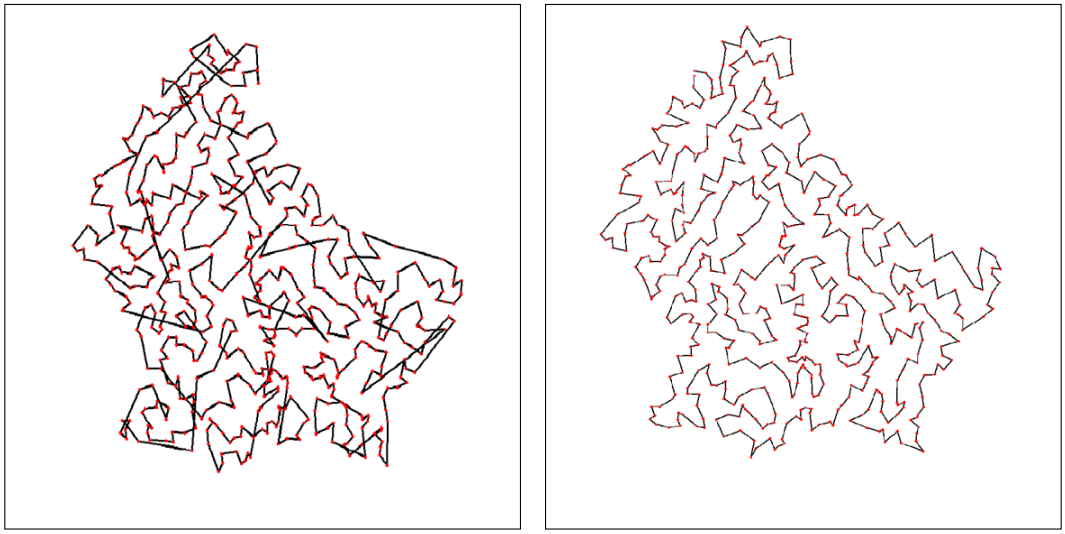

Figure 16: *The best tour found using ACS (left) and the optimal tour (right) through all 634 populated locations of Luxembourg*

Table 6: *Results of applying ACS to four instances of TSP with 100 iterations. The relative error is defined as (best tour length - optimum)/(*$0.01 * optimum$*).*

Instance	Optimum	Average tour length	Best tour length	Relative error	Average time per run (s)
23 cities in the Netherlands	1,295	1,333	1,295	0 %	2.69
att48 (48 capitals of the USA)	33,522	36,060	34,987	4.4 %	12.9
kroA100 (100 cities)	21,282	24,658	23,691	11.3 %	50.5
634 cities in Luxembourg	11,340	14,871	14,555	28.4 %	1691

6 Conclusion

In this thesis we studied algorithms for the travelling salesman problem. These algorithms can be divided into exact and heuristic algorithms. Branch and bound is a framework that is used to design exact algorithms. It is based on the idea of arranging the set of all tours into smaller sets of tours that have common properties. We arrange the tours in a binary tree of which the root node represents the set of all tours. Each node has two children that partition the set of tours in the parent node based on a constraint on these tours. This process is called branching. For each node, a lower bound can be calculated. When this lower bound exceeds the length of a known tour, the node will be pruned. In this way, we don't need to look at each individual tour. We search through the tree in depth-first order. The process continues until all but one node has been pruned, this node will then contain the shortest tour. Different choices can be made regarding the branching and bounding strategies. We implemented four different branch and bound algorithms. For branching it was better to add the constraints in order of increasing length instead of in lexicographic order. For bounding, the lower bound that adds up the two smallest allowed edges of each city performed better than a lower bound based on a 1-tree. The biggest instance for which we found an exact solution consisted of 23 cities.

Heuristic algorithms find a tour with length close to the optimum. The shortest components do not form the optimal tour and therefore a greedy algorithm will in most cases not find the best tour. We studied Ant Colony Optimisation, an algorithm based on the foraging behaviour of ants. This algorithm simulates an ant colony and lets them construct tours. Edges that belong to good tours are rewarded with an increase in pheromone value. The ants choose the next city according to a probability function. Short edges with high pheromone values are more likely to be traversed. We implemented the algorithm in Python. For smaller instances ($\leq$ 100 cities) the algorithm performs well but for bigger instances the error grows larger. The expectation is that adding local search will improve the algorithm.

A Python code for branch and bound algorithms

In this Appendix all the Python code for the branch and bound algorithms is listed. First you find the classes `Node` and `TSP` for the algorithm LG-SB. Below that are the method `next_constraint` as used for branching strategy IL and the method `computeLowerBound` as used for the lower bound OT. Finally you find the brute force algorithm.

```
import math
import copy
import numpy as np

class Node():

def __init__(self, size, costs,sortedEdges,allSortedEdges,
parent_constr, extra_constr = None):
    self.size = size    # Number of cities
    self.costs = costs  # Distance matrix
    self.sortedEdges = sortedEdges
    self.allSortedEdges = allSortedEdges
    self.extra_constr = extra_constr
    self.constraints = self.determine_constr(parent_constr)
    self.lowerBound = self.computeLowerBound()

# This methods calculates the simple lower bound (SB)
# This this the sum over all vertices of the two smallest
# allowed edges
def computeLowerBound(self):
    lb = 0
    for i in range(self.size):
    lower = 0
    t = 0
    for j in range(self.size):
        if self.constraints[i][j] == 1:
            lower += self.costs[i][j]
            t += 1
    tt = 0
    while t < 2:
        shortest = self.sortedEdges[i][tt]
        if self.constraints[i][shortest] == 2:
            lower += self.costs[i][shortest]
            t += 1
        tt += 1
        if tt >= self.size:
            lower = math.inf
```

```
                break
        lb += lower
    return lb

    # This method determines the constraints of a node
    # based on the constraints of the parent and
    # the extra constraint for this node
    def determine_constr(self,parent_constr):
        constraints = copy.deepcopy(parent_constr)
        if self.extra_constr == None:
            return constraints
        fr = self.extra_constr[0]
        to = self.extra_constr[1]
        constraints[fr][to] = self.extra_constr[2]
        constraints[to][fr] = self.extra_constr[2]
        for i in range(2):
            constraints = self.removeEdges(constraints)
            constraints = self.addEdges(constraints)
        return constraints


    # This method excludes edges when:
    # 1) Already two other edges adjacent to the same vertex
    #    are included
    # 2) Including the edge would create a subtour
    def removeEdges(self,constraints):
        for i in range(self.size):
            t = 0
            for j in range(self.size):
                if (i != j) and (constraints[i][j] == 1):
                    t += 1
            if t >= 2:
                for j in range(self.size):
                    if constraints[i][j] == 2:
                        constraints[i][j] = 0
                        constraints[j][i] = 0

        for i in range(self.size):
            for j in range(self.size):
                t = 1
                prev = i
                fr = j
                cycle = False
```

```
            nextOne = self.next_one(prev,fr,constraints)
            while (nextOne[0]):
                t += 1
                next = nextOne[1]
                if next == i:
                    cycle = True
                    break
                if t > self.size:
                    break
                prev = fr
                fr = next
                nextOne = self.next_one(prev,fr,constraints)
            if (cycle) and (t < self.size) and
            (constraints[i][j] == 2):
                constraints[i][j] = 0
                constraints[j][i] = 0
    return constraints


# This methods checks if all but two edges adjacent to
# a vertex are excluded.
# If so, these edges are included
def addEdges(self,constraints):
    for i in range(self.size):
        t = 0
        for j in range(self.size):
            if constraints[i][j] == 0:
                t += 1
        if t == self.size - 2:
            for j in range(self.size):
                if constraints[i][j] == 2:
                    constraints[i][j] = 1
                    constraints[j][i] = 1
    return constraints

# Determines whether there exists an included edge that starts
# in fr and does not end in prev. If so, it also returns the
# endpoint of this edge
def next_one(self,prev,fr,constraints):
    for j in range(self.size):
        if (constraints[fr][j] == 1) and (j != prev):
            return [True,j]
    return[False]
```

```
# Determines if a node represents a full tour
# by checking whether from every vertex there are
# exactly 2 included edges and all other edges are excluded.
def isTour(self):
    for i in range(self.size):
        num_zero = 0
        num_one = 0
        for j in range(self.size):
            if self.constraints[i][j] == 0:
                num_zero += 1
            elif self.constraints[i][j] == 1:
                num_one += 1
        if (num_zero != self.size - 2) or (num_one != 2):
            return False
    return True

# Checks if a node contains a subtour
def contains_subtour(self):
    for i in range(self.size):
        next = self.next_one(i,i,self.constraints)
        if next[0]:
            prev = i
            fr = next[1]
            t = 1
            next = self.next_one(prev, fr, self.constraints)
        while next[0]:
            t += 1
            prev = fr
            fr = next[1]
            if (fr == i) and (t < self.size):
                return True
            next = self.next_one(prev,fr,self.constraints)
            if t == self.size:
                return False
    return False

# Assumes the node represents a tour and returns
# the length of this tour
def tourLength(self):
    length = 0
    fr = 0
    to = self.next_one(fr,fr,self.constraints)[1]
```

```
    for i in range(self.size):
        length += self.costs[fr][to]
        temp = fr
        fr = to
        to = self.next_one(temp,to,self.constraints)[1]
    return length

# This method determines the next constraint according
# to the the branching strategy lexicographic order (LG)
def next_constraint(self):
    for i in range(self.size):
        for j in range(self.size):
            if self.constraints[i][j] == 2:
                return [i,j]

# If a node represents a tour, this method returns a
# string with the order of the vertices in the tour
def __str__(self):
    if self.isTour():
        result = '0'
        fr = 0
        to = None
        for j in range(self.size):
            if self.constraints[fr][j] == 1:
                to = j
                result += '-' + str(j)
                break
        for i in range(self.size - 1):
            for j in range(self.size):
                if (self.constraints[to][j] == 1) and (j != fr):
                    result += '-' + str(j)
                    fr = to
                    to = j
                    break
        return result
    else:
        print('This node is not a tour')
```

```
from Node import Node
import math
import time
import copy

class TSP():

def __init__(self,size,costs,bestTour = math.inf):
    self.size = size
    self.costs = costs
    self.bestTour = bestTour
    self.bestNode = None
    self.bestNodeTime = 0
    self.num_createdNodes = 0
    self.num_prunedNodes = 0
    self.sortedEdges = self.sort_edges()
    self.allSortedEdges = self.sort_allEdges()

def findSolution(self):
    root = self.create_root()
    self.num_createdNodes += 1
    T1 = time.perf_counter()
    self.BranchAndBound(root)
    T2 = time.perf_counter()
    print('---------------------------------------------------------')
    print('The shortest tour is:', self.bestNode)
    print('It has a length of', self.bestTour, 'km')
    print('Found in',T2 - T1, 'seconds')
    print('Best tour was found after:',self.bestNodeTime,'seconds')
    print('Number of nodes created:',self.num_createdNodes)
    print('Number of nodes pruned:',self.num_prunedNodes)
```

```
# Sorts the edges of the distance matrix per row returns matrix
# where each row i contains the numbers 0 <= k <= (self.size-1)
# in the order of increasing costs of the edges (i,k)
def sort_edges(self):
    result = []
    for i in range(self.size):
        result.append([x for (y, x) in sorted(zip(self.costs[i],
        list(range(self.size))))])
    return result


# sorts all edges of distance matrix
# returns list of pairs [i,j] in order of increasing costs
def sort_allEdges(self):
    edges = []
    lengths = []
    for i in range(self.size):
        for j in range(i + 1, self.size):
            edges.append([i, j])
            lengths.append(c[i][j])
    result = [z for (l, z) in sorted(zip(lengths, edges))]
    return result

def create_root(self):
    no_constraints = []
    for i in range(self.size):
        row_i = []
        for j in range(self.size):
            if (i != j):
                row_i.append(2)
            else:
                row_i.append(0)
        no_constraints.append(row_i)
    root = Node(self.size,self.costs,self.sortedEdges,
    self.allSortedEdges,no_constraints)
    return root
```

```
def BranchAndBound(self,node):
    if node.isTour():
        if node.tourLength() < self.bestTour:
            self.bestTour = node.tourLength()
            self.bestNode = node
            self.bestNodeTime = time.perf_counter()
            print('Found better tour:', self.bestNode,'of length',
            self.bestTour, 'km')
    else:
        new_constraint = copy.copy(node.next_constraint())
        new_constraint.append(1)
        leftChild = Node(self.size,self.costs,self.sortedEdges,
        self.allSortedEdges,node.constraints, new_constraint)
        new_constraint[2] = 0
        rightChild = Node(self.size,self.costs,self.sortedEdges,
        self.allSortedEdges,node.constraints, new_constraint)
        self.num_createdNodes += 2

        if self.num_createdNodes % 401 == 0:
            print('Number of nodes created so far:',
             self.num_createdNodes)
            print('Number of nodes pruned so far:',
             self.num_prunedNodes)
        if self.num_createdNodes % 51 == 0:
            print('.')
        if  (leftChild.contains_subtour()) or
        (leftChild.lowerBound > 2 * self.bestTour):
            leftChild = None
            self.num_prunedNodes += 1
        if (rightChild.contains_subtour()) or
        (rightChild.lowerBound > 2 * self.bestTour):
            rightChild = None
            self.num_prunedNodes += 1
        if (leftChild != None) and (rightChild == None):
            self.BranchAndBound(leftChild)
        elif (leftChild == None) and (rightChild != None):
            self.BranchAndBound(rightChild)
        elif (leftChild != None) and (rightChild != None):
            if leftChild.lowerBound <= rightChild.lowerBound:
                if leftChild.lowerBound < 2 * self.bestTour:
                    self.BranchAndBound(leftChild)
                else:
                    leftChild = None
```

```
                self.num_prunedNodes += 1
            if rightChild.lowerBound < 2 * self.bestTour:
                self.BranchAndBound(rightChild)
            else:
                rightChild = None
                self.num_prunedNodes += 1
        else:
            if rightChild.lowerBound < 2 * self.bestTour:
                self.BranchAndBound(rightChild)
            else:
                rightChild = None
                self.num_prunedNodes += 1
            if leftChild.lowerBound < 2 * self.bestTour:
                self.BranchAndBound(leftChild)
            else:
                leftChild = None
                self.num_prunedNodes += 1
```

```
# Determines the next constraint using IL
def next_constraint(self):
    for edge in self.allSortedEdges:
        i = edge[0]
        j = edge[1]
        if self.constraints[i][j] == 2:
            return edge
```

```
# Calculates lower bound OT
def computeLowerBound2(self):
    lb = 0
    onetree = np.zeros((self.size,self.size),np.int8)
    t = 0
    for i in range(1,self.size):
        for j in range(i + 1, self.size):
            if self.constraints[i][j] == 1:
                onetree[i][j] = 1
                onetree[j][i] = 1
                t += 1
                lb += self.costs[i][j]
    for edge in self.allSortedEdges:
        if t >= self.size - 1:
            break
        i = edge[0]
        j = edge[1]
        if (self.constraints[i][j] == 2) and (i != 0):
            onetree[i][j] = 1
            onetree[j][i] = 1
        if self.onetree_contains_cycle(onetree):
            onetree[i][j] = 0
            onetree[j][i] = 0
        else:
            t += 1
            lb += self.costs[i][j]
    t = 0
    for j in range(self.size):
        if self.constraints[0][j] == 1:
            onetree[0][j] = 1
            onetree[j][0] = 1
            lb += self.costs[0][j]
            t += 1
    tt = 0
```

```
    while t < 2:
        shortest = self.sortedEdges[0][tt]
        if self.constraints[0][shortest] == 2:
            onetree[0][shortest] = 1
            onetree[shortest][0] = 1
            lb += self.costs[0][shortest]
            t += 1
        tt += 1
    return lb
```

```
# Brute force algorithm
# The parameters 'n' and 'distances' have to be specified
import itertools
import math

minLength = math.inf
minTour = []

for tour in itertools.permutations(list(range(1,n))):
    fr = 0
    length = 0
    count = 0
    while count < n-1:
        to = tour[count]
        length += distances[fr][to]
        fr = to
        count += 1
    length += distances[fr][0]
    if length < minLength:
        minLength = length
        minTour = tour

minTour = (0,) + minTour + (0,)
print('Shortest tour is:', minTour)
print('It has a length of:', minLength, 'km')
```

B Data for testing branch and bound algorithms

We used the following cities in the Netherlands:
1: Arnhem, 2: Assen, 3: Den Haag, 4: Groningen, 5: Haarlem, 6: 's Hertogenbosch, 7: Leeuwarden, 8: Lelystad, 9: Maastricht, 10:Middelburg, 11: Utrecht, 12: Zwolle, 13: Amsterdam, 14: Enschede

Table 7 shows the distances rounded to whole kilometers.

Table 7: Distances between the cities in km.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	141	118	171	126	69	158	79	166	208	65	67	98	97
2	141	0	266	34	212	208	82	114	305	324	165	75	187	118
3	118	266	0	232	56	107	194	109	229	122	61	151	60	201
4	171	34	232	0	200	233	63	128	332	347	188	104	180	146
5	126	212	56	200	0	105	145	84	228	170	55	123	19	181
6	69	208	107	233	105	0	212	113	123	144	54	133	96	163
7	158	82	194	63	145	212	0	104	324	323	161	93	137	161
8	79	114	109	128	84	113	104	0	236	225	65	51	58	133
9	166	305	229	332	228	123	324	236	0	187	177	230	219	239
10	208	324	122	347	170	144	323	225	187	0	165	249	198	298
11	65	165	61	188	55	54	161	65	177	165	0	91	48	140
12	67	75	151	104	123	133	93	51	230	249	91	0	104	73
13	98	187	60	180	19	96	137	58	219	198	48	104	0	161
14	97	118	201	146	181	163	161	133	239	298	140	73	161	0

C Python code for Ant Colony Sytem

```
import numpy as np
import math
import time
from random import *

class AntColonySystem():

numAnts = 10 # number of ants
numIterations = 100 # number of tours each ant is going to walk
a = 0.1 # parameter alpha
b = 2   # parameter beta
q0 = 0.9

def __init__(self,size,cityLocations):
    self.size = size
    self.cityLocations = cityLocations
    self.costs = self.createCostMatrix(cityLocations)
    self.bestTour = None
    self.bestTourLength = math.inf
    self.tau = 1 / (self.lengthNearestNeighbour() * self.size)
    self.pheromone = self.tau * np.ones((self.size,self.size))

def findSolution(self):
    T1 = time.perf_counter()
    location = np.zeros(self.numAnts,np.int32)
    startingpoint = np.zeros(self.numAnts,np.int32)
    for ant in range(self.numAnts):
        startingpoint[ant] = location[ant] = randint(0,self.size-1)

    for i in range(self.numIterations):
        visited = np.zeros((self.numAnts, self.size), dtype=bool)
        tours = [np.zeros((self.size,self.size),np.int8) for
            ant in range(self.numAnts)]
        distances = np.zeros(self.numAnts)
        for ant in range(self.numAnts):
            visited[ant][location[ant]] = True

        for step in range(self.size):
            for ant in range(self.numAnts):
                current = location[ant]
                if step != self.size - 1:
```

```
                    next = self.nextCity(ant,location[ant],visited[ant])
                else:
                    next = startingpoint[ant]
                location[ant] = next
                visited[ant][next] = True
                tours[ant][current][next] =
                    tours[ant][next][current] = 1
                distances[ant] += self.costs[current][next]
                self.localTrailUpdate(current,next)
        shortestLength = min(distances)
        if shortestLength < self.bestTourLength:
            self.bestTourLength = shortestLength
            self.bestTour = tours[np.argmin(distances)]
        self.globalTrailUpdate()
    T2 = time.perf_counter()
    print('Best tour found:', self.bestTourList())
    print('It has a length of:', self.bestTourLength)
    print('Found in', T2 - T1, 'seconds')

def createCostMatrix(self,cityLocations):
    result = np.zeros((self.size,self.size))
    for i in range(self.size):
        for j in range(self.size):
            result[i][j] = self.distance(i,j)
    return result

# returns the Euclidean distance between city i and city j
def distance(self,i,j):
    return math.sqrt(
    math.pow(self.cityLocations[j][0]-self.cityLocations[i][0],2) +
    math.pow(self.cityLocations[j][1]-self.cityLocations[i][1],2))

# returns the closest, nonvisited city
def closestNotVisited(self,loc,visited):
    minimum = math.inf
    result = None
    for city in range(self.size):
        if (not visited[city]) and (self.costs[loc][city] < minimum):
            minimum = self.costs[loc][city]
            result = city
    return result
```

```
def localTrailUpdate(self,i,j):
    self.pheromone[j][i] = self.pheromone[i][j] =
    (1-self.a)*self.pheromone[i][j] + self.a*self.tau

def globalTrailUpdate(self):
    for i in range(self.size):
        for j in range(i+1,self.size):
            self.pheromone[i][j] = self.pheromone[j][i] =
            (1-self.a)*self.pheromone[i][j] +
            self.a * self.bestTour[i][j]/self.bestTourLength

# returns the next city the ant will visit
def nextCity(self,ant,loc,visited):
    result = None
    q = np.random.random_sample()
    if q <= self.q0:
        max = -math.inf
        for city in range(self.size):
            if not visited[city]:
                f = self.attraction(loc,city)
                if f > max:
                    max = f
                    result = city
        if max != 0:
            return result
        else:
            return self.closestNotVisited(loc,visited)
    else:
        sum = 0
        for city in range(self.size):
            if not visited[city]:
                sum += self.attraction(loc,city)
        if sum == 0:
            return  self.closestNotVisited(loc,visited)
        else:
            R = np.random.random_sample()
            s = 0
            for city in range(self.size):
                if not visited[city]:
                    s += self.attraction(loc,city)/sum
                    if s > R:
                        return city
```

```
def attraction(self,i,j):
    if i != j:
        return self.pheromone[i][j] /(math.pow(self.costs[i][j],self.b))
    else:
        return 0

# returns the length of the nearest neighbour tour
def lengthNearestNeighbour(self):
    start = randint(0,self.size-1)
    current = start
    visited = np.zeros(self.size, dtype=bool)
    tour = [current]
    length = 0
    for i in range(self.size-1):
        visited[current] = True
        minimum = math.inf
        closest = None
        for i in range(self.size):
            if (not visited[i]) and (self.costs[current][i] < minimum):
                minimum = self.costs[current][i]
                closest = i
        tour.append(closest)
        length += minimum
        current = closest
    tour.append(start)
    length += self.costs[current][start]
    return length

# Returns the best tour as a list of cities
# in the order they are visited
def bestTourList(self):
    current = 0
    previous = 0
    tour = [0]
    for i in range(self.size):
        next = 0
        while(self.bestTour[current][next] == 0) or (previous == next):
            next += 1
        tour.append(next)
        previous = current
        current = next
    return tour
```

References

[1] David L. Applegate, Robert E. Bixby, Vasek Chvátal, William Cook, Daniel G. Espinoza, Marcos Goycoolea, and Keld Helsgaun. Certification of an optimal TSP tour through 85,900 cities. *Operations Research Letters*, 37(1):11–15, 2009.

[2] Marco Dorigo. *Optimization, learning and natural algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[3] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1), 1997.

[4] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT press, 2004.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[6] S. Goss, S. Arona, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the Argentine Ant. *Naturwissenschaften*, 76:579–581, 1989.

[7] Michael Held and Richard M. Karp. The travelling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[8] Joseph B. Kruskal Jr. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[9] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

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

[11] Richard Wiener. Branch and bound implementation for the travelling salesperson problem - Part 1. *Journal of Object Technology*, 2(2):65–86, 2003.