Constructing Minimal Spanning/Steiner Trees with Bounded Path Length*

Iksoo Pyo
JF1-61
Intel Corportation
Hillsboro, OR 97124

Jaewon Oh and Massoud Pedram
Department of EE-Systems
University of Southern California
Los Angeles, CA 90089

Abstract

This paper presents an exact algorithm and two heuristics for solving the Bounded path length Minimal Spanning Tree (BMST) problem. The exact algorithm which is based on iterative negative-sum-exchange(s) has polynomial space complexity and is hence more practical than the method presented by Gabow. The first heuristic method (BKRUS) is based on the classical Kruskal MST construction. For any given value of parameter ϵ, the algorithm constructs a routing tree with the longest interconnection path length at most $(1+\epsilon)\cdot R$, and empirically with cost at most 1.19 times $cost(BMST^*)$ where R is the length of the direct path from the source to the farthest sink and $BMST^*$ is the optimal bounded path length MST. The second heuristic combines BKRUS and negative-sum-exchange(s) of depth 2 to improve results. Extensions of these techniques to the bounded path length Minimal Steiner Trees, using the Elmore delay model are presented as well. Empirical results demonstrate the effectiveness of these algorithms on a large benchmark set.

1 Introduction

In the design of high-performance VLSI systems, circuit speed and power consumption are important considerations. Routing optimization plays an important role in achieving optimal circuit speed and minimal power consumption. Indeed, critical path delay is a function of maximum interconnection path length while power consumption is a function of the total interconnection length.

A linear RC model (where interconnection delay between a source and a sink is proportional to the wire length between the two terminals) is often used as a simple approximation for interconnection delay. First, we also use wire length to approximate interconnection delay during the construction of routing trees. Later, we extend this delay model to a more accurate RC delay model.

A routing tree used in a synchronous system has an input, called the driver or source, that sends signals to each sink. Critical path delay is defined as the maximum delay from the source to any sink. The critical path delay of the Shortest Path Tree (SPT) is minimum(In a SPT, each sink is connected to the source by the shortest possible path.), but SPT has excessive routing cost and power dissipation as the power consumed by the driver has a linear relation with the routing capacitance. Minimal Spanning Tree (MST) has minimal routing cost, but may contain very long source-to-sink paths which degrade the performance. Alpert et al. [8] showed how to trade the average source-to-sink path length for lower total routing cost by using a linear combining cost function consisting of the source-to-sink path length and the weight of the edge to be added during the tree construction.

In this paper, we present algorithms for constructing a Bounded path length Minimal Spanning Tree (BMST). The routing tree achieves bounded path length, that is, the length of the path from the source to each terminal is bounded. Such a bounded path length tree provides a good initial topology for designers to adjust for minimizing critical paths using a more accurate RC delay model. Also, the tree has small routing cost which is important from area and power consumption viewpoints.

Let R be the length of direct path from the source to the farthest sink and ϵ be a non-negative user-specified parameter. Our method constructs a spanning tree with radius at most $(1+\epsilon)\cdot R$ by using an analogue of the classical Kruskal MST construction [1]. We will show the same method can be extened to Elmore delay model and to bounded path length Steiner tree. Furthermore, the tree cost is empirically observed to be at most 1.19 of that of an optimal BMST.

We next describe an exact algorithm due to Gabow [5] which produces an optimal BMST with exponential time and space complexity. Then, we propose a new exact algorithm which requires exponential time but has polynomial space complexity. This method constructs an optimal tree by negative-sum-exchange(s) on an initial feasible solution. We also propose another heuristic which resolves the complexity problems of the exact algorithm and produces better average results than the Kruskal based method.

2 Background

On a Mahattan (L_1 metric) or an Euclidean (L_2 metric) plane, let $G=(V,E)$ ($|V|=N$) be a network where V is a set of randomly distributed terminal pins called *sinks* with a distinguished pin called the *source(s)*, and E is the set of edges connecting V. BMST seeks to connect all nodes of V in G by a set of edges in E of minimal total length with a bounded path length from the source to any sink. This problem is known to be NP-complete [7]. We propose a novel algorithm - that is, Bounded path length Kruskal (BKRUS) - for solving this problem heuristically. A tree generated by our BKRUS method is called a Bounded path

*This research was funded in part by SRC under contract No. 94-DJ-559 and by NSF NYI under contract No. M/P-9457392

ED&TC '96
0-89791-821/96 $5.00 © 1996 IEEE

length Kruskal minimal spanning Tree (BKT).

Cong et al. [2] proposed two heuristics for solving the BMST problem. In the first method of Cong et al., i.e. the Bounded Prim (BPRIM) algorithm, even though the empirical results are promising, the worst-case performance ratio is unbounded where performance ratio is defined as cost(BPRIM)/cost(MST) (see Table 2 and Figure 5). In the second method of Cong et al., i.e. the Bounded Radius, Bounded Cost (BRBC) algorithm, the worst-case performance ratio is bounded. However, BRBC method uses minimum path (shortest path) from the source to sink whenever the source-to-sink path length violates the length bound $\epsilon \cdot cost(S, sink)$ during the depth first tree traversal. Hence, it may introduce unnecessary routing cost. Their benchmark results [2] show about a worst-case performance ratio of 2.66 and an average performance ratio of 1.57.

3 A Heuristic: BKRUS

Before describing our approach, we give some definitions. The sum of all edge weights of T is the cost of the tree, $cost(T)$. The shortest path distance between u and v in graph G is $dist_G(u, v)$. The shortest path distance between u and v in tree T is $dist_T(u, v)$. The radius of node $v \in G$ is $radius_G(v)$ (i.e. $max\{dist_G(v, u)\}$, $\forall\ u \in V$). Similarly, the radius of node $v \in T$ is $radius_T(v)$ (i.e. $max\{dist_T(v, u)\}$, $\forall\ u \in V$). The partial tree which contains node v is represented by t_v. S denotes the source.

BKRUS algorithm solves the BMST problem by solving the following problem:

> Given the routing graph $G(V, E)$ in L_1 or L_2 space, find a minimal cost routing tree BKT with $radius_{BKT}(S) \leq (1 + \epsilon) \cdot R$.

The classical Kruskal algorithm adds an edge (u, v) in G to MST, or equivalently, merges two partial trees t_u and t_v by the edge (u, v) if:

(1) (u, v) is the least weight edge among the available edges and

(2) $t_u \neq t_v$.

For (1), all the edges are sorted in nondecreasing order. For (2), a disjoint set on V is implemented. Three operations on the set are *MAKE_SET*, *FIND_SET* and *UNION*, the meanings of which are self-explanatory. Merging two partial trees is done by the *UNION* operation followed by the Merge routine to be discussed later, while condition (2) is easily tested by the *FIND_SET* operation. BKRUS algorithm adds one more condition as follows:

(3) the merged tree satisfies the path length bound $(1 + \epsilon) \cdot R$ from the source to the farthest sink.

Let t_M be the merged tree, i.e., $t_M = t_u \cup t_v \cup (u, v)$. Two cases are possible:

(3-a) If t_u contains the source, then the following condition should be satisfied:

$$dist_{t_u}(S, u) + dist_G(u, v) + radius_{t_v}(v) \leq (1 + \epsilon) \cdot R$$

Since nodes in t_u already satisfy the upper bound constraint, this condition ensures that nodes in t_v will also satisfy the upper bound constraint after the merge. The case where t_v contains the source is similar.

(3-b) If neither t_u nor t_v contains the source, then there must be a node $x \in t_M$ such that:

$$dist_G(S, x) + radius_{t_M}(x) \leq (1 + \epsilon) \cdot R$$

This condition ensures that all the nodes in the merged tree t_M can be connected to the source without violating the upper bound path length constraint by having at least a direct path from the source to node x. That is, the existence of such node x guarantees that all nodes in t_M can satisfy the upper bound constraint. If no such node exists in t_M, then (u, v) should be rejected as there is no way to satisfy the upperbound constraint for all the nodes in t_M. We can now give two important definitions.

Definition 3.1 *A feasible node: If there exists a node x in t_M such that $dist_G(S,x) + radius_{t_M}(x) \leq (1 + \epsilon) \cdot R$, then node x is a feasible node in t_M.*

Definition 3.2 *A feasible edge: If edge (u, v) satisfies conditions (2) and (3), then it is a feasible edge.*

Feasible edges can be safely added to the spanning tree under construction.

BKRUS maintains the radius of each node in the partial tree it belongs to, and the path lengths between every pair of nodes within the partial tree they belong to. Let the array D[V,V] contain information about the Manhattan or the Euclidean distances between every pair of nodes, i.e. $D[x, y] = dist_G(x, y)$. Let the array P[V,V] be the path length between every pair of nodes in the routing tree, i.e. $P[x, y] = dist_T(x, y)$. Also let the vector r[V] be the radii of nodes in the tree they belong to. Initially, the array P and the vector r are initialized to zero. As the tree grows, P and r are updated by the Merge routine given below:

```
// Merge two subtrees t_u and t_v by edge (u, v)
Algorithm Merge(u, v)
1 for each x ∈ t_u and y ∈ t_v do
2     P[x, y] = P[y, x] = P[x, u] + D[u, v] + P[v, y]
3 end for
4 for each x ∈ t_u do
5     r[x] = max(r[x], P[x, i], ∀ i ∈ t_v)
6 end for
7 for each y ∈ t_v do
8     r[y] = max(r[y], P[i, y], ∀ i ∈ t_u)
9 end for
```

Figure 1 shows an example of how Merge routine works. The two partial trees are merged by the edge (c, e). The lefthand side tree is t_c and the righthand side tree is t_e. Before the merging takes place, all of the non-zero elements (except the diagonal elements) in matrix P and the vector r were computed from previous mergings. Note that elements of r are the maximum of each row of P. The Merge routine leaves those non-zero elements unchanged and updates $P[x, y]$ only when x and y are in different partial trees. For example, $P[a, f]$ can be computed by $P[a, f] = P[a, c] + D[c, e] + P[e, f]$. Once the P matrix is updated by line 1-3 in the algorithm, new radius $r[x]$ can be found by taking the maximum among the old radius (old $r[x]$) and the $P[x, y]$s for all $y \in t_v$. For example, new $r[a]$ can be found by taking the maximum among {old $r[a]$, $P[a, e]$, $P[a, f]$}, which is {9,

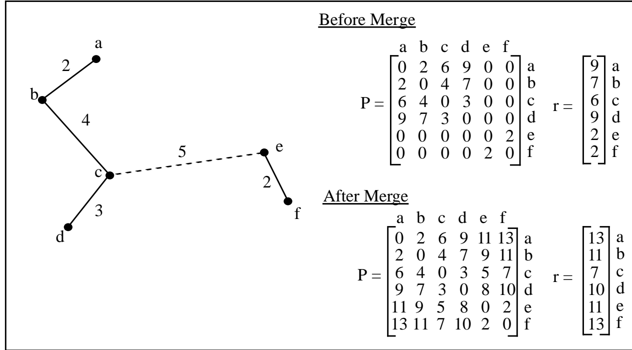

Figure 1: Example of Merging Two Partial Trees

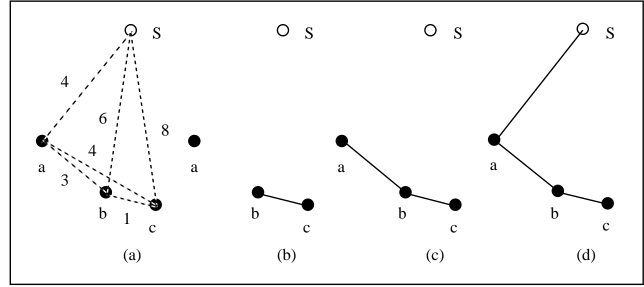

Figure 2: BKRUS Example

11, 13}. So the new r[a] is 13. We can easily see that the time complexity of Merge is $O(V^2)$.

Since we need a new radius of a node x in the merged tree to test the feasibility of x, it seems that a merging is needed before the feasibility test is performed. However, we can find the new radius of any node without an actual merging. Using the same notation as before, suppose x belongs to t_u. Then it can be easily seen that

new radius of $x = max$ $\{$r[x], P[x, u] + D[u, v] + r[v]$\}$

where r and P values are read from the arrays before the merge. The case where x belongs to t_v is similar. With this, feasibility test for a node can be done in $O(1)$. So the condition (3-b) can be tested in $O(V)$. We also note that condition (3-a) can be tested in $O(1)$. The complete BKRUS algorithm is summarized in the following:

Algorithm BKRUS(G)
1 **for** each vertex $x \in$ V **do**
2 MAKE_SET(x)
3 r[x] = 0
4 **end for**
5 **for** every pair of vertices $x, y \in$ V **do**
6 P[x, y] = 0
7 **end for**
8 sort the edge set E in nondecreasing order of weights
9 **for** each edge (u, v) in the sorted edge list **do**
10 **if** FIND_SET(u) $\neq$ FIND_SET(v) **then**
11 **if** either condition (3-a) or (3-b) is satisfied **then**
12 UNION(u, v)
13 Merge(u, v)
14 output the edge (u, v)
15 **end if**
16 **end if**
17 **end for**

The dominating complexity of BKRUS is on line 11 and 13. Since line 11 is executed E times and line 13 is executed $V - 1$ times, the complexity of BKRUS is bounded by $O(EV + V \cdot V^2) = O(V^3)$.

Here, we explain BKRUS algorithm with a simple example. Suppose we have a source and three sinks as shown in (a) of Figure 2. If the upper bound path length is set to $(1 + \epsilon) \cdot R = 8$, BKRUS works in the order of (b), (c), and (d) and produces a BKT with total cost 8 which is optimal. The selected lightest edge b-c of (a) satisfies above three conditions since b is the feasible node. So the edge b-c is a feasible edge. The next lightest edge a-b of (b) satisfies above three conditions since a is the feasible node. Finally, edge S-a is included since S is the feasible node.

3.1 Extension of BKRUS to use the Elmore Delay Model

The BKRUS algorithm can be extended to the Elmore delay model so that the path length from the source to any sink is replaced with the signal propagation delay. To ensure the existence of a solution, the source should be able to supply very large amount of current, i.e. it must have a very small driver resistance so that SPT can be a solution. R is set to the longest S-sink delays of SPT. For two nodes u and v, the delay from u to v is not simply proportional to the path length between u and v, but also dependent on the tree topology. So the method in BKRUS for computing the radius of a node does not work. The new radii r in BKRUS algorithm must be completely recomputed after temporarily merging the two subtrees. The radius r[v] of node v is the longest delay from v to any sink in the merged tree rooted at v, and can be found in linear time. At the same time, the total capacitance, which is sum of the wire and load capacitances in the merged tree (denoted by $C(merged_tree)$) is also computed. The feasibility tests (3-a) and (3-b) are then restated as:

(3-a)$'$ r[source] $\leq (1 + \epsilon) \cdot R$ in the merged tree

(3-b)$'$ there exists a node x in the merged tree such that $R_s \cdot dist_G(S, x) \cdot (C_s \cdot dist_G(S, x)/2 + C(merged_tree))$ + r[x] $< (1 + \epsilon) \cdot R$, where R_s and C_s denote the sheet resistance and capacitance of the wire, respectively.

(3-a)$'$ takes $O(V)$ while (3-b)$'$ takes $O(V^2)$. As a result of these modifications to BKRUS, the feasibility test dominates the total complexity of the algorithm, whose complexity becomes $O(EV^2)$.

3.2 Constructing Bounded Path Length Steiner Trees: BKST

Bounded Path Length Steiner Trees can be constructed on a channel intersection graph or on a Hanan's grid graph [9] using a modified BKRUS. A spanning tree that spans all the sinks and the source on these routing graphs becomes a Steiner tree. We call this variation of Bounded Kruskal method a Bounded Kruskal STeiner (BKST). Initially, the distances between every pair of sinks on the routing graph are computed and stored in a heap. These distances are analogous to the edge weights in BKRUS. Then we extract the smallest distance from the heap and check its feasibilty. If it is feasible, the path in the routing graph that achieves

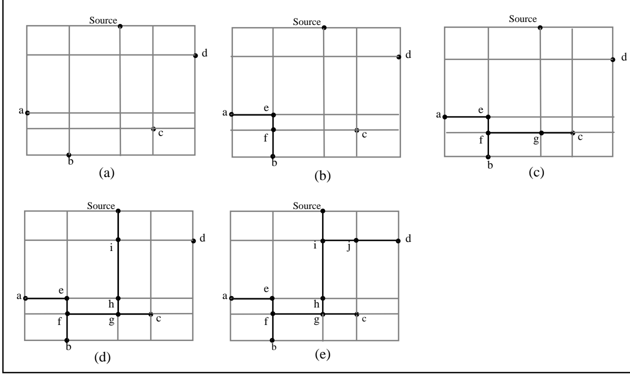

Figure 3: Example of Steiner BKRUS (BKST)

this distance is found and added to the Steiner tree under construction. If there are multiple such paths, we choose only L-shaped paths (no zigzag paths). Also, among the two possible L-shaped paths, we choose the path whose corner is closest to the source. The nodes that lied on the path which were just added to the Steiner tree, are treated as new sinks. Next, the distances between the new sinks and all other sinks which are not in the current merged tree are computed and stored in the heap. The next iteration picks the smallest distance from the heap. This continues until every sink is covered.

If there are m sinks (both given and added sinks) in the Steiner tree, the complexity of BKRUS is $O(Vm^2)$. In the worst case, m is of $O(V^2)$. However, in practice, m is not large. In our benchmark circuits, m was usually no more than 10 times of V. In many VLSI designs, especially in standard cell designs, the sink locations are regular. So there are not so many Hanan points. These facts enabled us to run BKST on large benchmarks as well.

Figure 3 shows an example of this algorithm on a Hanan grid graph. In the Figure, (a) shows the given source and four sink locations. The dotted lines and their intersection points are the edges and nodes of the Hanan grid graph. Initially, the distances between the 5 points ($Source$, a, b, c, d) are computed and stored in the heap. From the heap, the shortest distance (a,b) is extracted. Assume that it is feasible. Then the path $path(a,b)$ shown in (b) is added to the Steiner tree. We then have two new sinks e and f. The distances from e, f to $Source$, c, d are computed and stored in the heap. The next shortest distance in the heap is (f,c) and it is feasible, so $path(f,c)$ is added to the Steiner tree in (c). The next shortest distance $path(c,d)$, however, is not feasible, so it is rejected. The next shortest and feasible distance is $path(Source,g)$, so it is added in (d). Finally, $path(i,d)$ is included in (e).

4 Gabow's Exact Method: BMST_G

Let us describe an optimal algorithm for the Bounded path length Minimal Spanning Tree (BMST) problem. This optimal algorithm is adopted from [5], although our implementation is somewhat different.

Gabow's algorithm produces all spanning trees in order of increasing cost with time complexity of O($KElog_{(1+E/V)}V$)

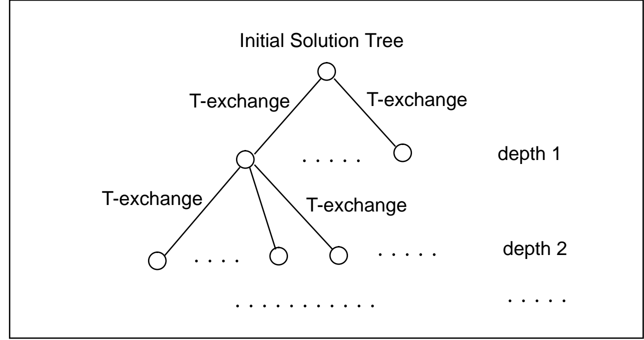

Figure 4: BKEX Negative-sum-Exchange Search Tree

and space complexity of O(K) where K is the total number of spanning trees generated[1]. We briefly describe his algorithm, omitting many details. Interested readers may refer to [5].

Let T be a spanning tree of G. A *T-exchange* is a pair of edges (e, f) where $e \in T$, $f \in G - T$ and $T - e \cup f$ is a spanning tree. The *weight of exchange* (e, f) is weight(f) − weight(e). The edge pair (e, f) which achieves the minimum weight of exchange is *minimal T-exchange*. Note that if T is a minimal spanning tree, there is no negative weight T-exchange. If T is a minimal spanning tree and (e, f) is a minimal T-exchange, then $T - e \cup f$ is a spanning tree with the next smallest cost. This is the basis of the algorithm.

We terminate Gabow's algorithm when the generated spanning tree satisfies the upper bound. The major shortcoming of Gabow's algorithm is the space complexity. Total number of spanning trees in a complete graph is V^{V-2} [6]. This makes Gabow's algorithm impractical even for as few as 10 nodes. We have been able to somewhat reduce the space and time complexities by eliminating some edges as a preprocessing to the Gabow's algorithm that cannot lead to a solution tree. Using this technique, we have used Gabow's algorithm on trees with as many as 15 sinks. In a practical CMOS circuit, a gate usually drives less than 10 gates. So this algorithm can be used in most practical cases.

5 Yet Another Exact Method and a Heuristic: BKEX and BKH2

Bounded Kruskal EXchange (BKEX) is a post-processing algorithm that starts from an initial solution tree and reduces the routing cost toward the optimal. If the initial tree is not an optimal solution, BKEX finds edge exchange(s) such that routing cost is reduced. We call such exchange(s) negative-sum-exchange(s).

Definition 5.1 *Negative-sum-exchange(s): A sequence of T-exchange(s) where the sum of the weight(s) of exchange(s) is negative.*

BKEX starts from any solution tree, finds negative-sum-exchange(s), converts the solution tree to a new solution tree by exchanging edges, and iterates until no more possible exchange(s) are found. Let's denote the search tree in Figure 4 as τ. Each node in τ represents a spanning tree.

[1]We believe this is the correct time complexity instead of Gabow's claim of O($KE\alpha(E,V)$).

The root of τ is the initial solution. A child node is generated by a T-exchange from its parent node. The edges of τ are labeled with the weight of T-exchange. BKEX searches negative-sum-exchange(s) in a depth first search manner. Note that one can reach any spanning tree from the root by a series of at most $V-1$ T-exchanges. So searching down to the depth of $V-1$ always finds the optimal solution. However, in most cases, BKEX finds an optimal solution in much smaller depth.

BKEX keeps track of the sum of T-exchange weights from the root to the current node during the depth first search. If the sum at a certain node is negative, a new tree with less cost is constructed at that node and we store the new tree as a minimum cost tree. Whenever a better solution is found during the search, this new tree is put on the root of τ and a new search begins.

Since the number of possible T-exchanges in a tree T is $O(EV)$, a node in τ has $O(EV)$ children. So τ has $O(E^nV^n)$ nodes where n is the depth of τ. For each node in τ, BKEX needs to check if the current spanning tree is feasible, which takes $O(V)$. So the time complexity of BKEX is $O(E^nV^{n+1})$. This is a higher time complexity than Gabow's, but space complexity is only $O(E)$. The initial solution significantly affects the performance of BKEX. When BKEX starts from a very good initial solution (such as BKT), the actual search space is much smaller than E^nV^n. Indeed our experimental results show that BKEX is much faster than Gabow's method. Besides, BKEX finds the solution when Gabow's algorithm fails for larger benchmarks due to its exponential space complexity.

We tested BKEX with 2,750 randomly generated benchmarks. The number of sinks of these benchmarks are between 5 and 15. The ϵ value has a range from 0.0 to 1.0. BKEX reaches optimal solutions of 96.945%, 97.309% and 99.709% with depth two, three and four respectively. Only one benchmark was left unoptimal with depth five and it was solved by depth six.

We implemented another heuristic method BKH2 which limits the depth of the search tree τ by two. It can be shown that BKT is a local optimum with respect to a single T-exchange. To obtain a better local optimum than BKT, at least double T-exchanges are needed. Thus BKH2 is proposed to find a local optimum with respect to two T-exchanges.

The complexity of BKH2 is $O(E^2V^3)$. Since this complexity is relatively high, we found that BKH2 is benificial when V is less than 300 (see Table 1).

6 Experimental Results

We implemented BKRUS, BMST_G, BKEX, BKH2, and BKST algorithms in C on HPPA and SUN workstations in the UNIX environment. We used three sets of benchmarks: (1) the sink placements for MCNC Primary1 and Primary2 benchmarks used in [3]; and (2) the sink placements for the five benchmarks r1-r5 used in [4]; and (3) five sets of 5 to 15 sinks and 50 random test cases for each set.

We added one more node as the source to the r* and primary* benchmarks because they did not come with a source. All the results are computed in Manhattan metric.

		BKRUS			BKH2			perf.
	ϵ	path ratio	perf. ratio	cpu	path ratio	perf. ratio	cpu	red. %
pr1	∞	2.424	1.000	0.73	2.424	1.000	0.76	0.00
	1.00	1.841	1.000	0.73	1.841	1.000	0.77	0.00
	0.50	1.491	1.018	0.75	1.465	1.002	11.51	1.60
	0.10	1.100	1.076	0.76	1.096	1.009	0.1k	6.24
	0.00	1.000	1.144	0.75	1.000	1.037	2.0k	9.37
pr2	∞	2.372	1.000	4.26	2.372	1.000	4.45	0.00
	1.00	1.935	1.000	4.27	1.935	1.000	4.6	0.00
	0.50	1.493	1.012	4.26	1.459	1.003	0.1k	0.88
	0.10	1.100	1.073	4.30	1.100	1.034	31.1k	3.60
	0.00	1.000	1.102	4.37	1.000	1.100	31.9k	0.18
r1	∞	1.941	1.000	1.12	1.941	1.000	1.17	0.00
	1.00	1.941	1.000	1.12	1.941	1.000	1.17	0.00
	0.50	1.489	1.019	1.11	1.489	1.002	2.5	1.68
	0.10	1.096	1.124	1.14	1.090	1.021	2.1k	9.21
	0.00	1.000	1.263	1.18	1.000	1.072	4.7k	15.11
r2	∞	1.838	1.000	7.32	1.838	1.000	7.67	0.00
	1.00	1.838	1.000	7.35	1.838	1.000	7.69	0.00
	0.50	1.489	1.021	7.33	1.468	1.000	0.3k	1.99
	0.10	1.098	1.084	7.51	1.097	1.034	52.6k	4.62
	0.00	1.000	1.148	7.55	1.000	1.118	32.6k	2.58
r3	∞	2.291	1.000	13.69	2.291	1.000	14.12	0.00
	1.00	1.833	1.000	13.65	1.833	1.000	14.27	0.00
	0.50	1.471	1.006	13.66	1.498	1.001	63.38	0.50
	0.10	1.100	1.060	13.68	1.099	1.051	22.0k	0.86
	0.00	1.000	1.156	14.27	1.000	1.137	28.6k	1.71
r4	∞	3.372	1.000	82.65	3.372	1.000	88.06	0.00
	1.00	1.995	1.011	82.62	1.898	1.000	23.5k	1.08
	0.50	1.490	1.014	84.83	1.496	1.010	55.0k	0.41
	0.10	1.100	1.076	103.67	1.100	1.076	54.5k	0.00
	0.00	1.000	1.104	102.74	1.000	1.104	18.7k	0.00
r5	∞	3.151	1.000	216.37	3.151	1.000	0.2k	0.00
	1.00	1.999	1.011	218.28	1.927	1.011	9.2k	1.05
	0.50	1.500	1.017	243.27	1.500	1.017	46.1k	0.00
	0.10	1.100	1.064	278.65	1.100	1.064	43.3k	0.00
	0.00	1.000	1.113	262.31	1.000	1.113	48.6k	0.00

perf. ratio (Tree) = cost(Tree) / cost(MST)

path ratio (Tree) = longest_path(Tree) / longest_path(SPT)

perf. reducttion = (1 − BKH2/BKRUS) * 100

CPU time is measured in seconds.

BKH2 limits CPU time to about 12 hours.

GABOW, BKEX and BPRIM are impractical to generate outputs.

Table 1: BKRUS and BKH2 results for large benchmarks

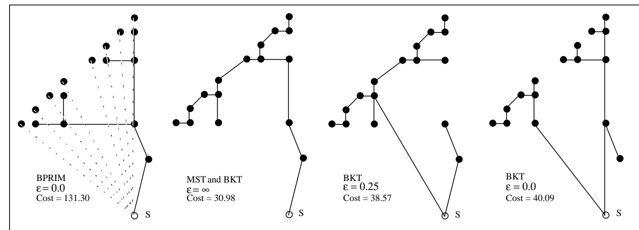

Figure 5: Example where the performance ratio of BRPIM is not bounded for any ϵ

As explained in [2], BPRIM can generate solution whose cost is not bounded for any ϵ. BPRIM generates $4.2 \cdot cost(MST)$ and $3.3 \cdot cost(BKT)$ for the benchmark shown in Figure 5 when $\epsilon = 0$.

A comparison of BKRUS and BKH2 over MST is given in Table 1 for benchmarks (1), (2). The results show that the performance ratio of BKT over MST is at most 1.263. For (3) benchmarks, the comparison of BPRIM, BRBC, BKRUS, BKH2, BMST_G and BKST in terms of routing cost is shown in Table 2. The benchmark results show about average performance ratios of 1.282, 1.245, 1.202, 1.202 and 1.032 for BPRIM, BKRUS, BKH2, BMST_G and BKST respectively in the worst case. In the case of 15 points with $\epsilon = 0.1$, the average cost reductions are 18.19%, 10.9%, 10.6% and 6.5% over BPRIM for BKST, BMST_G (BKEX), BKH2 and BKRUS respectively. BKRUS method offers a continuous, smooth trade-off between the competing requirements of longest path length and total wire length in terms of ϵ.

net		BPRIM		BRBC	BKRUS			BKH2			BMST_G			BKST			
size	ϵ	ave	max	max	ave	max	cpu	ave	max	cpu	ave	max	cpu	min	ave	max	cpu
5	0.0	1.157	1.854	1.854	1.153	1.854	0.021	1.151	1.854	0.024	1.150	1.854	0.219	0.802	0.953	1.366	0.308
5	0.1	1.108	1.514	1.715	1.092	1.514	0.021	1.088	1.514	0.022	1.088	1.514	0.221	0.802	0.934	1.125	0.303
5	0.2	1.073	1.332	1.715	1.063	1.332	0.022	1.059	1.332	0.024	1.059	1.332	0.224	0.802	0.925	1.114	0.307
5	0.3	1.040	1.332	1.715	1.035	1.332	0.023	1.033	1.332	0.021	1.033	1.332	0.226	0.802	0.923	1.114	0.307
5	0.4	1.034	1.332	1.438	1.032	1.300	0.023	1.028	1.300	0.022	1.028	1.300	0.202	0.802	0.924	1.114	0.306
5	0.5	1.024	1.209	1.595	1.020	1.206	0.021	1.018	1.168	0.023	1.018	1.168	0.220	0.802	0.924	1.114	0.308
5	1.0	1.004	1.089	1.315	1.002	1.048	0.023	1.002	1.048	0.022	1.002	1.048	0.218	0.802	0.914	1.000	0.303
8	0.0	1.254	2.043	2.494	1.239	1.938	0.024	1.202	1.938	0.022	1.202	1.938	0.221	0.864	0.986	1.205	0.307
8	0.1	1.156	1.928	2.116	1.115	1.457	0.023	1.094	1.457	0.023	1.094	1.457	0.227	0.853	0.982	1.205	0.308
8	0.2	1.120	1.933	1.908	1.077	1.318	0.021	1.057	1.294	0.024	1.057	1.294	0.225	0.853	0.959	1.111	0.307
8	0.3	1.072	1.515	1.773	1.043	1.318	0.023	1.029	1.187	0.021	1.029	1.187	0.221	0.853	0.951	1.111	0.307
8	0.4	1.056	1.351	1.767	1.032	1.205	0.024	1.023	1.156	0.020	1.023	1.156	0.219	0.840	0.945	1.111	0.307
8	0.5	1.057	1.469	1.665	1.023	1.156	0.020	1.019	1.156	0.023	1.019	1.156	0.216	0.840	0.944	1.111	0.306
8	1.0	1.003	1.145	1.361	1.002	1.112	0.011	1.000	1.025	0.013	1.000	1.025	0.211	0.840	0.934	1.025	0.306
10	0.0	1.235	2.121	2.136	1.221	1.970	0.012	1.173	1.970	0.014	1.169	1.970	0.249	0.861	1.009	1.224	0.311
10	0.1	1.168	1.621	2.136	1.127	1.495	0.011	1.094	1.369	0.014	1.092	1.369	0.239	0.861	0.983	1.224	0.312
10	0.2	1.143	1.764	1.843	1.069	1.338	0.010	1.054	1.277	0.013	1.052	1.277	0.231	0.861	0.946	1.136	0.312
10	0.3	1.110	1.906	1.803	1.031	1.242	0.014	1.029	1.242	0.012	1.029	1.242	0.223	0.861	0.941	1.136	0.314
10	0.4	1.065	1.489	1.803	1.025	1.253	0.012	1.020	1.242	0.018	1.019	1.242	0.229	0.861	0.939	1.136	0.313
10	0.5	1.056	1.591	1.535	1.022	1.247	0.022	1.016	1.242	0.023	1.015	1.242	0.228	0.861	0.937	1.136	0.312
10	1.0	1.005	1.149	1.455	1.001	1.020	0.022	1.001	1.020	0.022	1.001	1.016	0.218	0.861	0.926	1.007	0.310
12	0.0	1.242	1.611	2.129	1.226	1.558	0.024	1.155	1.517	0.029	1.149	1.517	0.818	0.829	1.002	1.217	0.314
12	0.1	1.200	1.536	2.129	1.115	1.542	0.023	1.072	1.300	0.024	1.072	1.300	0.313	0.829	0.980	1.256	0.314
12	0.2	1.136	1.407	1.707	1.073	1.408	0.024	1.041	1.246	0.024	1.041	1.246	0.292	0.829	0.956	1.135	0.314
12	0.3	1.092	1.398	1.601	1.056	1.286	0.022	1.029	1.171	0.023	1.029	1.171	0.240	0.829	0.947	1.129	0.315
12	0.4	1.068	1.272	1.525	1.038	1.254	0.022	1.020	1.133	0.024	1.020	1.133	0.218	0.829	0.937	1.033	0.314
12	0.5	1.044	1.312	1.508	1.016	1.139	0.022	1.009	1.087	0.025	1.009	1.087	0.222	0.829	0.932	1.028	0.314
12	1.0	1.001	1.052	1.346	1.001	1.033	0.023	1.001	1.030	0.022	1.001	1.030	0.218	0.829	0.927	0.976	0.316
15	0.0	1.282	1.763	2.179	1.245	1.745	0.025	1.157	1.745	0.032	1.148	1.686	10.465	0.926	1.032	1.266	0.326
15	0.1	1.204	1.705	2.037	1.126	1.298	0.025	1.077	1.298	0.030	1.073	1.298	22.490	0.872	0.985	1.245	0.324
15	0.2	1.138	1.618	1.877	1.078	1.282	0.025	1.047	1.212	0.024	1.043	1.212	8.518	0.872	0.971	1.245	0.329
15	0.3	1.141	1.794	1.802	1.056	1.282	0.024	1.031	1.148	0.025	1.028	1.148	4.411	0.846	0.955	1.245	0.323
15	0.4	1.099	1.556	1.711	1.034	1.159	0.024	1.020	1.105	0.026	1.018	1.092	0.683	0.846	0.938	1.044	0.323
15	0.5	1.067	1.345	1.708	1.024	1.133	0.025	1.012	1.092	0.024	1.012	1.092	0.634	0.846	0.939	1.120	0.321
15	1.0	1.012	1.151	1.385	1.004	1.046	0.023	1.003	1.046	0.023	1.003	1.046	0.301	0.846	0.925	1.000	0.320

50 random test cases were generated for each point.

CPU time is the average of 50 random test cases measured in seconds.

Minimum values are 1.008, 1.038, 1.007 and 1.007 for BPRIM, BKRUS, BKH2 and BMST_G respectively at ϵ = 0.0 of net 12. The others are 1.000.

BRBC is shown only with maximum values since minimum and average values of BRBC are always worse than those of BPRIM.

Table 2: The Ratio of the Routing Cost over MST

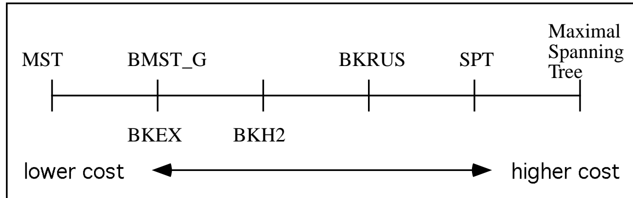

Figure 6: Routing Cost Chart

From these results, the various BMST methods can be ordered by their routing costs as shown in Figure 6. This chart shows the average relative positions.

The result of Bounded Kruskal Steiner Tree (BKST) on benchmark set (3) shows that its cost is lower than any other spanning tree heuristics. The savings are 5% to 30% over other heuristics. Note that the savings are even greater when ϵ is close to zero. This is due to the fact that when ϵ is close to zero, there are many direct source-to-sink paths in the spanning tree solutions while in the Steiner solutions, these direct paths are replaced by fewer direct source-to-sink paths. Although BKST produces lower cost trees, we feel that spanning tree heuristics are worthwhile because they run much faster.

7 Conclusion

We have presented bounded path length minimal spanning/Steiner tree schemes which can control longest path length and routing cost. Our method achieves smaller cost than that of BPRIM and BRBC. Future research includes considering the effects of buffering and wire sizing, extending this work to lower and upper bounded Steiner Trees and preserving planarity during the construction procedure.

References

[1] J. B. Kruskal, "One the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, Vol. 7, pp. 48-50, 1956.

[2] Jingsheng Cong, A. Kahng, G. Robins, M. Sarrafzadeh, and C. K. Wong, "Provably Good Performance-Driven Global Routing," *IEEE Transactions on Computer Aided Design* , Vol. 11, NO. 6, pp. 739-752, June, 1992.

[3] M. A. B. Jackson, A. Srinivasan, and E. S. Kuh, "Clock routing for high-performance ICs," *27th Design Automation Conference* , pp. 573-579, 1990.

[4] R-S Tsay, "Exact zero skew," *International Conference on Computer-Aided Design* , pp. 336-339, 1991.

[5] Harold N. Gabow, "Two algorithms for generating weighted spanning trees in order," *SIAM J. Comput.* , Vol. 6, No. 1, pp. 139-150, March 1977.

[6] F. Harary, "Graph Theory," *Addison-Wesley*, Massachusetts, pp. 152-154, 1969.

[7] J. Ho, D. T. Lee, C. H. Chang, and C. K. Wong, "Bounded diameter spanning trees and related problems," *Procceedings of ACM Symposium Computational Geometry* , pp. 276-282, 1989.

[8] C.J. Alpert, T.C. Hu, J.H. Huang and A.B. Kahng, "A direct combination of the Prim and Dijkstra constructions for improved performance-driven global routing," *International Symposium on Circuit and Systems*, pp. 1869-1872, 1993.

[9] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM Journal of Applied Mathematics*, pp. 255-265, March 1966.