Praktikum 1

Reverse Engineering Concorde: The Chained Lin-Kernighan Heuristic for TSP Problems

Verfasser
Tobias Prisching, B.Sc.

Wien, 2023

Studienkennzahl lt. Studienblatt:	066 921
Fachrichtung:	Masterstudium Informatik - Data Science
Betreuerin / Betreuer:	Ass.-Prof. Dr. Kathrin Hanauer, B.Sc. M.Sc.

Abstract

This report presents the findings of a project that investigates the implementation of the Chained Lin-Kernighan heuristic of the popular TSP solver package Concorde. This includes a discussion of the theory behind the heuristic, as well as notes and remarks of how individual steps of the algorithm are realized in practice. Additionally, a basic implementation of the CLK heuristic written in Rust is presented and compared to Concorde regarding runtime and solution quality on various TSP problem instances.

This quote reflects my personal experience with reading and working through the original source code of Concorde:

"One enters the first room of the mansion and it's dark. Completely dark. One stumbles around bumping into furniture but gradually you learn where each piece of furniture is. Finally, after six months or so, you find the light switch, you turn it on, and suddenly it's all illuminated. You can see exactly where you were. Then you move into the next room and spend another six months in the dark."

—Andrew J. Wiles [17]

Affidavit

I, Tobias Prisching, hereby declare that I have authored this work independently, without usage of any other than the cited and indicated sources, resources and aids, marking any direct and indirect quotations.

This work was not submitted to any other examination board or published, fully or partially, in this or any other substantially similar version before.

Signature

Contents

1 Introduction

The traveling salesperson problem (TSP) is one of the more well-known problems in the field of computational optimization with applications in e.g. vehicle routing, logistics and scheduling. The problem can be described as follows: Let $G = (V, E, w)$ be a simple, loop-less graph with n nodes $v_i \in V$ (the "cities" our traveling salesperson needs to visit each exactly once) with $1 \leq i \leq n$, $E = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ edges between these nodes and $w : V \times V \to \mathbb{R}^+$ denoting weights of these edges, which are interpreted as distances between two nodes. In the case of the *symmetric* TSP (which will be the only one considered for the rest of this work and therefore the property of symmetry will not be mentioned any further) the distances themselves are symmetric, i.e. going from node v_i to v_j has the same cost as going from v_j to v_i for any pair of nodes:

$$w(v_i, v_j) = w(v_j, v_i) \quad \forall v_i, v_j \in V \tag{1}$$

These weights could be described by a variety of distance metrics. Probably the most common one is the two-dimensional euclidean distance (which will be the only one further discussed in the upcoming chapters) due to its trivial nature, being intuitive to understand and its many applications.

The problem now is to find a Hamiltonian cycle - a set of edges that when traversed visits every node exactly once, in this context also called a *tour* - that has minimum cost, i.e. there exists no other tour where the sum of all edge weights is smaller - however, the optimal solution does not have to be unique. Furthermore, note that such a tour could also be represented using a permutation π of the vertices in V where the edges are given implicitly as $(\pi(i), \pi(i+\%n+1))$ with $1 \leq i \leq n$ (we will encounter a similar representation in an upcoming chapter). It can be shown that the TSP is NP-complete, so, assuming P≠NP, there exists no algorithm that can solve any given problem instance within polynomial-time.

That is why heuristics - methods which can be used to obtain a solution within a reasonable amount of time - are of major interest for the TSP problem. There exist a variety of heuristics following different approaches, from relatively simple ones like 2-opt[6], to more sophisticated ones like the one initially proposed by Lin and Kernighan in [11] and its derivatives (see e.g. [16] for a short introduction on these versions).

This brings us to the Chained Lin-Kernighan heuristic (CLK), discussed in [1] and implemented by the same authors as part of the software package *Concorde*[1][2]. Despite its age (with the current version from December 19th, 2003) it is still considered one of the more popular TSP solvers due to its performance and solution quality. However, the authors of Concorde have made a variety of design decisions during development, the vast majority of which without being documented. In general, the source code is hard to read due to the non-existence of comments, cryptic variable names, as well as an excessive usage of macros and pre-processor directives.

[1] According to an archived version of the corresponding website, "Concorde" is an acronym for "**C**ombinatorial **O**ptimization and **N**etworked **C**ombinatorial **O**ptimization **R**esearch and **D**evelopment **E**nvironment"[3]

This is where the motivation for this project stems from: Understanding how Concorde implements the CLK heuristic and how it achieves its runtime performance. Only a handful of sources give a rough overview on theory, leaving out implementation-relevant and practical details, some of which potentially having a large impact on the runtime [4].

Apart from the theory and source code discussed in upcoming pages this led to a small-scale re-implementation of the algorithm to further deepen the understanding. This new version (called *Blackbird*[2]) is written in Rust[14] and features the basic functionality for solving TSP problems similarly to how Concorde does. The details of this will be discussed after an introduction to heuristic and Concorde themselves, with later sections then describing a benchmark for comparing the two programs and finally discussing the results.

2 Chained Lin-Kernighan and Concorde

This section discusses the CLK heuristic in the context of Concorde. The idea is to discuss both of them together, as the heuristic on its own is not of practical use without an efficient implementation. This discussion is based on the already mentioned report [1] by Applegate et al. as this is the closest to useful documentation of the theory employed by Concorde. Furthermore, as Concorde provides a variety of features and options that need to be controlled both at compile- and runtime, only their default values and settings will be considered. Compilation options that are controlled via `#define`/`#undef`, `#ifdef`/`#ifndef`, etc. are left as-is so that this corresponds to a user downloading, compiling and using the package on their own without bothering to make further adjustments.

2.1 General Foundation

The foundation for the CLK heuristic is the original method as described in [11], which in turn builds upon the works of 2-opt [6] and 3-opt [10]. The basic idea of the 2-opt technique is to achieve an "intersection"-less tour. Given some tour T as

$$T = (i_1, i_2, ..., i_{x-1}, i_x, i_{x+1}, ..., i_{y-1}, i_y, i_{y+1}, ...i_{n-1}, i_n)$$

with $i_j \in \{1, ..., n\}$ and tour edges $\{(v_{i_a}, v_{i_b}) | v_{i_a}, v_{i_b} \in V \wedge (b - a = 1 \vee (a = n \wedge b = 1))\}$ (i.e. two consecutive nodes v_{i_j} and $v_{i_{j+1}}$ are connected, as well as v_{i_n} and v_{i_1}) we define the two edges $(v_{i_{x-1}}, v_{i_x})$ and $(v_{i_y}, v_{i_{y+1}})$ to have an *intersection* if they fulfill the inequality

$$w(v_{i_{x-1}}, v_{i_x}) + w(v_{i_y}, v_{i_{y+1}}) > w(v_{i_{x-1}}, v_{i_y}) + w(v_{i_x}, v_{i_{y+1}}) \tag{2}$$

which can be fixed by changing the tour to

$$T' = (i_1, i_2, ..., i_{x-1}, i_y, i_{y-1}, ..., i_{x+1}, i_x, i_{y+1}, ...i_{n-1}, i_n)$$

[2] As Concorde is also the name of a supersonic airplane, the idea was to name this project in a similar fashion, choosing the Lockheed SR-71 "Blackbird" - a plane that was also capable of supersonic flight[5] - as namesake: "**B**ringing to you a derivation of the **L**in-Kernighan Heuristic by **A**nalyizing **C**oncorde, **K**een on **B**uilding a somewhat **I**dentically performing, **R**ust-written **D**uplicate."

This operation will be called `flip`(x, y)[3], as the path from v_{i_x} to v_{i_y} was cut out, reversed in direction and reinserted. Furthermore, let us denote by `next`(v_{i_x}) and `prev`(v_{i_x}) the immediate successor and predecessor of a node v_{i_x} in T.

By applying the `flip` operation iteratively on all pairs of edges that fulfill inequality eq. (2) until no such pair of edges can be found, we essentially end up with the 2-opt algorithm. A tour T is said to be 2-opt if no such pairs exist anymore. This can be extended to 3-opt by also considering the possibility of taking a sequence $i_x, ..., i_y$, cutting it out and reinserting it at another location (either as-is or reversed), involving the change of 3 edges instead of 2. Further generalizing this approach yields the λ-opt method, where, for a given λ, the resulting tour is λ-opt, meaning that no exchange of a tuple of λ edges with another tuple results in a shorter tour. Using values $\lambda > 3$ does not produce significantly better results, especially when compared to the runtime, which is affected by the binomial coefficient $\binom{n}{\lambda}$.

2.2 Heuristic specific Methods

This is where the description of the CLK heuristic as described in [1], according to the authors, starts to differ from the default LK method from [11]. This is mostly related to reformulating the steps of the heuristic to be realized later on in executable code, keeping the main ideas of the LK heuristic. This section tries to use the naming convention of functions as they appear in the Concorde source code, making notes on (important) deviations from the literature to emphasize on the connections between theory and practice.

Taking the high level view of CLK in pseudo code 1, it becomes apparent that it basically performs multiple calls to (a modified version of) LK, which in turn processes each node from a given queue Q as a starting point of a λ-opt move. Each such move consists of multiple steps, implemented by `step` and `weird_second_step` (in the literature, such as [1], this gets referred to as `alternate_step`).

In Concorde, the functions described in pseudo code 1 perform additional side tasks, like for example memory (de)allocation, keeping track of how much the tour has been improved, when to stop the while loop of CLK, etc.

2.2.1 The `step` Operation

With the given tour T and the two nodes $first$[4] and $last$ (initially the successor of $first$ in T) the `step` operation performs part of a λ-opt move by trying to replace edges with potentially better ones and then performing recursive calls to itself to lengthen the optimization move (i.e. increasing λ). This replacement is performed by a `flip` as follows: With given $first$ and $last$, let us pick a probe-node (and its successor `next`$(probe)$) and perform the flip `flip`$(last, probe)$,

[3]For simplicity, we will assume for this operation to be well defined that the tour has a direction. So, while traversing the tour one way or the other does not affect its cost, it does change how it is written down using the respective indices.

[4]In the literature referred to as *base*, without an explicit name for *last*.

Pseudo Code 1 High level view of CLK

```
1: function CLK(T)                                  ▷ Expects an initial & valid tour T
2:     Q ← Queue of nodes V
3:     T′ ← LK(T, Q)
4:     while should continue with search do
5:         T′ ← perform a kick on T′                                  ▷ See section 2.4
6:         T′ ← LK(T′, Q)
7:     end while
8: end function
9:
10: function LK(T, Q)                    ▷ A valid tour T, reference/pointer to Q
11:     while base = pop(Q) do
12:         δ ← improve_tour(T, Q, base)      ▷ Potentially modifies T in place
13:     end while
14: end function
15:
16: function improve_tour(T, base)
17:     first ← base, last ← next(base)
18:         ▷ Copies, remain unchanged after calls to step/weird_second_step
19:
20:     step(T, Q, first, last, 1, 0, 0)
21:     if call to step was not successful then
22:         weird_second_step(T, Q, first, last)
23:     end if
24:
25:     if one of the two calls was successful then
26:         add first and last to Q
27:     end if
28: end function
```

which now puts *probe* as the new successor to $first$ and gets used as new *last* in recursive calls. Plugging into inequality eq. (2), this yields

$$\begin{aligned} w(first, last) + w(probe, \texttt{next}(probe)) > \\ w(first, probe) + w(last, \texttt{next}(probe)) \end{aligned} \tag{3}$$

as a necessary condition for the flip to be an improvement, which Lin and Kernighan further simplify implicitly to the criterion

$$w(first, last) - w(last, \texttt{next}(probe)) > 0$$

i.e. there is a positive gain/a reduction of the tour length by improving only a single edge in T. This criterion is sufficient as shown by [11] due to the fact that we only need partial sums of gains to be positive for the total gains to be positive - for further details on the related proof please refer to the cited paper.

Concorde achieves this by first generating a list of possible candidate edges $(probe, \texttt{next}(probe))$, depending on $first$ and *last*, which then gives us the nodes for the flip operation and marking edges as deleted/added. These markings are required as we do not want to re-introduce an edge that has previously been

marked as removed. To create the aforementioned list, we start by taking *last* and searching nodes that are nearby (as we want to add an edge to this node, which should ideally have a low distance weight) - in case of Concorde, this is done via a list of close neighbors called the *sparse edge set*[5] that gets created using a kd-tree and will be used for other tasks as well. In essence, this is a list of edges for each node in V that have a low cost and could potentially be of interest for the tour. The idea behind this is that the total number of edges that need to be considered for a tour can be reduced substantially when compared to the total number of possible edges $\frac{n \cdot (n-1)}{2}$.

Pseudo Code 2 An overview of the `look_ahead`, based on Concorde

```
1:  function look_ahead(T, first, last, level, g_i)
2:      ordering ← []
3:      for node next_probe that is candidate for next(probe) for last do
4:          if w(last, next_probe) > g_i then
5:              break
6:          ▷ Assumes candidates are traversed with increasing distance to last
7:          end if
8:
9:          if
10:             (last, next_probe) ∉ deleted_edges and
11:             next_probe ≠ first and
12:             next_probe ≠ next(last)
13:         then
14:             probe ← prev(next_probe)
15:             edge ← (probe, next_probe)
16:             diff ← w(last, next_probe) − w(edge)        ▷ differs to eq. (4)
17:             add (edge, diff) to ordering
18:         end if
19:     end for
20:
21:     sort ordering by diff-values in descending order
22:                                  ▷ Concorde's alternative sorting approach
23:     return first breadth(level)-many elements of ordering
24: end function
```

The custom kd-tree structure features a nearest-neighbor-like query which divides the space of points into 4 quadrants and distributes the search among them. The resulting neighbors will be the `next`(*probe*)-nodes, so we can obtain *probe*-nodes simply via `prev`(`next`(*probe*)). Note that these `next`(*probe*)-nodes need to fulfill the following requirements:

- `next`(*probe*) $\neq$ *first*: This would mess up the tour, as we would mark the edge $(first, last)$ both as added and as deleted
- `next`(*probe*) $\neq$ `next`(*last*): Again, this would also mess up the tour as the edge (*last*, `next`(*last*)) would get marked both as added and deleted

[5]The corresponding variable in the Concorde source is called `goodlist`

- $(last, \texttt{next}(probe)) \notin deleted\_edges$: We do not want to re-introduce a deleted edge

The resulting edges $(probe, \texttt{next}(probe))$ are then ordered so that they maximize the value

$$w(probe, \texttt{next}(probe)) - w(last, \texttt{next}(probe)) \tag{4}$$

Pseudo Code 3 An overview of the `step` operation as used by Concorde

```
1: function step(T, Q, first, last, level, g_i, G*)
2:     if level is deep enough so that breadth(level) = 1 then
3:         return step_noback(T, Q, first, last, level+1, g_i, G*)
4:                                   ▷ Performs a step without backtracking
5:     end if
6:
7:     hit ← 0              ▷ keeps track of improvements at this recursion level
8:     candidate_edges ← look_ahead(T, first, last, level, g_i)
9:                                                ▷ literature: lk_ordering
10:    for edge ∈ candidate_edges do
11:        probe ← edge.end
12:        next_of_probe ← edge.start
13:        g_{i+1} ← g_i − edge.diff          ▷ .diff is computed by look_ahead
14:        G ← g_{i+1} − w(first, probe)
15:        if G > G* then
16:            G* ← G ▷ G* keeps track of best improvement of T, so replace it
17:            hit += 1
18:        end if
19:
20:        flip(last, probe)
21:
22:        if level is below maximum depth threshold then     ▷ Concorde: 25
23:            mark (last, next_of_probe) as added
24:            mark (probe, next_of_probe) as deleted
25:            result ← step(T, Q, first, probe, level+1)
26:            hit += result.hit
27:            G* ← result.G*
28:            unmark (last, next_of_probe) and (probe, next_of_probe)
29:        end if
30:
31:        if hit is 0 then                ▷ i.e. no improvement has been made
32:            unflip(last, probe)
33:        else
34:            add probe to Q
35:            add next_of_probe to Q return (1, G*)
36:        end if
37:    end for
38:    return (0, G*)                          ▷ no improvement has been made
39: end function
```

which is part of inequality eq. (3) and tells us how much the exchange of these two edges improves the tour, which is why we want to maximize the difference. This order is implemented with a slight variation, namely by instead minimizing

$$w(last, \texttt{next}(probe)) - w(probe, \texttt{next}(probe)) \tag{5}$$

and sorting edges $(probe, \texttt{next}(probe))$ in descending order according to the value of the term in eq. (5). The source code does not provide a reasoning for this alternative sorting approach.
To reduce the number of possible combinations the recursive `step` function keeps track of the depth via a level variable. This gets used in combination with `look_ahead` to provide backtracking by limiting the amount of edges to try during a specific call to `step`. The Concorde authors discuss different settings for the breadth at different levels in [1]. In practice, they use by default

$$\text{breadth(level)} = \begin{cases} 4 & \text{if level } = 1 \\ 3 & \text{if } 2 \leq \text{level} \leq 3 \\ 2 & \text{if level } = 4 \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

However, the case for level > 4 is special in another way, which we will take a look at next.

2.2.2 The `step_noback` Operation

Additionally, looking at pseudo code 3, we can see in line 3 a call to `step_noback`, which covers the special case of the step operation without performing backtracking once a certain recursion depth is reached. To compensate the miss of backtracking, the function also makes use of so-called Mak-Morton moves[6], introduced in [12]. Their basic idea is, instead of performing flips that change $last$, flip operations are done so that $first$ changes instead, leaving $last$ (i.e. the initial successor of $first$) untouched. Other than some special considerations in preparing the ordering (Concorde has its own method for this case: `look_ahead_noback`), the step itself is basically the same and will thus not be further discussed at this point.

2.2.3 The `weird_second_step` Operation

In case the recursive calls to `step` did not yield an improvement for a given pair $first$ and $last$ and cascades all the way back to `improve_tour`, the algorithm considers a different type of step to start the recursion, implemented by `weird_second_step`. The idea behind this is that, while the previously discussed default `step` operation performs a flip of the form $\texttt{flip}(\texttt{next}(base), \texttt{prev}(probe))$, removing $(\texttt{prev}(probe), probe)$, the alternative is to remove $(probe, \texttt{next}(probe))$, which is achieved by multiple flips. These flips employ additional nodes on the tour, with control flow depending on in which section of the tour they are currently positioned. To implement this, Concorde utilizes multiple functions for creating different orderings for these different nodes. However, as the actual

[6] This is the default behavior of Concorde. The source code shows that it would in fact support this type of moves at every recursion depth. However, this is deactivated by default via compiler flags.

move itself is relatively complex and its details beyond the scope of this project, it is sufficient to know of its existence and to think of it as simply a slightly different type of `step` operation that can be called at the start of a move's recursion.

2.3 Tour Initialization

The pseudo code 1 expects a valid tour T as initial starting point due to CLK belonging to the class of improvement heuristics (i.e. taking a given, valid solution and improving upon it). This is a challenge on its own, as there exist a variety of methods to obtain a first cycle, ranging from utilizing a (pseudo) random permutation of the nodes $v_i \in V$ up to 2-approximation methods that give us an initial tour that is at most double the optimal length.

In the case of Concorde the default initialization is done via the Quick-Borůvka heuristic, which is based on an algorithm for obtaining a minimum spanning tree by Borůvka[15][7] This method does not explicitly construct a minimum spanning tree but utilizes the previously introduced sparse edge set (see section 2.2.1) which at some point in the beginning got created using a kd-tree of all nodes V.

In essence, the heuristic iterates over the nodes of the TSP instance as long as there are not enough edges and finds one for a given node by querying its nearest neighbors as possible partners to form a new edge. This involves keeping track of the degree of the individual nodes and the tails of already existing subpaths that will later on be part of the initial tour. Pseudo code 4 gives a more detailed overview of the Concorde version of the heuristic.

Pseudo Code 4 Quick-Borůvka heuristic as implemented in Concorde

```
 1: function QuickBoruvka(nodes, kd-tree)
 2:     degree ← map that keeps track of the degree of nodes
 3:     tails ← map that stores the tail of a consecutive sequence of nodes
 4:     edges ← ∅
 5:     sorting_indices ← ordering of node IDs that sorts them in the x dim.
 6:
 7:     while len(edges)+1 < len(nodes) do
 8:         for i ∈ sorting_indices do
 9:             node ← nodes[i]
10:
11:             if degree of node = 2 then
12:                 continue
13:             end if                          ▷ Can't connect node to additional edge
14:
15:             if tails contains a tail for node then
16:                 tail ← tail from tails for node
17:                 deactivate tail in kd-tree      ▷ Can't be result of a NN query
18:                 edge_partner ← nearest neighbor of node in kd-tree
19:                 activate tail in kd-tree
```

[7]This is not the original paper but a translation from Czech to English.

```
20:             else
21:                 edge_partner ← nearest neighbor of node in kd-tree
22:             end if                              ▷ Find edge partner for new edge
23:
24:             if degree of node > 0 then
25:                 deactivate node in kd-tree
26:             end if
27:             if degree of edge_partner > 0 then
28:                 deactivate edge_partner in kd-tree
29:             end if
30:     ▷ deg. =1, after iteration =2 → can't take add. edge in future iteration
31:
32:             increase degree of node and edge_partner
33:             insert edge (node, edge_partner) into edges
34:
35:             if tails does not contain a tail for node then
36:                 if tails does not contain a tail for edge_partner then
37:                                                  ▷ both don't have a tail yet
38:                     insert node as tail for edge_partner
39:                     insert edge_partner as tail for node
40:                 else
41:                                 ▷ edge_partner does have a tail but node doesn't
42:                     tail ← tail from tails for edge_partner
43:                     insert tail as tail for node
44:                     insert node as tail for tail
45:                 end if
46:             else if tails does not contain a tail for edge_partner then
47:                                 ▷ node does have a tail but edge_partner doesn't
48:                 tail ← tail from tails for node
49:                 insert edge_partner as tail for tail
50:                 insert tail as tail for edge_partner
51:             else
52:                                                      ▷ both have a tail
53:                 tail_1 ← tail from tails for node
54:                 tail_2 ← tail from tails for edge_partner
55:                 insert edge_partner as tail for tail_1
56:                 insert node as tail for tail_2
57:             end if          ▷ Handling of the tails for the newly added edge
58:         end for
59:     end while
60:
61:     Find two nodes with degree 1 and connect them to complete tour
62: end function
```

2.4 Kicking the Tour

At the heart of the CLK heuristic is the way how multiple runs of the LK heuristic are chained together without restarting from scratch. To do so, a *kick* - a slight perturbation of the existing tour - is required between two succeeding runs of the LK heuristic. This idea was initially proposed in [13] with special 4-

opt moves called double-bridges. As with tour initialization there are multiple approaches on how to achieve such a perturbation, with Concorde providing four different approaches that are all based around double-bridge moves, with the default working as follows:

- First, a pair of two nodes $t_1, t_2 \in V$ is obtained with edge $(t_1, t_2) \in T$ being part of the current tour using a random-based search approach (i.e. randomly selecting an edge). This pair has the property that their edge is relatively long when compared to what the distance to the closest nearest neighbor from the sparse edge set of t_1 is. The idea behind this is that t_1 is not in the tour where it should belong due to the length of the edge needed to get there, which is why we want to find an edge that maximizes the aforementioned property.

- With such a pair t_1, t_2 we need to find additional nodes $t_3, ..., t_8$ to form the double bridges. In the end, the edges

  $$(t_1, t_2), (t_3, t_4), (t_5, t_6), (t_7, t_8)$$

  will get replaced by

  $$(t_1, t_6), (t_2, t_5), (t_3, t_8), (t_4, t_7)$$

- To obtain $t_3, ..., t_8$, the algorithm used by Concorde tries to obtain first (after t_1, t_2) the pair t_3, t_4, then t_5, t_6 and then t_7, t_8 by taking the latter of the previous pair and randomly choosing one of the nearest neighbors. This is done for multiple iterations, with the result of the previous iteration as starting point, resulting at some point in the odd-indexed node. The even-indexed node is then simply the successor of its corresponding odd-indexed partner, e.g. $t_4 = \texttt{next}(t_3)$.

- The resulting sequence $t_1, ..., t_8$ then needs to fulfill multiple checks, i.e. the nodes need to be distinct and the kick must not result in the tour being split into two sub-cycles.

- A valid sequence $t_1, ..., t_8$ is then used to perform multiple flips to obtain the aforementioned replacement of edges. Furthermore, these nodes and their neighbors up to a certain depth are added to the queue Q as a starting point for LK.

With the kick completed, the next call to the LK heuristic can be performed.

2.5 Managing the Current Tour

During this entire section we assumed that our tour representation data structure is capable of performing certain operations like `flip` and `next` without considering their runtime. In practice however, choosing the right data structure (referred to as *flipper*) for the task of storing and handling the tour is critical. A basic investigation by the Concorde authors in [1] shows that with a sub-optimal data structure the majority of the runtime might be spent on just the `flip` operations. The authors discuss multiple approaches, starting with a

trivial array-based structure, working towards what in the end they describe as a two-layered list: This divides the tour into multiple "buckets", each with a single parent node that together with the other parents forms a second layer that can be used for shortcuts during the individual operations. Within a bucket multiple nodes are stored representing a node of the TSP instance, with two pointers to the next and previous node in the current tour.

Looking at the source code it becomes apparent that setting up this data structure and updating it quickly is relatively complex due to different edge cases that require non-trivial pointer arithmetic and operations. Due to this complexity and the scope of this project, a different approach was taken for the Blackbird implementation, as shall now be discussed in the next section.

3 Blackbird

To further deepen the understanding of Concorde, a partial re-implementation of the CLK heuristic called *Blackbird* was written in Rust[8] that focuses on behaving similarly[9]. In this section the design decisions made during the development of Blackbird will be discussed, pointing out differences in the approaches taken when compared to Concorde.

3.1 General Remarks

First, it needs to be noted that Blackbird only implements a small portion of the functionality that Concorde provides. Apart from the default settings that this report focuses on, a variety of other options are provided by Concorde at different points and stages of the heuristic, like for example tour initialization and types of kicks. This starts with the different metrics used by TSP instances which require different versions of the functions discussed in the previous section. Blackbird focuses, similarly to this report, on the default Concorde configuration, while trying to remain easily expandable for potential future developments.

Next, the main deviation from the Concorde code base regarding actual program logic should be discussed. This can be found in the functions for creating the `look_ahead` orderings of the different move types. As discussed in section 2.2, the functions `step`, `step_noback` and `weird_second_step` require orderings of possible candidate edges that are considered for performing a flip. These edges are ordered in Concorde according to the value of term eq. (5) in descending order. However, during development it was discovered that reversing this sorting for Blackbird consistently yielded better results regarding solution quality, which is why this change was kept at the cost of not being 100% identical to Concorde. It is still unclear why Concorde deviates in this regard from the literature.

[8] This language was chosen due to its memory safety features while providing a reasonable good performance, as well as its growing popularity in the field of algorithmic research

[9] and not identically, as this is (at least trivially) not possible when using a different programming language as is the case for Blackbird

3.2 Data Structures

Some of the main challenges during development were the different data structures used throughout the heuristic, primarily the kd-tree and the flipper. Memory management was less of a concern as the size of these data structures only become of relevance for problem instances far beyond of what will be tested in section 4. Primarily, the focus was put on to find a good balance between performance, complexity and similarity to Concorde.

3.2.1 kd-tree

Initially, a ready-to-use crate for a kd-tree was used. However, due to the lack of the ability to "deactivate" a node (temporarily) for the nearest neighbor query during the Quick-Borůvka tour initialization (see pseudo code 4, lines 17 & 19), as well as the special 4-quadrant query used by Concorde, this approach was discarded and replaced with a custom kd-tree. This in turn also affects the creation of the sparse edge set which is needed throughout the entire heuristic. In the end, the resulting set is nearly identical to the one produced by Concorde. Slight variations still exist due to the kd-tree itself being not unique and depending on the state of the random generator.

This is a general problem with the random generator in Rust: During development, it was observed that the compiled binary does not behave deterministically. Setting the seed of the random generator does not guarantee that it behaves exactly the same in two separate runs. This is a known issue and was not further investigated as it only poses a minor inconvenience.

3.2.2 Flipper

As previously discussed, the two layered list that is being used by Concorde as flipper for storing and modifying the tour is a complex topic on its own, which is why compromises have been made in this regard. At first, a simple and inefficient array-based flipper was implemented to have a basic structure that can be used for testing and comparing more complex flippers. In the end, the final flipper implementation follows a double linked list approach, where each node has a reference to its immediate predecessor and successor, as well as a "reversed" bit to determine which reference to use for `next`/`prev`. The idea behind this boolean is to simply negate its value instead of changing references in case the node lies on the path that is getting flipped.

This data structure design is a compromise in the sense that operations need to take place on a more granular level, as each node needs to be visited/modified individually instead of being able to handle all nodes of an entire bucket at once. This is a potential bottleneck of Blackbird, as only the `next` and `prev` operations can be served within constant time (via references that are based around the identifiers of the TSP instance nodes).

4 Benchmark & Comparison

With the Blackbird implementation being sufficiently advanced to solve TSP instances in a similar manner as the default configuration of Concorde, it is now time to set up and perform a comparison of the two. This section discusses the general setting of how this benchmarking is performed.

4.1 TSP Instances and Environment

Choosing the "right" instances to compare two given TSP solvers is not trivial, as the structure of a problem[10] might be (unintentionally) advantageous for one of the implementations. Furthermore, the results of this preliminary study should be reproducible and comparable to other experiments. This is why for the upcoming experiments a variety of pre-existing and popular TSP instances was selected from TSPLIB [19], a collection of problems that have been studied extensively. The majority of these instances use the two dimensional euclidean distance, with only three larger instances (pla7397, pla33810 and pla85900) being exceptions that have been modified to use this metric as well. To make distinguishing the individual instances in the upcoming plots easier, each instance size is unique, i.e. there exist no two instances that have the same number of nodes.

4.2 Setup of the Experiments

The experiments were carried out on a system featuring an Intel Core i7-7700K with 16GB of DDR4 memory, running macOS Ventura 13.4, automated using a shell script that alternately calls both Blackbird and Concorde five times for each TSP instance and writes the results to a CSV file. These values include (apart from some metadata) the runtimes for building the kd-tree, the initial tour, the CLK heuristic and total runtime, as well as the initial and final tour lengths.

5 Running Experiments & Analysis of Results

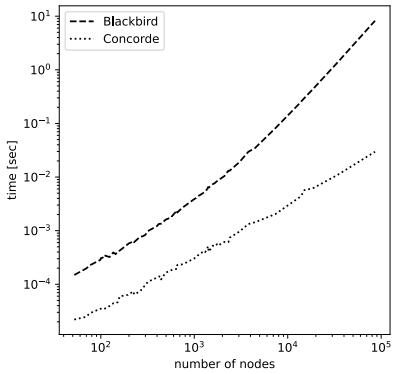
This sections discusses the results of the previously introduced experiments for comparing Blackbird with Concorde. The results were analyzed using a Jupyter Notebook [9] running Python [20] in combination with the NumPy [7] and pandas [18] libraries for data manipulation and Matplotlib [8] for plotting.

5.1 Runtime

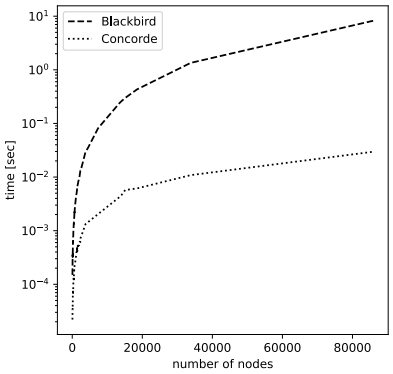
To start with the runtime analysis, fig. 1 depicts the plots of the arithmetic means of the runtime results for the tour initialization and the total heuristic runtime. Comparing the curves of Blackbird and Concorde reveals that

- Both solvers have a runtime that can be expressed using some polynomial within the problem instance size. This is obvious as the curves in the lin-log-scaled plots do not show a linear behavior but instead taper of with increasing node count.

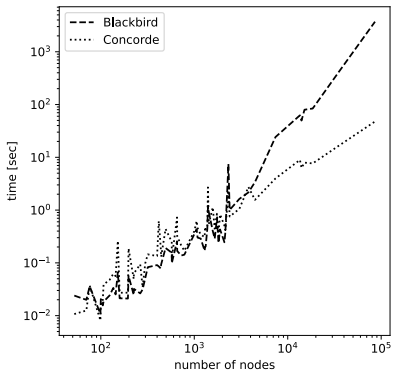
[10] Or problems that are artificially generated by some algorithm



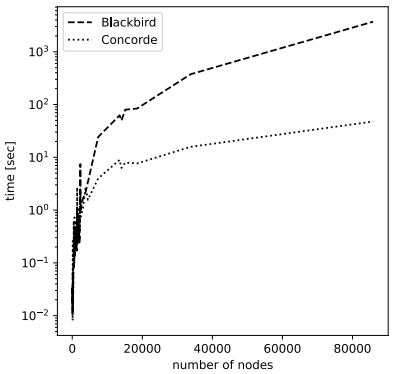
(a) Tour Initialization Runtime (log-log-scale)



(b) Tour Initialization Runtime (lin-log-scale)



(c) Total Runtime (log-log-scale)



(d) Total Runtime (lin-log-scale)

Figure 1: Comparing the initial and total runtimes

- Both solvers are affected similarly by the inherent structure of the TSP instances. This becomes apparent due to the runtime spikes of certain problem sizes (that correspond to specific instances) in the log-log-scaled plot of the total runtime.
- Blackbird has some significant performance drawbacks within the order of one to three magnitudes, especially during the creation of the initial tour. For smaller instances with just a few hundred nodes however the total runtime seems to be comparable without any significant differences.

To explain the differences in initialization and total runtime it helps to take a closer look at the inner workings of Blackbird. For the tour initialization this extra cost likely stems from the way the set of edges gets converted into a permutation of node identifiers, as this function was implemented independently from the corresponding Concorde method. The differences in total runtime are of greater interest as they seem to increase with more nodes and are practically

negligible for instances with less than 10^4 nodes. This likely occurs due to the less sophisticated flipper data structure employed by Blackbird which does not feature any of the optimizations that were applied to Concorde. However, these are only theories and would require a more in-depth study on the effect of these differences in implementation, which exceed the scope of this project as this requires the implementation of a two-layered list-based flipper for Blackbird.

5.2 Solution Quality

Quantifying and comparing the quality of a solution is not as straightforward as the runtime. Each of the instances has one or more optimal solutions of some finite length. It is unlikely that either of the two solvers achieves such an optimal solution for every of the tested instances[11]. For this reason, comparing to the optimal solutions does not make much sense, which is why the Blackbird results are again compared to Concorde. Note that this does not tell us something about the objective quality of the solution but only relates it to results obtained via another solver.

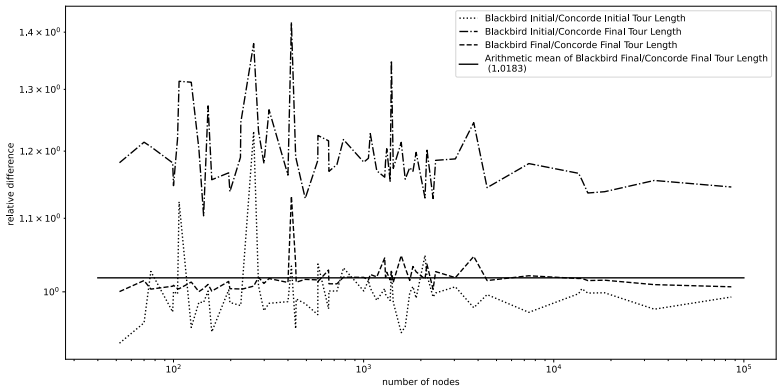


Figure 2: Comparing the initial and final tour lengths

Studying the curves of the plot in fig. 2 reveals that

- The initial tour creation of Blackbird seems to yield slightly better tours in some cases, as the quotient of their lengths is less than 1 (fig. 2).
- Problem instances that are "problematic" for Blackbird already during the initialization seem to stay that way as peaks of the dashed and dash-dotted curve show some correlation. This leads to the conclusion that creating a good initial tour is essential for performing further improvements.
- Tours that are produced by Blackbird are mostly by just a few percent longer. More importantly, this deviation from Concorde does not increase with problem size.

[11] This is why Concorde as a whole does not just stop at applying the CLK heuristic but instead uses it to achieve an intermediate result that gets further optimized using branch-and-bounds-based techniques [2].

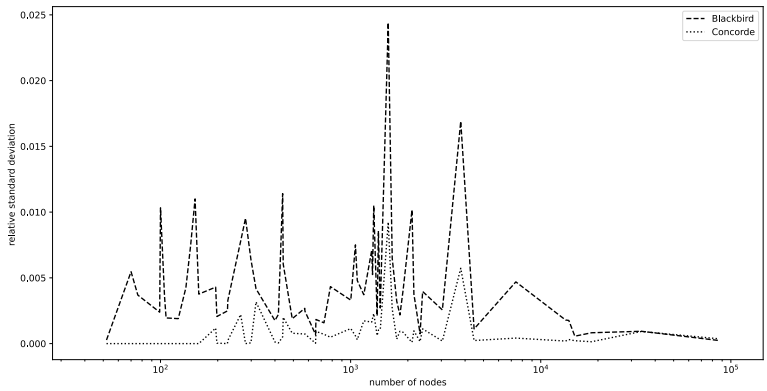


Figure 3: Relative standard deviation of tour lengths

Additionally, fig. 3 shows the relative standard deviation of the final tours over the five runs. This tells us that the resulting tours do not differ much from one another when compared to their overall length. This is slightly worse in the case of Blackbird but still negligible.

Overall, these results show that Blackbird does behave similar to Concorde and responds to certain TSP instances and their inherent structure in a similar way. This still leaves however the massive difference in runtime performance and the, while small, non-negligible increase in the final tour length.

6 Conclusion

In this report an in-depth investigation of the inner working of the Chained Lin-Kernighan implementation of the Concorde software package has been conducted. The theory behind this heuristic, as well as practical implementation details in the context of Concorde have been discussed, giving an overview of the different software components, data structures and functions. Additionally, a partial re-implementation of the CLK heuristic that mimics Concorde has been introduced. Experiments were conducted to compare the two solvers, revealing, despite best attempts, significant differences in runtime performance and final tour length. The main takeaway from this project is that, while understanding a complex algorithm and replicating an existing implementation of it is manageable, achieving the same performance is even less trivial and requires an even more profound understanding than just of the theory and combining it with source code. In the end, this leaves room for improvement and expansion of the Blackbird implementation for future developments, which concludes the work and research of this project.

List of Figures

References

[1] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Finding tours in the TSP. Technical report, 1999.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Concorde TSP Solver. URL: https://www.math.uwaterloo.ca/tsp/concorde/index.html (visited on 2023-06-25).

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Concorde: combinatorial optimization and networked combinatorial optimization research and development environment. A code for solving traveling salesman problems. Aug. 18, 2000. URL: https://web.archive.org/web/20000818181649/http://www.keck.caam.rice.edu/concorde.html (visited on 2023-07-01).

[4] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem. A Computational Study*. Princeton University Press, Princeton, 2007. ISBN: 978-1400841103. DOI: doi:10.1515/9781400841103. URL: https://doi.org/10.1515/9781400841103 (visited on 2023-06-25).

[5] P. F. Crickmore. *Lockheed SR-71 Blackbird*. Bloomsbury Publishing, 2015. ISBN: 978-0850456530.

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

[7] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[8] J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[9] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[10] S. Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, 1965. DOI: 10.1002/j.1538-7305.1965.tb04146.x.

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

[12] K.-T. Mak and A. J. Morton. A modified lin-kernighan traveling-salesman heuristic. *Operations Research Letters*, 13(3):127–132, 1993. ISSN: 0167-6377. DOI: https://doi.org/10.1016/0167-6377(93)90001-W. URL: https://www.sciencedirect.com/science/article/pii/016763779390001W.

[13] O. Martin, S. W. Otto, and E. W. Felten. Large-step markov chains for the tsp incorporating local search heuristics. *Operations Research Letters*, 11(4):219–224, 1992.

[14] N. D. Matsakis and F. S. Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34 of number 3, pages 103–104. ACM, 2014.

[15] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001. ISSN: 0012-365X. DOI: https://doi.org/10.1016/S0012-365X(00)00224-7. URL: https://www.sciencedirect.com/science/article/pii/S0012365X00002247. Czech and Slovak 2.

[16] C. Rego, D. Gamboa, F. Glover, and C. Osterman. Traveling salesman problem heuristics: leading methods, implementations and latest advances. *European Journal of Operational Research*, 211(3):427–441, 2011. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2010.09.010. URL: https://www.sciencedirect.com/science/article/pii/S0377221710006065.

[17] S. Singh. *Fermat's last theorem: The story of a riddle that confounded the world's greatest minds for 358 years*. 1997. ISBN: 978-1857025217.

[18] The pandas development team. pandas-dev/pandas: Pandas. URL: https://github.com/pandas-dev/pandas.

[19] Universität Heidelberg. TSPLIB: library of traveling salesman problems. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/ (visited on 2023-06-30).

[20] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN: 978-1441412690.