Indian Institute of Technology, Madras

Department of Computer Science & Engineering

CS6100 Course Project

Rectilinear Steiner Tree Construction

Arihant Samar - CS18B052

Abhishek - CS18B049

Kshitij Deogade - CS17B104

Prashant T - CS17B105

June 14, 2021

Introduction

The Rectilinear Minimum Steiner Tree problem is to find the minimum cost spanning tree of a set of points known as the terminals along with the help of some extra points (called as Steiner points) introduced by intermediate junctions where the distance metric used is the Manhattan Metric.
The problem is known to be NP-Hard and hence various heuristics exist to solve the RMST problem.

We implemented and improved the algorithm given by Zhiliu Zhang [1] which follows the divide and conquer paradigm for solving the RMST problem.
The algorithm consists of sequentially dividing the points into partitions whose size is at most 7 and solving for the exact Steiner tree for these set of points.The trees are then combined and also optimised to reduce the cost of combining the trees.

Algorithm

The `Const_optRST` function finds the optimal rectilinear Steiner tree for atmost 7 points.It does this with the help of 2 functions, `EXTREME` and `FORK`.

The function `EXTREME` checks if there is only point which has the minimum/maximum x/y-coordinate.If that is the case, then it updates the graph on the basis of the lemmas proved in the paper.For example if there is a unique point having the minimum x-coordinate, the graph is updated by moving that point 1 unit to the right.Hence if the point is (x, y) it updates the point to $(x + 1, y)$.

For example, let u be the vertex with the minimum x-coordinate and v is the vertex with the second minimum x-coordinate.In our implementation, we directly change the x-coordinate of u by shifting it to v's x-coordinate.Hence uo is the old vertex and un is the new updated vertex according to the reduction by `EXTREME` and an edge is added between uo and un.

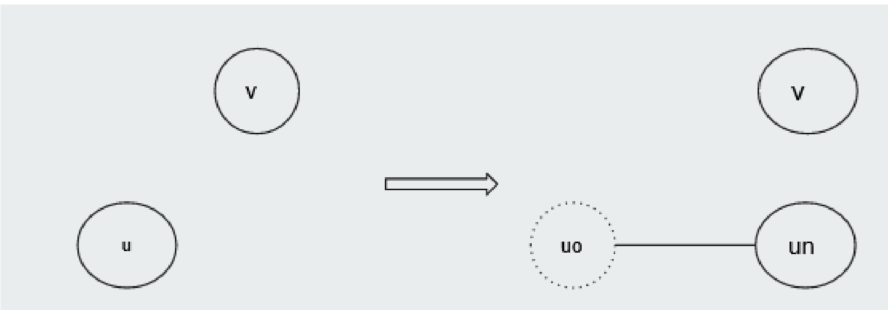

Figure 1: EXTREME function

It does this for all 4 cases, maximum or minimum , x or y co-ordinate.It recursively updates the graph until none of the 4 reductions are possible and returns True .

The `FORK` function works similar to the `EXTREME` function. It performs reduction when there are 2 points which have the minimum/maximum x/y coordinate. It performs 3 different reductions and appends these 3 reduced graphs to the *TreeList*. It is proved in the paper that the optimal *RST* must contain one of these reduced graphs.

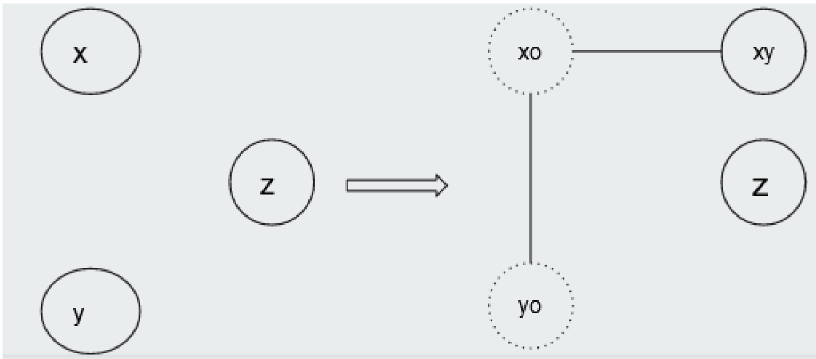

Figure 2: Fork function's Reduction Type 1

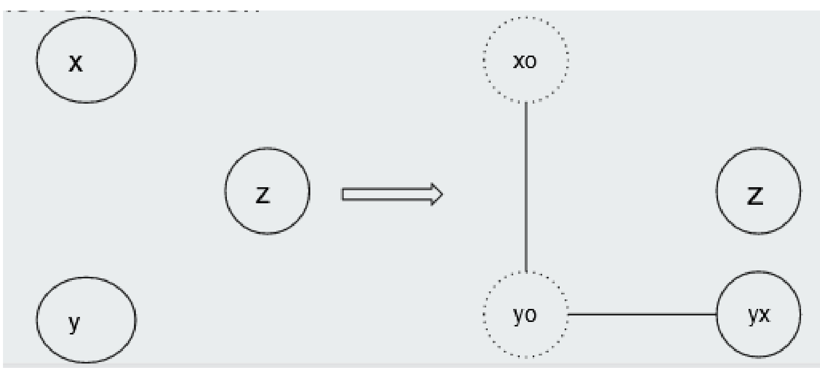

Figure 3: Fork function's Reduction Type 2

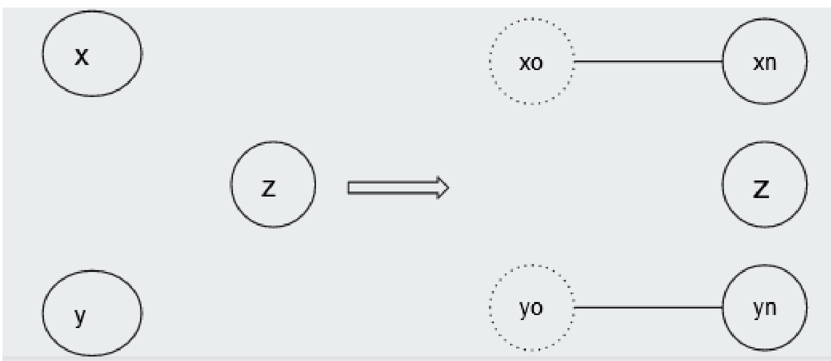

Figure 4: Fork function's Reduction Type 3

In the algorithm `Const optRST`, *TreeList* is a list of graphs. When a graph is constructed as a rectlinear tree, *G.grown* is set to be true. At first, we add the given graph $G(P)$ into the *TreeList*. And then we reduce the graph by *extreme*(G). When *extreme*(G) is finished and returns true, we further reduce the graph with three forked subgraphs by fork(G) which are added into *TreeList* and delete the original graph. When *extreme*(G) returns false, that means the graph has been formed as a subtree. When each subgraph in the *TreeList* has become a subtree, we end the loop return the steiner tree with the minimal length.

```
CONST_OPTRST(G)
  TreeList = [G]
  G.grown = false
  for G ∈ TreeList
      if G.grown == false
          if EXTREME(G) == true
              FORK(G, TreeList)
          else
              G.grown = true
  return min(TreeList)
```

Optimisation

For a subgraph, when two or more subtrees are generated, the RST of the entire subgraph is also constructed, because any two subtrees share a common terminal between them. However, this may bring the extra cost of the total length for merging two subtrees together.

For example in 5 ,point A is the common point of 2 adjacent districts. It connects to C and B which both have a higher y coordinate than A. In the left figure, we can see 2 edges going upwards from A. This leads to extra cost which can be optimised as shown in the right hand side figure.

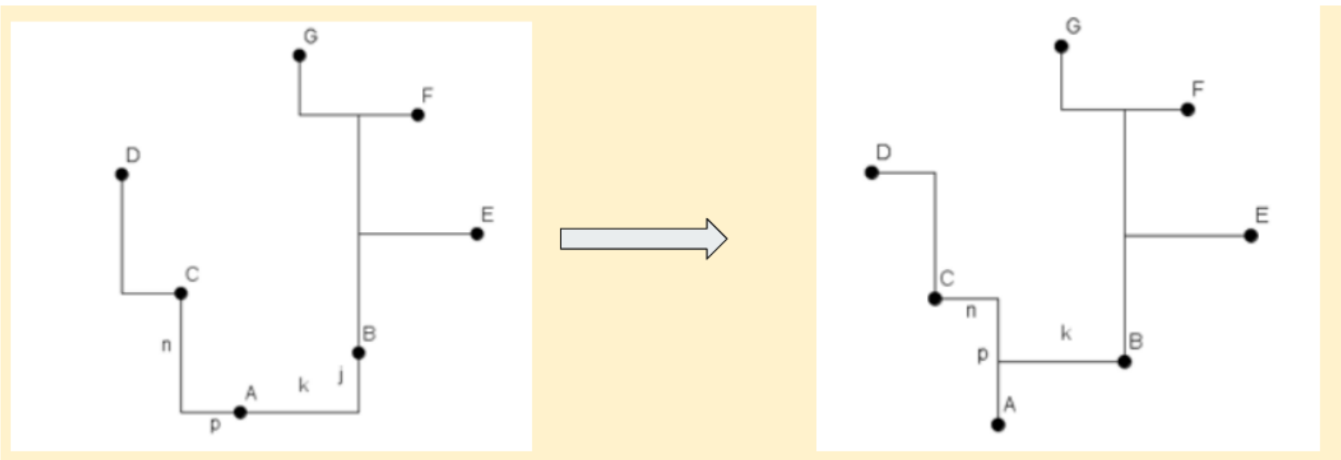

Figure 5: Optimisation of 2 merged Steiner trees

Improvements to the algorithm

1. In the original algorithm , the `EXTREME` function only shifts points by 1 unit in each iteration.Using this way, a point with the minimum x-coordinate will keep shifting to the right till it has the same x-coordinate as the second minimum point.Naturally, we just club all these iterations and directly move the minimum x-coordinate to the second minimum x-coordinate in 1 step.

2. We do a similar optimisation in the fork function where we directly shift to the next minimum coordinate.

3. We store the grid points in an self balancing binary search tree which provides insertion and deletion in $O(\log n)$.Hence the `EXTREME` function which has a recursion depth of at most 4 in our implementation runs in $O(\log n)$ time complexity instead of the $O(n)$ mentioned in the paper.

4. When optimising the combined trees of 2 districts, we first try to combine edges with a common end-point.This means if $(x_1, y) \leftrightarrow (x_2, y)$ and $(x_2, y) \leftrightarrow (x_3, y)$ are edges, then we combine them to a single edge $(x_1, y) \leftrightarrow (x_3, y)$

Implementation

The `Graph` structure stores the points in 2 sets, one sorted by x coordinates (P_x_sorted) and other sorted by y coordinates(P_y_sorted). This is to make it easy to write the conditions in the `if` blocks for `EXTREME` and `FORK` functions.

To check the conditions in `EXTREME` function,for example to check if there is only one point with the minimum x-coordinate, we can just check if the x-coordinate of the first element is equal to the x-coordinate of the second element in P_x_sorted.

Similarly for `FORK` function, to check if there are 2 points with the same minimum x-coordinate, we can check if the x-coordinate of the first element is equal to the x-coordinate of the third element in P_x_sorted. This is because we are already sure that since `EXTREME` has already run before `FORK`, at least 2 points have the same x-coordinate. Hence just checking with the third element suffices.

For the optimisation, we have to find the corresponding terminals attached to the common terminal between two adjacent districts. For example , in 5 we have to find points B and C corresponding to point A. The optimisation can only be done if B and C are on the same side , either both above or below of A. The `optimise` function does just this.

Evaluation

Values of n and sigma range as:- n=[100,600,...,5600] ; sigma=[0,1,3,5,7]; For each n and sigma, a directory *no_of_points_n_sigma_sigma* was made. It had 300 files. Each file had n points. For sigma=0, each of the n points in a file was got via np.random.uniform(0,10000,n).

Thus, n points were sampled from uniform distribution between 0 and 10000. For other sigmas, each of the 300 files was made by taking the n points from *no_of_points_n_sigma_0* , and adding gaussian noise with sigma to them. Thus, perturbations were added.

As coordinates were floating point values, they were rounded off to integers.
For a given '*no_of_points_n_sigma_sigma*' directory, all its 300 files were run, and the time of execution and length of Steiner tree got were averaged.

n \ σ	0	1	3	5	7
100	141239	141282	141314	141290	141323
600	755260	755294	755141	755753	755445
1100	1373811	1374084	1373551	1372972	1373842
1600	1994520	1994994	1995326	1994696	1994962
2100	2611249	2611129	2611453	2610510	2610424
2600	3230808	3231577	3231139	3230119	3230204
3100	3851139	3850956	3851239	3851257	3851284
3600	4472265	4472964	4470970	4469332	4469342
4100	5091286	5090640	5091162	5092547	5089805
4600	5710810	5711178	5712957	5711298	5711081
5100	6333024	6330828	6333463	6330463	6333543
5600	6953293	6954623	6953566	6957019	6955170

Table 1: Average Cost of Steiner Tree for various (n, σ)

n \ σ	0	1	3	5	7
100	40	40	40	40	40
600	237	237	237	238	238
1100	426	424	425	425	424
1600	604	603	600	603	604
2100	775	776	771	791	776
2600	976	971	956	954	955
3100	2100	2098	2096	2090	2099
3600	2254	2254	2255	2244	2253
4100	2400	2404	2391	2407	2402
4600	2550	2549	2548	2538	2550
5100	2690	2678	2689	2695	2692
5600	2846	2834	2839	2846	2839

Table 2: Average Running Time of Steiner Tree for various (n, σ)

n \ σ	1	3	5	7
100	0.000304448	0.000531015	0.00036109	0.000594737
600	0.0000450176	-0.000157562	0.000652755	0.000244949
1100	0.000198717	-0.000189255	-0.00061071	0.000022565
1600	0.000237651	0.000404107	0.0000882418	0.000221607
2100	-0.000045955	0.0000781235	-0.000283006	-0.000315941
2600	0.000238021	0.000102451	-0.000213259	-0.00018695
3100	-0.0000475184	0.0000259663	0.0000306403	0.0000376512
3600	0.000156297	-0.000289562	-0.00065582	-0.000653584
4100	-0.000126883	-0.0000243553	0.000247678	-0.000290889
4600	0.0000644392	0.000375954	0.000085452	0.0000474539
5100	-0.000346754	0.0000693192	-0.000404388	0.0000819514
5600	0.000191276	0.000039262	0.000535861	0.000269944

Table 3: Percentage change of score for (n, σ) from (n,0)

terminals	O(n^2)$Prim$	Scheffer	Guibas-Stolfi	Our Algorithm
5	0.000006	0.000400	0.000053	0.0004
10	0.000024	0.000685	0.000138	0.0053
50	0.000567	0.003601	0.001118	0.0141
100	0.002269	0.007959	0.002613	0.0212
500	0.055323	0.051744	0.017536	0.0922
1000	0.223079	0.118234	0.038819	0.1739
5000	5.774400	0.889230	0.243520	1.0206
10000	23.641100	2.477000	0.531860	2.4647
50000	N/A	22.685750	3.461500	30.7502
100000 (Without Optimise)	N/A	75.654200	9.253000	16.1663
500000 (Without Optimise)	N/A	486.621200	91.634800	83.9285

Table 4: Comparison of our Algorithm run on a ..... with Some standard algorithms run on a 195MHz SGI Origin 2000, excluding I/O and memory allocation averaged

n \ σ	0	1	3	5	7
100	144050	144057	144070	144016	144056
600	768584	768535	768282	768248	768405
1100	1393843	1393841	1393966	1393440	1394709
1600	2020336	2020497	2020692	2020711	2019708
2100	2644595	2644378	2645636	2645163	2642855
2600	3267166	3267113	3267389	3267410	3267655
3100	3894102	3893444	3894055	3892033	3895502
3600	4518930	4517397	4517825	4516146	4519926
4100	5142628	5142717	5144849	5143576	5144924
4600	5767769	5767011	5768583	5769094	5768130
5100	6392926	6393979	6390860	6389534	6389558
5600	7013324	7014368	7010615	7015529	7014708

Table 5: Average Cost of Steiner Tree for various (n, σ) (without optimization)

n \ σ	0	1	3	5	7
100	1.951406	1.926321	1.912959	1.892845	1.897179
600	1.733578	1.722888	1.710440	1.626428	1.686611
1100	1.437178	1.417450	1.464526	1.468883	1.496154
1600	1.277807	1.262214	1.255313	1.287418	1.225227
2100	1.260911	1.257347	1.292052	1.310052	1.227120
2600	1.112830	1.087688	1.123066	1.141302	1.146112
3100	1.103284	1.091270	1.099522	1.047679	1.135104
3600	1.032656	0.983597	1.037114	1.036592	1.119133
4100	0.998361	1.012636	1.043510	0.992092	1.071328
4600	0.987540	0.968145	0.964292	1.001821	0.989038
5100	0.937004	0.987664	0.898111	0.924496	0.876665
5600	0.855956	0.851752	0.813752	0.834007	0.848759

Table 6: Percentage decrease in cost (via optimization) for various (n, σ)

Observations

- We can observe there is not a lot of change in the cost of the tree and the runtime of the algorithm adding Gaussian Noise. Hence we can conclude that the algorithm is stable.
- This may be due to the fact that the points are divided into small districts. Hence runtime of the algorithm just depends on the number of districts which is the same after adding the noise as well.
- The optimise function is currently the bottleneck of the algorithm as we can see that with optimise the algorithm takes 30 seconds on 50000 terminals whereas it takes only 16 seconds on 100000 terminals when optimise function is not called.
- The absolute difference benefit from using the optimise function increases with increasing n. This is natural since there will be more districts with larger n and hence more pair of adjacent districts can be optimised.
- On the other hand, the percentage benefit decreases with increasing n. This may be because the cost of the Steiner tree itself increases very fast with n.

References

[1] Zhiliu Zhang. Rectilinear steiner tree construction. 2016.