University of Würzburg
Institute of Computer Science
Research Report Series

An $O(n \log n)$ Heuristic for the Euclidean Traveling Salesman Problem

Evgeny Yanenko, Eckart Schuhmacher[1]
and
Ulrich Spörlein, Kurt Tutschku[2]

Report No. 358 April 25, 2005

[1] atg test systems GmbH & Co KG.
Zum Schlag 3, 97877 Wertheim, Germany.
{eianenko,eschuhmacher}@atg-test-systems.de

[2] University of Würzburg, Department of Distributed Systems.
Am Hubland, 97074 Würzburg, Germany.
{spoerlein,tutschku}@informatik.uni-wuerzburg.de

An $O(n \log n)$ Heuristic for the Euclidean Traveling Salesman Problem

Evgeny Yanenko, Eckart Schuhmacher
atg test systems GmbH & Co KG.
Zum Schlag 3, 97877 Wertheim, Germany.
{eianenko,eschuhmacher}@
atg-test-systems.de

Ulrich Spörlein, Kurt Tutschku
University of Würzburg, Department of Distributed Systems.
Am Hubland, 97074 Würzburg, Germany.
{spoerlein,tutschku}@informatik.
uni-wuerzburg.de

Abstract

In this paper we present a fast approach for solving large scale Traveling Salesman Problems (TSPs). The heuristic is based on Delaunay Triangulation and its runtime is therefore bounded by $O(n \log n)$. The algorithm starts by construction the convex hull and successively replaces one edge with two new edges of the triangulation, thus inserting a new city. The decision which edge to remove is based on edge ranks. Finally the tour is subject to a node insertion improvement heuristic.

By extensive case studies it will be shown that only highly optimized 2/3-Opt heuristics are superior to this approach in both running time and tour lengths for very large TSP instances.

1 Introduction and Motivation

Atg test systems GmbH & Co. KG is the global market leader for automatic test equipment (ATE) for unpopulated printed circuit boards (PCBs). The product line comprises two families of machines one of them the so called flying probe testers. These machines determine boards with open circuits or shorts between networks of the PCB. Multiple moving probe heads are being positioned on all test points of the PCB. By contacting these test points, electro-static properties are captured by various kinds of measurements. Today typical PCBs contain several tens of thousands of test points and in peak can hold up to 250 000 test points. Goal of the cooperation is to find solutions for this TSP that are close to the optimum yet in very short computation times.

2 Related Work

A comprehensive overview of tour constructing heuristics and tour improving algorithms can be found in [1]. The results in [1] shows that established tour constructing heuristics (Nearest Neighbor, Insertion heuristics, Christofides, Savings) usually range about 10 %–30 % above the lower bound of the instances. The Savings heuristic, being the best, had an average of 11.11 % above the lower bound for some selected instances of the TSPLIB [2].

Several well known tour improvement algorithms are compared: Node/Edge Insertion, 2-Opt and 3-Opt exchanges as well as the Lin & Kernighan heuristic. The tour quality improves from heuristic to heuristic and is rather independent from the initial tour. That is, even with random starting tours, the average quality of tours obtained with 3-Opt and Lin-Kernighan lies between 6.82 % and 2.47 %, respectively. For Christofides starting tours an average tour quality of 3.55 % for 3-Opt and 1.95 % for the Lin-Kernighan heuristic are achievable. An iterated approach to Lin-Kernighan yields tours that lie on average only 1.26 % above the lower bound.

In fact the Lin-Kernighan heuristic implemented by CONCORDE [3] has been used to solve many instances of the TSPLIB. Several improvements have been made to the original algorithm and the question of how to obtain a good initial tour is still open. As of this writing, the record for the optimal solution lies at 24 978 cities [4].

However, running time constrains often forbid long runs of improvement heuristics, only to shorten the tour by 1 %–2 %. Therefore we focus more on achieving acceptable results (5 %–8 %) in a short time.

An overview of existing algorithms and how they perform can be obtained from the 8th DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) Implementation Challenge [5].

3 Algorithm

3.1 Triangulation

This heuristic is based on the properties of plane triangulation. As the TSP is closely connected to proximity problems, the first choice is a Delaunay triangulation. But it is important to understand that the optimum tour for the TSP is not a subset of the Delaunay graph (see Figure 1).

From another point of view, the optimum tour is a subset of some triangulation, because it contains no intersecting edges. Therefore it is also interesting to investigate other types of triangulation, which may appear to be more suitable for the TSP.

On each step, the heuristic will operate with a non-empty set of directed tours T_i. Tour T_i itself contains i edges E_{mn} of the triangulation graph, connecting cities C_m and C_n. Each edge E_{mn}, which does not belong to the convex hull, has two adjacent triangle faces, one on the left, city $C_L(E_{mn})$, and one on the right, city $C_R(E_{mn})$, respective to the tour direction (see Figure 2). The edges forming the convex hull have only one adjacent triangle face.

For simplicity, let us consider only left faces and cities. It is easy to see that if the direction of the tour is changed—so that the edge E_{mn} changes to E_{nm}—then $C_L(E_{nm})$ is equivalent to $C_R(E_{mn})$ and vice versa.

3.2 Construction

Starting with some tour T_i, the next tour T_{i+1} can be constructed, if an edge E_{mn} is replaced with edges E_{mL} (connecting C_m to $C_L(E_{nm})$) and E_{Ln} (connecting $C_L(E_{nm})$

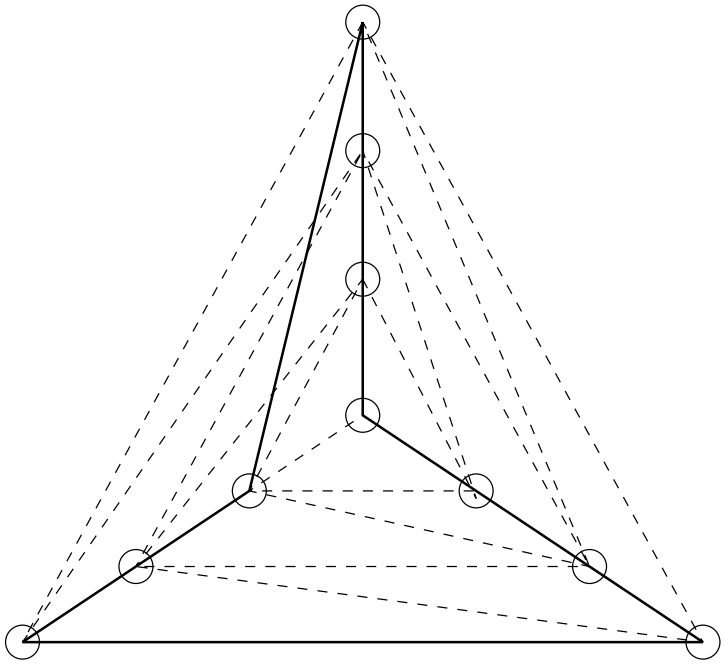


Figure 1: Optimum tour is not a subset of Delaunay graph.

to C_n), with the restriction that the city $C_L(E_{nm})$ does not yet belong to the tour (see Figure 3).

Associated with each edge E_{mn} is the pre-calculated edge quality Q_{mn}. In the general case Q_{mn} and Q_{nm} can be different. Later we will discuss different possibilities to calculate the edge quality.

So the heuristic starts with some plane triangulation, taking the cities as vertices, and some directed initial tour T_k. As we will extend the tour always to the left, the initial tour may be any clockwise tour around triangle face, or counterclockwise tour, consisting of the edges on the convex hull.

To keep track of the edge with the worst quality, references to all edges of the initial tour have to be put into an additional data structure D (heap or balanced tree). Now we can proceed to the main loop.

1. If D is empty, finish the loop.
2. Find the edge E_{mn} with worst quality Q_{mn} and remove it from D.
3. If $C_L(E_{nm})$ does not exist (this can happen if E_{mn} is a clockwise edge on the convex hull), return to step 1.
4. If $C_L(E_{nm})$ already belongs to some tour, return to step 1.
5. Construct the tour T_{i+1}, replacing the edge E_{mn} with edges E_{mL} and E_{Ln}. Add references to E_{mL} and E_{Ln} to D and return to step 1.

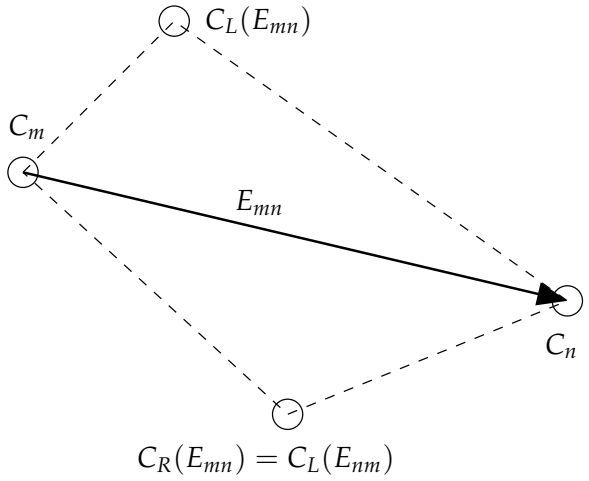


Figure 2: Edge with two adjacent faces.

3.3 Finishing the Tour

It can happen that several cities are not included into the tour. Each of these cities C_x can be included into the tour, as it is shown in Figure 4.

After finding an edge E_{mn}, which belongs to the tour and contains a city C_m, adjacent to C_x, this edge is replaced with edges E_{mx} and E_{xn}. If several edges can be replaced, one which minimizes the increase of tour length $L_{mx} + L_{xn} - L_{mn}$ is selected, where L_{mn} is the length of the edge E_{mn}. Note that the edge E_{xn} does not necessarily belong to the triangulation graph.

3.4 Quality function

There remains one unresolved item in the described algorithm—how to calculate edge quality Q_{mn}. Several approaches are possible. It is not yet clear which of them gives the best result. Here are the classic methods. The city $C_L(E_{nm})$ is denoted as C_l. In each case the edge with minimum Q_{mn} will be on the top of the heap D.

- Farthest insertion: $Q_{mn} = -\min(L_{ml}, L_{ln})$.
- Nearest insertion: $Q_{mn} = \min(L_{ml}, L_{ln})$.
- Cheapest insertion: $Q_{mn} = L_{ml} + L_{ln} - L_{mn}$.

Other quantities for the edges can also be used. Best results till now were achieved using edge ranks. For each city we can rank the edges on their length and assign rank 0 for the shortest edge, rank 1 to the next and so on. Each edge E_{mn} will get two such ranks: R^m_{mn} from the city C_m and R^n_{mn} from the city C_n. Here is the function, which performed best till now.

$$Q_{mn} = L_{ml} + L_{ln} - L_{mn}(R^m_{mn} + R^n_{mn}) \tag{1}$$

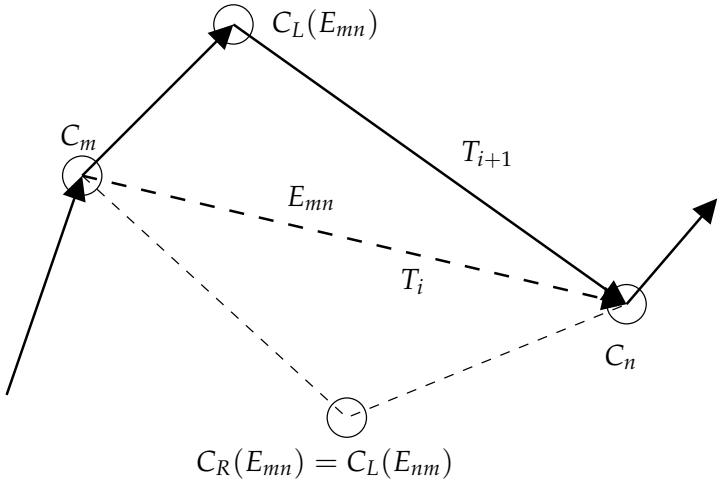


Figure 3: Construction of the new tour.

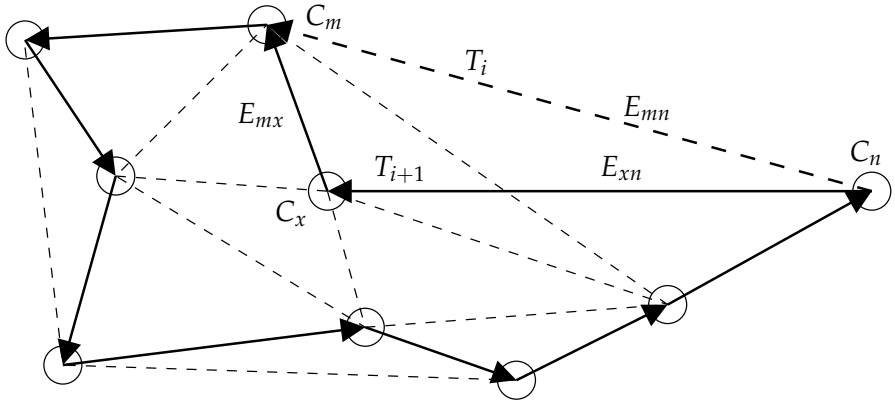


Figure 4: Inserting cities into the tour.

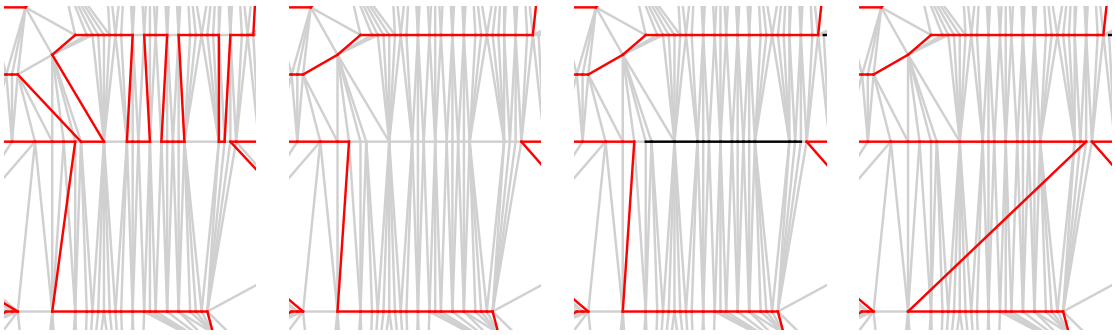
This quality function is purely empirical, so further investigation is necessary to understand it theoretically and to improve the calculation of edge qualities.

3.5 Improvements

Many improvements to the core algorithm were tried. The following two proved to be beneficial for a wide range of problem instances.

3.5.1 Clustering

What we call *clustering* is nothing more than additional subtours that get started inside the main tour. The main tour starts with the convex hull and successively expands into the interior of the graph by removing one edge and including a new city with two new edges (see Section 3.6). Once it is decided that the cost for doing so exceeds the cost



(a) Result (b) Threshold (c) Linear cluster (d) New result

Figure 5: Behaviour towards end of the main loop (rl5934)

of creating a new subtour, such a new subtour is created inside the graph—a cluster is formed. Several such tours may exist and they can be combined (see Algorithm 9). This clustering approach appeared to be better suited for several instances at first, however, some refinements in the normal version of the algorithm closed this gap.

3.5.2 Linear Clusters

Towards the end of the edge removing process we run into the problem depicted in Figure 5. The process is trying too hard and includes cities with relatively high cost. The process is therefore ended, once a certain threshold of the cost has been reached.

The remaining cities could be inserted via the tour finishing function (see Algorithm 11), but an approach with so called *linear clusters* provided slightly better results. These linear clusters are non-closed tours, that are inserted into the tour *en bloc*.

3.6 Pseudo Code

Algorithm 1 shows the main structure of the heuristic. After generating all the necessary data structures and the Delaunay triangulation of the cities, the initial tour is constructed via the convex hull. These first edges get placed onto a heap, sorted by the quality of the edges (see Section 3.4). The triangles are sorted by function (2).

$$Q_t = \frac{\min(L_a, L_b, L_c)}{\text{mid}(L_a, L_b, L_c)} \tag{2}$$

Where L_i is the length of edge i and min and mid are the minimum and middle operations. Lower Q_t are better, because the resulting triangle has one very small angle and two rather long edges. These are good starting points for our edge removal process, as such constructs appear at regular shaped lines of points, as can be seen in Figure 5.

The main loop in line 8 then gathers the cost of all possible four actions: removing an edge from the tour, removing an edge from a cluster, construction a new cluster

(a new triangle) or merging two clusters or cluster and tour. If the threshold has not been reached yet, the step with minimum cost is performed (line 16–24).

Algorithm 1 Heuristic pseudocode

```
 1: procedure SolveTSP
 2:     calculate the delaunay triangulation, edge quality and triangle quality
 3:     set initial, directed tour ← convex hull
 4:     heap E ← all edges in the convex hull                ▷ ≙ initial tour
 5:     heap C ← all edges in the clusters                   ▷ empty upon start
 6:     heap T ← all triangles                               ▷ sorted by function (2)
 7:     heap M ← all cities for cluster merging              ▷ empty upon start
 8:     while true do
 9:         ec ← TourEdgeCost(E)
10:         cc ← ClusterEdgeCost(C)
11:         tc ← TriCost(T)
12:         mc ← ClusterMerge(M)
13:         if ec > threshold and cc > threshold then
14:             break                                        ▷ Do linear clusters instead
15:         end if
16:         if ec is best then
17:             TourEdgeRemove(E)
18:         else if cc is best then
19:             ClusterEdgeRemove(C)
20:         else if tc is best then
21:             TriBuild(T)
22:         else if mc is best then
23:             ClusterMerge(M)
24:         end if
25:     end while
26:     LinearCluster()
27:     TourFinish()
28:     TourImprove()
29: end procedure
```

Algorithms 2, 3, 4 and 5 show what is necessary to calculate the cost of (1) removing an edge from the tour, (2) from the cluster, (3) generating a new triangle, or (4) merging two clusters or cluster and tour, respectively. They consist of pulling the first object from the heap and performing some plausibility checks to see, if the given object is usable in the following step. Algorithms 2 and 3 have one additional task. If removing an edge from the tour or cluster is not possible—because the city to add is already a member of the tour or another cluster—it must be decided, if these two tours can be merged. The cost associated with merging is calculated and the city in question is placed onto heap M.

Algorithm 2 Pseudocode calculating the cost of removing an edge from the tour

```
1: procedure TourEdgeCost(E)
2:     get top element e from heap E
3:     e_1 and e_2 are the corresponding edges, which form a triangle      ▷ see Figure 3
4:     v ← vertex shared by e_1 and e_2
5:     if v ∈ tour then
6:         restart procedure
7:     else if v ∈ cluster then
8:         calculate cost to merge with cluster
9:         add the edge e to M
10:        restart procedure
11:    end if
12:    return the cost associated with edge e                    ▷ city v is still free
13: end procedure
```

Algorithm 3 Pseudocode calculating the cost of removing an edge from the cluster set

```
1: procedure ClusterEdgeCost(C)
2:     get top element e from heap C
3:     e_1 and e_2 are the corresponding edges, which form a triangle      ▷ see Figure 3
4:     v ← vertex shared by e_1 and e_2
5:     if v ∈ cluster then
6:         if v ∈ same cluster then
7:             restart procedure
8:         end if
9:         calculate cost to merge with other cluster
10:        add the edge e to M
11:        restart procedure
12:    else if v ∈ tour then
13:        calculate cost to merge with tour
14:        add the edge e to M
15:        restart procedure
16:    end if
17:    return the cost associated with edge e                    ▷ city v is still free
18: end procedure
```

Algorithm 4 Pseudocode calculating the cost of creating a new cluster

```
1: procedure TriCost(T)
2:     get top element t from heap T
3:     if one city is in tour or cluster then
4:         restart procedure
5:     end if
6:     return the cost associated with triangle t
7: end procedure
```

Algorithm 5 Pseudocode calculating the cost of merging cluster and tour or cluster and cluster

```
1: procedure MergeCost(M)
2:     get top element m from heap M
3:     if both cities already in tour then
4:         restart procedure
5:     else if both cities belong to the same cluster then
6:         restart procedure
7:     end if
8:     return cost associated with m
9: end procedure
```

Algorithm 6 Pseudocode removing an edge from E and adding two new edges

```
1: procedure TourEdgeRemove(E)
2:     remove top element e from heap E
3:     e1 and e2 are the corresponding edges, which form a triangle      ▷ see Figure 3
4:     v ← vertex shared by e1 and e2
5:     add city v to the tour
6:     add e1 and e2 to E
7: end procedure
```

Algorithm 7 Pseudocode removing an edge from C and adding two new cluster edges

```
1: procedure ClusterEdgeRemove(C)
2:     remove top element e from heap C
3:     e1 and e2 are the corresponding edges, which form a triangle      ▷ see Figure 3
4:     v ← vertex shared by e1 and e2
5:     add city v to the cluster
6:     add e1 and e2 to C
7: end procedure
```

Algorithm 8 Pseudocode building a new cluster out of three edges

```
1: procedure TriBuild(T)
2:     remove top element t from heap T
3:     create a cluster consisting of a simple triangle
4:     add all three edges to C
5: end procedure
```

Algorithm 9 Pseudocode merging two clusters or one cluster with the tour

```
1: procedure ClusterMerge(M)
2:     remove top element m from heap M
3:     if cluster ↔ cluster then
4:         merge clusters by removing two edges and adding another two
5:         put these two new edges onto C
6:     else                                                    ▷ tour ↔ cluster
7:         merge cluster with tour by removing two edges and adding another two
8:         put these two new edges and all edges from the cluster onto E
9:     end if
10: end procedure
```

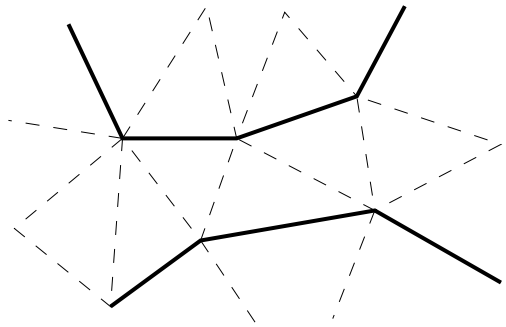
Algorithms 6 and 7 are nearly identical and perform the removal of the worst edge e from either heap, adding another city v and putting the newly used edges back onto the heap. Algorithm 8 simply pops off the first triangle structure and transforms it into a minimal cluster, adding three edges to the cluster heap. Algorithm 9, however, is fairly complex. There are not only three possible calls (tour ↔ cluster, cluster ↔ tour, cluster ↔ cluster) but also two possible ways to merge the two constructs (see Figure 6).

Algorithm 10 Pseudocode building linear clusters and merging them into the tour

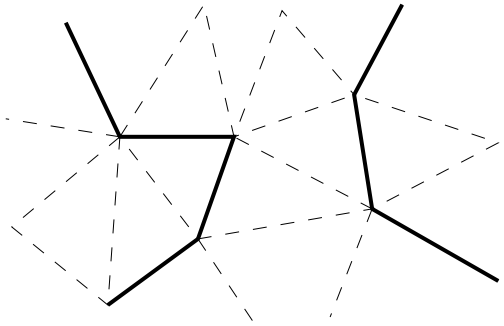
```
1: procedure LinearCluster
2:     for all cities not in tour do
3:         create linear, open clusters with delaunay edges
4:         merge them into the tour
5:     end for
6: end procedure
```

Algorithm 10 for *linear* clusters—that is, clusters that do not form a closed cycle—has even more cases to check. There are two endpoints to consider. Then each facing edge has to be checked for the best possible way to integrate the cluster. This can be done in four different ways.

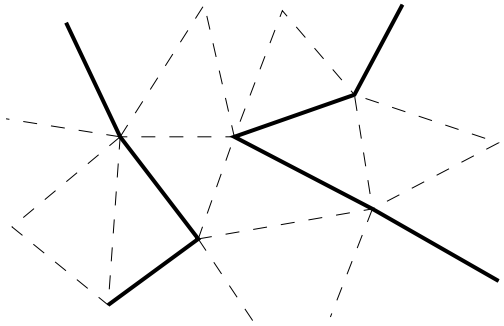
Algorithms 11 and 12 are very similar too. They have been separated, because TourImprove is not strictly necessary to obtain a valid tour, whereas TourFinish is (see also Section 3.3 on page 4). They are rather straightforward and check for every city v, if it could be better inserted before or after every adjacent city. If the tour improvement would be positive, the city is then detached from its current position in the tour, and reinserted into the better position. The difference to TourFinish is that it then places all cities, that share an edge with the five changed cities—city v, its old previous and next city and its new previous and next city—back onto the heap. This is basically a node insertion heuristic with the candidate set limited to cities, that are at most two delaunay edges away from the current city.



(a) Two tour sections



(b) Solution 1



(c) Solution 2

Figure 6: Possible ways of merging two tours

Algorithm 11 Pseudocode connecting remaining cities to the tour

```
1:  procedure TourFinish
2:      put all remaining cities onto heap H
3:      for all cities c in H do
4:          search all facing edges e for the best spot to insert c
5:          if no e could be found then          ▷ no facing edge is yet part of the tour
6:              re-add c to the bottom of H
7:              continue for loop
8:          end if
9:          remove edge e and add two new edges connecting c to the tour
10:     end for
11: end procedure
```

Algorithm 12 Pseudocode performing Node Insertion optimization

```
1: procedure TourImprove
2:     put all remaining cities onto heap H
3:     for all cities c in H do
4:         search all facing edges e for a better spot to insert c
5:         if no better e could be found then
6:             continue for loop
7:         end if
8:         remove c from current tour
9:         put c between the two cities connected through e
10:        put all adjacent cities to the five changed cities back onto H
11:        put the five changed cities back onto H
12:    end for
13: end procedure
```

4 Results

4.1 Test setup

All tests were conducted on a Dell Inspiron 8600c laptop (Intel Banias CPU, 1 MB L2 cache, 1.5 GHz and 1 GB RAM) running FreeBSD 5.4. The program itself, as well as CONCORDE, was compiled using the Intel C Compiler (ICC) version 8.1.028 with very high optimization settings (`CFLAGS=-O3 -xB`).

All times reported are the *user times* (i. e., the time the program spend executing code in userland). It does not contain the time the process waited for I/O to finish or was otherwise blocked from running.

Because of the Delaunay triangulation this approach uses, only instances with either euclidean or ceiling norm can be computed. Thus, only those instances were used from the following libraries: TSPLIB, National TSPs and the Very Large Scale Integration (VLSI) TSPs. The first can be obtained from [2], the latter two from http://www.tsp.gatech.edu/. The results will be limited to instances with more than 10 000 (TSPLIB, National TSPs) and 50 000 (VLSI TSPs) cities, to avoid showing endless tables. Besides, the goal of this heuristic is to get good tours of huge instances in short time.

The resulting tours are compared against the known optimal solution or the best known lower bound. This provides an exact or slightly pessimistic overview of the quality of tours achieved. However, no lower bounds exist for the VLSI instances, therefore the solution is compared to the best known tour as of this writing. Comparisons for the VLSI instances are thus rather optimistic, as the real solution could actually be an even shorter tour. The figures shown are the percentages above the known lower bound, calculated via:

$$\text{quality of tour} = \left(\frac{\text{calculated tour}}{\text{lower bound}} - 1 \right) \cdot 100$$

4.2 Version with linear clusters

The first version of `woptsa` only used the TourEdgeRemove (Algorithm 6) and the LinearCluster function (Algorithm 10), i. e., it used no triangles/cyclic clusters at all. The quality function (3) was used (compare also equation (1) on page 4).

$$Q_{mn} = L_{ml} + L_{ln} - p \cdot L_{mn}(R^m_{mn} + R^n_{mn}) \tag{3}$$

The parameter p describes a weighting of the edge rank computation. The original version ran the algorithm eleven times varying p ($p \in \{0.20, 0.25, \ldots 0.70\}$) and returned the shortest tour achieved. The results can be seen in Table 1. N is the number of cities, M the number of cities left for OptTourFinish to include, O denotes the number of tour changes done in OptTourImprove. p is the weighting parameter for function (3). q and t show the excess above the lower bound/known optimal tour and the running time.

4.3 Version with cyclic clustering

A different quality function was used for the cyclic clustering case:

$$Q_{mn} = \frac{L_{ml} + L_{ln} - 2 \cdot L_{mn}}{L_{mn}} \tag{4}$$

This provided a normalized increase in tour length for Q_{mn}. Once the removal of edge E_{mn} and the addition of edges E_{ml} and E_{ln} would result in the new edges being at least twice as long as the old edge, the value of Q_{mn} will be greater than 1.

This value is critical for the threshold t. The main loop ran an additional three times with t varying between $\{1.0, 1.5, 2.0\}$. The best tour of those 14 runs was then passed to OptTourImprove for further improvement.

As it stands now, the additional clustering helps with a few small instances of the TSPLIB, where the original code with linear clustering only achieved tour qualities of 21 % (fl3795 and fl1577, see Figure 7). These could be brought down to 13 % and 9 % above the lower bound, respectively. Comparing all the results from the TSPLIB, the average excess could be reduced from 5.69 % to 5.19 % and the standard deviation from 4.18 % to 3.04 %. A lower deviation is very good, as it improves the estimation of qualities for unknown instances.

4.4 Comparison

Besides `woptsa`, the running times and tour qualities were compared to several other algorithms as implemented by CONCORDE (see Table 2). Namely a 2-Opt and 3-Opt heuristic starting with a Greedy tour (`kdtree`) and a Chained Lin-Kernighan heuristic provided via `linkern` (Quick-Boruvka starting tour, one round, # kicks = # cities). All CONCORDE-tools used 99 as random seed, `woptsa` is not relying on random numbers and is not seeded.

TSPLIB

Instance	N	M	O	p	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
rl11849	11849	26	1068	0.55	9.32	0.33
usa13509	13509	39	1667	0.45	6.31	0.35
brd14051	14051	49	1369	0.50	5.31	0.36
d15112	15112	45	1605	0.50	5.38	0.43
d18512	18512	56	1834	0.55	5.30	0.52
pla33810	33810	94	3085	0.25	7.39	0.92
pla85900	85900	166	8819	0.60	8.21	2.68
total					6.75	5.59
stddev					1.60	

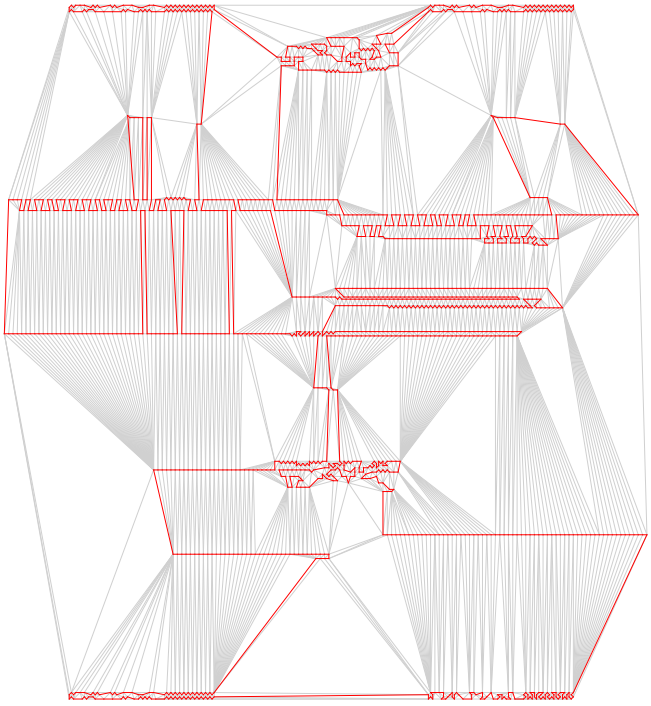
National TSPs

Instance	N	M	O	p	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
fi10639	10639	20	1083	0.40	5.89	0.26
mo14185	14185	38	1423	0.40	6.71	0.36
it16862	16862	39	1636	0.40	5.75	0.47
vm22775	22775	51	1970	0.50	6.42	0.65
sw24978	24978	64	2609	0.35	6.51	0.72
ch71009	71009	193	7348	0.45	6.05	2.38
total					6.22	4.84
stddev					0.38	

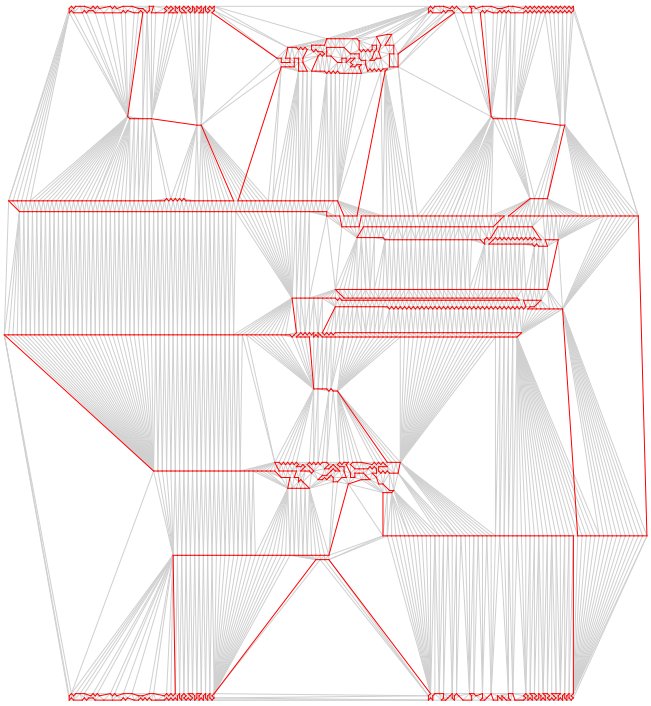
VLSI

Instance	N	M	O	p	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
fna52057	52057	102	3523	0.45	8.58	1.73
bna56769	56769	123	3893	0.50	8.06	1.91
dan59296	59296	161	4201	0.35	8.94	2.00
sra104815	104814	274	7975	0.40	9.80	3.65
ara238025	238025	581	18594	0.40	9.51	9.25
lra498378	498378	1078	42601	0.50	11.06	21.74
lrb744710	744710	1638	51267	0.40	9.11	32.48
total					9.29	72.76
stddev					0.97	

Table 1: Qualities and times for the first version of `woptsa`



(a) Without clusters



(b) With clusters

Figure 7: Final tours for fl1577

TSPLIB

		woptsa		2 Opt		3 Opt		linkern	
Instance	N	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
rl11849	11849	9.32	0.33	9.41	0.11	5.81	0.20	0.35	20.82
usa13509	13509	6.31	0.35	10.75	0.17	7.48	0.27	0.24	40.84
brd14051	14051	5.31	0.36	9.13	0.16	6.28	0.24	0.19	28.58
d15112	15112	5.38	0.43	9.10	0.19	6.27	0.32	0.19	35.43
d18512	18512	5.30	0.52	9.38	0.23	6.51	0.33	0.21	35.39
pla33810	33810	7.39	0.92	12.13	0.36	8.50	0.60	0.57	73.39
pla85900	85900	8.21	2.68	10.00	1.10	7.24	1.58	0.29	211.88
total		6.75	5.59	9.99	2.32	6.87	3.54	0.29	446.33
stddev		1.60		1.11		0.92		0.14	

National TSPs

		woptsa		2 Opt		3 Opt		linkern	
Instance	N	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
fi10639	10639	5.89	0.26	12.10	0.12	8.84	0.19	0.28	22.47
mo14185	14185	6.71	0.36	10.06	0.17	6.98	0.26	0.25	30.26
it16862	16862	5.75	0.47	12.52	0.20	8.25	0.47	0.19	44.37
vm22775	22775	6.42	0.65	12.20	0.29	8.91	0.41	0.24	50.18
sw24978	24978	6.51	0.72	11.00	0.31	7.51	0.59	0.31	55.61
ch71009	71009	6.05	2.38	9.27	1.28	5.91	1.87	0.23	186.17
total		6.22	4.84	11.19	2.37	7.73	3.79	0.25	389.06
stddev		0.38		1.31		1.17		0.04	

VLSI

		woptsa		2 Opt		3 Opt		linkern	
Instance	N	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$	$q \cdot 100$	$t \cdot \mathrm{s}^{-1}$
fna52057	52057	8.58	1.73	10.06	0.61	7.02	0.89	0.50	69.51
bna56769	56769	8.06	1.91	10.44	0.68	7.01	0.96	0.46	78.28
dan59296	59296	8.94	2.00	10.13	0.70	7.69	1.00	0.50	81.86
sra104815	104814	9.80	3.65	10.89	1.47	8.26	2.49	0.48	207.25
ara238025	238025	9.51	9.25	10.66	3.72	7.85	6.00	0.48	531.19
lra498378	498378	11.06	21.74	9.95	9.13	7.09	18.67	0.56	1276.58
lrb744710	744710	9.11	32.48	10.48	15.38	7.77	60.05	0.51	2016.67
total		9.29	72.76	10.37	31.69	7.53	90.06	0.50	4261.34
stddev		0.97		0.34		0.49		0.03	

Table 2: Comparison of tour qualities and computation times between woptsa, 2-Opt, 3-Opt and Lin-Kernighan heuristics as implemented by CONCORDE.

Instance	slower, better	slower, worse	faster, worse	faster and better	
pla85900	65	14	26	9	(2/3opt-JM, 2/2.5opt-B, LK-ACR, LK-ABCC, CLK-ACR)
C316k.0	42	6	32	7	(2/3opt-JM, 2/2.5opt-B)
E3M.0	21	9	30	2	(2opt-JM, 3opt-JM)

Table 3: Speed and quality compared to `woptsa` for three selected instances

Since the cyclic clusters made no significant impact on instances with more than 10 000 cities, we use only the version from Section 4.2.

Figure 8 shows the running time versus tour quality comparison for pla85900 from the TSPLIB and two instances from the 8th DIMACS challenge: E3M.0, three million uniformly distributed points and C316k.0, 316 000 clustered points. The exact details of the compared heuristics can be taken from the DIMACS Homepage [5]. Note that these are highly optimized heuristics that are either very fast or very good. The figures are divided into four quadrants with `woptsa` being the intersecting point. The number of heuristics in each quadrant is also shown. As can be seen, there are only a few heuristics that beat `woptsa` both in time and quality.

Table 3 shows how many and which heuristics fared better or worse than `woptsa`.

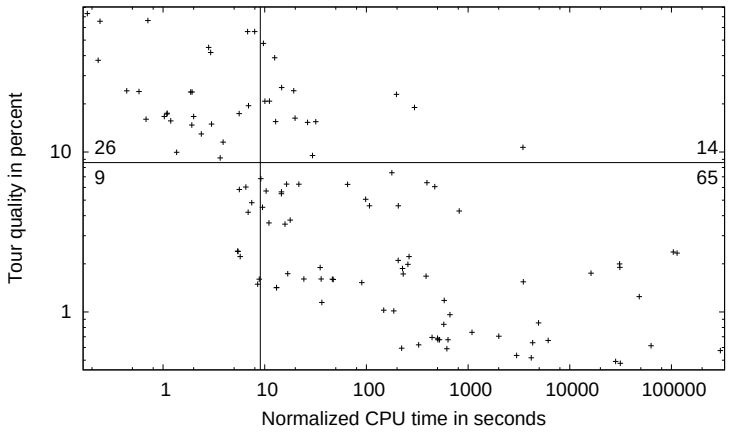
5 Conclusion

In this work we presented an algorithm for the Traveling Salesman Problem, based on Delaunay triangulation. An advantage of this algorithm is that there is only one parameter to tune: the edge quality, or in other words, the decision which edges to remove, to advance the given tour. Here, it is not obvious, why some instances are very sensitive to the weighting factor p of the edge rank. Further research is necessary in this area.

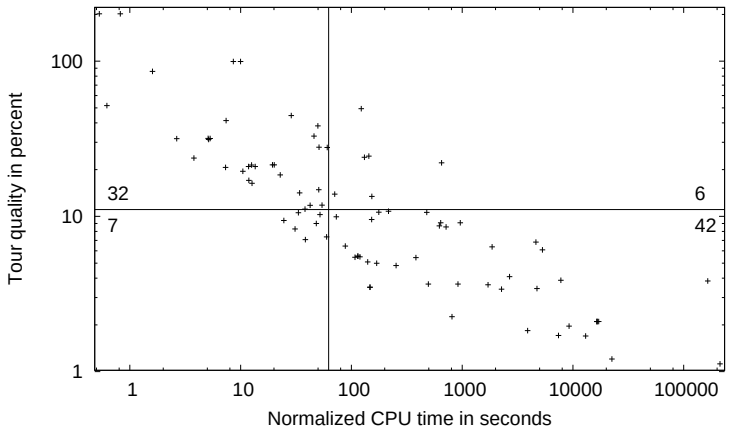
The heuristic also shows a good tradeoff between speed and length of the obtained tour. There are only a few selected and highly optimized 2-Opt, 3-Opt and Lin-Kernighan heuristics that do a better job overall.

The algorithm stabilizes with increasing node count. With fewer nodes, there can be high variations in the achieved tour quality as already mentioned in Section 4.3 (see also Figure 7). This also explains the high overall standard deviation of tour qualities within the TSPLIB, whereas the instances with 10 000 nodes or more only show a standard deviation of 1.60.

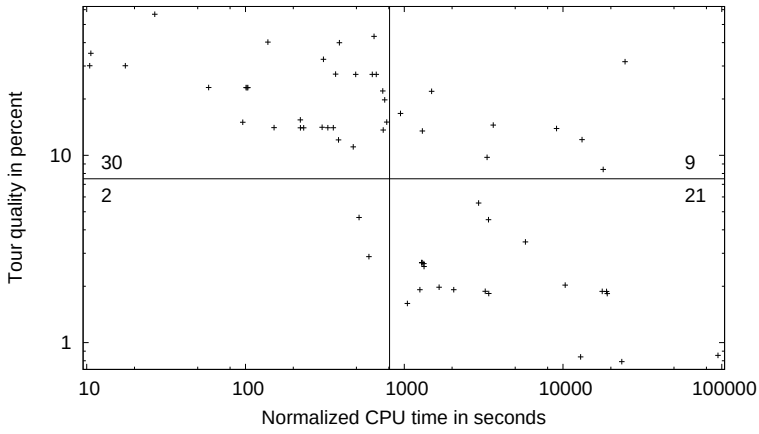
Another notable observation is the correlation between the tour quality and graphs with lots of regular shapes. The National TSPs are solved very well and—due to their nature of being maps—their distribution of delaunay edge lengths varies highly. Contrast that with the VLSI instances, which have lots of regular patterns and thus many equally long delaunay edges grouped together. The same can be seen with



(a) pla85900



(b) C316k.0



(c) E3M.0

Figure 8: Runtime vs. quality comparison of several heuristics.

the TSPLIB instances. Those with regular patterns (rl11849, pla33810, pla85900) have qualities above 7 % and the city map instances (usa13509, brd14051, d15112, d18512) are below 6.5 % (see Table 1).

References

[1] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer, 1994.

[2] G. Reinelt, "TSPLIB – a Traveling Salesman Problem library." http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/, 1995.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "Concorde: A code for solving Traveling Salesman Problems." http://www.tsp.gatech.edu/concorde.html, 2004.

[4] D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun, "Optimal tour of Sweden." http://www.tsp.gatech.edu/sweden/, 2004.

[5] D. Johnson, "8th DIMACS implementation challenge: The Traveling Salesman Problem." http://www.research.att.com/~dsj/chtsp/, 2004.