Traveling Salesperson Problem

A Comparison of Heuristics

Florian Hochstrasser

2019-01-03

Abstract

The performance of several heuristics from the families of construction and improvement heuristics were implemented in python and compared for their perfromance with respect to finding the shortest cycle among a set of nodes in a network. Consistent with literature, it was found that Best Insertion outperforms other construction heuristics. For the improvement family of heuristics, Greedy Local Search with a combination of moves performed best. Simulated Annealing resulted in inferior cycle lengths compared to other improvement heuristics when run with the parameters outlined in the project description. At the expense of more computation time, the results could be improved.

Contents

1 INTRODUCTION

The traveling salesperson problem, first mentioned in (Voigt 1831) is a discrete optimisation problem for which to date no algorithm with polynomial execution time has been discovered. It describes a network of nodes that have to be connected in a tour so that the path connecting all nodes is as short as possible. Each node can only be visited once and thus has exactly two connecting edges.

It is therefore a permutation problem of the set of vertices V containing $n = |V|$ elements with $\frac{(n-1)!}{2}$ possible solutions in the discrete domain $\mathbb{D}$.

Let $\mathbf{G} = (V, E, d)$ the graph where V is the set of vertices of cardinality $n = |V|$, $E \subset V$ x V the set of edges connecting the vertices in V and $d : E \longrightarrow \mathbb{R}^+$ the function assigning a distance $d(e)$ to each edge $e \in E$.

A Hamiltonian cycle, see (Hamilton 1856), is a cyclic path visiting all vertices $v \in V$ exactly once. This problem is of the family *NP-complete* and as such is considered a *hard* problem to solve.

Since the complexity of the problem grows fast with increasing size of V, exact solutions (i.e. branch and bound, brute forcing) are only feasible for small networks. Therefore, the problem is usually solved using heuristics.

There are two main families of heuristics for solving the problem. The construction heuristics start with one or a few nodes drawn at random and build the tour from there. The algorithm stops when all nodes are connected in the tour. The family of improvement heuristics

start with a randomly chosen tour and then look for possible improvements using several different strategies. The algorithm stops after a certain number of executions.

In the scope of this project, the following heuristics will be implemented and compared:

- Construction heuristics
  - Insertion (Best insertion): The vertex resulting in the smallest increase in tour length when inserted between to nodes (vertices) already in the tour is selected.
  - Nearest Neighbor (Best-best insertion): The nearest vertice to the open ends of the current partial tour is inserted.
  - Greedy (Shortest Edge): The shortest edge not resulting in a prematurely closed loop or a vertex with more than two connections is inserted.
- Improvement heuristics
  - Greedy local search with the moves:
    - Swapping
    - Translation
    - Inversion
    - Mixed move: Each of the other moves is selected at random with equal probability
  - Simulated Annealing with the criteria
    - Metropolis
    - Heat bath

1.1 NETWORK

The network to be simulated consists of 411 nodes in $\mathbb{R}^2$. Figure 1 shows the locations of the nodes.

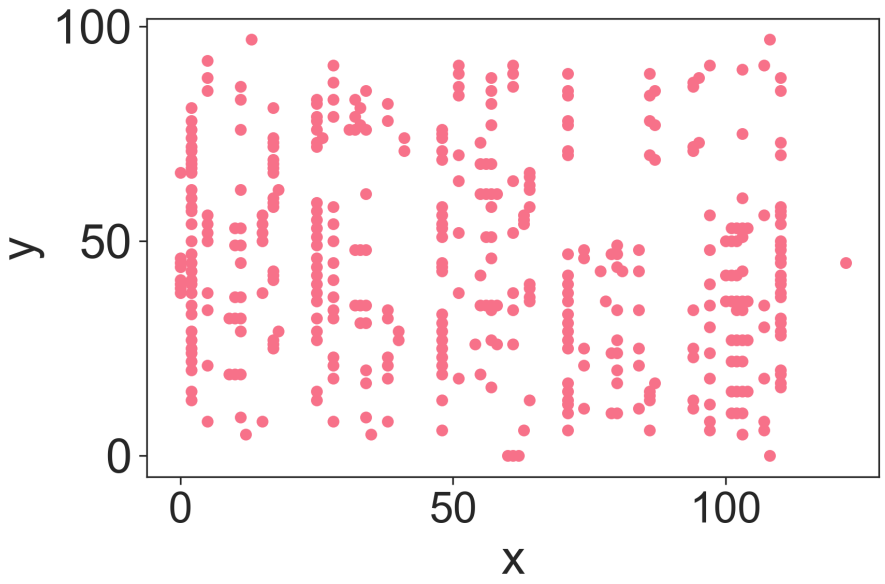

Figure 1: Locations of the 411 nodes in the network to be solved. There are several areas of interest. We can see vertical lines near 0, 25, 45, 70 and 110 on the x-axis. There is also one outlier at x = 120 and a dense cluster of nodes around x = 100 and y = 10 ... 50.

2 METHODS

2.1 IMPLEMENTATION

The project was implemented in Python (Python Software Foundation, https://www.python.org/), version 3.6.7. Packages *numpy* (Oliphant 2006), *pandas* (McKinney 2011), *scipy* (Jones et al., n.d.) and *matplotlib* (Hunter 2007) were also relied on.

The source code may be obtained online[1] and is also included in the appendix.

2.1.1 DATA STRUCTURES

The distance matrix was implemented as a symmetric matrix built with functions in scipy's *spatial* module.

Cycles for the construction heuristics were represented using association matrices $\mathbf{A}$. These matrices are symmetric n x n matrices where each row index and column index i, j represent the id of a node in the network. The element $A_{i,j}$ has a value of 1 if the nodes i and j are connected, and zero otherwise. The marginal sums $\sum_{i=1}^{n} A_{i,j}$ for $j = 1...n$ and $\sum_{j=1}^{n} A_{i,j}$ for $i = 1...n$ are equal to 2 when the Hamiltonian cycle is finished.

For improvement heuristics, cycles were implemented as python lists, holding the integer id's of the nodes in the network and thus representing a permutation of the nodes in V.

Nodes and edges were implemented as python objects of a respective class.

[1] https://github.com/datarian/TravelingSalespersonProblem

2.1.1.1 Traveling Salesperson Problem Configuration Three classes represent the network on which heuristics operate. These are:

- `Node`: A vertex in the network with x- and y-coordinates and a number
- `Edge`: A line connecting two nodes with a length (*cost*/*distance*)
- `TravelingSalespersonProblem`: Has a list of `Node` objects representing the vertices in the network and has a distance metric registered that is used to compute the distance matrix $\mathbf{D}$, an n x n symmetric matrix describing the distance between any two nodes in the network.

The edge objects were only employed for the Shortest Edge heuristic, all other heuristics operate directly on the node objects and the distance matrix.

2.1.1.2 Cycle representation For the construction heuristics, the cycle was represented through an association matrix $\mathbf{A}$ of data type boolean. The matrix has the same shape as the distance matrix held in the `TravelingSalespersonProblem` class. The matrix is initialized as $False$ and each connection between two nodes i, j is indicated by a $True$ value in $\mathbf{A}_{i,j}$.

For the improvement heuristics, a list with the current permutation of V was chosen to represent the cycle. This lowers the cost of calculating the loss L of the cycle. While for the construction heuristics, L is only calculated once when the cycle is finished, it has to be calculated at each iteration for the improvement heuristics.

2.1.2 CODE STRUCTURE

Two modules were created:

- `tsp`: Holds the classes for the TSP configuration
- `tsp_heuristic`: Holds the implemented heuristics

The base class `TspHeuristic` was subclassed into `ConstructionHeuristic` and `ImprovementHeuristic` with respective child classes for each heuristic. The moves were also implemented in an object-oriented manner and instantiated as necessary.

2.1.3 HEURISTICS

The heuristics were implemented according to the presentation of algorithms in the course slides.

For the Shortest Edge construction heuristic, a deterministic approach to break ties was chosen by always picking the edge with the smallest node id for equally sized edges. This resulted in a deterministic algorithm.

2.2 PAIRWISE COMPARISON

For the pairwise comparison, Monte-Carlo experiments were conducted, evaluating each heuristic 30 times. From these samples, the relevant statistics were computed as follows: The minimum, maximum and mean loss per heuristic were computed through functions of the package *pandas*, the confidence interval was computed with *scipy.stats*'s *t.interval* function, giving the standard deviation with *numpy.std*.

Hypothesis tests were performed by means of a two-sample T-test through package *scipy*'s stats module using the function *ttest_ind* on the mean loss of the heuristics.

Test assumptions:

- Independence between samples (since the heuristics rely on random decisions).
- Unequal variance, since the heuristics employ distinct approaches in solving the problem.

The statistic t is computed as seen in Equation 1. $\bar{x}$ and $\bar{y}$ are the sample means, s_x^2 and s_y^2 are the sample variances. x and$y$ are the two samples, here the losses resulting from two different heuristics.

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y}}} \tag{1}$$

It is evident that negative values for t can be interpreted as $\bar{x} < \bar{y}$ while for positive values, the opposite is true.

Test hypotheses:

H_0: The sample means are equal.

H_1: The difference of the sample means is $\neq 0$

The hypotheses were evaluated at a confidence level of $\alpha = 5\%$. Since the test is two-sided, meaning that H_1 states that the difference in means $\neq 0$, H_0 is rejected if the p-value is larger than $\alpha/2 = 2.5\%$. We then conclude that there is a signifcant difference in sample means at a cofidence level of $\alpha = 5\%$ and can make a statement about which of the two heuristics performs better.

Complementing the outcome of the statistical test mentioned above, an acceptance interval $[-\beta, \beta]$ can be computed (see Equation 2). If the assumptions of the test are correct, in order to not reject H_0 at $\alpha = 5\%$, we would expect the difference in mean loss between two heuristics to fall into this interval. Likewise, a difference in means outside of the interval means to reject H_0. Furhtermore, the location (left / right of the acceptance interval) allows to make a statement about which heuristic performed better.

$$\beta = t_m^{\alpha/2} * \sqrt{\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y}} \tag{2}$$

where $t_m^{\alpha/2}$ is the quantile of the Student's distribution for $\alpha/2$ with m the degrees of freedom, n_x, n_y the sample sizes of the two samples and s_x^2, s_y^2 the sample variances. Furthermore, the degrees of freedom $m = \frac{(\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y})^2}{\frac{s_x^2/n_x}{n_x+1} + \frac{s_y^2/n_y}{n_y+1}} - 2$.

2.3 PERFORMANCE PLOTS

For the Greedy Local Search variants, the resulting loss at each iteration of the heuristic's execution was recorded. Plotting the losses of a heuristic against the number of iterations

then allows to make inference on it's performance.

For Simulated Annealing, the minimum, maximum and mean of each run at a given temperature were saved and then plotted against the temperature.

From the resulting performance plots, convergence can be studied, which is useful to assess the best performing heuristic and, furhtermore, can inform about improved parameters (number of repetitions, maximum and minimum temperature) for subsequent runs of the heuristics.

3 RESULTS AND DISCUSSION

All calculations were done in a Jupyter notebook[2]. The notebook can be retrieved or examined in the github repository for this project[3].

A summary of the resulting losses for 30 runs of each heuristic are given in Table 1.

We can see that two of the construction heuristics are among the top 3, with Best Insertion performing best on average, followed by Greedy Local Search with mixed moves, then the Shortest Edge heuristic. On the other side of the ranking, we find Greedy Local Search with Swap moves. Because the Monte-Carlo experiments for Simulated Annealing were run with Swap moves as well, we find the heuristic with both criteria variants (Metropolis / Heat Bath) producing high losses as well.

It is evident that the Shortest Edge heuristic is deterministic (0 variance). Since the edges are sorted at the beginning, the same sequence of edges is added to the cycle.

Table 1: Summary statistics for all heuristics evaluated. Based on n=30 runs of each heuristic. Blue cells indicate good values, red cells bad values.

	Mean	Min	Max	SD	CI lower	CI upper
Construction						
BestInsertion	1564.02	1499.71	1697.51	44.74	1547.32	1580.73
BestBestInsertion	1945.67	1916.29	1976.27	16.02	1939.69	1951.65
ShortestEdge	1696.93	1696.93	1696.93	0	1696.93	1696.93
Improvement						
GreedyLocalSearchSwap	3878.33	3317.23	4287.82	210.01	3799.91	3956.75
GreedyLocalSearchTranslate	2285.35	2060.27	2529.23	103.21	2246.81	2323.89
GreedyLocalSearchInvert	1797.65	1592.13	4140.74	447.88	1630.4	1964.89
GreedyLocalSearchMixed	1594.49	1485.31	2851.65	239.54	1505.05	1683.94
SimulatedAnnealingMetropolis	3422.29	3156.51	3815.65	152.38	3365.39	3479.19
SimulatedAnnealingHeatBath	3648.97	3427.31	3855.9	120.7	3603.9	3694.04

3.1 PAIRWISE COMPARISON

3.1.1 CONSTRUCTION HEURISTICS

The t-tests are all significant at $\alpha = 5\%$, as can be seen in Table 2. The test results are shown in Figure 2. In both comparisons, Best Insertion won over the alternatives (Shortest

[2]https://jupyter.org

[3]https://github.com/datarian/TravelingSalespersonProblem/blob/master/Probabilistic_Algorithms_Project.ipynb

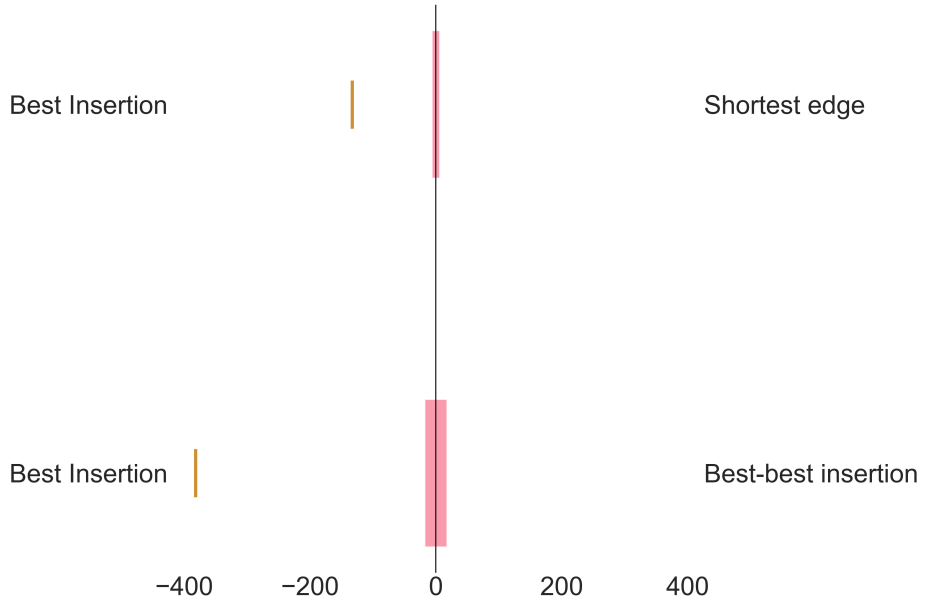

Figure 2: Pairwise comparison of construction heuristics. The acceptance intervals are shown around the center as they are symmetric around zero. The differences in means are shown as vertical lines. As all lines are outside of the acceptance intervals, we can conclude that the difference in means between the results is significant. The location of the vertical lines indicates the heuristic that performed better.

Edge and Best-best Insertion). It can therefore be concluded that Best Insertion is the most efficient construction heuristic evaluated.

Table 2: P-values (labelled pval) and statistics for the student's T-test comparing the construction heuristics.

	pval	statistic
Best vs. BestBest	3.02e-46	-43.99
Best vs. Shortest	2.77e-23	-16.27

3.1.2 IMPROVEMENT HEURISTICS

Again, all t-tests were significant at $\alpha = 5\%$ (see Table 3). The results of the comparison are shown in Figure 3. For the Greedy Local Search variants, Swap performed worse than any other moves. We can further conclude that Mixed moves perform better than Tranlsation. Mixed moves outperform Inversion moves, albeit by a small margin, and are therefore the best variant of the Greedy Local Search.

For Simulated Annealing, the Metropolis criterion was better than Heat Bath.

Table 3: P-values (labelled pval) and statistics for the student's T-test comparing the improvement heuristics.

	pval	statistic
Swap vs. Tranlsate	3.17e-42	37.29
Swap vs. Invert	7.10e-31	23.04
Swap vs. Mixed	1.76e-43	39.27
Translate vs. Invert	2.78e-07	5.81
Translate vs. Mixed	5.90e-21	14.51
Invert vs. Mixed	3.25e-02	2.19
Metropolis vs. Heat Bath	3.11e-08	-6.39

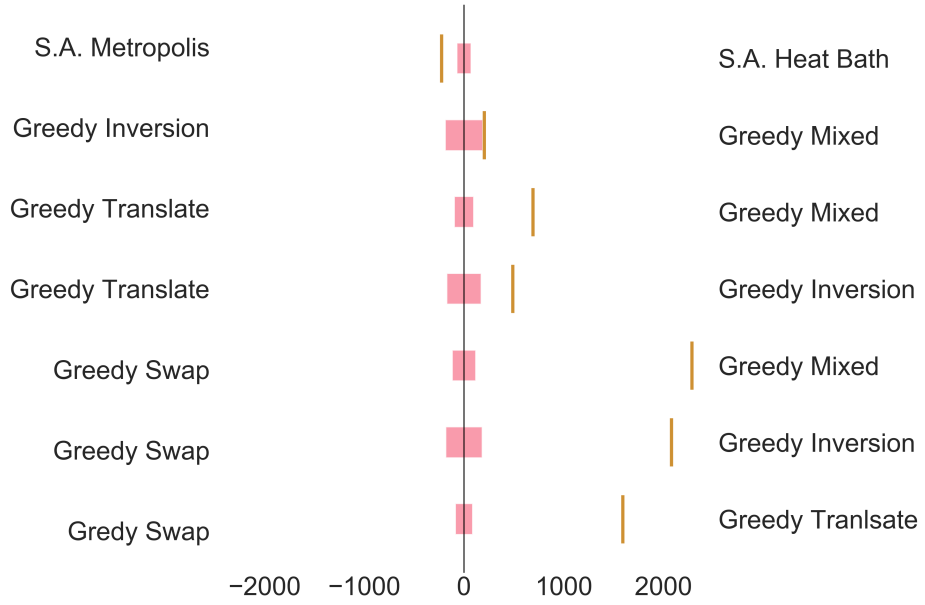

Figure 3: Pairwise comparison of improvement heuristics. The acceptance intervals are shown around the center as they are symmetric around zero. The differences in means are shown as vertical lines. As all lines are outside of the acceptance intervals, we can conclude that the difference in means between the results is significant. The location of the vertical lines indicates the heuristic that performed better.

3.2 SHORTEST CYCLES

The shortest cycles resulting from Monte Carlo experiments with 30 repetions for each heuristic are shown in Figure 4. Looking at the construction heuristiscs, a couple of observations can be made. BestBestInsertion optimises locally, resulting in longer overall cycle length compared with BestInsertion's global optimisation. The shortest edge heuristic is deterministic in the implementation used (ties are broken in a deterministic way), so it always calculates the same cycle. The quality of the solution seems similar to the one obtained with Best Insertion, although towards the end of execution, a few longer edges are inserted, particularly between $\sigma(n)$ and $\sigma(1)$, which increases the loss.

For the improvement heuristics, Inversion and Mixed moves result in nice-looking cycles. Swap moves, on the contrary, perform bad. This is seen in Greedy Local Search as well as both Simulated Anneling runs, where Swap moves were used as well.

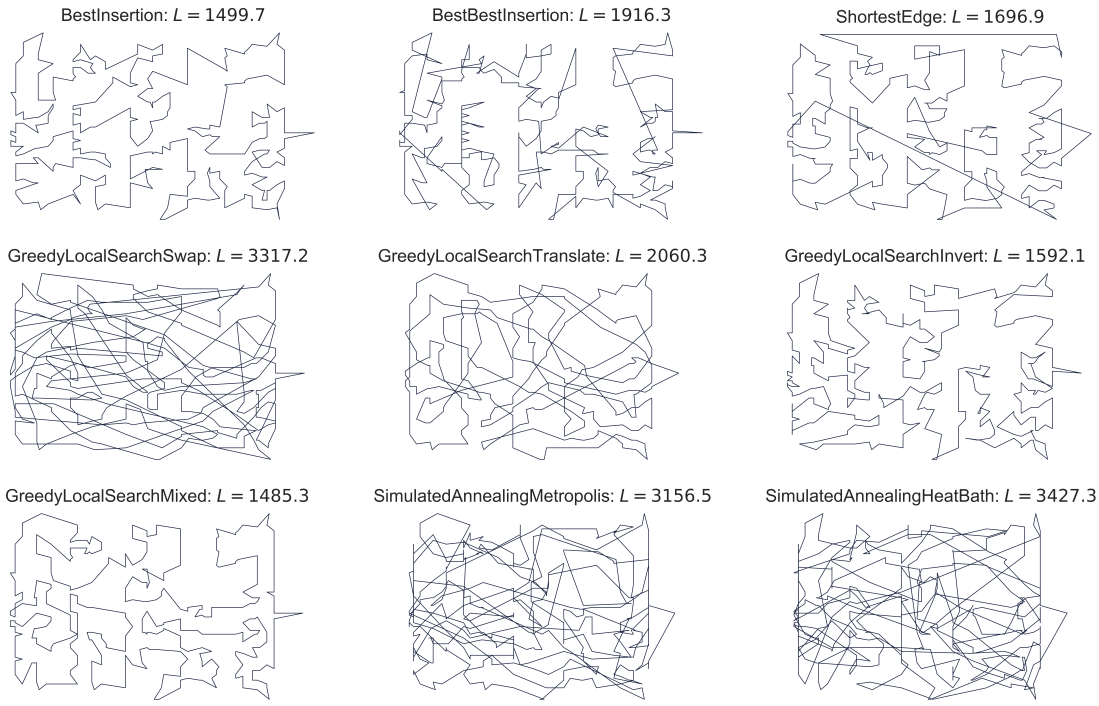

Figure 4: Shortest cycle among 30 runs for each heuristic. It is evident that the Simulated Annealing Heuristics would require more iterations to find a solution closer to the optimum. Also, Greedy Local Search with Swap moves was a weak performer. The best results for the improvement heuristics were achieved with Greedy Local Search using Mixed moves.

3.3 PERFORMANCE PLOTS

3.3.1 GREEDY LOCAL SEARCH

The results for the performance of greedy local search, exhibited in Figure 5, do not directly reflect the results of the pairwise comparison. As a result of randomness, the Mixed move performed worse than Translation in this case, with Inversion being last. It is to be kept in mind that these are single runs and based on the variance in the results seen above, are nevertheless plausible.

We can observe that both Swap and Translation moves take longer to converge. It can be concluded that the fast convergeance of the Mixed move is due to the fast convergence of the Inversion move.

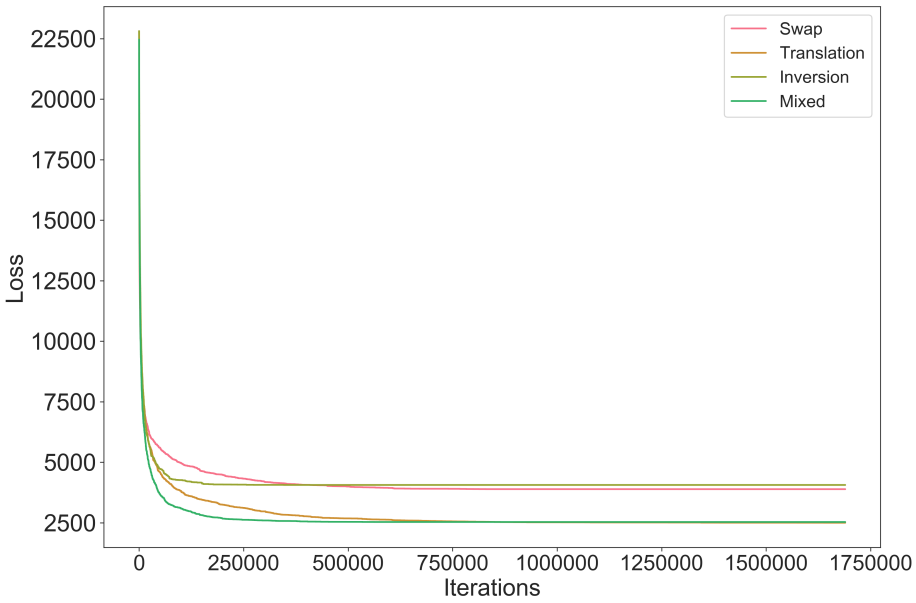

Figure 5: Performance plot for all moves considered in the Greedy Local Search heuristic in direct comparison. Inversion and Mixed moves stabilise at approximatey 200'000 iterations, while Swap takes at least 5 times as long to stabilize. Translation converges at about 500'000 iterations.

3.3.2 SIMULATED ANNEALING

Both performance plots (see Figures 6 and 7) are similar in shape. We can easily see the effect of the decreasing temperature on the probability of selecting a solution with a higher loss than the previous: The magnitude of the solutions at high temperatures is larger in both cases, and the mean of solutions gets closer to the minimum solution as temperature decreases.

The results again reflect the performance of the moves chosen. Swap moves result in the highest losses, while Mixed and Inversion moves are found at the lower end.

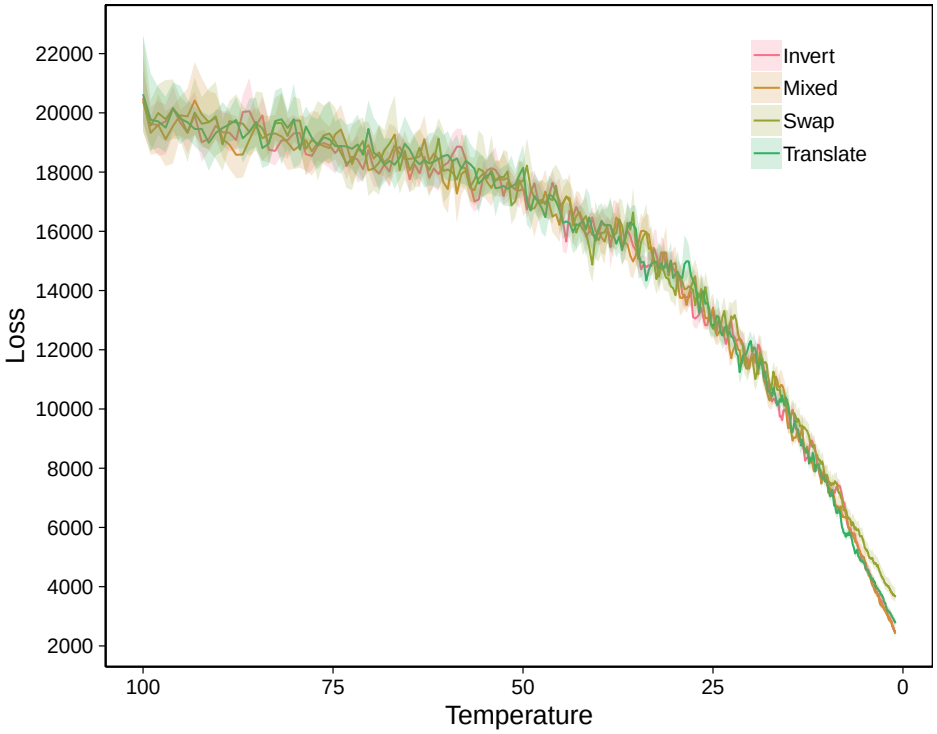

Figure 6: Performance for the moves Swap, Translate, Invert and Mixed for the Metropolis Criterion configuratio of Simulated Annealing. Loss ranges are shown as shaded areas around the mean loss values displayed as solid lines.

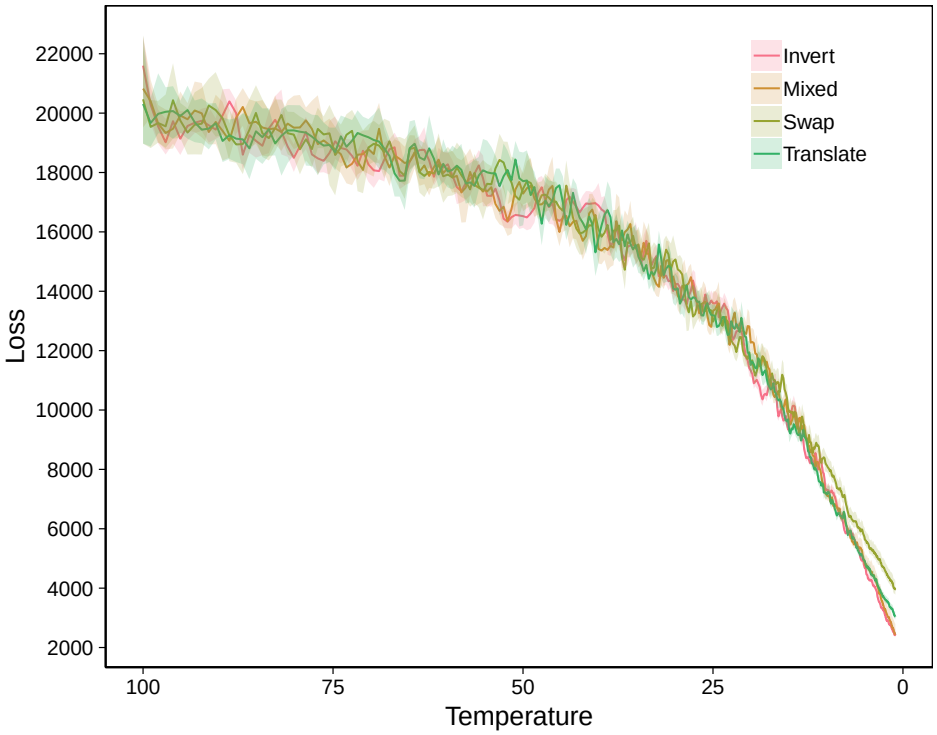

Figure 7: Performance for the moves Swap, Translate, Invert and Mixed for the Heat Bath Criterion configuratio of Simulated Annealing. Loss ranges are shown as shaded areas around the mean loss values displayed as solid lines.

4 CONCLUSIONS

4.1 IMPLEMENTATION

- In hindsight, it would have been more efficient to use incidence matrices instead of association matrices. Calculation of the cycle in the correct order is complex in the case of the adjacency matrix, whereas it can be directly obtained from an incidence matrix (in such a boolean matrix, each row represents one node where exactly one value $\neq False$ indicating $\sigma(i)$, the position of the node in the tour).
- Tie-breaking in the ShortestEdge heuristic would better be implemented with a stochastic approach in order to gain more information on the heuristic's performance through Monte-Carlo experiments.

4.2 FURTHER IMPROVEMENTS

Instead of starting from a random cycle, the improvment heuristics are fed a cycle built with a construciton heuristic. By combining the best construction heuristic *Best Insertion* with the best improvement heuristic *Greedy Local Search Mixed Moves*, a further reduction in loss could be achieved. The combination of heuristics was studied in a Monte Carlo experminent with 30 repetitions. Statistics are shown in Table 4. The shortest cycle produced is displayed in Figure 8.

Table 4: Statistics for a Monte Carlo experiment with 30 repetitions of a combination of Best Insertion and Greedy Local Search with Mixed moves.

	Mean	Min	Max
Loss	1473.2	1441.8	1532.7

Running the Simulated Annealing heuristic with a slower cooling factor or more iterations per temperature, or both, also results in a reduction of loss.

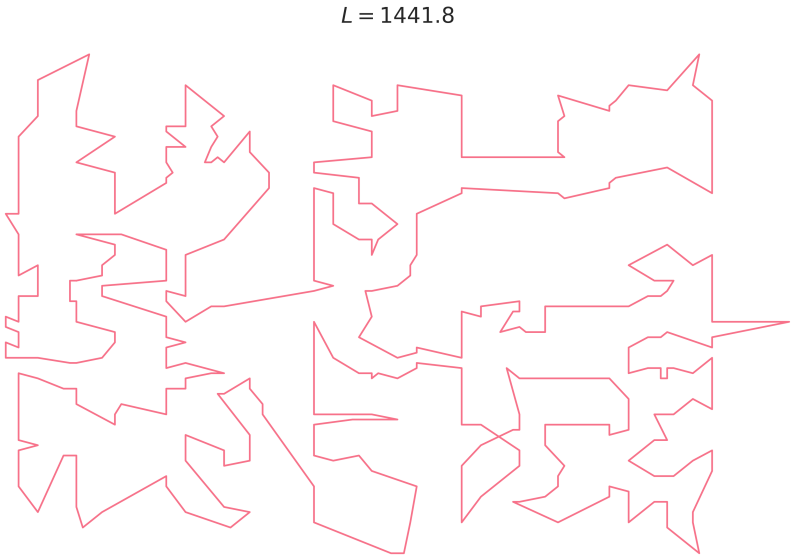

Figure 8: Cycle resulting from a combination of Best Insertion to build the initial cycle, then improving it using GreedyLocalSearch with Mixed moves. Best cycle from 30 repetitions.

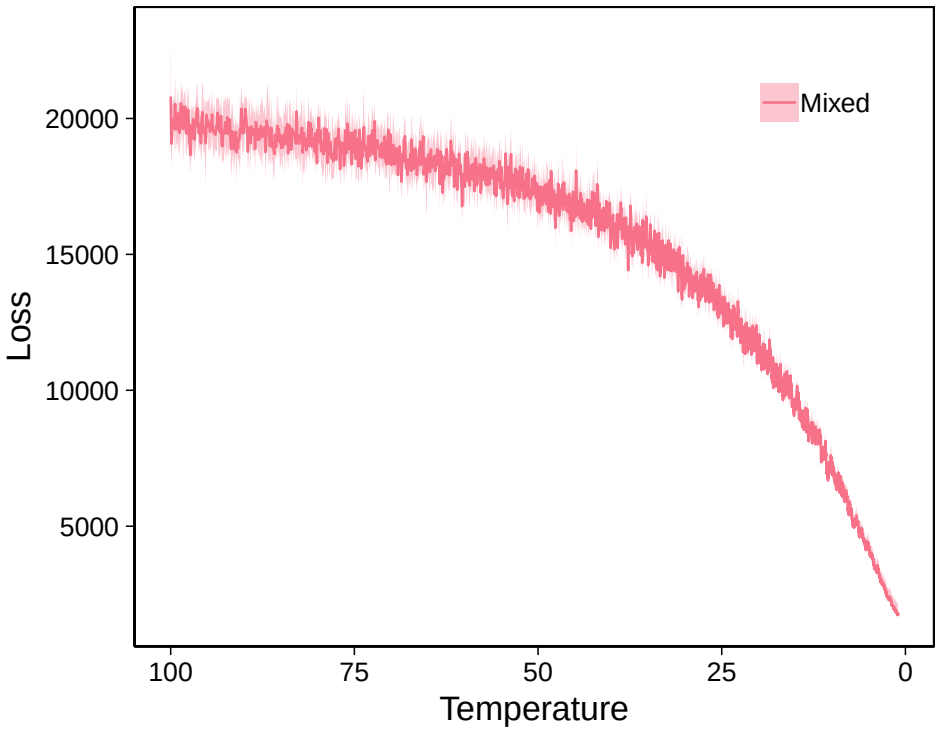

Figure 9: Simulated Annealing with Mixed moves and a cooling factor $\theta = 0.999$. The range of solutions at a given temperature is indicated by the shaded region around the solid line representing the mean loss.

5 REFERENCES

Hamilton, William Rowan. 1856. “LVI. Memorandum Respecting a New System of Roots of Unity.” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 12 (81). Taylor & Francis:446–46.

Hunter, John D. 2007. “Matplotlib: A 2D Graphics Environment.” *Computing in Science & Engineering* 9 (3). IEEE:90–95.

Jones, Eric, Travis Oliphant, Pearu Peterson, and others. n.d. “SciPy: Open Source Scientific Tools for Python.” http://www.scipy.org/.

McKinney, Wes. 2011. “Pandas: A Foundational Python Library for Data Analysis and Statistics.” *Python for High Performance and Scientific Computing*, 1–9.

Oliphant, Travis E. 2006. *A Guide to Numpy*. Vol. 1. Trelgol Publishing USA.

Voigt, Bernhard Friedrich. 1831. “Der Handlungsreisende, Wie Er Sein Soll Und Was Er Zu Thun Hat, Um Aufträge Zu Erhalten Und Eines Glücklichen Erfolgs in Seinen Geschäften Gewiss Zu Sein.” *Commis-Voageur, Ilmenau*.

6 APPENDIX

6.1 TRAVELING SALESPERSON IMPLEMENTATION

The implementation of the TSP configuration is contained in module `tsp.py`

```
"""This module implements data structures for a traveling salesman problem."""
import scipy.spatial as sp
import functools
import numpy as np
import pandas as pd
from collections import namedtuple
class Node:
    """Represents one node in the network.
    Parameters:
    ----------
    node:   A named tuple with fields 'node', 'x' and 'y'
    """
    def __init__(self, node):
        self.node = int(node['node'])
        self.x = int(node['x'])
        self.y = int(node['y'])
    @property
    def coords(self):
        "Returns the coordinates for the node"
        return (self.x, self.y)
@functools.total_ordering
class Edge:
    """Represents an edge, the connecting line terminated by two nodes."""
    def __init__(self, node1, node2, tsp_config):
        self.tsp = tsp_config
        self.node1 = node1
        self.node2 = node2
        self.edge = self.tsp.distance_matrix[node1,node2]
    def __repr__(self):
        return "{}: nodes: ({}, {}). Length: {}".format(self.__class__.__name__,
                                                        self.node1,
                                                        self.node2,
                                                        self.edge)
    def __lt__(self, other):
        return self.edge < other.edge
    def __eq__(self, other):
        return self.edge == other.edge
    @property
    def length(self):
        return self.edge
class TravelingSalesPersonProblem:
    """Holds the nodes and distance matrix for a traveling salesman problem.
    It also provides methods for calculating the distance between nodes.
    Parameters:
    ----------
    nodes:              a pandas dataframe with columns [node, x, y]
    distance_matrix:    an nxn symmetric matrix
                        with the distances between nodes. If not passed, it is
                        constructed using an L2 distance metric by default.
    distance_metric:    The metric used to calculate distances in the network.
                        Has to accept the coordinates of two nodes as parameters
```

```
    """
    def __init__(self, nodes = None, distance_metric = sp.distance.euclidean):
        self._nodes = dict([[i, Node(nodes.iloc[i,:])]
                    for i in range(len(nodes))]) if nodes is not None else None
        if nodes is not None:
            self.build_distance_matrix()
        else:
            self._distance_matrix = None
        self._distance_metric = distance_metric
    @property
    def nodes(self):
        return self._nodes
    @nodes.setter
    def nodes(self, nodes):
        if isinstance(nodes, pd.DataFrame):
            self._nodes = dict([[i, Node(nodes.iloc[i,:])]
                                for i in range(len(nodes))])
        else:
            return False
    @property
    def distance_matrix(self):
        if self._distance_matrix is None:
            self.build_distance_matrix()
        return self._distance_matrix
    @distance_matrix.setter
    def distance_matrix(self, distance_matrix):
        assert isinstance(distance_matrix, np.ndarray)
        self._distance_matrix = distance_matrix
    def calc_distance(self, c1, c2):
        return self._distance_metric([c1.x,c1.y], [c2.x,c2.y])
    def build_distance_matrix(self):
        assert self.nodes is not None
        coords = [self.nodes[i].coords for i in range(len(self.nodes))]
        self._distance_matrix = sp.distance_matrix(coords,coords, p=2)
```

6.2 HEURISTICS

The implmentations for all heuristics are contained in module `tsp_heuristics.py`

```
import numpy as np
import pandas as pd
import collections
import operator
import random
from tsp.tsp import Edge
__all__ = [
    'ConstructionHeuristic',
    'ImprovementHeuristic',
    'BestInsertion',
    'BestBestInsertion',
    'ShortestEdge',
    'GreedyLocalSearch',
    'Swap', 'Translate', 'Invert', 'Mixed',
    'GreedyLocalSearchSwap',
    'GreedyLocalSearchTranslate',
    'GreedyLocalSearchInvert',
```

```
    'GreedyLocalSearchMixed',
    'SimulatedAnnealing',
    'SimulatedAnnealingMetropolis',
    'SimulatedAnnealingHeatBath'
]
class TspHeuristic:
    def __init__(self, tsp_config):
        self.tsp = tsp_config
        self.num_nodes = self.tsp.distance_matrix.shape[0]
        self.l = 0
        self.start = None
    @property
    def nodes(self):
        return self.tsp.nodes
    def calculate_cycle(self):
        raise NotImplementedError()
    def _cycle_finished(self):
        raise NotImplementedError()
    def loss(self, cycle=None):
        raise NotImplementedError()
    def get_cycle(self, tour=None):
        raise NotImplementedError()
    def get_cycle_for_plotting(self, nodes=None):
        """Builds the list of node coordinates in the cycle's sequence.
        If no nodes are passed in, takes the calculated cycle from the instance
        variable self.cycle."""
        if not nodes:
            nodes = self.get_cycle()
        cycle = [self.nodes[i].coords for i in nodes]
        cycle.append(self.nodes[nodes[0]].coords)
        return cycle
    def get_starting_node_for_plotting(self):
        raise NotImplementedError()
    def get_steps(self):
        return NotImplementedError()
# The three construction heuristics inherit from a common class
class ConstructionHeuristic(TspHeuristic):
    def __init__(self, tsp_config):
        super().__init__(tsp_config)
        self.cycle = np.zeros_like(tsp_config.distance_matrix, dtype=int)
        self.num_edges = self.num_nodes - 1
        # All nodes have 2 edges attached
        self.stopping_criterion = 2*(self.num_nodes-1)
    def loss(self, cycle=None):
        if self.l == 0:
            if not self._cycle_finished():
                self.calculate_cycle()
            cycle = self.get_cycle()
            for i in range(len(cycle)-1):
                self.l += self.tsp.distance_matrix[cycle[i]][cycle[i+1]]
            self.l += self.tsp.distance_matrix[cycle[-1]][cycle[0]]
        return self.l
    def get_cycle(self, start=False):
        """Builds the cycle in the correct order."""
        # init
        cycle = []
        included = set()
        if not start:
            start = self.start
```

```
        cycle.append(start)
        included.add(start)
        current = start
        finished = False
        while not finished:
            possible_nexts = [i for i, x in enumerate(
                self.cycle[current, :]) if x]
            next = set(possible_nexts).difference(included)
            if len(possible_nexts) == 1 and current != start:
                finished = True
            elif not next:
                finished = True
            else:
                # We're inside the cycle. Remove any nodes already selected
                # (basically, the one we came from)
                insert = next.pop()
                cycle.append(insert)
                included.add(insert)
                current = insert
        return cycle
    def get_cycle_for_plotting(self):
        """Builds the list of node coordinates in the cycle's sequence."""
        nodes = self.get_cycle()
        cycle = [self.nodes[i].coords for i in nodes]
        cycle.append(self.get_starting_node_for_plotting())
        return cycle
    def get_starting_node_for_plotting(self):
        """Finds the starting node of the cycle."""
        return self.nodes[self.start].coords
    def _select_new_node(self, size=1):
        """Randomly selects one or more of the nodes
        that have no edge attached so far."""
        available = [i for (i, v) in zip(
            range(self.cycle.shape[0]), self.cycle.sum(axis=1) == 0) if v]
        selected = random.sample(available, size)
        return selected[0] if size == 1 else selected
    def _insert_into_cycle(self, left, new, right=False):
        """Inserts a node into the cycle. If only left is given,
        the node is appended after left.
        If left and right are given, the new node
        goes in between left and right."""
        left = int(left)
        new = int(new)
        if not right:
            self.cycle[left][new] = 1
            self.cycle[new][left] = 1
        else:
            self.cycle[left][right] = 0  # break connection
            self.cycle[right][left] = 0
            self.cycle[left][new] = 1  # append new after left
            self.cycle[new][left] = 1
            self.cycle[new][right] = 1  # prepend new before right
            self.cycle[right][new] = 1
    def _check_loop_closed(self, node1, node2):
        """Checks if there is a closed connection between node 1 and node 2."""
        if self.cycle[node1, :].sum() == 0 or self.cycle[node2, :].sum() == 0:
            # one of the two is not connected, so it's impossible we end
            # up with a closed cycle
            return False
```

```
        else:
            if self.cycle[node1, :].sum() == 1:
                start = node1
                end = node2
            elif self.cycle[node2, :].sum() == 1:
                start = node2
                end = node1
            else:
                # Both nodes are already fully connected.
                # While this says nothing about a closed
                # loop, we can't insert anyways.
                return True
            cycle = self.get_cycle(start)
            if cycle[-1] == end:
                # We have a connection between start and end
                return True
            else:
                return False
    def _get_occupied_nodes_in_cycle(self):
        """Returns all nodes with two connections (fully connected)"""
        connected = np.where(self.cycle.sum(axis=1) == 2)
        connections = [(i, j) for i in connected[0]
                       for j in np.where(self.cycle[i, :] == True)[0] if i < j]
        return connections
    def _cycle_finished(self):
        # cycle is finished when all nodes except start and end have 2 neighbors
        if np.sum(self.cycle) == self.stopping_criterion:
            return True
        return False
    def _get_last_in_cycle(self):
        """Finds the last node in the cycle."""
        open_nodes = tuple([i for (i, v) in zip(
            range(self.num_nodes), self.cycle.sum(axis=1) == 1) if v])
        if not open_nodes:
            return False
        else:
            last = set(open_nodes) - set([self.start])
            if len(last) == 1:
                return last.pop()
            else:
                return False
class BestInsertion(ConstructionHeuristic):
    def __init__(self, tsp_config):
        super().__init__(tsp_config)
    def _init_algo(self):
        """ Initializes the best insertion algorithm.
        Selects three random nodes as the starting cycle.
        """
        self.l = 0
        self.cycle = np.zeros_like(self.tsp.distance_matrix)
        # draw 3 random nodes
        start_nodes = self._select_new_node(size=3)
        self._insert_into_cycle(start_nodes[0], start_nodes[1], start_nodes[2])
        self.start = start_nodes[0]
    def calculate_cycle(self):
        """Runs the best insertion algorithm."""
        self._init_algo()
        while not self._cycle_finished():
            try:
```

```
                next = self._select_new_node()
            except ValueError:
                print("No more nodes available! {}".format(
                    set(self.tsp.nodes.keys()) - set(self.cycle)))
                print("Number of None values in cycle: {}".format(
                    len([i for i in self.cycle if i is None])))
            left, right = self._calc_delta_loss(next)
            self._insert_into_cycle(left, next, right)
        return self.loss()
    def _calc_delta_loss(self, new_node):
        """Calculates the increase when the new point is inserted between any of the
        existing nodes.
        The returned list's indices can be used to select
        the left node for insertion, chosen where the added distance is minimal.
        """
        deltas = []
        # returns coordinate tuples of fully connected nodes
        c = self._get_occupied_nodes_in_cycle()
        start = self.start
        end = self._get_last_in_cycle()
        def d(n1, n2):
            return self.tsp.distance_matrix[n1, n2]
        if len(c) > 0:
            for i in range(len(c)):
                deltas.append(d(c[i][0], new_node) +
                              d(new_node, c[i][1]) - d(c[i][0], c[i][1]))
        else:  # We are at the start of the algorithm, there are 3 nodes.
            visited = self.get_cycle()
            second = visited[1]
            c = c + [(start, second), (second, end)]
            deltas.append(d(start, new_node) +
                          d(new_node, second) - d(start, second))
            deltas.append(d(second, new_node) +
                          d(new_node, end) - d(second, end))
        # Check between current end and start of cycle
        deltas.append(d(end, new_node) + d(new_node, start) - d(end, start))
        c = c + [(end, False)]
        insert_between = c[np.argmin(deltas)]
        return insert_between
class BestBestInsertion(ConstructionHeuristic):
    def __init__(self, tsp_config):
        super().__init__(tsp_config)
    def _init_algo(self):
        """ Initializes the best-best insertion algorithm.
        Selects one random node to start the cycle.
        """
        self.cycle = np.zeros_like(self.tsp.distance_matrix)
        self.l = 0
        self.start = self._select_new_node()
        next = [operator.itemgetter(0)(n) for n in sorted(enumerate(
            self.tsp.distance_matrix[self.start, :]), key=operator.itemgetter(1))][1]
        self._insert_into_cycle(self.start, next)
    def calculate_cycle(self, save_steps=False):
        """Runs the best insertion algorithm."""
        self._init_algo()
        while not self._cycle_finished():
            left, next, right = self._select_next()
            self._insert_into_cycle(left, next, right)
        return self.loss()
```

```
    def _select_next(self):
        """Finds the next node to insert. Determines the distance of all nodes
        not in the cycle so far to all the nodes already in the cycle.
        Returns:
        A tuple of the format(left, next, right)
        next:   Node number of next to insert
        left:   The node to the left"""
        visited = self.get_cycle()
        available = np.where(self.cycle.sum(axis=1) == 0)[
            0] # Select available nodes
        # Select the nodes already in the cycle
        candidates = np.where(self.cycle.sum(axis=1) > 0)[0]
        # By default, mask everything
        available_mask = np.ones_like(self.cycle, dtype=bool)
        # Unmask where we could possibly insert
        for row in available:
            for col in candidates:
                available_mask[row][col] = False
        # Build masked distance matrix
        masked_distance = np.ma.array(
            self.tsp.distance_matrix, mask=available_mask, shrink=False)
        # Get numbers of next and left nodes
        next_after = np.where(masked_distance == masked_distance.min())
        left = next_after[1][0]
        next = next_after[0][0]
        # figure out right node
        if not left == int(visited[-1]):
            l_index = visited.index(left)
            right = visited[l_index+1]
        else:
            right = False
        return (left, next, right)
class ShortestEdge(ConstructionHeuristic):
    def __init__(self, tsp_config):
        super().__init__(tsp_config)
        self.edges = sorted([Edge(i, j, self.tsp) for i in range(self.num_nodes)
                             for j in range(self.num_nodes) if not i == j])
        self.condition_cycle_premature = self.num_nodes*2
    def _init_algo(self):
        self.cycle = np.zeros_like(self.tsp.distance_matrix)
        self.l = 0
        self.start = self.edges[0].node1
        self._insert_into_cycle(
            self.edges[0].node1, self.edges[0].node2, right=False)
    def _check_constraints(self, new_edge):
        n1 = new_edge.node1
        n2 = new_edge.node2
        # Check node degrees:
        if self.cycle[n1, :].sum() == 2 or self.cycle[n2, :].sum() == 2:
            return False
        # Check prematurely closed loop.
        if self._check_loop_closed(n1, n2):
            return False
        return True
    def calculate_cycle(self):
        self._init_algo()
        edge_stack = self.edges.copy()
        edge_stack.remove(self.edges[0])  # The first edge already inserted
        while not self._cycle_finished():
```

```
            for e in edge_stack:
                if self._check_constraints(e):
                    self._insert_into_cycle(e.node1, e.node2, False)
                    edge_stack.remove(e)
        open_nodes = [i for i in range(
            self.num_edges) if self.cycle[i, :].sum() == 1]
        self.start = open_nodes[0]
        return self.loss()
# Improvement Heuristics
# Moves for the greedy local search algorithm
class Move():
    def __init__(self, heuristic):
        self.heuristic = heuristic
    @property
    def cycle(self):
        return self.heuristic.cycle
    def do(self):
        """Executes the move."""
        raise NotImplementedError()
    def _select_nodes_for_move(self, size=1, exclude=[]):
        """Selects one or more nodes in the cycle.
        Parameters:
        size:       How many nodes to select
        exclude:    List of nodes that are not available for selection"""
        available = set(range(self.heuristic.num_nodes)) - set(exclude)
        return list(np.random.choice(list(available), size, replace=False))
    def _d(self, n1, n2):
        node1 = self.cycle[n1]
        node2 = self.cycle[n2]
        return self.heuristic.tsp.distance_matrix[node1][node2]
    def _get_successor(self, node):
        n = self.cycle.index(node)
        if n <= self.heuristic.num_nodes-2:
            index = n+1
        else:
            index = 0
        return index
    def _get_predecessor(self, node):
        n = self.cycle.index(node)
        if n != 0:
            index = n-1
        else:
            index = self.heuristic.num_nodes-1
        return index
class Swap(Move):
    def __init__(self, heuristic):
        super().__init__(heuristic)
    def do(self):
        tau = self.cycle.copy()
        i, j = self._select_nodes_for_move(size=2)
        tau[i], tau[j] = tau[j], tau[i]
        return tau
class Translate(Move):
    def __init__(self, heuristic):
        super().__init__(heuristic)
    def do(self):
        tau = self.cycle.copy()
        i, = self._select_nodes_for_move(size=1)
        i_suc = self._get_successor(i)
```

```
        j, = self._select_nodes_for_move(size=1, exclude=[i + i_suc])
        insert_j_at = self._get_successor(i_suc)
        node_j = tau[j]
        tau.remove(node_j)
        tau.insert(insert_j_at, node_j)
        return tau
class Invert(Move):
    def __init__(self, heuristic):
        super().__init__(heuristic)
    def do(self):
        tau = self.cycle.copy()
        i, j, i_suc, j_suc = 0, 0, 0, 0
        while i_suc == j or j_suc == i:
            i, j = self._select_nodes_for_move(size=2)
            if i > j:
                j, i = i, j
            i_suc = self._get_successor(i)
            j_suc = self._get_successor(j)
        tau[i_suc:j_suc] = tau[i_suc:j_suc][::-1]
        return tau
class Mixed(Move):
    def __init__(self, heuristic):
        super().__init__(heuristic)
        self.moves = [
            Swap(heuristic),
            Translate(heuristic),
            Invert(heuristic)]
    def _choose_move(self):
        return np.random.randint(0, 3)
    def do(self):
        m = self._choose_move()
        self.current = m
        return self.moves[m].do()
class ImprovementHeuristic(TspHeuristic):
    def __init__(self, tsp_config, start_cycle_heuristic=False):
        super().__init__(tsp_config)
        self.start_cycle_heuristic = start_cycle_heuristic
        self.cycle = []
    def loss(self, cycle=None):
        l = 0
        if not cycle:
            cycle = self.cycle
        for i in range(len(cycle)-1):
            row = cycle[i] * self.num_nodes
            col = cycle[i+1]
            l += self.tsp.distance_matrix.item(row+col)
        row = cycle[-1] * self.num_nodes
        col = cycle[0]
        l += self.tsp.distance_matrix.item(row+col)
        return l
    def get_cycle(self):
        return self.cycle
    def get_starting_node_for_plotting(self):
        return self.nodes[self.cycle[0]].coords
    def _accept_cycle(self, dl):
        """Decides whether to accept the cycle or not
        based on the difference in loss."""
        if dl <= 0:
            return True
```

```
        return False
    def _init_algo(self, save_steps=False):
        # Create a random permutation of the nodes
        if self.start_cycle_heuristic:
            self.start_cycle_heuristic.calculate_cycle()
            self.cycle = self.start_cycle_heuristic.get_cycle()
        else:
            self.cycle = list(np.random.choice(
                list(self.nodes.keys()), size=self.num_nodes, replace=False))
        self.l = self.loss()
        self.finished = False
        if save_steps:
            self.steps = []
            self.steps.append(self.l)
    def _cycle_finished(self):
        return self.finished
class GreedyLocalSearch(ImprovementHeuristic):
    """Performs a greedy local search with the specified move.
    Parameters:
    move:   A child class of Move"""
    def __init__(self, tsp_config, move, start_cycle_heuristic=False):
        super().__init__(tsp_config)
        self.stopping_criterion = 10 * self.num_nodes**2
        self.start_cycle_heuristic = start_cycle_heuristic
        self.finished = False
        self.move = move(self)
    def calculate_cycle(self, save_steps=False):
        self._init_algo(save_steps)
        iter = 0
        while iter < self.stopping_criterion:
            tau = self.move.do()
            loss_t = self.loss(tau)
            if loss_t <= self.l:
                self.cycle = tau
                self.l = loss_t
            if save_steps:
                self.steps.append(np.amin([loss_t, self.l]))
            iter += 1
        self.finished = True
        return self.loss()
class GreedyLocalSearchSwap(GreedyLocalSearch):
    def __init__(self, tsp_config, start_cycle_heuristic=False):
        super().__init__(tsp_config, Swap, start_cycle_heuristic)
class GreedyLocalSearchTranslate(GreedyLocalSearch):
    def __init__(self, tsp_config, start_cycle_heuristic=False):
        super().__init__(tsp_config, Translate, start_cycle_heuristic)
class GreedyLocalSearchInvert(GreedyLocalSearch):
    def __init__(self, tsp_config, start_cycle_heuristic=False):
        super().__init__(tsp_config, Invert, start_cycle_heuristic)
class GreedyLocalSearchMixed(GreedyLocalSearch):
    def __init__(self, tsp_config, start_cycle_heuristic=False):
        super().__init__(tsp_config, Mixed, start_cycle_heuristic)
class SimulatedAnnealing(ImprovementHeuristic):
    def __init__(self, tsp_config, criterion='metropolis', move=Swap, cooling_factor=0.99, max_i
        assert(criterion in ['metropolis', 'heatbath'])
        super().__init__(tsp_config)
        self.t_max = 100
        self.t_min = 1
        self.cooling_factor = cooling_factor
```

```
        self.criterion = criterion
        self.max_it = max_it
        self.move = move(self)
    def _cool(self, t):
        return self.t_max*self.cooling_factor**t
    def _accept(self, dl, temp):
        if self.criterion == 'metropolis':
            if dl < 0:
                return True
            else:
                d = np.random.uniform()
                if d < np.exp([-dl/temp]):
                    return True
        elif self.criterion == 'heatbath':
            d = np.random.uniform()
            if d < 1/(1+np.exp([dl/temp])):
                return True
        return False
    def _init_algo(self, save_steps=False):
        # Create a random permutation of the nodes
        self.cycle = list(np.random.choice(
            list(self.nodes.keys()), size=self.num_nodes, replace=False))
        self.l = self.loss()
        self.finished = False
        if save_steps:
            self.steps = {}
    def _check_equilibrium(self, i):
        if i < self.max_it:
            return False
        return True
    def _create_candidate(self):
        return self.move.do()
    def calculate_cycle(self, save_steps=False):
        self._init_algo(save_steps)
        temp = self.t_max
        t = 0
        while temp > self.t_min:
            i = 0
            losses = []
            while not self._check_equilibrium(i):
                d_new = self._create_candidate()
                new_loss = self.loss(d_new)
                dl = new_loss - self.l
                if self._accept(dl, temp):
                    self.cycle = d_new
                    self.l = self.loss()
                if save_steps:
                    losses.append(new_loss)
                i += 1
            t += 1
            if save_steps:
                self.steps[temp] = {'min': np.amin(losses),
                                    'max': np.amax(losses),
                                    'mean': np.mean(losses)}
            temp = self._cool(t)
        return self.l
class SimulatedAnnealingMetropolis(SimulatedAnnealing):
    def __init__(self, tsp_config, move=Swap, cooling_factor=0.99, max_it=1000):
        super().__init__(tsp_config, criterion='metropolis',
```

```
                        move=move, cooling_factor=cooling_factor, max_it=max_it)
class SimulatedAnnealingHeatBath(SimulatedAnnealing):
    def __init__(self, tsp_config, move=Swap, cooling_factor=0.99, max_it=1000):
        super().__init__(tsp_config,  criterion='heatbath',
                        move=move, cooling_factor=cooling_factor, max_it=max_it)
```