KTH Royal Institute of Technology

DD2440

Advanced Algorithms

# TSP: Travelling Salesperson 2D

*Authors:*
Josip Matak (961114-T376)
Josip Mrđen (960430-T250)

*Kattis ID (score):*
3351348 (40.223431)

November 5, 2018

# Contents

## 1 Introduction

Travelling salesman problem (TSP) could be described in following way: given a set of cities and distances between them, find the cheapest route visiting all cities and return to starting position. Formally, this means that with provided complete undirected graph $G = (V, E)$, where $V$ is set of vertices and $E$ is a set of edges with defined cost, goal is to find hamiltonian cycle of $G$ that corresponds to smallest cost (4).

Instance of TSP provided in this particular problem is Euclidian TSP, where distances between cities or vertices are defined by their euclidian distance in space. While it is usual to work with *metric* instances of this problem, which means that every 3 vertices satisfy triangle inequality, for this instance cost at each edge is rounded integer, therefore we can't be sure that this requirement is satisfied. Problem itself is NP hard (can be solved on non-deterministic Turing's machine in polynomial time), even with those assumptions. Moreover, running time required for testing all possible solutions is $\frac{(n-1)!}{2}$, because defined cost matrix is a symmetric one. To better present idea of how complex this problem is, computing output while testing each solution for 14 cities would take up to half an hour, but adding just two more cities costs additional 4 days.

As input to this problem, list of cities with their corresponding Cartesian 2D coordinates were provided. Each of the problem instances is not larger than 1000 cities, and proposed maximum running time is 2 seconds. Through this paper, key ideas behind implementation of different heuristic algorithms for solving TSP will be discussed and supported with corresponding performances.

As a testing data, 4 instances of the algorithm have been selected: berlin52, ch130, a280 and pr439. All of the algorithms were run on those datasets, with tables and pictures in continuation.

# 2 Christofides algorithm

With approximation rate of $\frac{3}{2}$, this algorithm guarantees to find solution close up to $1.5 \cdot OPT$, where $OPT$ stands for distance of optimal tour. Assumption of this algorithm is it's metric nature. On general scale, by this time, this is still algorithm that generates best known approximation. Because of it's complexity, not having guarantee of metric problem, lots of thing implemented in our solution were left to heuristic approach. Core of the algorithm should look like this:

```
Output: Hamiltonian cycle
1. Compute Minimum Spanning Tree (MST)
2. Add a minimum-weight perfect matching on odd degree vertices in MST
3. Find one Eulerian circuit through graph
4. Transform that circuit into Hamiltonian cycle by removing duplicates
```

**Algorithm 1:** Christofides' algorithm (5)

From this point on, algorithm is going to be described, as well as the techniques used in this paper with goal to avoid complexity.

## 2.1 Computing MST

To obtain speed and reduce unnecessarily complexity, Prim's algorithm is used to find minimum spanning tree of graph $G$. Firstly, minimum spanning tree can be defined as a subset of the edges of graph $G$ that connects all the vertices together without any cycles and with the minimum possible total edge weight (6).

Starting with an empty tree, this algorithm always looks for the minimum weighted edge that connects set of points which are already included in tree and those which are not included. After finding an edge, algorithm is repeated with search starting from joined vertex. Edge which is connecting two disjoint sets in graph is called a *cut*. That cut is always calculated to be minimum, therefore Prim's algorithm is declared as a greedy technique. Algorithm can be seen in Algorithm 2.To prove that Prim's algorithm really yields a minimal spanning tree, next idea is presented (3).

Let $e = (u, v)$ be the edge chosen in $k^{th}$ iteration by Prim's algorithm which is not included in real minimal spanning tree. Let $P$ be the path from $u$ to $v$ in minimum spanning tree and one endpoint of $e$ is shared with $e'$ which is edge generated in $(k-1)^{th}$ iteration of algorithm. If weight of $e'$ is less than the weight of $e$, then Prim's

```
Output: Minimum spanning tree
1. Choose some $v \in V$ and let $S = \{v\}$
2. Let $T = \emptyset$
while $S \neq V$ do
    - Choose a least-cost edge $e$ with one
    endpoint in $S$ and one endpoint in $V-S$
    - Add $e$ to $T$
    - Add both endpoints of $e$ to $S$
end
```

**Algorithm 2:** Prim's algorithm

algorithm would have chosen it on $k^{th}$ iteration. That process can be repeated for each vertex in graph $G$ and therefore is proof that Prim's algorithm always generates a minimal spanning tree. Problem *berlin*52 was used as basis to represent steps of Christofides' algorithm. First step, or building of minimum spanning tree can be seen in Figure 1.

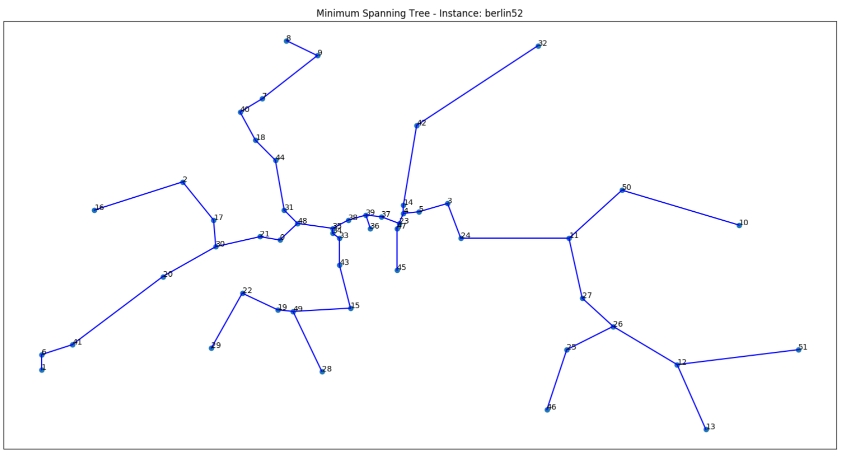

Figure 1: Minimum spanning tree in berlin52 problem

## 2.2 Minimum weight perfect matching

Once when minimum spanning tree is calculated, edges from which is constructed are not sufficient to make a Hamiltonian cycle. That is the reasoning behind perfect minimum weight matching. Perfect matching is a problem where it is necessary to find subset of edges in graph $G$ such that each vertex is met only once in subset. Adding minimum weights to it's name tells us that that subset should be a subset

with minimal weights. One of the examples for solving this problem could be Edmonds' algorithm, but due to it's complexity, proposed method for finding perfect matching is a greedy one.
Algorithm constructed for this purpose as input gets all the vertices from original $G$ that have odd degree in generated minimum spanning tree. Because calculation of Christofides' route is used mainly as starting point for other heuristics, it is not crucial to obtain minimum weight perfect matching. To simplify algorithm,, proposed solution could be picking every second edge from route of TSP in odd vertices graph solved with greedy nearest neighbour approach. Algorithm would look like this:

1. Calculate TSP tour with odd degree vertices in MST
2. Pick every second edge from route and proclaim it as solution for matching problem

**Algorithm 3:** Matching algorithm

Assumption taken into account is one that route obtained with greedy algorithm could result with low cost in each edge and therefore be suited to this task. Output of the algorithm is minimum spanning tree graph expanded with edges on odd degree vertices, which is in fact Eulerian graph. How it looks in practice is shown in Figure 2. With black color greedy TSP route is presented and red ones are matched odd degree nodes.

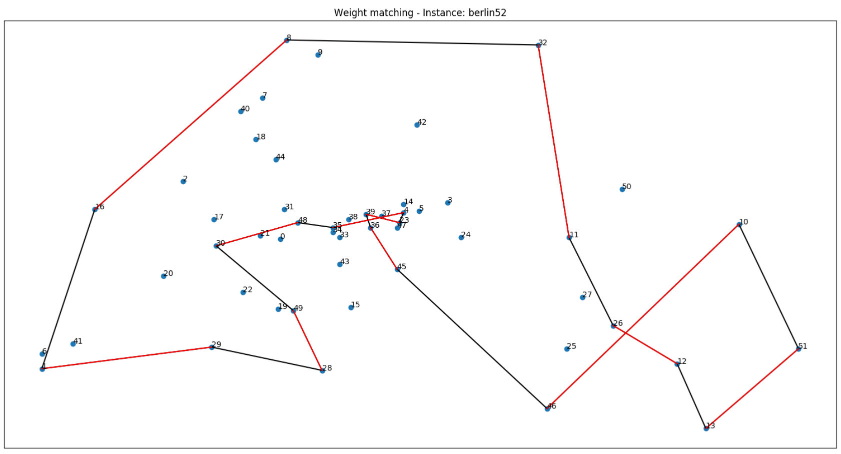

Figure 2: Weight matching done with greedy algorithm. Black shown is greedy tour while red lines are matched nodes.

## 2.3 Collecting hamiltonian tour

Already known fact is the one that it is possible to define hamiltonian cycle trough Eulerian graph (every node has even degree). Again, for the purpose of this algorithm it is not of great importance to get best possible route from Eulerian graph, only thing needed is to find circle throughout it. Since in every Eulerian graph it is possible to find a route that starts at one vertex and visits each of other while returning back in last step, by greedy going from vertex to vertex through connected edges, every time one is going to end up with some cycle. Moreover, with this approach, after running this simple algorithm, the output contains a few circuits which can be seen on Figure 3, they are connected with node that they are sharing (since starting point is shared graph).

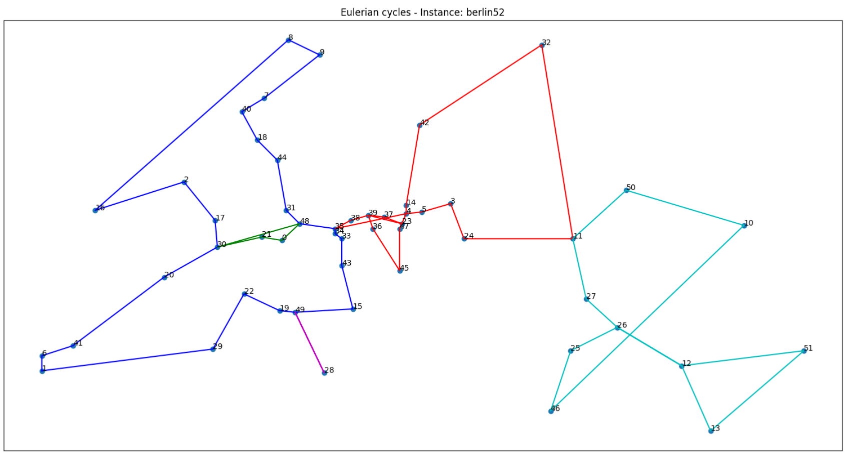

Figure 3: Result of running algorithm and obtaining multiple circuits, which are going to be connected afterwards.

After unfolding those circuits to single big one, only thing left to do is to remove duplicate nodes after which Hamiltonian graph is obtained and algorithm is finished0. Finally, result of Christofides' tour can be seen in Figure 4

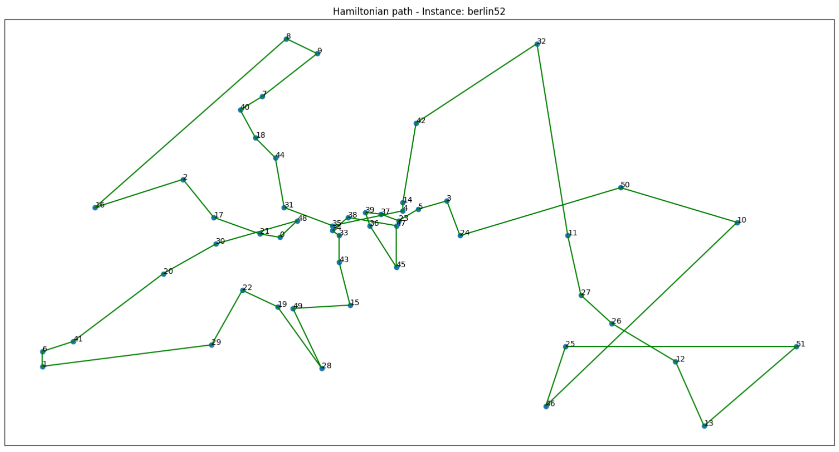

Figure 4: Final result of Christofides' algorithm

## 3 Simple Immunological Algorithm

SIA is a shortage for Simple Immunological Algorithm. It is one of the artificial immune systems algorithm that joins abstraction of real life system to solve engineering problem (7). Goal is to create as many antibodies which are nucleus suited for solving one special task, in this case it is problem of travelling salesman. Algorithm starts with population of antibodies, where every antibody refers to one TSP solution. Fitness of each antibody, or how well antibody performs, is distance of solution route, where clearly shorter is better. Algorithm's core should look like following:

```
1. Initialize population $P_0 = \{x_1, x_2, ..., x_d\}$
2. Evaluate($P_0$)
while Until stoppage condition do
    $P_{CLO} \leftarrow clone(\mathrm{P}_i)$
    $P_{HYP} \leftarrow hypermutate(\mathrm{P}_{CLO})$
    Evaluate($P_{HYP}$)
    $P_{i+1} = P_i + P_{HYP}$
end
```

**Algorithm 4:** Simple immunological algorithm

To briefly explain the algorithm, let the starting position be initialized population. While it is common to initialize first population with random samples of TSP tours, in this algorithm, initialized population would consist out of tours slightly changed from solution obtained from Christofides' algorithm. After evaluating (collecting fitness of each antibody in population or calculating route's distance) loop is entered, and it proceeds on running till stoppage condition is hit. That condition can be some predefined number of iteration, or in this case, running time of 2 seconds. Cloning population in this case means creating new population with equal number of antibodies that performed best in previous iteration. That number is up for user to decide on. For example, with population of 100 antibodies, 5 of best ones could be picked and cloned 20 times to produce new population. Furthermore, that population doesn't introduce anything new to algorithm, therefore term of mutation needs to be introduced. Mutating each of the antibodies in population by switching two towns in route and performing local 2-opt optimization which is going to be explained later on, can lead to better solutions that were at the start similar to ones in previous generation (introduction of new DNA). New generation is then picked between old one and mutated one, depending on preferences. It is important to mention that algorithm is elitistic, which means that best solution is always preserved, from iteration to iteration. With this simple algorithm on top of some other construction algorithm, extraordinary results can be obtained. Those results are presented in next subsection, mostly based on running immunological algorithm on top of Christofides.

| **TSP Instance** | **SIA after Christofides** | **SIA with random starting point** | **Optimal solution** |
|---|---|---|---|
| berlin52 | 7563.3 ± 73.7 | 7542.0 ± 0.0 | 7542 |
| ch130 | 6141.9 ± 23.3 | 6160.0 ± 19.6 | 6110 |
| a280 | 2605.1 ± 12.8 | 2630.3 ± 18.6 | 2579 |
| pr439 | 109752.9 ± 1255.9 | 110502.0 ± 801.1 | 107217 |

Table 1: Performance of Simple Immune Algorithm with Christofides starting solution vs. Random starting solution

## 4 2-opt heuristic

Given a certain path, one can apply different neighbor searching techniques and perform local changes to improve the tour (Figure 5). Those techniques are fairly simple and often they yield satisfying results. Among those, 2-opt is an algorithm that deletes two edges on the tour and tries to reconnect them in some other way (1). The process is repeated until no tours with shorter distances can be found. Furthermore, the tour is then referred to be 2-optimal.

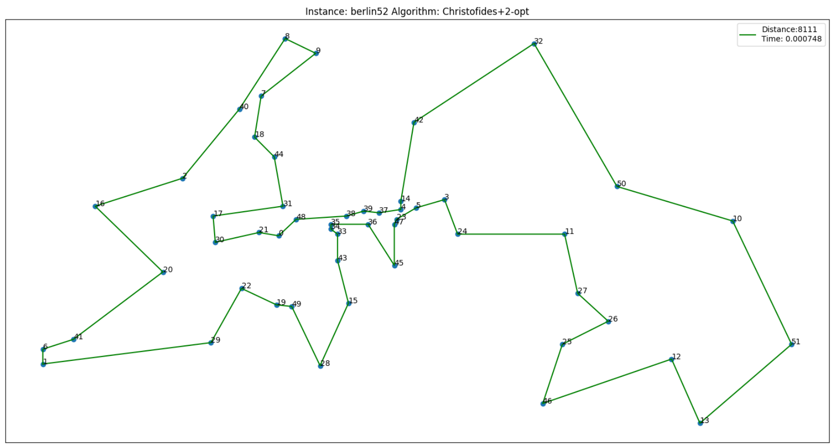

Figure 5: Improvement of Christofides' tour with 2-opt local optimization.

## 5 Implementation and results

Before discussing results of described algorithm, few words should be said about implementation. First important thing to decide when starting a project like this one is certainly choice of programming language. Since C++ has shown best performances in the past, and it is faster than most programming languages, except C, which was much more demanding for usage. Dealing with the simple structure like graph that include vertices and edges, picking the suitable data structure wasn't challenging task. For most part, either simple $Vertex$ class described with city coordinates or just position of city presented as $integer$ is used. At the very beginning, distance between each of the cities is precalculated, for faster computation in later point of algorithm, meaning that we can obtain each distance in $O(1)$ time. That distance matrix is then propagated to most of the functions since distance describes the most valuable information. To encapsulate structure of an graph's edge, *pair* structure of two integers was used. Followed by the $KISS$ programming principle, we tried to keep it as simple as possible, which resulted with better performances. Result on problem that was described previously can be seen in Figure 6.

To ensure good starting point for optimization algorithm, result was obtained by running SIA algorithm on top of Christofides with limitation on computational time which was 2 seconds. With such requisition for running time, it was crucial to find good starting point, and ensure that stoppage is time based, since running algorithm with high number of cities can be very time consuming. How fitness was moving though iterations in this setting has been ploted on Figure 7. This test was result of running SIA with population of 9, and every time cloning and

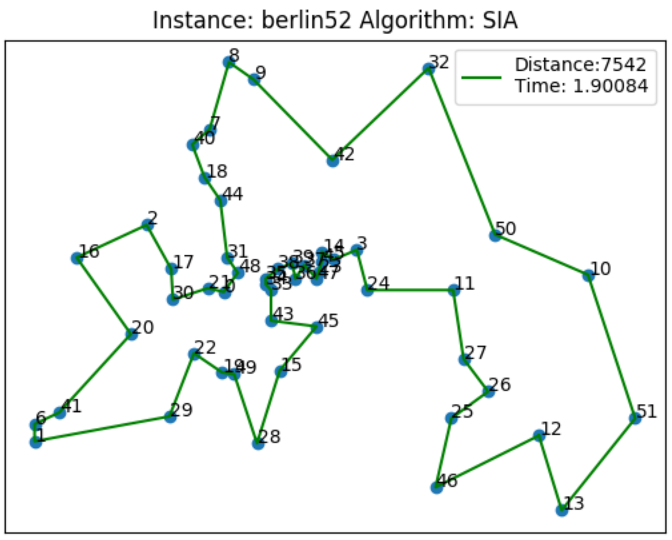

Figure 6: Final result after running SIA on top of the Christofides algorithm

mutating best 3 antibodies. That setting was also used for Kattis submission. For purpose of visualization and better insight in result, drilling problem named $a280$ was picked. Drilling problems are usually hard to solve because of specific (usually densely populated) cities positions. They are named drilling because it is required to find shortest drilling route through electric circuit board. Every trial of algorithm differs in number of iterations, therefore that number is restricted to 42 iterations which is average number of iterations possible under 2s. As we can see, distance is constantly decreasing, which means, on average, in every iteration better solution is found. Reason for this is definitely it's randomized nature which means that with enough time there is high chance of finding even better solution.

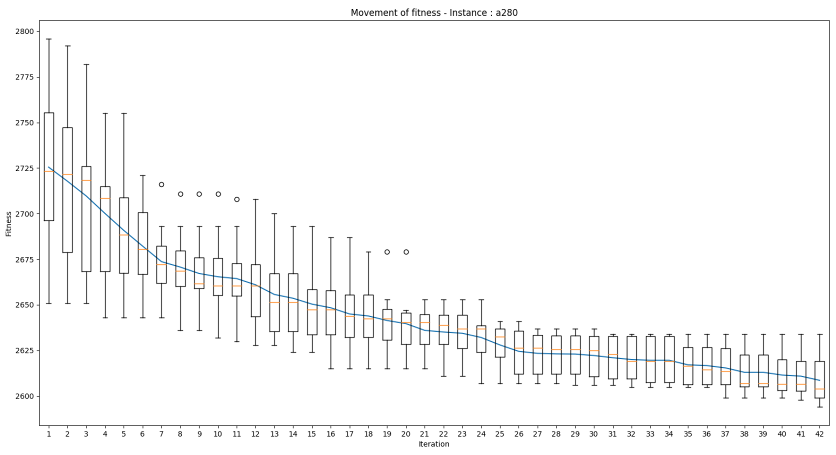

Figure 7: Movement of fitness through SIA iterations in $a280$ problem. Each epoch is visualized with boxplot, while blue line is presenting average fitness.

# 6 Additional algorithms

## 6.1 Simulated Annealing

The name itself comes from metallurgy, where various metal components are put to high temperature in order to get a proper crystal structure. The temperature is progressively being lowered, until the final structure has been obtained. A well conducted procedure will give the metal better properties than before. While temperature is being lowered, metal is transferring from one energy state to another. If possible, moving to a lower energy state will always be done, while passing into a higher energy state would happen with probability of:

$$P(\Delta E) = exp\left(\frac{-\Delta E}{k \cdot t}\right) \tag{1}$$

where $k$ is the Boltzmann constant, $t$ is the current temperature and $\Delta E$ is the difference between the two energy states.
Benefit of moving towards higher energy state can be motivated with wider exploration of solution space. The probability of reaching a higher energy state is greater when the temperature is high, while low temperatures serve for fine tuning the properties (7).

The algorithm solution could be one of the valid tours. Measurement of the fitness (the quality of our solution) can be done by calculating its tour distance. One solution outperforms another if it has better fitness, i.e. smaller route distance.

Higher temperature in the beginning allows for more frequent transfers into high energy states, i.e. selecting solutions with worse tour distance for getting a better ground. As the temperature lowers, the probability of transferring to higher states is decreasing towards zero and therefore solution is dragged into a local optimum attractor.

For creating the neighbor solutions as next candidates for energy states, switching 2 towns on the route was applied. After each iteration, best solution was enhanced with a 2-opt local optimization.

All things considered, the algorithm would look like this:

```
Output: Found city tour after certain number of iterations
$solution \leftarrow pickSolution()$
for $i \leftarrow 1$ to $n$ do
    $temp \leftarrow getNewTemperature()$
    for $j \leftarrow 1$ to $m$ do
        $neighbor \leftarrow generateNeighbor(solution)$
        if $neighbor.fitness < solution.fitness$ then
            $solution \leftarrow neighbor$
        else
            $\Delta \leftarrow neighbor.fitness - solution.fitness$
            $neighbor \leftarrow solution\ with\ probability\ exp(-\frac{\Delta}{temp})$
        end
    end
end
return $solution$
```

**Algorithm 5:** Simulated annealing

First experiment conducted was running the Simulated Annealing algorithm by itself. New temperatures were obtained by multiplying the current temperature with factor of 0.99, ensuring slow decreasing of the temperature so the algorithm could perform efficiently. The outer loop varied from 100 to 1000 iterations while the inner loop that generates the neighbors was around the same number itself. Since the running time of the algorithm was around 2 seconds per test case, Both of those loops were decreased to around 250 for the outer loop and 100 for the inner loop. Generation of neighbor states only included switching of two cities. After each iteration of the outer loop, 2-opt local search heuristic would be performed on the solution. This wasn't used on the inner loop due to time limit as it harmed the performance of the output solution. Results after Christofides outputted solution are also provided in Table 2

| **TSP Instance** | **SA after Christofides** | **SA with random starting point** | **Optimal solution** |
|---|---|---|---|
| berlin52 | 8086.2 ± 129.5 | 7938.6 ± 155.4 | 7542 |
| ch130 | 6432.4 ± 121.8 | 6486.9 ± 135.8 | 6110 |
| a280 | 2709.9 ± 60.7 | 2752.6 ± 41.1 | 2579 |
| pr439 | 117977.5 ± 2311.2 | 120871.3 ± 3011.4 | 107217 |

Table 2: Performance of Simulated Annealing with Christofides starting solution vs. Random starting solution

Driven by the poor performances from the plain Simulated Annealing, an experiment was conducted by fine tuning the final solution of the Christofides algorithm by putting it as the starting solution for Simulated Annealing, hoping for a better performance. The main reason for merging the two algorithms together was the speed of Christofides algorithm which used only 10% of allowed running time, letting the Simulated Annealing to do the rest of the job.
Often the case would be that SA would output the solution even worse than the Christofides output, since high temperatures would probably discard that solution. Further changes were made by elitist keeping of the best solution found during the algorithm was run, but it didn't provide any meaningful improvements. Results are provided in the Table 2.

## 6.2 Ant Colony Optimization

Motivation for this algorithm is taken from nature, where ants often tend to find an optimal way from their anthill to food. Every ant is dropping chemical substances called pheromones on their way to food. The amount of pheromones dropped is proportional to the amount of food he found and their smell lead other ants towards the same way. The result of the iterative process is the optimal path found to get food. Out of all derived algorithms from this principle, Min Max Ant System (a sub-algorithm of ACO family) was chosen and experimented on (7).

The algorithm itself can be easily written in pseudocode ($n$ is the number of iterations, $m$ is the number of ants in the colony), shown in Algorithm 6.

However, there are a few things to distinguish.
There must be a defined way for an ant to create a solution, it can't be random

```
Output: Best path found for TSP
for $i \leftarrow 1, ..., n$ do
    for $ant \leftarrow 1, ..., m$ do
        ant.createSolution()
        ant.fitness ← evaluateSolution()
    end
    evaporatePheromones(); bestAnt ← getBestAnt()
    updatePheromones(bestAnt)
end
return $bestRoute$
```

**Algorithm 6:** Min Max Ant System

because we wouldn't gain any knowledge out of it. Also, there are several factors that account for an ant to move from one city and another - the strength of pheromones on that path and the distance between cities. We can construct a greedy heuristic function $\eta$ that has a greater value for cities that are close, than for cities that are farther away from each other:

$$\eta_{ij} = \frac{1}{d_{ij}} \text{ (d is the distance between cities)} \tag{2}$$

If we denote $\tau$ as the strength of the pheromone trace, we can construct a categorical probability distribution for each city that an ant has not visited yet:

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_l \tau_{il}^\alpha \cdot \eta_{il}^\beta} & \text{for every unvisited city } l \\ 0 & \text{otherwise} \end{cases}$$

Where $\alpha$ and $\beta$ are the hyperparameters that were chosen empirically. With this, we prevented the ant from choosing the solution in a greedy way and allowed for some randomization in creating the solution.
Evaporation of pheromones could be done by multiplying the current pheromone track between every city by a certain factor:

$$\tau_{ij} = \tau_{ij} \cdot (1 - \rho) \tag{3}$$

while the updating can be done by every ant or by the best ant found in the iteration:

$$\tau_{ij} = \tau_{ij} + \Delta\tau$$
$$\Delta\tau = \frac{1}{\text{ant tour distance}} \text{ (for every edge that ant visited)}$$

We can evidently see that more pheromones will be placed on a certain edge if a tour distance of an ant is shorter.

Although the algorithm is used vastly for construction problems such as TSP, it has been shown that the running time of the algorithm was to small to provide any meaningful solutions. Results are shown in Table 3.

| **TSP Instance** | **ACO with 2 second restriction** | **ACO without time restrictions** | **Time elapsed** | **Optimal solution** |
|---|---|---|---|---|
| berlin52 | 7658.6 ± 128.1 | 7659.6 ± 122.1 | 2.92 ± 0.36 | 7542 |
| ch130 | 6729.4 ± 186.5 | 6500.1 ± 144.1 | 3.98 ± 0.23 | 6110 |
| a280 | 3303.5 ± 59.4 | 2993.7 ± 94.1 | 14.02 ± 0.87 | 2579 |
| pr439 | 142093.9 ± 1960.1 | 125474.4 ± 6072.3 | 34.21 ± 0.95 | 107217 |

Table 3: Results of Ant Colony Optimization given 2 second time like on Kattis vs. unlimited time

## 6.3 Branch and bound

Branch and bound is an algorithm that can, given enough time, yield an optimal TSP solution. One can model the starting city position as a tree node, from which we can branch into $n-1$ children nodes as the remaining number of cases where we can go from that city, given $n$ cities. At every node we must know its cost, which can be correlated to the distance passed so far. When we branch to the terminating node (tour completed and no more cities to visit), we calculate the finishing cost and update the upper bound for a cost, as we can't have anymore solutions with greater cost because evidently that would be a worse solution from the one found so far. Any node with cost greater than the upper bound is removed, until we are left with the only solution, i.e. the optimal solution to the TSP problem (2).

For the first solution, a reduced cost matrix is assigned from the distance matrix itself, with the distance between one city to itself given as infinity ($\infty$). The cost of reduction is always added to the total cost of the node.
When we branch to deeper nodes, it's cost is calculated as the sum of the parent cost, reduction cost of the new matrix and the distance from one city to the parent city in the parent's matrix (PM):

$$C = C_{parent} + C_{reduction} + PM[parent][child]$$

The terminating tour will then form a valid TSP tour with the cost that can possibly update the upper bound. The pseudocode for the algorithm is given below:

Although the algorithm is exact and it yields the optimal solution every time, it could often be the case that it takes exponential time to reach the solution (worst case). Even with removing some of the nodes with the upper bound, which considerably reduces the amount of solutions examined (the case of exhaustive, brute force

```
Output: Best path found for TSP
$currentNode \leftarrow rootNode$
while solution not found do
    1) Branch currentNode until you get to a solution (picking the lowest cost)
    2) Update lower bound with found solution
    3) Delete possible next current nodes with higher upper bound
    4) Select next current node with lowest cost
end
```

**Algorithm 7:** Branch and Bound Algorithm

search), the number of solutions to the TSP is sometimes too much so it is mostly used for smaller number of cities.

## 7 Conclusion

All in all, the algorithms which were experimented on provided results which could be assumed before trying them. Best results, considering the time limit and distance of the route, were provided with the Simple Immunological Algorithm combined with the Christofides' algorithm solution as a starting point and 2-opt heuristic as a help for finding a satisfying solution. Although the algorithms were vastly explored, there is still open room for doing a bit more research and improving the selected algorithms, as well as trying some new ones.
For the optimization algorithms, one could spend a lot of time in searching for the optimal algorithm hyperparameters in order to get a better searching pace of the solutions (e.g. temperature management in Simulated Annealing, number of iterations in algorithms, number of ants in ACO etc.). However, a quest for that search can take an enormous amount of time, therefore it was not conducted in this paper and only a few values have been experimented on, which empirically led to selecting one of them. For the Christofides' algorithm, one could improve it by doing a minimum perfect weight matching and boosting finding of Euler path instead of using greedy approaches, which could improve a starting solution since the algorithm only gives a 1.5-approximation.
As for the implementation of new algorithms, one could consider implementing a 3-opt-heuristic (for boosting the other algorithms), Lin-Kerninghan heuristic, linear programming algorithms, etc.

## References

[1] Lyle A. McGeoch David S. Johnson. *The Travelling Salesman Problem: A Case Study in Local Optimization*. 1995.

[2] CSA Game Theory Lab. Optimal solution for tsp using branch and bound algorithm. URL: `http://lcm.csa.iisc.ernet.in/dsa/node187.htmll`.

[3] Stanford University. Greedy algorithms. URL: `https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/14/Small14.pdff`.

[4] Thomas Sauerwald Cambridge University. Approximation algorithms: Travelling salesman problem. URL: `https://www.cl.cam.ac.uk/teaching/1516/AdvAlgo/tsp.pdf`.

[5] Wikipedia. Christofides algorithm. URL: `https://en.wikipedia.org/wiki/Christofides_algorithm`.

[6] Wikipedia. Minimum spanning tree. URL: `https://en.wikipedia.org/wiki/Minimum_spanning_tree`.

[7] Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi. Metaheuristike*. 2013.