1 Brief Outline and summary

Summary of the project is presented here for quick reference.

- ⋆ **Problem Statement**: Find minimum wirelength routing using Steiner trees
- ⋆ **Steiner tree algorithm**: Prim's algorithm followed by Edge replacement heuristic
- ⋆ **Improvement over Minimum Spanning Tree**: around 4-5%

Edge replacement heuristic is one of the state of art heuristics giving the best improvements. More details will be explained in the following sections.

2 Introduction

The Euclidean Steiner tree problem is NP Hard problem and also has a place in Karp's 21 NP Hard problems. Rectilinear Steiner tree problem is also proven to be NP hard. Therefore we use a heuristic approach for solving the problem.
The first thing that comes to mind is a minimum spanning tree in which no additional points are introduced. This gives us a good enough solution to the problem. But definitely we can do better.
One non-trivial approach by *Kahng* is dividing the space into directed Dirichlet partitions. Then we can insert Steiner points in the regions generated by refining the partitions. The algorithm has $O(n^2)$ time complexity.
Another approach by *Borah and others* builds a Minimum Spanning Tree. New edges are then introduced in the tree and old and long edges are removed. This algorithm also has $O(n^2)$ time complexity.
Although the first approach has $O(n^2)$ time complexity, the time constants involved are large and the algorithm does not run fast enough(compared to second). The second approach uses relatively simple data structures and is relatively easier to implement. The accuracies of both the algorithms are almost the same. So we choose the second algorithm for our problem.

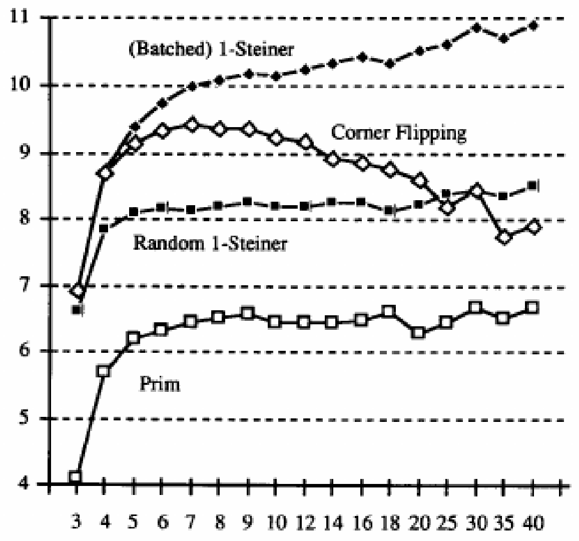


Fig. 10. Performance comparison of the heuristics; the horizontal axis represents the number of points per set, while the vertical axis represents percentage cost improvement over MST.

Figure 2.1 Comparison of performance of Prim's and 1-Steiner algorithms. Image taken from reference 1.

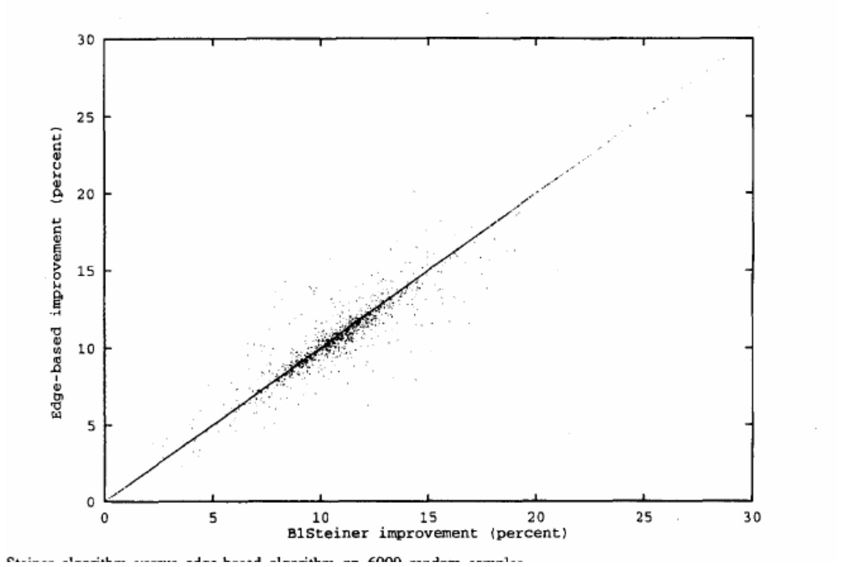


Figure 2.2 Comparison of performance of Batched 1-Steiner and Edge Replacement algorithm. Image taken from reference 3.

3 Algorithm

3.1 Data Structures and pseudo code

Point: This basically contains a 2-tuple of x and y co-ordinates and some extra parameters for book keeping.

Edge: This is a simple but powerful data structure we use throughout the implementation. We basically have a 3-tuple of points. Point 1, point 2 represent the endpoints of the edge. via Point represents the point at the bend that the line from point 1 to point 2 goes *via*. If points 1 and 2 are in a straight line and there is no bend viaPoint is taken as `None`. This 3-tuple along with some extra parameters forms our Edge class.

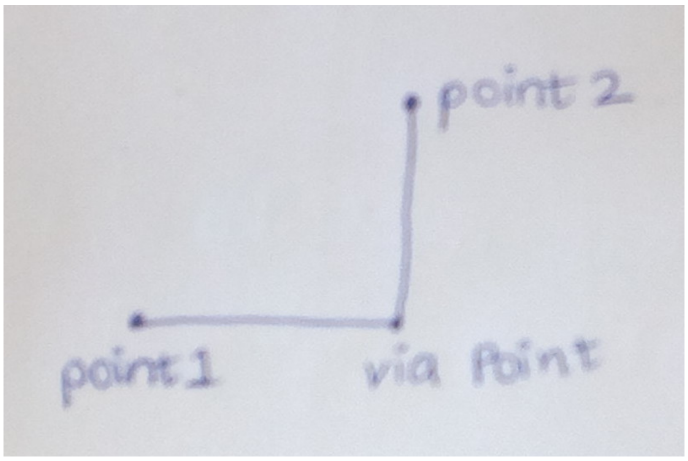


Figure 3.1 Edge data structure

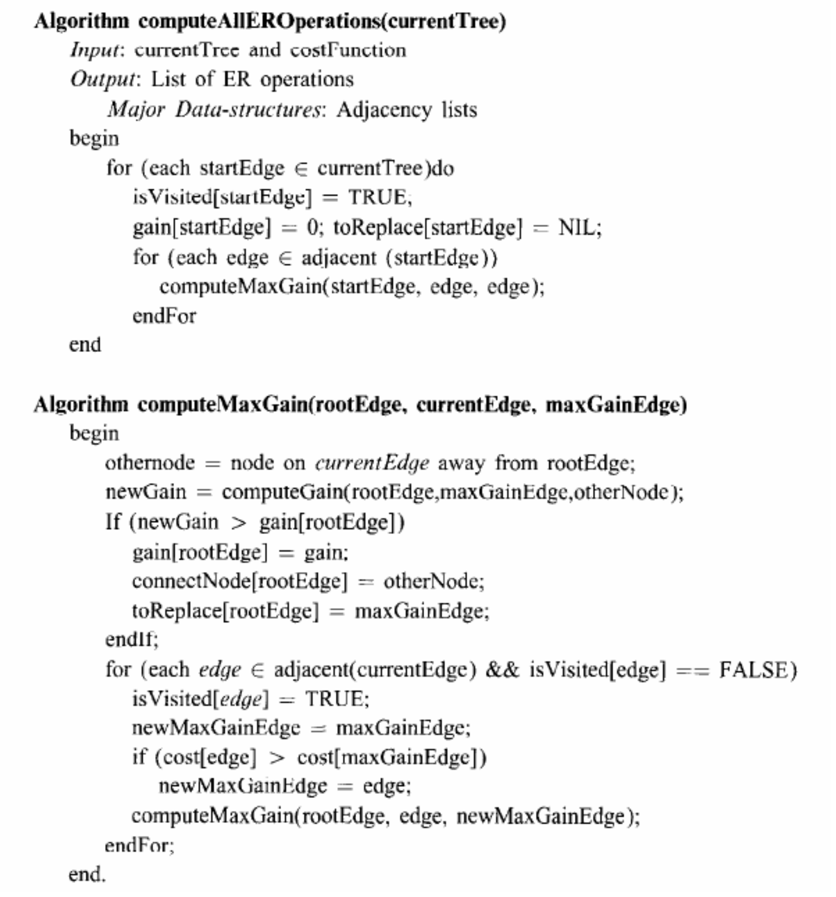
Heap: This was used for the Prim's algorithm of minimum spanning tree.

Adjacency List for edges: This was used for making the tree using edge replacements.

Tracks Matrix: This was used for detecting and estimating gain in case of overlapping edges, and edge collisions while forming a new edge in the Edge Replacement algorithm.

3.2 Implementation details

We add an additional edge(to the already formed minimum spanning tree) from a point in the graph, to an edge. This causes the formation of a loop. We now remove



```
Algorithm computeAllEROperations(currentTree)
    Input: currentTree and costFunction
    Output: List of ER operations
        Major Data-structures: Adjacency lists
    begin
        for (each startEdge ∈ currentTree)do
            isVisited[startEdge] = TRUE,
            gain[startEdge] = 0; toReplace[startEdge] = NIL;
            for (each edge ∈ adjacent (startEdge))
                computeMaxGain(startEdge, edge, edge);
            endFor
    end

Algorithm computeMaxGain(rootEdge, currentEdge, maxGainEdge)
    begin
        othernode = node on currentEdge away from rootEdge;
        newGain = computeGain(rootEdge,maxGainEdge,otherNode);
        If (newGain > gain[rootEdge])
            gain[rootEdge] = gain;
            connectNode[rootEdge] = otherNode;
            toReplace[rootEdge] = maxGainEdge;
        endIf;
        for (each edge ∈ adjacent(currentEdge) && isVisited[edge] == FALSE)
            isVisited[edge] = TRUE;
            newMaxGainEdge = maxGainEdge;
            if (cost[edge] > cost[maxGainEdge])
                newMaxGainEdge = edge;
            computeMaxGain(rootEdge, edge, newMaxGainEdge);
        endFor;
    end.
```

Figure 3.2 Algorithm used for edge replacement. Image taken from reference 3.

an edge from the loop which causes the maximum gain(in decrease of length of the tree). The gain is given by $gain = length(e_1) - length(e_2)$ if e_2 has been added and e_1 has been removed. We do this for multiple edges($e_1, e_2, \cdots$) and points as long as the operation for edge e_i do not interfere with operations for edge e_j. Thus generally we do multiple edge replacements only if the loops formed by them have no effect on each other.

We ensure that root edge which has already participated in the iteration for edge e_i does not get removed in the iteration for edge e_j. We also ensure that the edge that we are going to remove in iteration i has not already been removed in iteration j. There are many details to this algorithm due to the rectilinear nature of the graph. If we have 2 points which are not collinear, there are 2 possible ways to construct a track. What if one of the track is not feasible because it intersects the already formed minimum spanning tree? What is both the feasible tracks intersect the minimum spanning tree? If there are overlapping edges in our loop formed(in edge

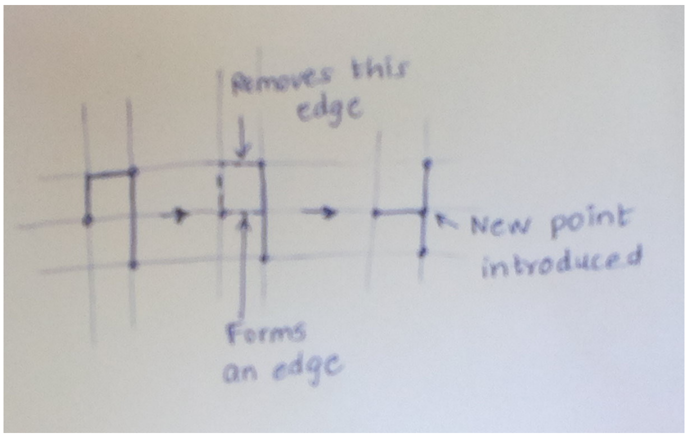


Figure 3.3 Implementation of this algorithm for a small tree

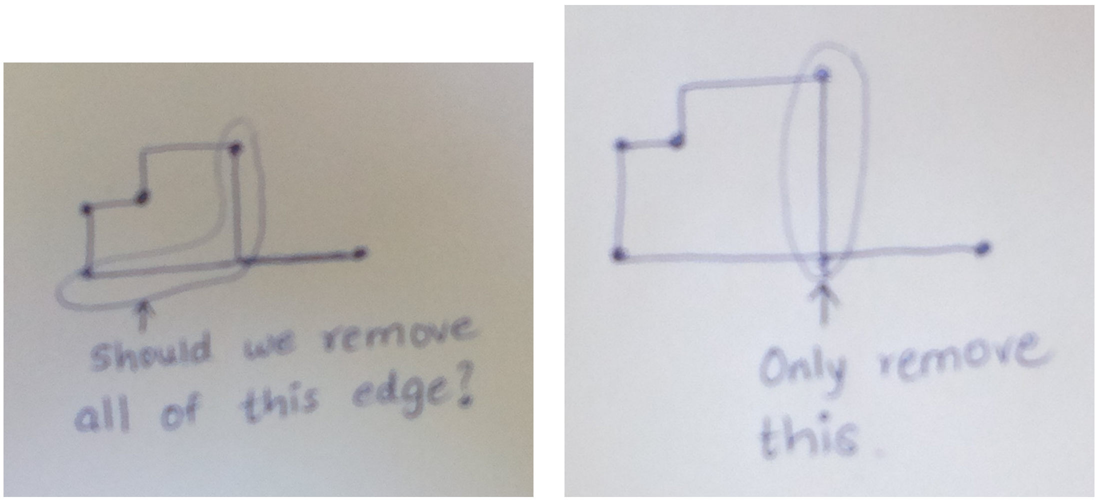


Figure 3.4 Should we remove all of this edge?

replacement stage), should we remove the entire of the maximum cost edge? No. This will cause some edges to remain dangling. We remove only the part of the maximum cost edge which does not overlap.

3.3 Input Output Details

Inputs: We have an option of choosing the locations of the circuit elements from a text file. But for large graphs writing such text files by hand is difficult. Therefore there is an option to randomly place circuit elements in a grid of some size specified by the user. The density of circuit elements in the grid can also be specified. We can either use

```
nodes=CreateNodes(takeInputs("placement2.txt"))
```

when you want to use the placement of circuit elements specified by the file *placement2.txt* of use

```
nodes=RandomCreateNodes(20,20,0.2)
```

to generate a 20×20 grid and put nodes(elements) in that grid with a density of 0.2. The input for the second figure in results is shown below as a sample.

```
DESIGN

COMPONENTS 29 ;
        - u1 NOR2_X1
                + PLACED ( 6 0 ) ;
        - u2 NOR2_X2
                + PLACED ( 7 0 ) ;
        - u3 NOR2_X3
                + PLACED ( 8 0 ) ;
        - u4 NOR2_X4
                + PLACED ( 9 0 ) ;
        - u5 NOR2_X5
                + PLACED ( 0 1 ) ;
        - u6 NOR2_X6
                + PLACED ( 2 1 ) ;
        - u7 NOR2_X7
                + PLACED ( 4 1 ) ;
        - u8 NOR2_X8
                + PLACED ( 5 1 ) ;
        - u9 NOR2_X9
                + PLACED ( 6 1 ) ;
        - u10 NOR2_X10
                + PLACED ( 3 2 ) ;
        - u11 NOR2_X11
                + PLACED ( 9 2 ) ;
        - u12 NOR2_X12
                + PLACED ( 1 3 ) ;
        - u13 NOR2_X13
                + PLACED ( 4 3 ) ;
        - u14 NOR2_X14
                + PLACED ( 2 4 ) ;
        - u15 NOR2_X15
```

```
            + PLACED ( 3 4 ) ;
     - u16 NOR2_X16
            + PLACED ( 8 4 ) ;
     - u17 NOR2_X17
            + PLACED ( 4 5 ) ;
     - u18 NOR2_X18
            + PLACED ( 6 5 ) ;
     - u19 NOR2_X19
            + PLACED ( 0 6 ) ;
     - u20 NOR2_X20
            + PLACED ( 3 6 ) ;
     - u22 NOR2_X22
            + PLACED ( 8 7 ) ;
     - u23 NOR2_X23
            + PLACED ( 4 8 ) ;
     - u24 NOR2_X24
            + PLACED ( 6 8 ) ;
     - u25 NOR2_X25
            + PLACED ( 7 8 ) ;
     - u26 NOR2_X26
            + PLACED ( 9 8 ) ;
     - u27 NOR2_X27
            + PLACED ( 0 9 ) ;
     - u28 NOR2_X28
            + PLACED ( 5 9 ) ;
     - u29 NOR2_X29
            + PLACED ( 8 9 ) ;
END COMPONENTS
```

Outputs: The outputs can easily be seen on the GUI window which gets displayed. One window displays the tree at the first stage i.e, when we create the minimum spanning tree. The other window displays the final Steiner tree after Edge replacement algorithm has been executed.

4 Some Results

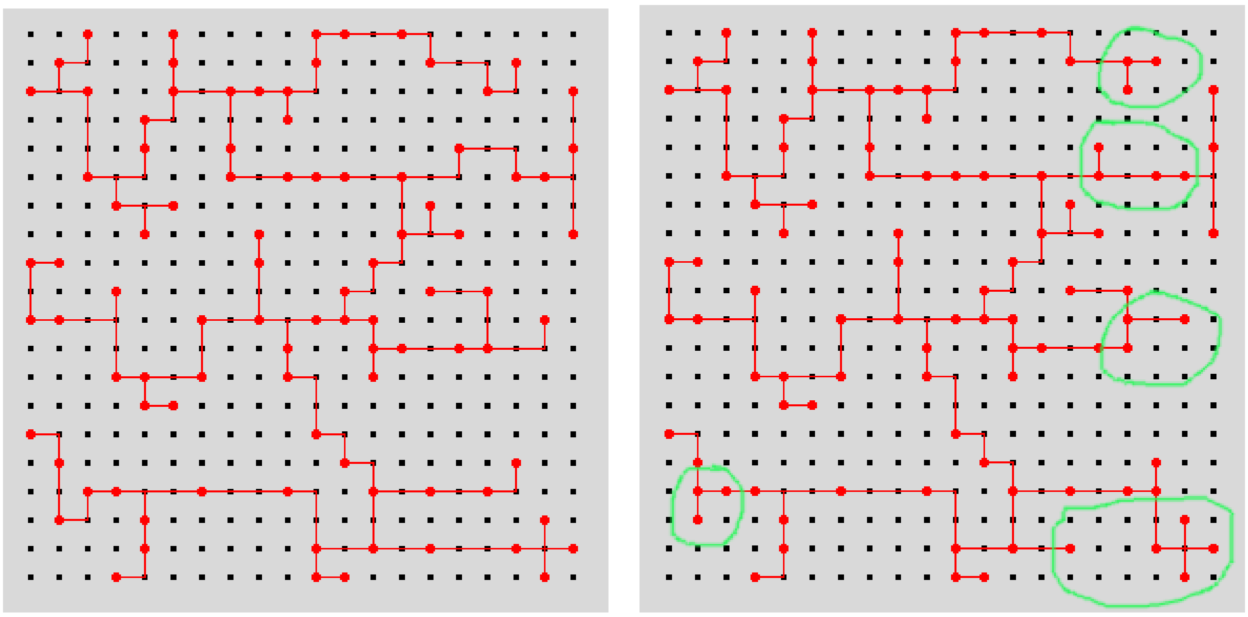


Figure 4.1 Left to Right: Minimum spanning tree construction by Prim's algorithm; Steiner Tree using Edge Replacement Heuristic. Spot the difference ;). Length reduction was 5 units.

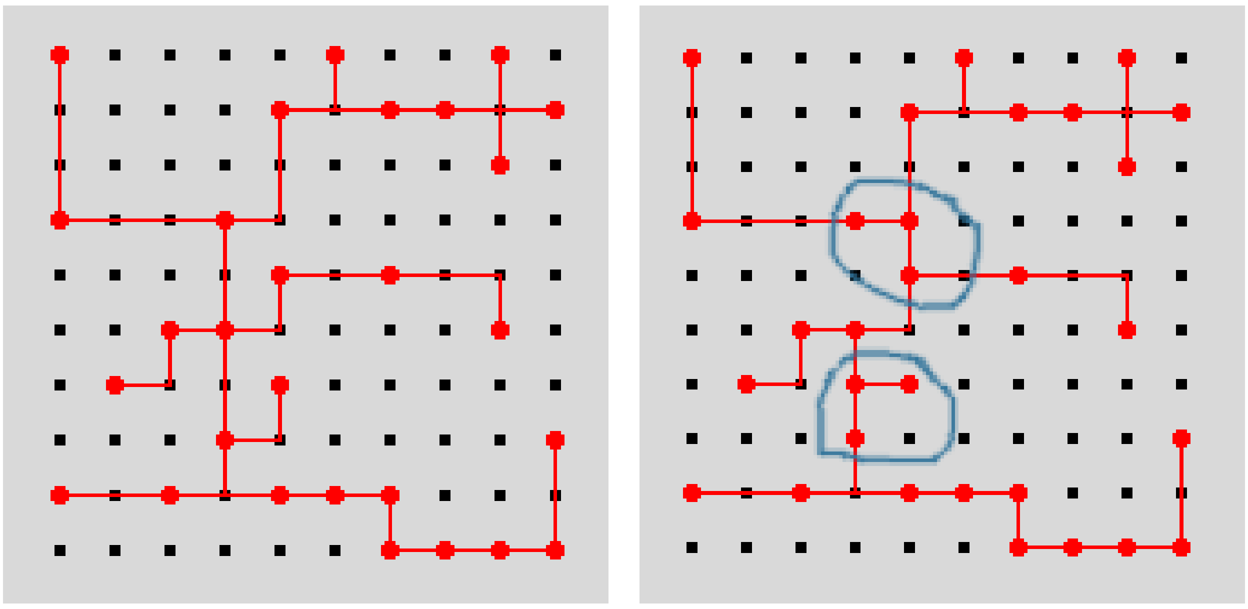


Figure 4.2 Left to Right: Minimum spanning tree construction by Prim's algorithm; Steiner Tree using Edge Replacement Heuristic. Length reduction was 2 units.

For the first case input was random points. For the second case inputs were taken from a text file. The black points represent our grid and the red points represent the points(nodes) in the tree.
Since we cannot find what is the actual best routing for huge diagrams, the efficiency of any Steiner tree algorithm is stated as an improvement over the minimum spanning tree. The efficiency of our implementation is around $4-5\%$. We only did one iteration of the algorithm which explains the slightly less(less by $1-2\%$) improvements compared to the results in the paper.

5 References

⋆ *A New Class of Iterative Steiner Tree Heuristics with Good Performance: IEEE Transactions on Computer-Aided Design, VOL. 11, NO. I, JULY 1992*, Andrew B. Kahng

⋆ *The 1-Steiner Tree Problem: Journal of Algorithms*, George Georgakopoulos and Christos H. Papadimitriou

⋆ *A fast and simple Steiner routing heuristic: Discrete Applied Mathematics 90 (1999)*, Manjit Borah, Robert Michael Owens , Mary Jane Irwin