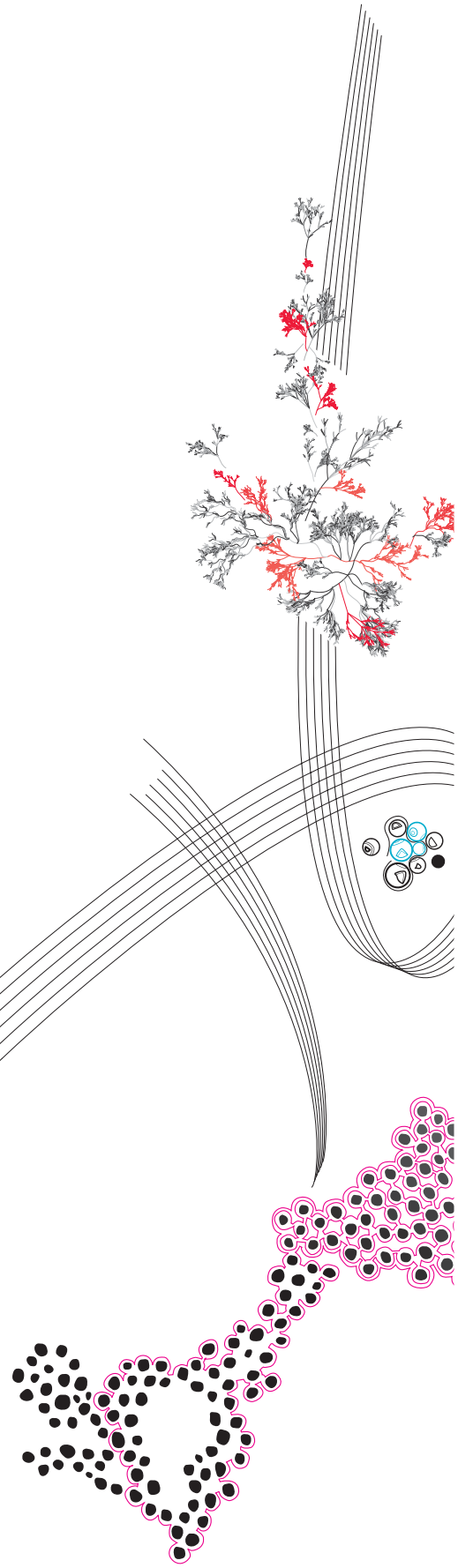


BSc Thesis Applied Mathematics

Experimental Analysis of the Lin-Kernighan heuristic

Renske Idzenga

Supervisors: Dr. B. Manthey, J. van Rhijn MSc

June, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

I would like to thank dr. Bodo Manthey and Jesse van Rhijn MSc for all of their feedback and guidance for this bachelor thesis.

Experimental Analysis of the Lin-Kernighan heuristic

Renske Idzenga

June, 2023

Abstract

The Traveling Salesman Problem is a combinatorial optimization problem with many applications. In the problem, the goal is to find a Hamiltonian tour of minimum cost in a graph. As finding the global optimum is difficult, many heuristics have been developed. These heuristics find tours of low, but not necessarily optimal, cost in reasonable time. The Lin-Kernighan heuristic is one of the best heuristics for the Traveling Salesman Problem. Theoretical analysis for this heuristic seems to be difficult, as we show in this paper. Therefore we investigate the heuristic experimentally. We find that the average number of iterations with respect to the input size is linear. The effect of four different settings of the heuristic are investigated. Two of the settings influence the slope of the linear relationship. The other two settings do not significantly influence the average number of iterations.

Keywords: Traveling Salesman Problem, Lin-Kernighan heuristic

1 Introduction

The traveling salesman problem (TSP) is a combinatorial optimization problem. In this problem, a traveling salesman needs to visit a number of cities. The salesman should visit each city exactly once, and finally return to its starting point. Of course, the salesman would like to do this as efficiently as possible. So, the goal is to find a route of minimum travel cost through all the cities that ends in the same place where it started. The traveling salesman problem is not only applicable to this specific situation, but has applications in several different areas, such as drilling holes in a circuit board, X-ray crystallography, and scheduling [1].

There are several variants of the traveling salesman problem. We consider here the symmetric TSP, where the cost to travel from one city to another is equal to traveling the opposite way. The TSP instances can be divided in two sets: metric and non-metric. Metric instances are instances where the edge weights abide by the triangle inequality. A subset of these instances are Euclidean instances. Euclidean instances consist of points scattered in a plane, and the travel distance between two points is the Euclidean distance between those points. In non-metric instances, the travel distances are chosen arbitrarily. With high probability, the triangle inequality does not hold on those instances. We consider here Euclidean instances and non-metric instances.

The traveling salesman problem is modeled as a graph. The cities are represented by vertices. The graph is complete, as we can travel between any two cities. The edges have weights that represent the cost of travel between the two vertices. Since the cost to travel one way is the same as traveling the other way, we work with simple graphs and undirected edges. The route that we need to find, is a tour T that visits each vertex exactly once, and

ends at the same place as it starts. In graph theory, this is known as a Hamiltonian tour. The cost of travel over an edge e is denoted as $|e|$. The cost C_T of a tour T is the sum of costs of all edges in the tour, i.e. $C_T = \sum_{e \in T} |e|$.

In the traveling salesman problem, the goal is to find the Hamiltonian tour with the minimal cost. The TSP problem is an NP-hard problem [2], so it is believed that no fast algorithm exists that finds the global optimum. Therefore we use heuristics.

A heuristic is an algorithm that tries to quickly find a good solution to a problem, without any guarantee that the solution is optimal. Also, it does not guarantee to be fast on every instance. Nevertheless, heuristics are widely used. In practice, many heuristics find close to optimal solutions in little time. And even if a heuristic does not always find a good solution, it can be run multiple times. Then it might find a better solution in one of those runs.

A local search heuristic starts with an initial solution to the problem. Each solution has a certain neighbourhood, that depends on the problem and the used heuristic. The local search heuristic then searches in the neighbourhood of the current solution for a better solution. If a better solution is found, we replace our initial solution by this improved one. Then the heuristic tries to improve this new solution. These steps are repeated until no improvement can be found. Then we are in a local optimum, with no guarantee of it being a global optimum. For TSP, a local search algorithm starts with some arbitrary Hamiltonian tour. It then removes some edges, and adds some different edges to maintain the Hamiltonian tour property. In most heuristics, the total cost of the new tour is then lower than the original tour. Some heuristics also allow tours with worse cost to be chosen. The local search algorithm would use this new tour as the current tour in the next iteration.

One widely used class of heuristics for the traveling salesman problem is k-opt. The k-opt heuristic is a local search heuristic. In each iteration, it can break and add at most k edges. So the neighbourhood of a solution are all tours that differ by at most k edges. The value of k is a trade-off between quality and efficiency. For lower values of k, k-opt is much faster, but the tours are of lower quality. In most applications of k-opt, 2-opt or 3-opt are used, as they produce very good tours in practice, especially if they are ran multiple times with different starting tours.

The Lin-Kernighan heuristic is an extension of the k-opt heuristic. Where any k-opt heuristic only exchanges at most k edges, the Lin-Kernighan heuristic can exchange in principle any number of edges. It improves a tour by removing one edge, adding a new edge, removing another edge, etc. This way it creates an alternating cycle of edges that we remove and add. To limit the number of options, several rules are in place that restrict and guide the search for a better tour in the Lin-Kernighan heuristic. Similar to k-opt, Lin-Kernighan starts with an arbitrary tour and iteratively optimizes that tour.

The Lin-Kernighan heuristic has been developed for the symmetric traveling salesman problem. However, it has been shown that the Lin-Kernighan heuristic can be extended to apply to other variants of TSP as well [3].

The Lin-Kernighan heuristic and variants of it are considered among the best TSP heuristics, and are used in state-of-the-art TSP solvers, such as Concorde [4]. Unfortunately, theoretical analysis of the algorithm has proven to be difficult and is an open problem [5]. Lin and Kernighan claimed that running times grow a bit worse that n^2 [6], but no theoretical results support this claim. To provide a basis for future theoretical research on this heuristic, we investigate the behaviour of the heuristic. In particular we see what impact several settings in the heuristic have on its efficiency.

For applications, it is important to have an algorithm with a low running time. Many algorithms are evaluated based on their asymptotic behaviour for running times. Running

times do however depend on the implementation and the machine. For theoretical evaluation, we look at the number of iterations in local search problems required to converge to a local optimum. The number of iterations is machine-independent and independent of the implementation. Therefore we do not consider the wall-clock time of the algorithm in this evaluation, but only the number of iterations, and the quality of the obtained tour.

In this paper, we first explain the Lin-Kernighan algorithm in depth. We list several configurations in the algorithm that we could change. Then we explain an efficient implementation for one step of the Lin-Kernighan heuristic.

After that, we do a theoretical analysis of the problem. We show here that it is not straightforward to find a useful theoretical bound. Therefore, we then analyse the algorithm experimentally. We consider the configurations listed before. We investigate how these configurations influence both the number of iterations and the tour quality.

2 Lin-Kernighan heuristic

2.1 Traveling Salesman Problem

The Lin-Kernighan heuristic has been developed for solving the traveling salesman problem. We represent an instance of the traveling salesman problem as a complete graph. All edges have weights that represent the cost to travel across that edge. These edge weights are determined in two different ways: in non-metric instances, we sample the edge weights from the uniform distribution between 0 and 1. In Euclidean instances, we place vertices randomly in the unit square, from 0 to 1. The edge weights are then the Euclidean distance between the vertices. As mentioned in the introduction, the weight of an edge e is denoted by jej. The algorithm starts with an initial Hamiltonian tour, called T. It then tries to find a new tour T^0 such that the cost of T^0 is lower than the cost of T.

2.2 The heuristic

As mentioned in the introduction, the Lin-Kernighan heuristic is a local search heuristic. Globally, it improves a tour by exchanging edges sequentially. It selects an edge to break, adds another edge, removes an edge, adds another edge etc. The algorithm ensures that we always keep a tour if we were to re-link at an intermediate step. How the algorithm decides when to stop, and which edges to choose, is explained below.

If we break and add exactly k edges, then this is called a k-change. In the Lin-Kernighan heuristic, this k is variable and can change every iteration.

We constantly break one edge and add another edge. Since each new selected edge is incident with the old selected edge, we can also see this as selecting vertices sequentially. We denote the selected vertices by $t = (t_1; t_2; :::)$. The i-th broken edge is denoted by x_i. The i-th added edge is denoted by y_i. The algorithm only changes the tour if the change decreases the cost of the tour. The algorithm changes the tour by re-linking. Re-linking happens only after breaking an edge, not after adding an edge. Let $X = fx_1; :::; x_ig$ and $Y = fy_1; :::; y_i \ _1; (t_{2i}; t_1)g$. Then the new tour after re-linking is $T^0 = (T n X) [Y$. The gain of a step is calculated by the sum of the weights of the edges we break, subtracting the sum of the weights of the edges we add. This is calculated by

$$G_i = \sum_{j=1}^{i} jx_jj \quad jy_jj:$$

The best gain that has been seen at some point in the current iteration is denoted by G^*.

The Lin-Kernighan heuristic can be broken up into three parts: a main loop that keeps updating the initial solution, an algorithm that breaks the next edge, and an algorithm that adds the next edge.

Algorithm 1 Main loop

```
1: for t_1 in vertices do
2:     for t_2 in neighbours of t_1 do
3:         broken ← [(t_1, t_2)]
4:         added ← []
5:         G* ← 0
6:         k ← 0
7:         if AddingEdge(broken, added, G*, k) is "improved" then
8:             restart MainLoop
9:         end if
10:    end for
11: end for
```

In Algorithm 1, the main loop of the algorithm is shown. It starts by selecting the first edge to break, (t_1, t_2). Since the algorithm always breaks an edge and then adds a new edge, the algorithm for adding a new edge (Algorithm 2) is called. If Algorithm 2 at some point improves the tour, the main loop restarts, and tries to improve the new tour.

The algorithm is allowed to try all options for t_1, and can try both options for t_2. If none of these options give an improvement, a local optimum has been found and the algorithm terminates.

In Algorithm 2, the pseudo-code for adding a new edge is shown. If we are at the step to add an edge, another edge in the tour has just been broken. This edge is called $x_i = (t_{2i-1}, t_{2i})$. Then vertex t_{2i} only has one incident edge left in the tour, while it should be incident with two edges for a valid tour. So we need to add an edge from t_{2i} to another vertex t_{2i+1}. This new vertex t_{2i+1} has to abide by a few properties:

- t_{2i+1} cannot be a vertex already chosen before by the algorithm in this iteration. Then it might happen that an edge that was just added needs to be broken, which is not allowed in the algorithm.
- t_{2i+1} cannot be a vertex that is a neighbour of t_{2i}. This would result in a double edge in our tour. In the next step, the algorithm would break the double edge, thus nothing would change. We do not allow this, as it can result in infinite loops. Also, as we work with simple graphs, we do not allow double edges.
- The partial gain G_i cannot be less than the best gain (G^*) we have seen thus far. This is called the gain criterion, and ensures that we do not search fruitless paths.

Since we want to get a high gain in each step, it is useful to consider the vertices in ascending distance. That way we first try the possibly better gain options.

If the partial gain turns out to be less than the best gain currently seen, we do not search for further edges. If we have found an improvement somewhere in our path, we re-link the tour with the improvements that led to this maximum gain. Then, we return "improved", so the other functions know the tour is updated and improved. If we have

Algorithm 2 AddingEdge(broken, added, G^*, k)

```
1:  i ← i + 1
2:  t_2i ← last vertex in t
3:  for t_2i+1 in vertices do
4:      if t_2i+1 was already chosen or t_2i+1 is in neighbours of t_2i then
5:          continue
6:      end if
7:      y_i ← (t_2i, t_2i+1)
8:      added.append(y_i)
9:      Calculate G_i
10:     if G_i < G* and G* > 0 then
11:         RelinkTour
12:         return "improved"
13:     else if G_i < G* and G* = 0 then
14:         added.remove(y_i)
15:         return "not improved"
16:     end if
17:
18:     if BreakingEdge(broken, added, G*, k) is "improved" then
19:         return "improved"
20:     else if BreakingEdge(broken, added, G*, k) is "no suitable edge" then
21:         added.remove(y_i)
22:         continue
23:     else if BreakingEdge(broken, added, G*, k) is "not improved" then
24:         added.remove(y_i)
25:         if backtracking is not allowed or max number of neighbours are considered then
26:             return "not improved"
27:         end if
28:     end if
29: end for
30: return "not improved"
```

not found an improvement, we will do backtracking. With backtracking, we allow more options to be considered in hopes of finding a tour with a lower total cost.

The backtracking in the Lin-Kernighan heuristic is limited. For the first vertex t_1, we are allowed to try every option. We are also allowed to try every edge to break, incident with t_1. For our first two edges we want to add, y_1 and y_2, we allow five neighbours. On the deeper levels, we only consider the first vertex t_{2i+1} that fulfills the three properties mentioned above. If this vertex does not give a gain, we do not consider more vertices on that level. We go down all the levels until we are allowed to consider multiple options, so levels 1 and 2. In the pseudo-code, this restriction on backtracking can be seen in line 25.

Algorithm 3 BreakingEdge(broken, added, G^*, k)

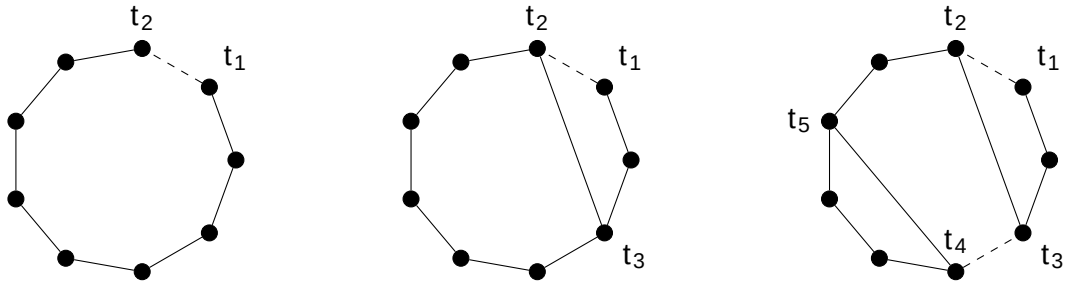
```
1:  t_{2i+1} ← last vertex in t
2:  x_i = (t_{2i+1}, t_{2i+2}) ← suitable edge to break
3:  if no suitable edge to break then
4:      return "no suitable edge"
5:  end if
6:  broken.append(x_i)
7:  if G_{i-1} + |x_i| - |(t_{2i+2}, t_1)| > G* then
8:      if greedy then RelinkTour
9:          return "improved"
10:     end if
11:     G* ← G_{i-1} + |x_i| - |(t_{2i+2}, t_1)|
12:     k ← i
13: end if
14:
15: if max depth is reached then
16:     if G* > 0 then RelinkTour
17:         return "improved"
18:     else if G* = 0 then
19:         broken.remove(x_i)
20:         return "not improved"
21:     end if
22: end if
23:
24: if AddingEdge (broken, added, G*, k) is "improved" then
25:     return "improved"
26: else if AddingEdge (broken, added, G*, k) is "not improved" then
27:     broken.remove(x_i)
28:     return "not improved"
29: end if
```

The last sub-algorithm of the Lin-Kernighan algorithm is shown in Algorithm 3. We arrive at this step if the new edge $y_i = (t_{2i}, t_{2i+1})$ has just been added. Then, the endpoint of this edge, t_{2i+1}, is now incident with three edges, while it should be incident with two edges for a Hamiltonian tour. So, one of its edges already in the tour T needs to be broken. In line 2, we mention that a suitable edge needs to be broken. With suitable edge, we mean that if we were to re-link the tour, it should not consist of multiple disjoint cycles, but it should be a Hamiltonian tour.

Only one of the two options for the edges will have this property. In Section 2.4, we discuss an efficient method to determine which of the two edges has to be broken.

Once it is determined which edge needs to be broken, we check what the gain would be if we would re-link the tour now, as seen in line 7. If this is a better gain than seen before, we update our G^*. We also record at which point we have found this best gain, by setting $k = i$.

If we have a valid edge we can break, we search for a new edge to break. We do this by calling the previously explained method AddingEdge.



(a) Breaking the first edge

(b) Adding the first edge

(c) Breaking, adding the second edges

Figure 1: Example of the Lin-Kernighan heuristic, for the first two iterations.

In Figure 1, we show the first two steps of the Lin-Kernighan heuristic. In Figure 1a, we break our first edge by selecting t_1 and t_2. We then add an edge from t_2 to another vertex t_3, as seen in Figure 1b. We would then break the edge such that re-linking would lead to a better tour. And then we again add another edge. These two steps are seen in Figure 1c. This would continue until we would re-link the tour.

2.3 Modifications of the algorithm

The Lin-Kernighan heuristic has some settings that restrict the search space. Lin and Kernighan mentioned that their choice for these settings relied on experimental analysis [6]. We are interested in seeing the effects of these settings on the heuristic. Also, we consider some more restrictions. There are four settings we are interested in.

- The level of backtracking is the level on which multiple options for neighbours, for adding an edge, are considered. In the Lin-Kernighan heuristic, this is only allowed on levels 1 and 2.
- The number of neighbours that are considered on levels where backtracking is allowed. Lower amounts of neighbours lead to a smaller neighbourhood of a solution. In the Lin-Kernighan heuristic, the number of neighbours is set to five neighbours. However, Lin and Kernighan said that often one of the first two options is chosen [6].
- The maximum depth, where we set a maximum of K, and we only allow a k-change if $k \leq K$. Restricting the maximum depth also decreases the size of the neighbourhood of a solution. A restricted depth might make the algorithm easier to analyse theoretically.
- A greedy version, where instead of searching deeper if there is potential for a higher gain change, we immediately re-link the tour once a gain has been found.

We look at the effects of these settings on the heuristic. We consider both the effect on the quality of the obtained tour, as well as the effect on the number of iterations.

2.4 Breaking a suitable edge

Algorithm 3, the breaking of an edge, mentions that a suitable edge needs to be broken. A suitable edge means that re-linking will result in a valid tour. If the incorrect edge is chosen, we end up with disjoint cycles instead of a Hamiltonian tour. A naive approach would be to try one of the edges, and check if that results in a Hamiltonian tour. However,

this subroutine has quadratic running time. Since the step of breaking an edge occurs often in the heuristic, an e cient implementation has signi cant impact on the e ciency of the algorithm. We present a linear-time algorithm here. This algorithm works by walking along the new tour, starting from t_{2i+1}. Depending on where we end with our walk (either at t_1 or t_{2i+1}), we know which edge we need to break to make a Hamiltonian tour. Since walking along the tour can be time consuming, we store extra information. With this information, we can walk along the tour in large steps and we do not need to traverse every vertex. To illustrate what information we need to store, we explain it using Figure 2.

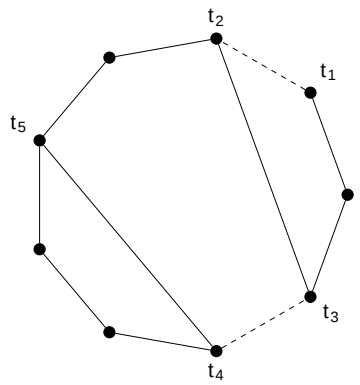


Figure 2: Intermediate step in Lin-Kernighan heuristic

We need to store information for each vertex $t_i \in t$. We store two pieces of information for each vertex: we store the vertex it is adjacent to by a newly added edge, and we store the vertex in t it is connected to if we would go along the tour. Taking vertex t_3 in Figure 2 as the example: t_3 is adjacent to t_2, by an added edge. And t_3 is connected to t_1, since we could walk along the tour and encounter t_1 as the rst vertex in t.

The adjacent vertex of t_i is denoted by $f(t_i)$. So $f(t_3) = t_2$. The other stored vertex is denoted by $g(t_i)$. So $g(t_3) = t_1$. The function $f(t_i)$ is updated once an edge is added. The function $g(t_i)$ is updated after an edge is broken. So in the situation of Figure 2, $f(t_5) = t_4$, but $g(t_5)$ is not de ned. Also, $g(t_2) = t_4$ and not t_5, as $g(t_i)$ is not yet updated. But $f(t_i)$ has been updated, so $f(t_4) = t_5$.

Algorithm 4 Determining which edge to break

```
1: v ← t_{2i+1} = last vertex in t
2: repeat
3:     v ← vertex clockwise from v
4: until v not in t
5: while True do
6:     if v = t_1 then
7:         return vertex counterclockwise from t_{2i+1}
8:     end if
9:     v ← f(v)
10:    if v = t_{2i+1} then
11:        return vertex clockwise from t_{2i+1}
12:    end if
13:    v ← g(v)
14: end while
```

In Algorithm 4, clockwise and counterclockwise vertices are mentioned. The initial tour is a list of vertices. We can write these vertices in a circle, and connect the edges between them. That represents the initial tour. Using this representation, we can then talk about the vertex clockwise from another vertex. It can also be seen as the next vertex in the list. A counterclockwise vertex is then the previous vertex in the list.

We prove that the algorithm provides the correct output in linear time.

Lemma 2.1 (Correctness of Algorithm 4). Given the index i, functions f and g, list t and Hamiltonian tour T as defined previously, Algorithm 4 provides a vertex t_{2i+2} in time linear in the number of vertices. This vertex t_{2i+2} ensures that re-linking after breaking edge (t_{2i+1}, t_{2i+2}) results in a Hamiltonian tour.

Proof. We order the vertices of the Hamiltonian tour T and place them in a circle. Algorithm 4 is only called if we already have broken at least one edge. So, starting at t_{2i+1} and going over the tour clockwise means that we hit a broken edge at some point. Since both endpoints of a broken edge are in the list t, we are ensured to hit some vertex $v \in t$ if we go clockwise along the old tour T. From v, we can walk along the new tour. The new tour alternates between paths of the old tour, and edges that are in added. By alternatingly moving to $f(v)$ and $g(v)$, we walk along this new tour. When we get to one of the stopping criteria (lines 7 and 11 in Algorithm 4), we are in one of two situations: if $v = t_{2i+1}$ (line 11), we have traversed a cycle. We do not allow cycles, so we need to break this cycle. Therefore we need to break the first edge traversed, so the edge clockwise from t_{2i+1}. Thus, we return the vertex clockwise from t_{2i+1}. The other situation is when $v = t_1$ (line 7). In this case we did not walk along the cycle, so we know the edge counterclockwise from t_{2i+1} must be in this cycle. So this edge needs to be broken. Thus, the vertex counter-clockwise form t_{2i+1} is returned. The algorithm finishes in linear time, as all necessary information is stored and no vertex is encountered twice. □

In line 3, we traverse the tour from vertex t_{2i+1} clockwise. This is a decision that has no influence on the outcome. The tour can also be traversed counter-clockwise. In that case, we would end up at the opposite stopping condition: if clockwise traversal would end in t_1, counter-clockwise traversal would end in t_{2i+1}, and vice versa. Thus, the decision to break which edge would also be reversed. Both versions of the algorithm would give the same result.

To illustrate Algorithm 4, we use the example in Figure 2. Here, a new edge has just been added between t_4 and t_5. Thus, we need to break one of the edges incident with t_5. If we want to determine which of the two edges we need to break, we will start at t_5. Continuing over our new tour in the clockwise direction, we encounter t_2. Then iteratively applying function f and then function g, we encounter vertices t_3 and t_1. Now that we are at t_1, we stop, since that was one of the criteria to stop the algorithm. Since we encountered t_1, we know we need to break the edge counter-clockwise from t_5.

To show that clockwise traversal is a decision that has no effect, we also show what would happen if we would go counter-clockwise. Then, we would first encounter t_4. We would then go to t_5 and stop, since that is the other stopping criteria. As the decisions are reversed if we go counter-clockwise, we need to break the edge counter-clockwise from t_5. In both cases, we would end up with the same result. And if we would re-link the vertex counter-clockwise from t_5 to t_1, we would see that the resulting configuration is a tour.

We then also update our stored connections: now t_2 is connected to t_3 and t_5, t_5 is connected to t_2 and t_4. Vertex t_6, the vertex counter-clockwise from t_5, is so far connected only to t_4. Now the algorithm can continue to its next iteration, by adding a new edge to t_6.

3 Theoretical analysis

For many algorithms, the worst-case behaviour is analysed. This gives a guarantee on the complexity of the algorithm. It is nice if the worst-case performance is low, as then we are sure the algorithm will perform well on all possible inputs. However, many algorithms have been shown to have a bad worst-case performance, while they perform really well in practice. An alternative analysis is therefore used: average-case analysis. This type of analysis is less dominated by worst-case instances. It does however give no guarantee on a single problem, but only on average. Its relevance does depend heavily on the probability distribution of the instances. If worst-case instances occur often, average-case analysis might not give a better bound.

In this analysis, we find an upper bound on the expected number of iterations. The Lin-Kernighan heuristic improves the tour in each iteration. We can find a lower bound on this improvement. We call this lower bound ε_{min}. If we then have an upper bound on the length of the starting tour, we can find an upper bound on the expected number of iterations. We consider the non-metric instances, where we sample edge weights from the uniform distribution between 0 and 1. The tour can then at most have length n, and at least has length 0. In every iteration, we decrease the tour cost by at least our lower bound ε_{min}. So we have at most $\frac{n}{\varepsilon_{min}}$ iterations. The upper bound on the tour length is not a random variable, so the only random variable is the lower bound on the gain. We analyse the probability that the minimal gain is small. We want to show that this probability is low, as that results in a low upper bound on the number of iterations.

An iteration in the Lin-Kernighan heuristic is the exchange of k edges in the current tour with k different edges, not in the tour. If the depth of the heuristic is not limited, at most n edges can be exchanged, as a tour consists of n edges. We analyse the situation if we would allow an exchange of at most k edges. If we then let $k = n$, we obtain an upper bound on the number of iterations for the Lin-Kernighan heuristic.

In a k-change, we have a list of k edges, $X_k = (x_1, x_2, \ldots, x_k)$, in our tour T that we will break. Also, we add a list of k edges, $Y_k = (y_1, y_2, \ldots, y_k)$, that are not in T.

We define the cost of an edge e by w_e. All edge weights are drawn from the uniform distribution, so $w_e \quad U(0,1)$. We define as the gain from a k-change. Thus,

$$(X_k, Y_k) = w_{x_1} + w_{x_2} + \ldots + w_{x_k} \quad w_{y_1} \quad w_{y_2} \quad \ldots \quad w_{y_k}$$

All edge weights are random variables. By the principle of deferred decisions, we can analyse this situation as if all edge weights are fixed, except w_{x_1}. We take

$$w = \sum_{i=2}^{k} w_{x_i} \quad \sum_{j=1}^{k} w_{y_j};$$

and we can rewrite as

$$(X_k, Y_k) = w_{x_1} + w :$$

Since all edge weights are fixed except w_{x_1}, w is a fixed number. We bound the probability that the gain of a step is small, i.e smaller than some,

$$\mathbb{P}((X_k, Y_k) \in (0,)) = \mathbb{P}(w_{x_1} \in (w, w +)) \quad ;$$

as the distribution of w_{x_1} is the uniform distribution on $(0,1)$, so its probability is bounded by 1.

In any step of the Lin-Kernighan heuristic, we have a positive gain. Thus for each choice of lists X_k, Y_k, it it is ensured that $(X_k, Y_k) > 0$. The minimal gain from any k-change is defined as follows:

$$\varepsilon_{min} = \min_{\substack{X_k, Y_k \\ (X_k, Y_k) > 0}} (X_k, Y_k)$$

So ε_{min} is a lower bound on the minimal gain that happens in every iteration of the heuristic. In order to bound the number of iterations, we want to show that the probability that ε_{min} is small is low. So we analyse the probability of this minimal gain to be small:

$$\mathbb{P}(\varepsilon_{min} \in) = \mathbb{P}(\exists (X_k, Y_k) : (X_k, Y_k) \in (0, \varepsilon]) = \mathbb{P}\left(\bigcup_{X_k, Y_k} (X_k, Y_k) \in (0, \varepsilon]\right) :$$

We apply a union bound over all possible choices of X_k and Y_k to obtain the following sum:

$$\mathbb{P}\left(\bigcup_{X_k, Y_k} (X_k, Y_k) \in (0, \varepsilon]\right) \sum_{X_k, Y_k} \mathbb{P}((X_k, Y_k) \in (0, \varepsilon]) \sum_{X_k, Y_k} = N ;$$

where N is the number of ways we can create sets X_k and Y_k that result in a positive gain. We can select the k edges in X_k in $\binom{m}{k}$, where m is the number of edges in the graph. We can then re-link these k vertices in $(2k-1)!! = (2k-1)(2k-3)\cdots(3)$ ways. Thus,

$$\mathbb{P}(\varepsilon_{min} \in) \quad N \quad \binom{m}{k} (2k-1)!! = O\left(\frac{m^k}{k^k}\right) k^k = O(m^k) \quad O(n^{2k});$$

where the last inequality comes from the fact that we are working in complete graphs where $m \quad n^2$.

Now that we have an upper bound on the probability of the gain being small, we use this to find an upper bound on the probability of a high number of iterations.

$$\mathbb{P}(T \quad t) \quad \mathbb{P}\left(\varepsilon_{min} \quad \frac{n}{t}\right) \quad O\left(n^{2k} \frac{n}{t}\right) = O\left(\frac{n^{2k+1}}{t}\right) ;$$

where T is the number of iterations. Then an upper bound of the expected number of iterations is

$$\mathbb{E}(T) = \sum_{t=1}^{\infty} \mathbb{P}(T \quad t) = \sum_{t=1}^{n!} \mathbb{P}(T \quad t) \int_{1}^{n!} O\left(\frac{n^{2k+1}}{t}\right) dt$$

$$= O\left(n^{2k+1} \log(t)\right)_{t=1}^{t=n!} \quad O\left(n^{2k+2} \log(n)\right) ;$$

where the last equality comes from the approximation of the factorial: $n! \quad n^n$.

Letting $k = n$, we obtain an upper bound on the expected number of iterations for the Lin-Kernighan heuristic. This upper bound is $\mathbb{E}(T) \quad O\left(n^{2n+2} \log(n)\right)$. This is worse than trying all possible tours, as that has a complexity of $n! \quad n^n$.

This straightforward probabilistic analysis of the average-case complexity does not give a useful bound. Many researchers have tried to get rid of the wasteful union bound, but no significant improvements have been found. Therefore, we analyse its behaviour experimentally. This might give ideas to tackle further theoretical analysis.

4 Implementation

We have implemented the Lin-Kernighan heuristic in Python 3.8. To store the graph, a NumPy [7] array is used. The entries in the array are the weights of the edges. Since we want to consider possible edges in order of distance, we also store a sorted version of the graph.

The tour is stored as a list of vertices, as well as a dictionary. In the dictionary, we store which vertex is connected to which other vertex. We use this to easily access the previous and next vertex in a tour.

5 Experimental results

We analyse the e ect of the settings in the Lin-Kernighan heuristic, on the average number of iterations and the average tour length. To analyse the e ect of the settings in the Lin-Kernighan heuristic, we see the average behaviour of the algorithm on randomly generated graphs. We consider two types of graphs:

- Graphs with edge weights drawn from the uniform distribution between 0 and 1. We call these graphs 'non-metric graphs'.
- Graphs with vertices placed randomly with both coordinates between 0 and 1, where the Euclidean distance represents the weight of an edge. We call these graphs 'Euclidean graphs'.

For these types of graphs, the growth of the optimal tour length is known. For non-metric graphs, the optimal tour length is at most 6 [8]. For Euclidean graphs, the optimal tour length tends to $c\sqrt{n}$ as n tends to 1 [9]. This c is an unknown constant. The approximation ratio is the ratio between the obtained tour length and the optimal tour length. We want to see what the e ect of the con gurations is on the approximation ratio. Therefore we normalize the tour lengths.

The algorithm has been run on a Dell PowerEdge R740, with 48 cores, 96 threads, max. 2.20 GHz and 256 GB of RAM. It has 2 Nvidia Tesla T4 GPU's [10].

For each setting, 1000 samples have been drawn using randomly generated graphss. The results were averaged in blocks of 25. With this size of blocks, the averages seem normally distributed. The standard error is calculated and plotted as error-bars. However, in most plots this error-bar is invisible as it is very small.

5.1 Number of neighbours

Figure 3: The average number of iterations for di erent values for neighbours in Euclidean graphs.

Figure 4: The average number of iterations for di erent values for neighbours in non-metric.

In Figure 3, we can see the average number of iterations for several numbers of neighbours. This gure is for Euclidean graphs. We can see that the average number of iterations is smaller for smaller number of neighbours. The number of iterations seems linear with respect to the size of the graph.

In Figure 4, we can see the same results for non-metric graphs. Here we see a similar trend, where lower values of neighbours result in slightly fewer iterations, although the di erences are smaller. Also, the number of iterations seems not linear.

Figure 5: The normalized tour length for di erent values for neighbours in Euclidean graphs.

Figure 6: The average tour length for di erent values for neighbours in non-metric.

The normalized tour length for di erent numbers of neighbours for Euclidean graphs can be found in Figure 5. It is not constant, as we would expect after normalization. However, our graphs are quite small, and the asymptotic behaviour is valid when m is large. We see that the tour is worse when the number of considered neighbours is 2 or 3. For higher values, there is no signi cant di erence.

In Figure 6, the actual tour lengths for non-metric graphs can be found. We can see the average tour length does not exceed 6. Similar to Euclidean graphs, the tour length is worse for two or three neighbours. For higher values, again no signi cant trends can be seen.

5.2 Backtracking depth

Figure 7: The average number of iterations for di erent levels of backtracking for Euclidean graphs.

Figure 8: The average number of iterations for di erent levels of backtracking for non-metric.

In both types of graphs, we can see that a higher level of backtracking leads to a slightly higher number of iterations. For Euclidean graphs, the number of iterations looks linear. For non-metric graphs, the number of iterations does not look linear.

Figure 9: The normalized tour length for di erent levels of backtracking in Euclidean graphs.

Figure 10: The average tour length for di erent levels of backtracking in non-metric.

For both types of graphs, we see that the tour length is worse for lower levels of backtracking. There is not a clear di erence for settings 5 and 6. Only allowing backtracking on level 2 clearly gives worse results on average.

5.3 Maximum depth

We have analysed two types of depth restriction. In the rst one, we restrict the depth to an absolute number of edges. In the second one, we restrict the depth to a percentage of the size of the graph. In case the depth calculated using the percentage is not an integer, we round it down.

Figure 11: The average number of iterations for di erent values of depth for Euclidean graphs.

Figure 12: The average number of iterations for di erent percentages of maximum depth for Euclidean graphs.

In Figure 11, the depth restricted by a number can be found for Euclidean graphs. The hyphen indicates no depth restriction. This is the normal setting for the Lin-Kernighan heuristic. A high number of maximum depth means that the algorithm can exchange more edges. We can see that the average number of iterations is linear for each setting of the depth. A lower maximum depth gives a higher inclination of the line.

In Figure 12, the average number of iterations with the depth restricted by a percentage can be found, for Euclidean graphs. Again, a higher percentage means more exchanges. The depth of 100% indicates no depth restriction. Similar to Figure 15, the average number of iterations is linear with respect to the size of the graph. However, the lines for the di erent settings have the same inclination. Only the line for a depth of 10% has a signi cant di erence. This line is shifted upwards compared to the other lines.

Figure 13: The average number of iterations for different values of depth for non-metric.

Figure 14: The average number of iterations for different percentages of maximum depth for non-metric.

Figure 13 shows the average number of iterations, for several depths. Similar behaviour is seen as for Euclidean graphs, in Figure 11. The number of iterations looks linear for each setting. A higher maximum depth results in fewer iterations.

In Figure 12, we again see similar behaviour as for Euclidean graphs. For non-metric however, we can clearly see that not only 10%, but also 20% depth restrictions are significantly different than the other lines. Again, the lines do not have a different inclination, but are shifted.

Figure 15: The normalized tour length for di erent values of maximum depth in Euclidean graphs.

Figure 16: The average tour length for di erent percentages of maximum depth in Euclidean graphs.

In Figure 11, we see a similar behaviour of the normalization as for the number of neighbours. It is not linear, but asymptotically it might be. Only a maximum depth of three edges is signi cantly di erent from the other settings. With three edges as maximum depth, the tour length is a lot higher than the other settings.

In Figure 16, no signi cant di erence can be found for any of the settings.

Figure 17: The average tour length for di erent values of maximum depth in non-metric.

Figure 18: The average tour length for di erent percentages of maximum depth in non-metric.

In Figure 17, we can see that the tour length is much higher for a maximum depth of 3 edges. For the other settings, a lower depth does result in a higher tour length, but there is not a big di erence.

Figure 18 shows that there is no signi cant trend in tour length. All percentages of maximum depth result in similar tour lengths. Most of the results are within each others error-bar.

5.4 Greedy variant

In Section 2, we proposed a greedy variant of the Lin-Kernighan algorithm. In this variant, we immediately re-link the tour if an improvement has been found.

Figure 19: The average number of iterations for a greedy and non-greedy Lin-Kernighan in Euclidean graphs.

Figure 20: The average number of iterations for a greedy and non-greedy Lin-Kernighan in non-metric.

For non-metric and Euclidean graphs, the number of iterations is larger for the greedy variant than for the non-greedy variant (as seen in Figures 19 and 20). For both variants, the average number of iterations is linear with respect to the size of the graph.

Figure 21: The normalized tour length for a greedy and non-greedy Lin-Kernighan in Euclidean graphs.

Figure 22: The average tour length for a greedy and non-greedy Lin-Kernighan in non-metric.

For Euclidean graphs, there is no signi cant di erence in the normalized tour lengths, for the greedy and non-greedy variant. This can be seen in Figure 21.

For non-metric graphs, the tour lengths do di er for the greedy and non-greedy variant. However, neither of the variants is always higher or always lower. Also, the error-bars are large. So there is no trend for the tour length between the greedy and non-greedy variant.

A possible explanation for this might be that the greedy and non-greedy variant both converge to local optima. The greedy variant does this in small steps, as it immediately re-links when it found an improvement. The non-greedy variant converges in fewer iterations, as it exchanges more edges each iteration.

6 Discussion and Conclusion

In this paper, we have done a theoretical and experimental analysis for the Lin-Kernighan heuristic.

From the theoretical analysis, we can see that a straightforward technique is not enough to obtain a useful upper bound. Our upper bound on the average number of iterations is exponential on the size of the input. This is worse than trying all possible options of Hamiltonian tours in a problem.

Therefore, we tried an experimental approach. We considered four di erent types of settings, on the average number of iterations, and on the (normalized) tour length. We analysed its e ect on two types of graphs: Euclidean graphs and non-metric graphs. For non-metric graphs, the edge weights were chosen from the uniform distribution between 0 and 1. The results for the two graphs did not di er a lot.

- The number of neighbours is the amount of options we can try, when backtracking is allowed. This setting had almost no e ect on the average number of iterations. The tour length was lower for higher numbers of neighbours.
- The level of backtracking also had little in uence on the average number of iterations. We could see that the number of iterations increased as the backtracking increased. The tour length did decrease for higher levels of backtracking.
- For the maximum depth, we had two types of limiters. Fixing the maximum depth at a value gave a linear relation, where lower depths corresponded with a higher inclination. The tour length was much worse for a depth of 3, but did not di er a lot for higher values of the depth. Taking the maximum depth as a percentage of the size also gave linear relations, but each setting had the same slope. Lower values for the depth shifted the lines upwards, to more iterations. There is no di erence in tour length between the settings, when taking a percentage as maximum depth.
- The last setting considered was a greedy variant for the heuristic. We saw that the number of iterations was much higher for the greedy variant. There was no clear di erence in tour length.

The goal of this paper was to provide an experimental basis for theoretical analysis on the heuristic. The Lin-Kernighan heuristic restricts the search space from naively trying all options. We investigated the settings to see if one of the settings had a high impact on the number of iterations. In all results, we have seen that the average number of iterations is linear or sub-linear. We have seen that the number of neighbours and level of backtracking had almost no e ect on the average number of iterations. The maximum depth and greedy variant both had an e ect, but they did not change the linear behaviour. Therefore, this experimental analysis does not provide a concrete setting that causes the linear behaviour.

As the Lin-Kernighan heuristic is quite complicated, we would like to simplify the heuristic if possible. This simpli ed heuristic might be easier to analyse. We can see that the numbers of neighbours and the level of backtracking both have no signi cant in uence on the number of iterations. Therefore, these two settings could be changed to simplify theoretical analysis. Also, it seems that the number of iterations is linear. This can be a bound that could be proven theoretically, although we do not know whether that is feasible.

For further experimental analysis, one could take into account the e ect of multiple settings. Due to time constraints, this analysis could not be done in this paper. But it could show that the interplay between settings explains the e ciency of Lin-Kernighan.

References

[1] Gerhard Reinelt. Related Problems and Applications . en. In: The Traveling Salesman. Vol. 840. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1994, pp. 31 41. isbn: 978-3-540-58334-9. doi: 10.1007/3-540-48661-5 .

[2] Michael R. Garey and David S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. en. 27. print. A series of books in the mathematical sciences. New York [u.a]: Freeman, 2009. isbn: 978-0-7167-1044-8 978-0-7167-1045-5.

[3] D. Karapetyan and G. Gutin. Lin Kernighan heuristic adaptations for the generalized traveling salesman problem . en. In: European Journal of Operational Research 208.3 (Feb. 2011), pp. 221 232. issn: 0377-2217. doi: 10.1016/j.ejor.2010.08.011 .

[4] Concorde Home url : https://www.math.uwaterloo.ca/tsp/concorde/index.html (visited on 04/13/2023).

[5] Bodo Manthey. Smoothed Analysis of Local Search . en. In: Beyond the Worst-Case Analysis of Algorithms. Ed. by Tim Roughgarden. 1st ed. Cambridge University Press, Dec. 2020, pp. 285 308. isbn: 978-1-108-63743-5 978-1-108-49431-1. doi: 10.1017/9781108637435.018

[6] S. Lin and B. W. Kernighan. An E ective Heuristic Algorithm for the Traveling-Salesman Problem . In: Operations Research 21.2 (Apr. 1973). Publisher: INFORMS, pp. 498 516. issn: 0030-364X. doi: 10.1287/opre.21.2.498 .

[7] Charles R. Harris et al. Array programming with NumPy . en. In: Nature 585.7825 (Sept. 2020). Number: 7825 Publisher: Nature Publishing Group, pp. 357 362. issn: 1476-4687. doi: 10.1038/s41586-020-2649-2 .

[8] Alan Frieze. On Random Symmetric Travelling Salesman Problems . In: Math. Oper. Res. 29 (Nov. 2004), pp. 878 890. doi: 10.1287/moor.1040.0105 .

[9] Jillian Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points . en. In: Mathematical Proceedings of the Cambridge Philosophical Society 55.4 (Oct. 1959). Publisher: Cambridge University Press, pp. 299 327. issn: 1469-8064, 0305-0041. doi: 10.1017/S0305004100034095

[10] UT-JupyterLab wiki [Jupyter wiki]. url : https://jupyter.wiki.utwente.nl/ (visited on 06/22/2023).

A The Python code

A.1 Tour class

```
"""
Class for a tour in a (complete) graph

Creates a random tour at initialisation from a given graph.
   Assumes the graph is complete.

From a tour you can access all vertices, all edges, the two
   neighbours of a vertex,
the next vertex, the next edge, and the cost of the tour given a
   cost matrix.

Can create a new tour given a current tour, edges that need to be
    broken and edges that need to be added.

A tour is implemented as a list of vertices
"""

import  numpy as np


class Tour:
    def __init__(self, graph, vertices=None):
        if vertices is None:
            vertices = list(np.random.default_rng().permutation(
               len(graph)))
        self._vertices = vertices
        self._next_vertex = {self._vertices[i]: self._vertices[(i
            + 1) % len(self._vertices)] for i in
                           range(len(self._vertices))}
        self._previous_vertex = {self._vertices[i]: self.
           _vertices[i = 1] for i in
                               range(len(self._vertices))}

    def __len__(self):
        return len (self.vertices)

    def __getitem__(self, item):
        return  self.vertices[item]

    @property
    def vertices(self):
        return  self._vertices

    @property
    def edges(self):
        return  [(self.vertices[i], self.vertices[(i + 1) %len(
```

```
        self)]) for i in range(len(self))]

    def previous(self, vertex):
        return self._previous_vertex[vertex]

    def next(self, vertex):
        return self._next_vertex[vertex]

    def neighbours(self, vertex):
        return self.previous(vertex), self.next(vertex)

    def update_tour(self, edges):
        if not edges:
            return False

        visited = [0]
        while len(visited) < len(self):
            if edges[visited[-1]] not in visited:
                visited.append(edges[visited[-1]])
            else:
                return False # If correct, you can never get
                    here since one of the two options will be fine

        self._vertices = visited
        self._next_vertex = {self._vertices[i]: self._vertices[(i
            + 1) % len(self._vertices)] for i in
                            range(len(self._vertices))}
        self._previous_vertex = {self._vertices[i]: self.
            _vertices[i - 1] for i in
                                range(len(self._vertices))}
        return True

    def make_tour_dictionary(self, broken_edges, added_edges):
        if len(broken_edges) != len(added_edges):
            return False

        new_tour = (set(self.edges) - set(broken_edges)) | {(e[1],
            e[0]) for e in broken_edges} | set(added_edges)
        if len(new_tour) != len(self):
            # print("Wrong tour")
            return False

        edges = {}
        current = 0
        while len(new_tour) > 0:
            for i, j in new_tour:
                if i == current:
                    edges[i] = j
                    current = j
```

```
                    break
                elif j == current:
                    edges[j] = i
                    current = i
                    break
            new_tour.remove((i, j))
        return edges

    def cost(self, graph):
        total_cost = 0
        for edge in self.edges:
            total_cost += graph[edge[0]][edge[1]]
        return total_cost

    def __str__(self):
        return str (self.vertices)
```

A.2 Lin-Kernighan heuristic

```
import numpy as np
import config
from Tour import Tour

broken_edges = []
added_edges = []
t = []

# Format will be {key:[value1, value2]}, where key is a vertex in
    t, value 1 is the endpoint of the added edge key
# is incident with (so also in t) and value 2 is the next vertex
   in t you would obtain when
# going over the current tour without the broken edges
connected_to = {}
sorted_graph = None

def gain(graph, broken, added):
    lost = sum([graph[e[0]][e[1]] for e in broken])
    gained = sum([graph[e[0]][e[1]] for e in added])
    return lost = gained

def determine_neighbours(vertex):
    return sorted_graph[vertex, :]

def relink_tour(tour, best_gain, best_gain_index):
    if best_gain > 0:
        first = t[0]
        last = broken_edges[best_gain_index = 1][1]
        edges = tour.make_tour_dictionary(broken_edges[:
           best_gain_index], added_edges[:best_gain_index = 1] +
           [(first, last)])
```

```
            if is_tour(tour, edges=edges):
                if tour.update_tour(edges):
                    return 'improved'
            raise Exception("Could not relink to valid tour")
        return 'zero gain'

    def break_next_edge(graph, tour, best_gain, best_gain_index):
        global broken_edges, t, connected_to
        first_vertex = t[0]
        last_vertex = t[-1]
        t2i_connection, t2i, last_vertex_connection =
           determine_edge_to_break(tour, last_vertex)
        if t2i in t:
            return 'does not allow breaking'

        broken_edges.append((last_vertex, t2i))
        t.append(t2i)

        connected_to[last_vertex][1] = last_vertex_connection
        connected_to[last_vertex_connection][1] = last_vertex
        connected_to[t2i] = [-1, t2i_connection]
        connected_to[t2i_connection][1] = t2i

        new_gain = gain(graph, broken_edges, added_edges + [(t2i,
           first_vertex)])
        if new_gain > best_gain:
            if config.greedy:
                return relink_tour(tour, new_gain, len(broken_edges))
            best_gain = new_gain
            best_gain_index = len(broken_edges)

        reached_max_depth = False
        # Maximum depth tracker
        if config.max_depth != 0 and len(broken_edges) >= config.
           max_depth:
            if relink_tour(tour, best_gain, best_gain_index) == '
               improved':
                return 'improved'
            reached_max_depth = True

        if not reached_max_depth:
            result = add_next_edge(graph, tour, best_gain,
               best_gain_index)
            if result == 'improved':
                return 'improved'

        broken_edges.pop(-1)  # Remove the last edge to backtrack
        t.pop(-1)  # Similarly for t
        connected_to[last_vertex_connection][1] = t2i_connection
```

```
    connected_to[t2i_connection][1] = last_vertex_connection
    connected_to.pop(t2i)
    connected_to[last_vertex][1] = =1
    return  'not improved'

def add_next_edge(graph, tour, best_gain, best_gain_index):
    global t, added_edges, connected_to
    last_vertex = t[ =1]
    considered = 0

    for vertex in determine_neighbours(last_vertex):
        if vertex in t:  # We are not allowed to choose this
           vertex again
            continue
        if vertex in tour.neighbours(last_vertex):
            continue

        new_gain = gain(graph, broken_edges, added_edges + [(
           last_vertex, vertex)])
        if new_gain <= best_gain:   # We can terminate the
           construction
            return  relink_tour(tour, best_gain, best_gain_index)

        added_edges.append((last_vertex, vertex))
        t.append(vertex)
        connected_to[last_vertex][0] = vertex
        connected_to[vertex] = [last_vertex,  =1]
        result = break_next_edge(graph, tour, best_gain,
           best_gain_index)

        if result == 'improved':
            return  'improved'

        added_edges.pop(=1)
        t.pop(=1)
        connected_to.pop(vertex)
        connected_to[last_vertex][0] = =1

        if result == 'not improved':
            if config.max_backtracking < len(added_edges):
                break
            else:
                considered += 1
        if considered >= config.max_neighbours:
            break
       # Our chosen edge to add did not permit a breaking of a
           next edge, so we will consider the next option

    if best_gain > 0:
```

```
            return relink_tour(tour, best_gain, best_gain_index)
        return 'zero gain'

    def determine_edge_to_break(tour, curr):
        closest_t = tour.next(curr)
        while closest_t not in t:
            closest_t = tour.next(closest_t)

        endpoint = closest_t
        while True:
            if endpoint == t[0]:
                return connected_to[closest_t][1], tour.previous(curr
                    ), closest_t
            endpoint = connected_to[endpoint][0]

            if endpoint == curr:
                return closest_t, tour.next(curr), connected_to[
                    closest_t][1]
            endpoint = connected_to[endpoint][1]


    def is_tour(tour, edges=None, broken=None, add=None):
        if edges is None:
            edges = tour.make_tour_dictionary(broken, add)
        if not edges:
            return False
        if len(edges) < len(tour):
            return False

        return True

    def improve(graph: np.array, tour: Tour):
        global broken_edges, added_edges, t, connected_to
        for t1 in tour.vertices:

            for t2 in tour.neighbours(t1):
                broken_edges, added_edges, t = [(t1, t2)], [], [t1,
                    t2]
                connected_to = {t1: [=1, t2], t2: [=1, t1]}
                if add_next_edge(graph, tour, 0, 0) == 'improved':
                    return True
        return False

    def sort_graph(graph):
        global sorted_graph
        sorted_graph = np.zeros((1, len(graph) - 1), dtype=int)
        for vertex in range(len(graph)):
            indices = np.argsort(graph[vertex, :])
            indices = indices[indices != vertex]
```

```
            sorted_graph = np.vstack((sorted_graph, indices))
        sorted_graph = sorted_graph[1:, :]

        pass

    def main(graph: np.array, tour=None):
        sort_graph(graph)
        if tour is None:
            tour = Tour(graph)
        else:
            tour = Tour(graph, tour)
        improved = True
        iterations = 0
        while improved:
            iterations += 1
            improved = improve(graph, tour)

        return tour.cost(graph), tour, iterations
```

A.3 Graph creation

```
import numpy as np
import tsplib95

def load_graph(file_name):
    """
    Loads a .tsp file using the tsplib95, and some functions to
       create a symmetric numpy array containing the weights.
    If the edge weights are given explicitly in the graph file,
       the get_weight function of tsplib95 does not work, so
    we use our own indexing to get those weights.

    :param file_name: filename of the graph, stored in the
       Examples map
    :return: symmetric numpy array with zeros on the diagonal,
       representing the weights as numbers in the array
    """
    graph = tsplib95.load(f"Examples\\{file_name}")
    n = graph.dimension
    matrix = np.zeros((n, n), dtype=int)
    if 'edge_weights' in graph.as_dict():
        weights = graph.as_dict()['edge_weights']
        for i in range(n):
            for j in range(n - i - 1):
                matrix[i][j + i + 1] = weights[i][j]
    else:
        for i in range(n):
            for j in range(i+1, n):
                matrix[i][j] = graph.get_weight(i+1, j+1)
    return matrix + matrix.T
```

```
def generate_random_graph(size, seed):
    """
    Method for generating a random symmetric matrix representing
       a complete graph with weights.
    :param seed: seed for the random number generator
    :param size: size of the graph
    :return: symmetric numpy array with numbers between 0 and 1,
       drawn uniformly
    """
    np.random.seed(seed)
    matrix = np.triu(np.random.random((size, size)), 1)
    return  matrix + matrix.T

def generate_euclidean_graph(size, seed):
    """
    Method for generating a weight matrix of a graph, where the
       vertices are placed randomly in the [0, 1]x[0, 1] plane.
    The weights are calculated using the two-norm.
    :param seed: seed for the random number generator
    :param size: size of the graph
    :return: symmetric numpy array with numbers representing the
       length between two vertices
    """
    np.random.seed(seed)
    points = np.random.random((size, 2))
    matrix = np.zeros((size, size))
    for i in range(size):
        for j in range(i+1, size):
            matrix[i, j] = np.linalg.norm(points[i] - points[j])
    return  points, matrix + matrix.T


if __name__ == "__main__":
    result = load_graph("own5.tsp")
    print(result)
```

A.4 Configuration file

```
"""
Here you can configure the TSP solver.
There are several settings that can be changed:
= The graph to run the algorithm on
= The algorithm
= The starting seed for random number generating
= The number of nearest neighbours to consider
= The maximum depth (percentage or number)
= The maximum level of backtracking
= Greedy variant or normal variant
= The file to store the results in
```

```
"""
type_of_graph = "random"  # file, random, euclidean
size_of_graph = [30, 50, 75, 100, 150, 200]  # , 250, 300]  #
    Number of vertices of a generated graph
filename = "own5.tsp"  # name of file if graph is a file type

algorithm = "LK"  # Options: LK (Lin-Kernighan), two_opt
iterations = 10  # Amount of times a new graph is generated (will
    be exactly the same if file graph)

start_seed = 0
seed_increments = 10

# Settings for the Lin-Kernighan Heuristic
max_neighbours = 5  # Number of nearest neighbours to consider in
    LK

depth_type = "number"  # Options: "percentage", "number". With
    percentage, percentage of size is taken as max depth
depth_percentage = 0.4
max_depth = 0  # Maximum search depth of LK (zero means infinite
    search depth)

max_backtracking = 2  # Limiter for full backtracking level
greedy = False  # Greedy flag (immediately relink when found a
    better tour, or search deeper first)

results_file = "results.csv"
```

A.5 Main file

```
import graph_creation
import two_opt
import lin_kernighan
import config
from tqdm import tqdm
import csv

seed = 0
def generate_graph(size):
    if config.type_of_graph == "file":
        graph = graph_creation.load_graph(config.filename)
    elif config.type_of_graph == "random":
        graph = graph_creation.generate_random_graph(size, seed)
    elif config.type_of_graph == "euclidean":
        _, graph = graph_creation.generate_euclidean_graph(size,
            seed)
    else:
        raise Exception("Not a valid type of graph to analyse")
    return graph
```

```
def execute_algorithm(vals, num_iters, graph, tour, size):
    if config.algorithm == "two_opt":
        val, optimal_tour, iters = two_opt.main(graph, tour)

    elif config.algorithm == "LK":
        val, optimal_tour, iters = lin_kernighan.main(graph, tour
            )
    else:
        raise Exception("Not an implemented algorithm type")
    vals[size].append(val)
    num_iters[size].append(iters)
    pass

def main_func():
    global seed
    seed = config.start_seed
    values = {x: [] for x in config.size_of_graph}
    number_of_iterations = {x: [] for x in config.size_of_graph}
    seeds = {x: 0 for x in config.size_of_graph}

    for i, size in enumerate(tqdm(config.size_of_graph)):
        if config.depth_type == "percentage":
            config.max_depth = int(size * config.depth_percentage
                )
        seeds[size] = seed
        for j in range(config.iterations):
            seed += config.seed_increments
            graph = generate_graph(size)
            tour = list(range(len(graph)))
            execute_algorithm(values, number_of_iterations, graph
                , tour, size)

    config.start_seed = seed + config.seed_increments  # So a
        next run can continue with the seeds

    for key in values.keys():
        row = [config.algorithm, config.type_of_graph, key,
            config.iterations, values[key], number_of_iterations[
            key],
                config.max_neighbours]
        if config.depth_type == "percentage":
            row.append(config.depth_percentage)
        else:
            row.append(config.max_depth)
        row += [config.max_backtracking, config.greedy, seeds[key
            ], config.seed_increments]
```

```
        with open(config.results_file, 'a', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(row)


if __name__ == '__main__':
    main_func()
```