Modified Greedy Delaunay Graph-Based Method for TSP Initialization

Jimi Tuononen
School of Computing
University of Eastern Finland
Joensuu, Finland
jimi.tuononen@uef.fi

Pasi Fränti
School of Computing
University of Eastern Finland
Joensuu, Finland
franti@cs.uef.fi

***Abstract*— Finding an initial solution for a traveling salesman efficiently and with reasonable quality is needed both by the state-of-the-art local search and branch-and-bound optimal algorithms. Classical heuristics require O(n^2), and their quality may be insufficient to keep the efficiency of local search algorithms high. In this paper, we present a modified greedy algorithm. It utilizes a Delaunay neighborhood graph to build initial fragments until no more connections are possible. Then it merges small fragments with other fragments. It then connects the fragments by a two-step process. First, the algorithm re-generates the graph and iterates the process until the number of fragments falls below threshold 1,000. After that, it performs a full search to connect the rest of the fragments in a greedy manner. The method achieves gaps of 2% and 17% for Dots and selected TSPLIB datasets, and gaps of 27.4% and 16.6% for Santa and World TSP when comparison is made to highly optimized solutions.**

Keywords—Travelling salesman problem, TSP, initial tour, initialization, Delaunay graph, greedy

I. Introduction

Traveling salesman is a well-known NP-hard problem. The search for optimal solutions requires exhaustive search such as branch-and-bound or Concorde [1], which becomes very slow on larger problem instances. A compromise is to find a good quality but sub-optimal solution using local search such as *Lin-Kernighan heuristic* (LKH) [2]. It is based on the so-called *k*-opt operation and has been state-of-the-art for more than two decades already. Local search strategies require an initial solution to start with.

There are many seemingly good heuristics available including nearest neighbor, greedy, and insertion algorithm that provide a reasonable solution [3, 4, 5]. However, their quality may be too low for branch-and-bound, or they are too slow. Optimal algorithms, such as branch-and-bound, strongly depend on the quality of initialization. The higher the quality of the initial solution, the more branches can be cut early. One approximate criterion is that the initial solution should have a gap of about 1% or less to the optimal solution to achieve significant savings compared to an exhaustive search.

The efficiency of local search depends on the initialization. The better the quality of the initial solution is, the fewer iterations of local search are needed. Starting from random initialization can easily multiply the running time. Moreover, simple heuristics also require O(n^2) time to run which adds to the total processing time. Therefore, it is an open question which algorithm to use for the initialization step.

Divide-and-conquer approaches have been used to address large-scale problem instances. They divide the problem instance into several sub-problems by clustering. However, the results for Santa Claus data consisting of 1.4M nodes were not any better than the state-of-the-art local search algorithm utilizing neighborhood search [6]. The key element is the usage of the neighborhood graph to speed up the initialization and k-opt operation. This avoids wasting time attempting k-opt moves which do not have a chance of improving the solution.

In this paper, we explore simple heuristics based on the Delaunay graph. The neighborhood graph has already been used in the LKH algorithm but the result of using different graphs and initialization methods is less studied for generating the initial solution. The LKH software [2] includes a few faster variants, but they are not well documented and lack empirical evidence of which one should be used. We present a variant based on the Delaunay graph. Comparison is made against existing variant by measuring the gap to the optimal solution (when known) and measuring the time taken.

II. Existing Initialization Heuristics

We next briefly recall the existing heuristics for solving TSP. We consider only simple heuristics; referred to as classical in [7]. They should be something simple but also fast to generate to serve the purpose of being an initial solution for local search. High quality solutions would also be desirable when using them as an initial solution for an exhaustive search like branch-and-cut. However, this is only a secondary property and something below O(n^2) is our primary goal. We consider the following algorithms:

- Nearest neighbor [3, 4]
- Nearest neighbor (fast) [2]
- Greedy graph [2]
- Kruskal-TSP [8]
- Moore [9]
- Sierpinski [10]

979-8-3503-6320-3/24/$31.00 ©2024 IEEE

- Boruvka [11]
- Quick-Boruvka [7, 12]
- Christofides [13, 14]
- Walk [2]
- Delaunay shortest edge [15]
- Delaunay random edge [15]
- Delaunay greedy with fragment merge (new)

The first thing to note is that straightforward implementation of even the simplest nearest neighbor and greedy algorithms require O(n^2) time. Fortunately, the LKH implementation has paid attention to this and utilizes a neighborhood graph [2]. The package implements fast variants of several of the above-mentioned heuristics.

The nearest neighbor starts from a random node and then continues to the nearest unvisited node until all nodes have been visited. It would be possible to create a heap from the distances to speed up the search to O(n log n) but the distance matrix still requires O(n^2) time to generate. A faster variant in [2] utilizes the neighborhood graph. This trick speeds up the method to O(nk), where k is the number of neighbors.

The LKH implementation includes randomization as the most favorable edge is used only 2/3 of the time. This provides a variation of tours which is needed for the restart of local search. However, the path length will also be longer because the solution will include unfavorable edges during the building process.

A greedy algorithm sorts the edges in increasing order and then selects the shortest edge at each step. There are two alternative ways to implement this. The first variant constructs a single path by adding new nodes to its end nodes. This corresponds to Prim's algorithm for finding a minimum spanning tree (MST) with the difference not allowing to create branches. The second variant maintains a forest of multiple trees (or paths in the case of TSP). The variants were referred to as Prim-TSP and Kruskal-TSP in [8] according to the corresponding MST algorithms they resemble. Here we used the second variant.

The greedy implementation in LKH is done slightly differently but also utilizes the neighborhood graph. The first step calculates the nearest neighbor of each node. Each neighbor and the corresponding shortest edge are placed into a min-heap structure. The nodes in the heap are then processed in ascending order. When a node is popped up from the heap, it is added to the tour via the shortest edge stored. If the node has a degree less than two, a new nearest neighbor is found, and the node is reinserted back into the heap. The nearest neighbor is done by a breadth-first search on a so-called *candidate graph*.

Boruvka is another greedy approach inspired by MST. The difference between the greedy approach and Boruvka is that the greedy approach follows the order of the edge lengths at every step whereas Boruvka sticks to the node order created by the sorting. In the first step, it calculates the nearest neighbors. The nodes are then processed in ascending order according to their length. A node is connected to the tour if that does not create a branch or a cycle analogous to the Boruvka MST algorithm. In the LKH implementation, this process is repeated until all the fragments are connected.

Quick-Boruvka does not add the node back into the heap but attempts to continue the tour immediately from that node; except when the degree of the node is already two. This results in unconnected fragments which are connected later as will be explained in Section 3. The method compromises quality for speed [7, 12] with mixed results with our datasets.

Christofides [13, 14] is a famous school-book algorithm. It uses MST as a starting point and adds supplementary edges by optimal pairing so that all nodes would have even edges to allow the construction of Euler tour. The result is shortened by shortcuts based on triangular inequality which holds for planar graphs. The algorithm has mediocre performance compared to other MST-based approaches [16] but is famous due to its 3/2 upper bound. The method is very slow and was therefore not included in the results.

Space-filling curves have also been considered [17]. They divide the space into four quadrants, which are recursively divided further until every node has its cell. The cells are indexed, and the space-filling curve travels through the cells according to the order created by this index. TSP path is obtained by sorting the nodes according to their index (order in this curve) and requires only O(n log n) time due to the sorting. We consider the Moore curve [9] and Sierpinski curve [10] as they are implemented in the LKH software [2].

The components of the graph-based methods are link selection, addition, graph, and dead-end strategy as summarized in Table 1.

Graph refers to how the candidate graph is created. LKH includes several variants, of which we consider only the Delaunay graph as it was found to outperform the other variants with Santa data [6].

The Selection strategy refers to how to select the next link to be added. The *Nearest neighbor* always connects to the recently added node. *Greedy* allows the connection of any two nodes. It creates multiple fragments as in Kruskal-TSP [8]. *Insertion* maintains only one tour but allows the addition of nodes also in the middle using a detour through the added node.

Multiple fragments with the limitation of the search to the graph may lead to a dead-end situation. It happens when none of the fragments have unused nodes in their neighborhood anymore. Some variants may prevent this by updating the neighborhood graph, but in general, we may need a *dead-end* strategy.

We consider two main strategies to resolve this: *Full search* and *Re-graph*. Full search simply stops using the graph and finds the remaining nodes by full search with greedy selection. Re-graph builds a completely new neighborhood graph from the endpoints of the fragments and iterates the original algorithm with this new graph.

TABLE I. OVERVIEW OF FAST GRAPH-BASED INITIALIZATION METHODS

Algorithm	Ref.	Selection strategy	Multiple fragments	Graph	Dead-end strategy
Nearest Neighbor (fast)	[2]	NN	-	Many options	No
Walk	[2]	NN+ random	-	Many options	No
Greedy (graph)	[2]	Greedy with updates	Yes	Many options	No
Boruvka	[11]	Greedy	Yes	Many options	No
Quick-Boruvka	[7, 12]	Greedy	Yes	Many options	Full search*
Delaunay shortest edge	[15]	Insertion	-	Delaunay	No
Delaunay random edge	[15]	Insertion	-	Delaunay	No
Delaunay greedy FM	New	Greedy	Yes	Delaunay	Re-graph + Full search

*See section 3.

III. MODIFIED GREEDY DELAUNAY

A neighborhood graph has been used in [2] for speeding up the local search. Especially with large problem instances, it is inefficient to consider randomly selected candidates for the k-opt operation. Connecting points located far away from each other is mostly a waste of time and should be avoided. Instead, the operator should only consider candidates in the same neighborhood. This is the key component in the state-of-the-art algorithm where the neighbors are decided with the candidate graph. The so-called alpha nearest criterion is used by default [2].

It is logical to utilize the neighborhood graph also for the initialization as well. In the package [2], this has been done, and a few fast variants of the classical heuristics have been implemented. They can also be used with different graphs. However, in the recent Santa Claus challenge [6], it was observed that using the simpler Delaunay graph can improve the tour length of LKH further from 109,284 to 108,996. While Delaunay is potentially overwhelming in structure and time-consuming to create, this is not the case for 2-dimensional planar graphs. Motivated by this, we designed two simple variants based on the Delaunay graph: Delaunay random edge and Delaunay shortest edge [15]. However, they were not as good as we wanted. In this paper, we designed an improved, more thought-out variant of them called *Delaunay greedy*.

The implementation is made in Java, and it utilizes different data structures to speed up computations. Edges are processed through a priority queue and thus are obtained in ascending order. Fragments are handled with a custom doubly linked list implementation, which provides constant time retrieval of head and tail nodes. The fragments are handled through a linked *hashmap* to allow fast references.

In some cases, the travel direction of the fragment needs to be flipped to properly connect the fragments. We always flip the smaller fragments when possible. The flip could also be made faster (constant time) by including a reversal bit in the implementation as done in [2], but the operations are already fast, and we did not see the need for it. The nodes are also presented with a custom implementation that allows attributes such as degree, previous, next, fragment, and neighbors.

A. Delaunay graph

The Delaunay graph is based on the Voronoi diagram which divides the space by drawing borders between the points so that it corresponds to the nearest neighbor mapping of every point in the space to its nearest reference point (node). Each node conquers its region in this space. Connecting neighboring nodes creates so-called *Delaunay triangulation* and the graph created from these connections is called the *Delaunay graph*. While the size of the graph can become overwhelming in higher dimensional spaces, it is useful and efficient to construct in O(*n* log *n*) time in 2-D space. Some examples of Delaunay graphs are given in Figure 1.

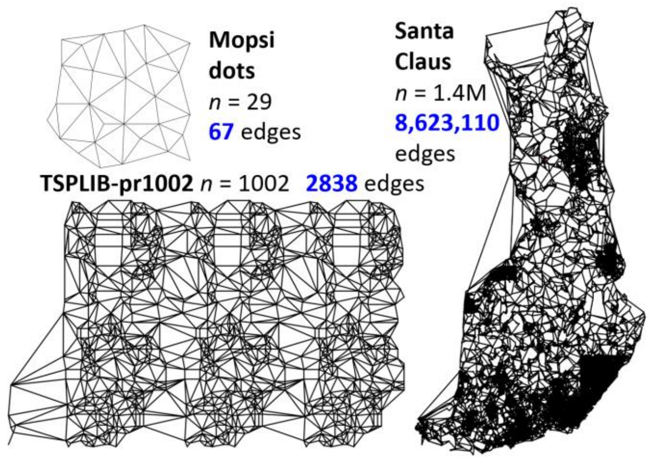

Figure 1. Examples of Delaunay graphs for used datasets.

Simpler variants of Delaunay that would work better in higher dimensional spaces are the *Gabriel graph* [18] and its heuristic variant called the *XNN graph* [19]. It was shown in [19] that 97% of the links in the optimal TSP path are included in the XNN graph. This makes it a suitable data structure for constructing TSP as we can find most links by much faster search within the graph.

The LKH uses the alpha-nearness criterion to prioritize the candidate selection by default. However, the results in [6] showed that the simple Delaunay graph with some alpha-nearness optimization works slightly better, and it takes only O(*n* log *n*) time in 2-D space.

B. Insertion or no insertion

This section describes the rules used for Delaunay edges when constructing so-called fragments [5]. Initially, every node has a degree of 0 as they do not belong to any fragment. Nodes with degree of 1 are the endpoints of fragments. Nodes with degree of 2 are in the middle of a fragment.

The edges are processed in ascending order. An edge with endpoints A and B is added to the tour if it fulfills the rules:

- Deg(A) = 0 and Deg(B) = 0: a new fragment is created.
- Deg(A) = 0 and Deg(B) = 1: node A is connected to an existing fragment.
- Deg(A) = 1 and Deg(B) = 1: the fragments are connected.

Where Deg refers to the degree of the node. Case 2-0 (degrees equal to 0 and 2) corresponds to an insertion of node A by making a detour from node B. This possibility is discarded as it provides longer tours in our experiments. This method would also create edges that are not included in the Delaunay graph and use longer Delaunay edges in the earlier phases of construction. The remaining cases 2-1 and 2-2 are not allowed as they would create branches.

C. *Connecting fragments*

The initial construction will cause a forest of fragments that need to be connected to form a tour. It is a side-effect of using the graph and does not happen in the full search. The time complexity of a full search would be at least O(n^2log$n$) because of sorting all pairwise edges, which makes it not suitable for medium to large-scale TSP. As such, this method can be seen as a faster variant of Kruskal-TSP [8].

The fragments can be connected in multiple ways, see Table 2. We consider the following three approaches:

- Nearest Neighbor
- Re-Delaunay
- Full search

Nearest Neighbor is the simplest approach. It starts from a random fragment and connects its head or tail to any other fragment's head or tail. It continues from the newly added node until all the fragments are connected. Its time complexity is O(r^2), where r is the number of fragments. It is much smaller than the full search O(n^2) assuming r<<n. However, the number of fragments in the Santa dataset [6] is somewhat large (r=50,841) and the approach is inefficient. We therefore use it only when the number of fragments falls below an empirically selected threshold value of 2,000. The *Re-Delaunay* method is used until the number of segments falls below this threshold.

TABLE II. OVERVIEW OF FRAGMENT CONNECTION METHODS. R REFERS TO THE NUMBER OF FRAGMENTS.

Approach	Time complexity	When used
Nearest Neighbor	O(r^2)	r<2,000
Re-Delaunay	O($r$log$r$)	Any n
Full search	O(r^2log$r$)	r<1,000

Re-Delaunay creates a new Delaunay graph from the tail and end points of the fragments and then applies the same tour extension with this new graph. The edges are processed in ascending order using the same ruleset of Section III-B. The process may need to be repeated several times (usually 1-3) as the dead-end situation may still re-occur with the new graph.

The *Full search* approach uses the same greedy extension algorithm as the first step but without any graph. Instead, the shortest distance is searched among the remaining fragments' heads and endpoints. Compared to the nearest neighbor, it selects the shortest edge among all remaining edges. This requires sorting the edges in ascending order and storing them in a priority queue. This leads to a slightly higher time complexity, O(r^2log$r$).

The Full search method can also be time-consuming for large-scale TSP instances. We therefore use it only for the cases r < 1,000. When there are more fragments, we apply the Re-Delaunay method until the number of fragments falls below this threshold. The motivation for this is that the fragments connected at the latest are the costliest ones, and worth applying full search for whereas the earlier connections can be dealt with the re-graph heuristic. The full search approach is suitable as standalone for medium-scale TSP instances. In the experiments, this variant was found to be the most effective and is therefore used as our baseline method.

Lastly, we mention the variant used in the LKH package [2]. It is a refined version of the Nearest Neighbor approach. It finds the least costly edge for each fragment's end nodes and uses these edges to connect fragments in ascending order. There will be collisions where the original edge is no longer suitable for forming a tour. In these cases, the edge is replaced with a new least costly nearest neighbor and placed back in a heap. This approach was also considered, but it was found to be worse than the full graph approach.

D. *Fragment merging*

Many small fragments are created during the fragment-building process (Table 3). The idea of the fragment merging step is to connect these small fragments to nearby fragments to reduce the number of fragments, and to avoid more costly fragment connections during the fragment connection step. This is performed as the first step of connecting the fragments.

TABLE III. FRAGMENT SIZES IN THE USED DATASETS.

Dataset	1–4	5–9	10+	Total
TSPLIB	20.5 %	14.3 %	65.2%	5,460
Santa	12.4 %	11.4 %	76.3 %	50,841
World-TSP	23.0 %	14.1 %	62.9 %	91,307

The fragment merging is implemented as a simple constant-time operation. Using the candidate set created by the Delaunay graph, we find the least costly edge for the fragment's head or tail node. This will serve as one of the connecting edges. The node that was found will be the connecting point and the other end of the fragment will be connected before or after this connecting point. This will result in cutting one edge away from the other fragment. Four different cases need to be considered when calculating the least costly option for the edges. The small fragment's orientation can be flipped (head can be tail and vice versa), and the other connecting point needs to be decided. The fragment merging operation is demonstrated in Figure 2.

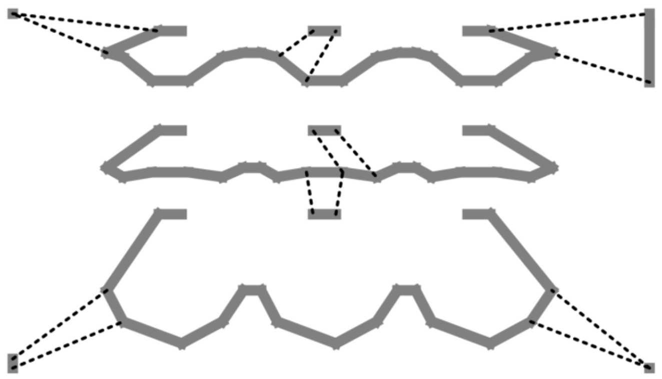
Figure 2. Example of fragment merging for a TSPLIB instance (pr76)

The effect of merging different sizes of fragments is demonstrated in Figures 3, 4, and 5 on different datasets. The merge value notes up to which size of fragments we are merging. The figures also show the effectiveness of the different fragment connection methods. The merge value 4 is consistently the best across all tested datasets.

The reason for four being the best merge value is caused by the simplicity of the operation. The merge is done using the two connection points, which works best for small fragments. When merging fragments sized 5 or higher, the detours caused by the operation provide higher cost than letting these fragments connect by fragment connection methods. This is illustrated in Figure 6.

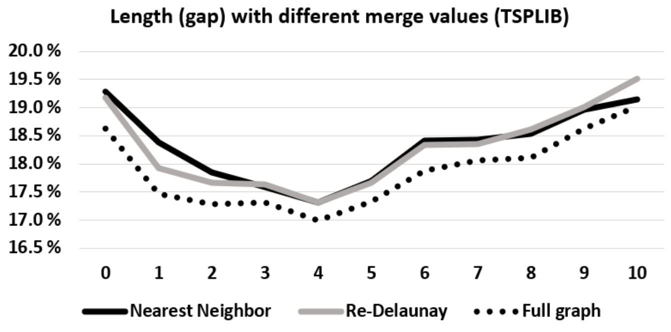

Figure 3. The behavior of fragment merging with different merge values and fragment merging approaches on TSPLIB instances.

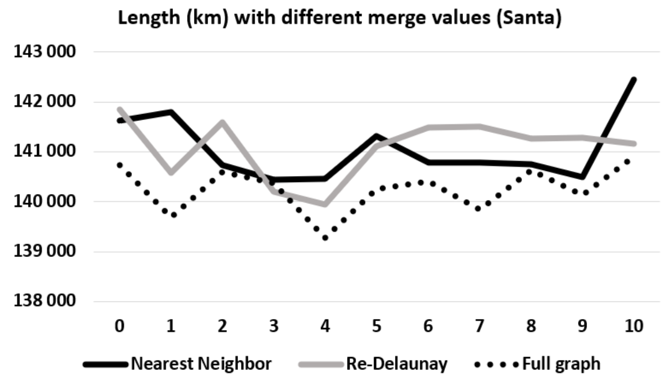

Figure 4. The behavior of fragment merging with different merge values and fragment merging approaches on the Santa dataset.

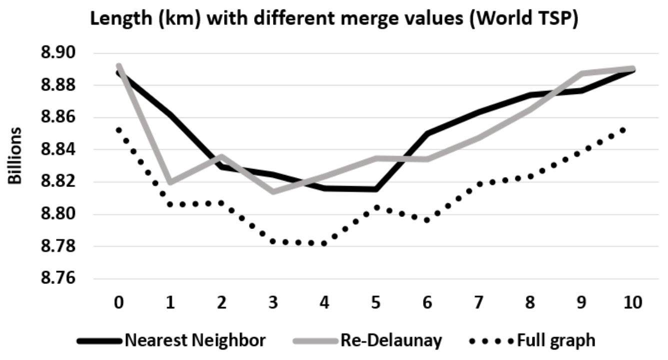

Figure 5. The behavior of fragment merging with different merge values and fragment merging approaches on the World TSP dataset.

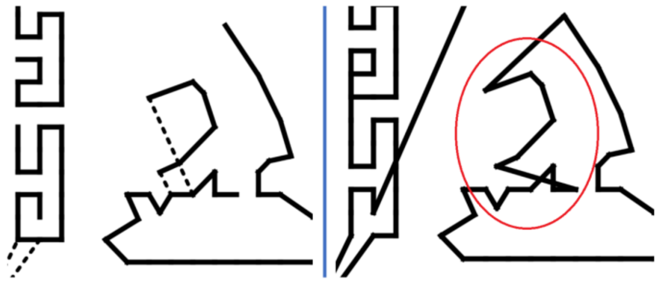
Figure 6. The weakness of merging bigger fragments (TSPLIB instance pcb442). Merging a bigger fragment causes a detour.

IV. Results

The details of each dataset are presented in Table 4. The TSPLIB results include 78 selected instances from the TSPLIB library [20]. This correlates to all instances using Euclidean distance out of the total of 111 instances.

TABLE IV. DATASETS USED IN THIS STUDY.

Dataset	Type	Distance	Instances	Sizes
Dots[1]	Open loop	Euclidean	6,449	5-31
TSPLIB	Closed loop	Euclidean/ haversine	78	51-18,512
Santa[2]	Closed loop	Euclidean	1	1,437,195
World-TSP[3]	Closed loop	GEOM	1	1,904,711

[1] https://cs.uef.fi/o-mopsi/datasets/ [2] https://cs.uef.fi/sipu/santa/
[3] https://www.math.uwaterloo.ca/tsp/world/

The results of all graph-based methods are summarized in Table 5 for Dots [21] and TSPLIB datasets corresponding to small and medium-scale TSP respectively. The results are rather modest and do not meet the requirement of the 1% gap for branch-and-bound but can still be useful for initialization of local search. The results can be roughly divided into four categories: poor (>100% gap), weak (35-45%), modest (12-25%) and good (2-7%). The results of fully randomized approaches such as Walk and Delaunay random edge are poor due to their random nature. Also, the fast variant of NN has some randomization, which causes longer paths.

The LKH package [2] provides many different graphs to use as the candidate set. The results shown in Tables V and VI use the Delaunay graph. LKH provides another graph based on the so-called alpha-nearness criterion, which originated from this package. Computing alpha-nearness values, which form the alpha-nearness graph, is time-consuming O(n^2). However, approximation is possible using subgradient optimization [22]. This process has been shown to eventually converge to the true alpha values, but stopping it early is the key to high efficiency [6]. This optimization can also be applied to the Delaunay graph by setting a small initial period for the ascent. The parameter INITIAL_PERIOD controls this and setting it to 100 provides almost all the optimization. The default value of $n/2$ is too time-consuming, increasing the time of the initialization methods from around 250 ms to approximately six seconds.

TABLE V. SUMMARY OF THE RESULTS FOR DOTS AND TSPLIB DATASETS. SIMPLE NEAREST NEIGHBOR AND LKH, WHICH CONTAINS BOTH INITIALIZATION AND OPTIMIZATION STEPS FOR REFERENCE. THE PERCENTAGES PRESENT THE GAP TO THE OPTIMAL SOLUTION.

Algorithm	**Ref.**	**Average gap**		**Time (ms)**
		Dots	***TSPLIB***	***TSPLIB***
Alpha = 0				
NN	[3, 4]	9.25 %	24.08 %	71
LKH	[2]	0.32 %	0.06 %	5440
Boruvka	[11]	21.74 %	20.01 %	151
Quick-Boruvka	[7, 12]	14.10 %	22.22 %	148
Greedy	[2]	19.72 %	19.69 %	150
NN (fast)	[2]	24.58 %	35.67 %	149
Walk	[2]	44.82 %	140.23 %	148
Delaunay shortest edge	[15]	9.87 %	32.68%	17
Delaunay random edge	[15]	24.14 %	110.97 %	30
Delaunay greedy FM	new	2.18 %	16.99 %	25
Alpha = 100				
LKH	[2]	0.01 %	0.05 %	6370
Boruvka	[11]	6.53 %	12.78 %	257
Quick-Boruvka	[7, 12]	6.69 %	16.17 %	255
Greedy	[2]	6.56 %	12.64 %	259
NN (fast)	[2]	6.74 %	21.29 %	256
Walk	[2]	14.16 %	116.17 %	258

Delaunay greedy FM (FM for fragment merging) achieves the best gap for the Dots dataset even considering the alpha-nearness optimized Delaunay variants of LKH. The alpha = 0 variants reach around a 15% gap and alpha = 100 variants around a 6% gap. Kruskal-TSP algorithm [8] tested in [15] achieved the earlier best gap of 2.66% which is close to the Delaunay greedy FM result, but it is achieved by a slower algorithm. If the Delaunay graph is complimented with a quadrant graph, which corresponds to using the least costly edge in each of the four geometric quadrants [23], the greedy and Boruvka variants achieve 1.78% and 1.94% gaps respectively. A fully utilized alpha-nearness graph [2] reaches below 1% results, but the graph is constructed in O(n^2) time making it not suitable for large-scale TSP and achieving nearly identical results in TSPLIB. Finally, using the LKH package including both initialization and optimization, a 0.01% gap is reached with default parameters.

The TSPLIB results (Table 5) are similar to the Dots results with small differences. The Delaunay greedy FM approach reaches the best gap in the alpha = 0 case at 16.99%. Other Delaunay graph-based approaches are close at around a 20% gap. Setting alpha to 100, these approaches improve to greedy achieving the best 12.64% gap. This shows that the small amount of alpha-nearness optimization has a big impact on the result. Additionally, again complimenting the Delaunay graph with quadrant graph edges, the best graph-based methods achieve an 11.90% gap (Boruvka) and an 11.99% gap (Greedy).

The results of all graph-based methods for Santa [6] and World-TSP are summarized in Table 6 corresponding to large-scale TSP. The optimal solutions for these instances are unknown. The results are therefore given in the length of the solution. Here, the Delaunay greedy FM approach is approximately tied with Boruvka in the alpha = 0 case in both datasets while greedy has the best solutions overall.

Quick-Boruvka provides a longer solution in Santa but is similar to others in World-TSP. The structure of the instance seems to make a difference.

Using alpha = 100 provides better results than alpha = 0 but at the cost of somewhat more time-consuming. Considering the time taken for the optimization step, this is still insignificant, and the overall optimization is expected to benefit from using alpha = 100.

Fragment merging is most effective in the case of smaller instances (Dots, TSPLIB) but not anymore in large-scale instances (Santa, World-TSP).

The new Delaunay greedy FM is fast similar to its predecessors in [15] but with significantly shorter tours. Moreover, it is written in Java in contrast to C used by the competitors. We can therefore expect even further speed-up simply by changing programming language. This is left to future studies.

V. CONCLUSION AND FUTURE WORK

Finding the initial solution for TSP by Greedy approaches can be time-consuming with O(n^2) time complexity. A faster variant using the Delaunay graph was proposed. It provides competitive performance for all datasets including large-scale datasets Santa (1.4M) and World-TSP (1.9M).

Alpha-nearness optimization turned out to be important for all the graph-based approaches. The effect of merging tiny fragments was effective only with small-scale instances but became relatively insignificant with large-scale instances.

As future work, we plan to compare the effect of these initializations technique to the result after the optimization. It was argued in [23] that tour construction heuristics improve the performance of k-opt optimization compared to random initialization. However, it is an open question whether a better initial solution also implies a better final solution after the optimization. This is left as future studies.

Nevertheless, the speed of the initialization matters. This paper has compared several fast graph-based variants, of which greedy heuristics in general seem the most promising when success is measured by the initial tour length and speed.

TABLE VI. SUMMARY OF THE RESULTS FOR SANTA AND WORLD-TSP DATASETS. THE LENGTHS OF WORLD TSP ARE IN MILLIONS OF KILOMETERS.

Algorithm	**Ref.**	**Length (km)**		**Time (s)**	
		Santa	***World TSP***	***Santa***	***World TSP***
Alpha = 0					
NN	[3, 4]	-	-	>2h	>2h
LKH	[2]	109 454	7 547	3600.0	3600.0
Boruvka	[11]	139 471	8 816	23.8	36.9
Quick-Boruvka	[7, 12]	154 409	8 776	21.6	32.5
Greedy	[2]	132 460	8 698	23.9	41.5
NN (fast)	[2]	179 228	10 140	21.9	32.7
Walk	[2]	455 729	42 790	20.9	38.4
Delaunay shortest edge	[15]	150 957	9 944	13.6	25.5
Delaunay random edge	[15]	310 293	15 040	16.9	30.0
Delaunay greedy FM	new	139 270	8 781	16.6	22.2
Alpha = 100					
LKH	[2]	109 360	7 532	3600.0	3600.0
Boruvka	[11]	128 798	8 478	495.2	512.4
Quick-Boruvka	[7, 12]	144 222	8 473	506.0	512.2
Greedy	[2]	127 038	8 416	493.2	505.9
NN (fast)	[2]	158 508	9 453	488.1	502.9
Walk	[2]	468 314	46 043	501.1	502.3

REFERENCES

[1] Applegate. D, Bixby. R.E, Chvátal. V, and William. J.C, “Concorde: a code for solving traveling salesman problems,” 1999, [Online]. Available: http://www.tsp.gatech.edu/concorde.html.

[2] Helsgaun. K, “An effective implementation of the Lin-Kernighan traveling salesman heuristic,” Eur. J. Oper. Res, vol. 126, no. 1, pp. 106–130, 2000, DOI:10.1016/s0377-2217(99)00284-2.

[3] Applegate. D.L, Bixby. R.E, Chavatal. V, and Cook. W.J, “The Travelling Salesman Problem, a Computational Study,” Princeton Univesity Press: Princeton, NJ, USA, 2006.

[4] Kizilates. G, and Nuriyeva. F, “On the nearest neighbor algorithms for the traveling salesman problem,” Advances in Computational Science, Engineering and Information Technology, Springer: Germany, vol. 225, 2013.

[5] Bentley J.L, “Fast Algorithms for Geometric Traveling Salesman Problems,” ORSA Journal on Computing,vol. 4, no. 4, pp. 387-411, 1992.

[6] Mariescu-Istodor R and Fränti P, “Solving large-scale TSP problem in 1 hour: Santa Claus Challenge 2020,” Frontiers in Robotics and AI, vol. 8, pp. 1-20, 2021.

[7] David S. Johnson and Lyle A. McGeoch, “Experimental Analysis of Heuristics for the STSP,” Combinatorial Optimization, vol. 12, 2007.

[8] Fränti, P, and Nenonen, H, “Modifying Kruskal algorithm to solve open loop TSP,” Multidisciplinary International Scheduling Conference (MISTA), Ningbo, China, vol. 12–15, pp. 584–590, 2019.

[9] Moore E.H, “On certain crinkly curves,” Trans. Amer. Math. Soc. N1, pp. 72–90, 1900.

[10] Sierpinski W, and O pewnej krzywej wypetniajacej kwadrat, “Sur une nouvelle courbe continue qui rempllt toute une aire plane,” Bulletin de l'Acad, des Sciences de Cracovie A, pp. 463-478, 1912.

[11] Borůvka. O, and O jistém problému minimálním, “About a certain minimal problem,” Práce Mor. Přírodověd. Spol. V Brně III (in Czech and German), vol. 3, pp. 37–58, 1926.

[12] Applegate. D, Cook. W, and Rohe. A, “Chained Lin-Kernighan for large traveling salesman problems,” INFORMS Journal on Computing, vol. 15, pp. 82–92, 2003.

[13] Christofides. N, “Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem,” Report 388, Graduate School of Industrial Administration, CMU: Pittsburgh, PA, USA, 1976.

[14] Goodrich. M.T, and Tamassia. R, “The Christofides Approximation Algorithm,” In Algorithm Design and Applications, Wiley: Hoboken, NJ, USA, pp. 513–514, 2015.

[15] Tuononen. J, and Fränti, P, “Simple and fast TSP initialization by Delaunay graph,” Int. Conf. on Image, Video Processing and Artificial Intelligence (IVPAI 2023), Shenzhen, China, in Proc. SPIE 13074, 2024.

[16] P. Fränti, T. Nenonen and M. Yuan, “Converting MST to TSP path by branch elimination,” Applied Sciences, vol. 11, no. 177, pp. 1-17, 2021.

[17] Platzman. L.K, and Bartholdi. J.J III, “Space filling curves and the planar traveling salesman problem,” Journal of the Association for Computing Machinery, vol. 36, no. 4, pp. 719–737, 1989.

[18] Gabriel. K.R, and Sokal. R.R, “New statistical approach to geographic variation analysis,” Syst. Zool, vol. 18, pp. 259–278, 1969.

[19] Fränti P, Mariescu-Istodor R, and Zhong C, “XNN graph, Joint Int. Workshop on Structural,” Syntactic, and Statistical Pattern Recognition (S+SSPR 2016), Merida, Mexico, LNCS 10029, pp. 207-217, 2016.

[20] Reinelt. G, INFORMS J. Comput 3, S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, G. T. Rado and H. Suhl, Eds. New York: Academic, vol. 3, pp. 271–350, 1963.

[21] Sengupta. L, Mariescu-Istodor. R, and Fränti. P, “Which local search operator works best for the open-loop TSP,” Appl. Sci, vol. 9, no. 19, pp. 1–24, 2019.

[22] Held. M, and Karp. R, “The Traveling-Salesman Problem and Minimum Spanning Trees: Part II,” Math. Programming 1, pp. 16–25, 1971, DOI:10.1007/bf0158407.

[23] Johnson. D, “Local optimization and the Traveling Salesman Problem,” In Proceedings of the International Colloquium on Automata, Languages, and Programming, ICALP 1990, England, UK, vol. 443, pp. 72–83, July 1990.

[24] Perttunen. J, “On the Significance of the Initial Solution in Travelling Salesman Heuristics,” J. Opt. Res. Soc, vol. 45, no. 10, pp. 1131-1140, 1997.