Constructing Minimal Spanning/Steiner Trees with Bounded Path Length

Jaewon Oh, Iksoo Pyo and Massoud Pedram*

Department of Electrical Engineering - Systems

EEB 206

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 $1.19 \cdot cost(BMST^*)$ 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 generate even better results (more stable local minimum). Extensions of these techniques to the bounded path length Minimal Steiner Trees, using the Elmore delay model, and the construction of MSTs with lower and upper bounded path lengths are presented as well. Empirical results demonstrate the effectiveness of these algorithms on a large benchmark set.

*Part of this work is published in European Design and Test Conference 1996, pp 244-249.

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[1], 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 a very long source-to-sink path which degrades the performance. Alpert et al. [9] 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. We will show the same method can be extened to bounded path length Steiner tree.

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]. 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 [6] which produces an optimal BMST with exponential time and space complexity. Then, we propose a new exact algorithm which requires polynomial space. 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. Finally, we show extension of these algorithms to the case where the path lengths are bounded from both above and below (e.g. in clock routing problem).

The key features of our algorithms are described as follows.

- The path length from the source to each terminal is bounded.
- The routing cost is small.

[1] In a SPT, each sink is connected to the source by the shortest possible path.

- A user-given parameter can trade-off the longest and shortest path length for the routing cost.

The remainder of this paper is organized as follows. Section 2 formulates the BMST problem and describes the previous work. Section 3 proposes a new bounded path length algorithm which uses an analogue of the classical Kruskal MST construction. Section 4 describes an exact method for finding BMST based on a variation of the Gabow's algorithm. Section 5 presents another exact method and heuristic based on negative-sum-exchange(s). Section 6 describes extension to the lower bounded path length constraints. Section 7 presents benchmark results and comparisons and Section 8 concludes the paper.

2 Background

On a Manhattan (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 [8]. 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 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, 4 and Figure 1). 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. A better heuristic algorithm was proposed in [9], but its average performance is still lower than BPRIM.

Before describing our approach, we give some definitions. The sum of all edge weights of T is the cost of the tree, $cost(T)$. A *node* refers to both the source and the sinks. The distance between node u and node v is $dist(u, v)$. The shortest path distance between node u and node v in tree T is $path_T(u, v)$. The radius of node $v \in T$ is $radius_T(v)$ (i.e. max $\{path_T(v, u)\}$, $\forall\ u \in T$). The partial tree which contains node v is represented by t_v. S denotes the source.

We define the BMST problem as follows:

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

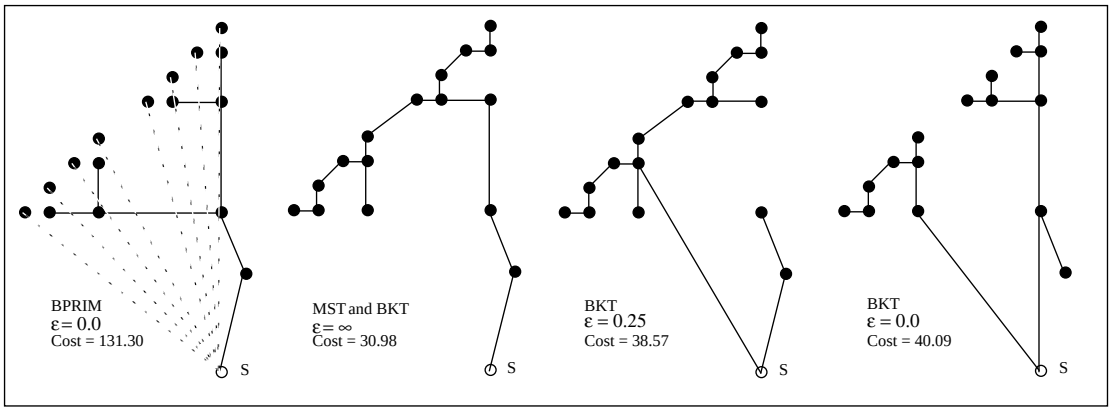

Figure 1: The leftmost figure is quoted from Cong, et. el. With BPRIM, as the tree is build up starting from the source, it ends up with a situation that the far away nodes cannot be connected to any other nodes except the source. However, our proposed method correctly returns the rightmost figure, which happened to be an optimal solution.

3 BKRUS and its extensions

In this section, we present Bounded KRUSkal(BKRUS) algorithm and its extensions to using the Elmore delay model, Bounded path length Steiner trees.

3.1 A heuristic: BKRUS

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 as shown in Figure 2:

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

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

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 (Figure 2-(a)). The case where t_v contains the source is similar.

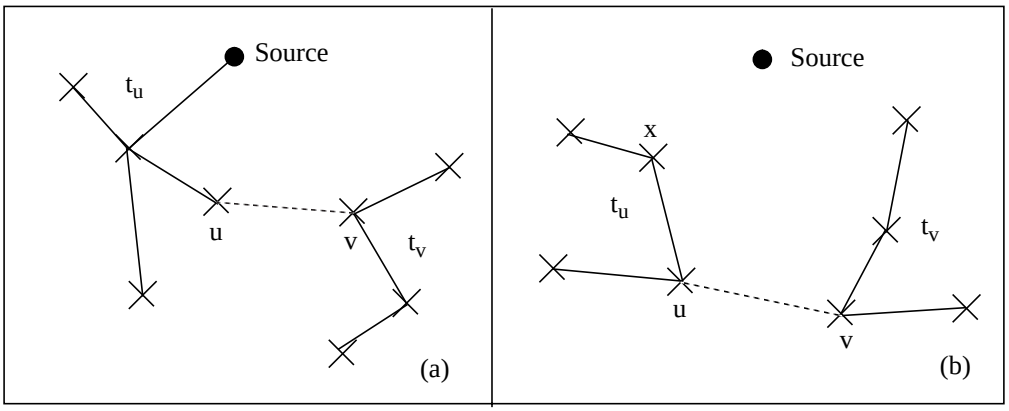

Figure 2: Feasibility test of edge (u, v). In (a), partial tree t_u contains the source. In (b), none of the partial tree contains the source.

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

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

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 (Figure 2-(b)). 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* $dist(S,x) + radius_{t_M}(x) \leq (1 + \epsilon) \cdot R$, *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.*

The importance of feasible edges is that they can be safely added to the spanning tree under construction.

The radius of a node in the merged tree can be found by the following equation. Suppose x belongs to t_u. Then it can be easily seen that

$$radius_{t_M}(x) = max\ \{radius_{t_u}(x), path_{t_u}(x, u) + dist(u, v) + radius_{t_v}(v)\}$$

The case where x belongs to t_v is similar. To conduct the feasibility test, 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 the distances between every pair of nodes, i.e. $D[x, y] = dist(x, y)$. This matrix is computed from the coordinates of nodes. Let the array $P[V, V]$ be the path length between every pair of nodes in the routing tree T, i.e. $P[x, y] = path_T(x, y)$. 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 at the beginning of the tree construction process. As the tree grows, P and r are updated by the Merge routine given below:

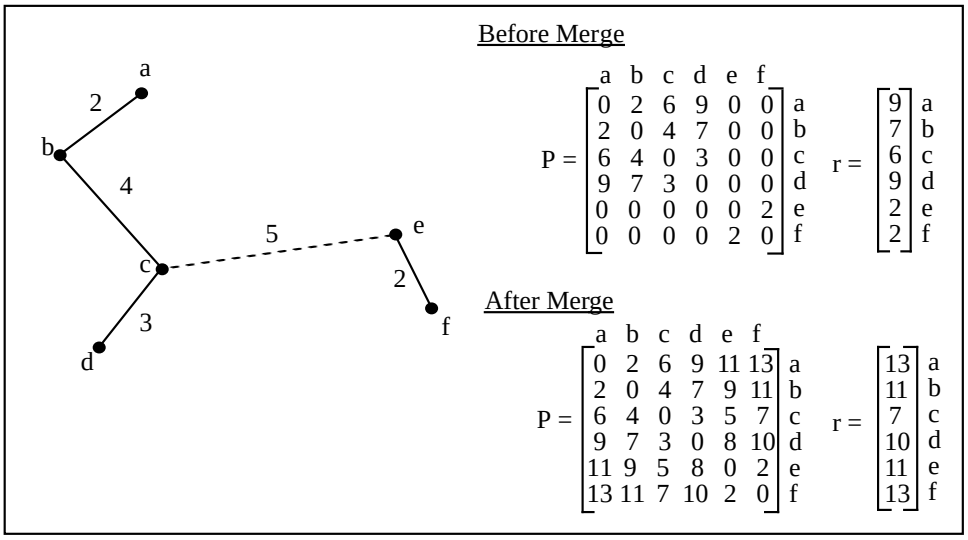

Figure 3: Example of Merging Two Partial Trees

{Merge two subtrees t_u and t_v by edge (u, v)}
Algorithm Merge(u, v)
1 for each $x \in t_u$ and $y \in t_v$ **do**
2 P$[x, y]$ = P$[y, x]$ = P$[x, u]$ + D$[u, v]$ + P$[v, y]$
3 end for
4 for each $x \in t_u$ **do**
5 r$[x]$ = max(r$[x]$, P$[x, i]$, $\forall\ i \in t_v$)
6 end for
7 for each $y \in t_v$ **do**
8 r$[y]$ = max(r$[y]$, P$[i, y]$, $\forall\ i \in t_u$)
9 end for

Figure 3 shows an example of how Merge routine works. The vertex labels and edge weights are shown in the figure. 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 the 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, 11, 13}. So the new $r[a]$ is 13. We can easily see that the time complexity of Merge is $O(V^2)$.

With this, the radius of a node x in the merged tree is

$$radius_{t_M}(x) = max\ \{r[x], P[x, u] + D[u, v] + r[v]\}$$

Note that D, P, r are already computed from the previous mergings. So no merging is needed to compute the radius of a node in the merged tree. Only when the edge (u, v) is feasible, the merge routine is invoked. Since feasibility test for a node can be done in $O(1)$, 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:source and sinks)
1 for each vertex $x \in$ G **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 **for** loop in line 9 can be implemented to make an early exit when $V - 1$ UNION is performed. Each node has a pointer to the next node in the same partial tree. Each node also has a pointer to a randomly selected representative node, which also serves as the name of the partial tree (hence the name of the set). With this implementation of sets, *FIND_SET(u)* can be done in $O(1)$ and *UNION(u,v)* can be done in $O(V)$. Line 1-4 take $O(V)$ while line 5-7 take $O(V^2)$. Sorting in line 8 takes $O(ElogE)$. The loop in line 9-17 goes $O(E)$ iterations in the worst case. Line 10 can be done in $O(1)$. Line 11 tests the condition (3-a) or (3-b) depending on the case and takes $O(V)$ in the worst case as discussed before. Line 12 takes $O(V)$ while line 13 takes $O(V^2)$. Line 14 puts (u, v) in the tree under construction. Since line 11 is executed E times and line 13 is executed $V - 1$ times, the complexity of line 9-17 is bounded by $O(EV + V \cdot V^2) = O(V^3)$. This dominates the whole complexity of BKRUS algorithm.

Here, we explain BKRUS algorithm with a simple example. Suppose we have a source and four sinks as shown in Figure 4. If the upper bound path length is set to 12, BKRUS works in the order of (a), (b), (c), and (d). In (c), edge (c, d) is rejected since there is no feasible node that satisfies upper bound constraint; edge (b, d) is included instead. In (d), edge (S,a) is rejected since it violates upper bound constraints. Finally, edge (S, b) is included in the tree.

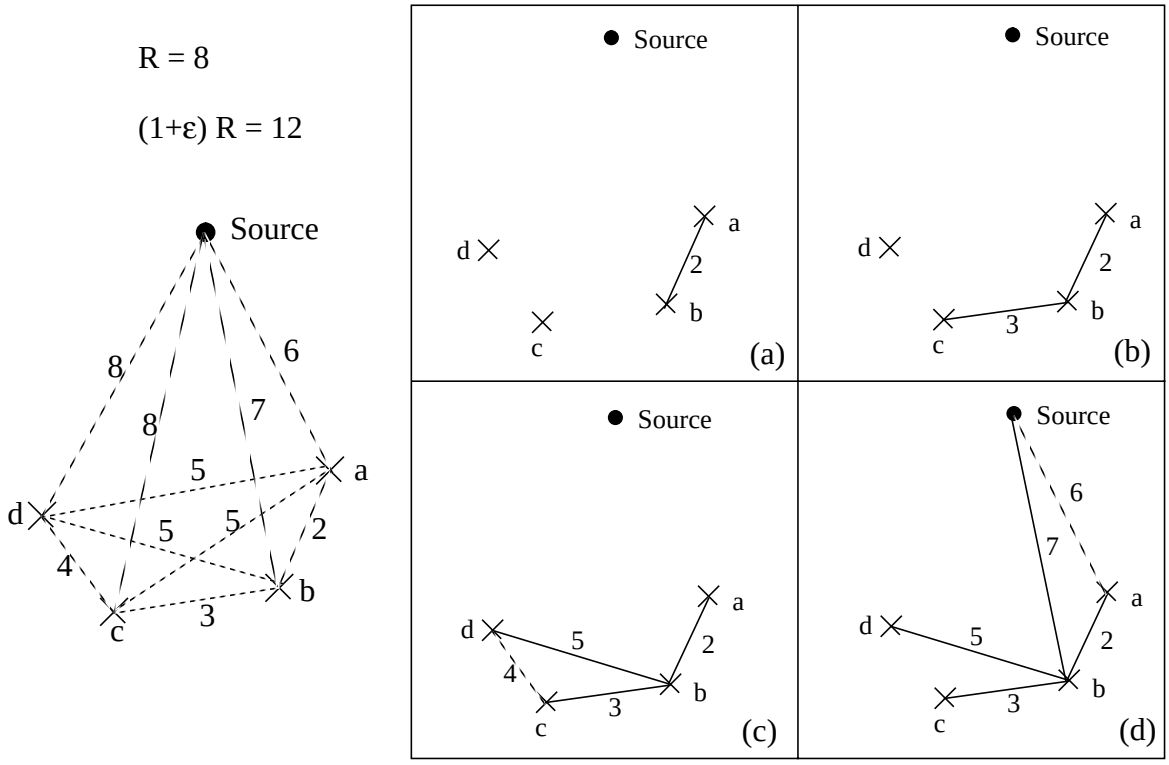

Figure 4: BKRUS Example: In (a) - (d), dotted lines are rejected because they are not feasible.

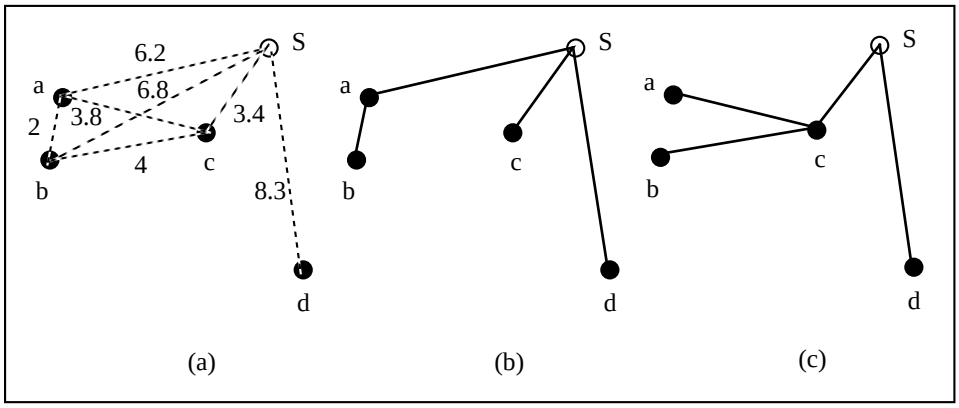

Figure 5: Example where BKRUS can not generate an optimal solution

During the BKRUS algorithm execution, once an edge is marked as infeasible and then rejected, it will never be considered again. The following lemma proves that indeed there is no need to reconsider such rejected edges.

Lemma 3.1 *If an edge is rejected during the BKRUS algorithm, then that edge cannot become feasible at a later time.*

Proof If the rejected edge was a cycle edge (violation of condition (2)), the rejected edge would again create a cycle when it is reconsidered for merging. If the rejected edge (u, v) was an upper bound violation edge (violation of condition (3)), there are two cases as follows: 1) If S $\in t_u$, then the edge (u, v) was rejected because $path_{t_u}$(S,u) + $dist(u, v)$ + $radius_{t_v}(v)$ > upper_bound. On the left hand side of the above inequality, the first two terms are fixed while the last term only increases but never decreases during the growth of trees. Hence there is no way that the edge (u, v) becomes feasible. The case S $\in t_v$ can be similarly proved. 2) If S $\notin t_u$ nor t_v, then the edge (u, v) was rejected because for all $x \in t_u$, $dist$(S,x) + $path_{t_u}(x, u)$ + $dist(u, v)$ + $radius_{t_v}(v)$ > upper_bound (the case for $x \in t_v$ can be similarly stated). After the growth of trees, let's assume without loss of generality that a node y is introduced to t_u and y is feasible in the tree merged by (u, v) (Note that any node x in t_u is still not feasible in the new merged tree). Then $dist$(S,y) + $path_{t_u}(y, u)$ + $dist(u, v)$ + $radius_{t_v}(v) \leq$ upper_bound. Let's pick any node x in the path from y to u. The above inequality can be rewritten as $dist$(S,y) + $path_{t_u}(y, x)$ + $path_{t_u}(x, u)$ + $dist(u, v)$ + $radius_{t_v}(v) \leq$ upper_bound. However, we have $dist$(S,y) + $path_{t_u}(y, x) \geq dist$(S,$x$) by the triangular inequality in Manhattan space (strict inequality in Euclidean space). If we insert this inequality into the previous inequality, then x is a feasible node, which is a contradiction. □

BKRUS may not generate an optimal solution. Consider configuration (a) of Figure 5 with an upper bound of 8.3. BKRUS generates the tree in (b) in order a-b, S-c, S-a and S-d with total cost 19.9 which is not optimal. However, if we had rejected a-b, we could have generated the tree in (c) in order S-c, c-a, c-b, and S-d with total cost of 19.5 which is optimal. In order for BKRUS to be an exact algorithm, we need a backtracking step which removes existing tree edges and adds a new feasible edge. However, this will make the algorithm an exponential one.

3.2 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. 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 and the load capacitance at the sinks. 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 a tentative merger of the two subtrees.

The Elmore delay is defined as follows. Let T be a routing tree and u, v be nodes of T. Suppose the signal is propagated from u to v. Then we restructure the tree T such that u is the root of the tree T (restructuring means changing the direction of edge arcs in the tree). Let T_k be a subtree of T rooted at node k, i.e. T_k is a subtree rooted at k and all the nodes in T_k are descendents of k in T. Let $p(k)$ be the parent of k. We define C_k be the sum of all the load capacitances and the wire capacitances of T_k. That is:

$$C_k = \sum_{x \in T_k, x \neq k} c_s \cdot dist(x, p(x)) + C_L(x)$$

where c_s is the unit sheet capacitance of the wire and $C_L(k)$ is the load capacitance of k.

In a tree, there is a unique path from a node to the other node. Let $path\_nodes(x, y)$ be the set of nodes in the path from x to y. The Elmore delay $delay(x, y)$ is defined as:

$$delay(x, y) = \sum_{k \in path\_nodes(x,y), k \neq x} r_s \cdot dist(k, p(k))(c_s/2 \cdot dist(k, p(k)) + C_k)$$

where r_s is the unit sheet resistance of the wire. Especially when x is the Source, we take driver resistance r_d and driver capacitance c_d into account.

$$delay(S, y) = r_d \cdot (c_d + C_S) + \sum_{k \in path\_nodes(S,y), k \neq S} r_s \cdot dist(k, p(k))(c_s/2 \cdot dist(k, p(k)) + C_k)$$

Then we can find the radius of a node u in the merged tree.

$$r[u] = max\ \{delay(u, v), \forall v \in t_M\}$$

Since the delay computation between any two nodes takes $O(V)$, the delay compatation between every pair of nodes in the merged tree takes $O(V^2)$. After that, the radii for all the nodes can be found in $O(V^2)$.

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 low driver resistance so that SPT can be a solution. R is set to the longest S-sink delays of SPT.

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

$$r_d \cdot (c_d + c_s \cdot dist(S, x) + C_x) + r_s \cdot dist(S, x) \cdot (c_s \cdot dist(S, x)/2 + C_x) + r[x] < (1 + \epsilon) \cdot R$$

When all the radii are available, (3-a)$'$ takes constant time, while (3-b)$'$ takes $O(V)$. Hence, the feasibility test is dominated by the radii computations, which is $O(V^2)$. As a result of these modifications to BKRUS, the feasibility test dominates the total complexity of the algorithm, whose complexity becomes $O(EV^2)$.

3.3 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 [10] using a modified BKRUS. A spanning tree that spans all the sinks and the source on these routing graphs becomes a Steiner tree. 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 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 closer to the source. The nodes that lie on the path which was 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 nodes in the routing graph, the complexity of BKRUS becomes $O(Vm^2)$. 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 6 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 [6], although our implementation is somewhat different. Besides, Lemma 4.1 to 4.3 which are used to reduce the space and time complexities of Gabow's method are new.

Gabow's algorithm produces all spanning trees in order of increasing tree cost with time complexity of $O(KElog_{(1+E/V)}V)$ and space complexity of $O(K)$ where K is the total number of spanning trees generated[2]. We briefly describe his algorithm, omitting many details. Interested readers may refer to [6].

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).

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

Figure 6: Example of Steiner BKRUS

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} [7]. This makes Gabow's algorithm impractical even for as few as 10 nodes. We found that some edges should be included and some edge should be excluded in the optimal solution tree. As a preprocessing, edges are included or excluded using the following rules. With it, we have been able to somewhat reduce the space and time complexities.

Lemma 4.1 *Consider the source S and two sinks a and b. For every three nodes S, a and b, if weight(a,b) > weight(S,a), weight(S,b), then eliminate edge(a,b).*

Lemma 4.2 *Eliminate edge (a,b) (a, b are sinks) if weight(S,a) + weight(a, b) $> (1 + \epsilon)R$ and weight(S,b) + weight(a,b) $> (1 + \epsilon)R$.*

Lemma 4.3 *Include edge(S,a) if for any node x ($x \neq a$), weight(S,x) + weight(x,a) $> (1 + \epsilon)R$.*

The Lemma 4.1, 4.2 and 4.3 above can be reasoned from geometric considerations. Lemma 4.1 means that there is no optimal solution that includes the edge (a, b). If an optimal solution includes such edge (a,b), then we can construct better solution as follows.

First remove the edge (a,b). This will disconnet the solution tree into two subtrees t_a and t_b. Suppose t_a contains the source S. Then we add edge (S,b) to the tree. We can see that nodes which were in t_b do not violate the upper bound constraint since weight(S,b) < weight(a,b). So the new tree has lower cost than the optimal solution, which proves the Lemma.

Lemma 4.2 means that some edge (a,b), when included, makes one of the node a, b violate the delay constraint no matter how the tree will be built later. Lemma 4.3 means that there are sinks which should be connected directly to the source because any indirect path to the sink will violate the delay constraint. Using these rules, 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

Let us describe another optimal algorithm for the same problem. This optimal algorithm is based on negative-sum-exchange(s) technique.

Bounded Kruskal EXchange (BKEX) is a post-processing algorithm that starts from an initial feasible solution and reduces the routing cost toward the optimal. Let T be a bounded path length tree such as SPT or BKT. If T is not an optimal solution, BKEX will find edge exchange(s) such that routing cost will be 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, and iterates until no more possible exchange(s) are found. Let's call the search tree in Figure 7 τ. Each node in τ represents a spanning tree. 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 including an optimal solution from the root by a series of at most $V-1$ T-exchanges. However, in most cases, BKEX finds an optimal solution in much smaller depth.

BKEX keeps track of sum of T-exchange weights from the root to the current node during the depth first search. If this sum is not negative, further search from this node is stopped. Whenever a better solution is found during the search, this new tree is put on the root of τ and a new search begins. Below is the complete algorithm.

Algorithm BKEX(G)
1 T $\leftarrow$ BKRUS(G)
2 while DFS_EXCHANGE(T,0) **do**
Algorithm DFS_EXCHANGE (T, weight_sum)
1 FA $\leftarrow$ make_father_array(T)
2 for each edge (x,y) in G $-$ T **do**

Figure 7: BKDFS Negative-sum-Exchange Search Tree

```
3     u = x, v = y
4     while (u ≠ v) do
5           if depth(u) > depth(v) then
6             swap u, v
7           diff = weight(x,y) − weight(v,FA[v])
8           if diff + weight_sum < 0 then
9             T ← T − (v,FA[v]) ∪ (x,y)
10            if T is feasible then
11              return TRUE
12            else if DFS_EXCHANGE(T, diff + weight_sum) then
13              return TRUE
14            else T ← T − (x,y) ∪ (v,FA[v])
15            end if
16          end if
17          v ← FA[v]
18    end while
19 end for
20 return FALSE
```

In the DFS_EXCHANGE algorithm, FA is the father array in spanning tree T (FA[v] is the father node of the node v). This can be generated by depth first search on T starting from S. At the same time, the depth level for each node is recorded (depth of a node is the number of ancestors in the path from the source S to the node. In particular, depth(S) = 0). Parameter $weight\_sum$ is the sum of weights of T-exchanges so far. Initially, for each non-tree edge (x, y), u, v are set to x, y respectively. Suppose v has a higher depth than u (line 5-6 ensure that v has a higher depth). Then the new exchange value is weight(x, y) − weight(v,FA[v]). If this value is added to $weight\_sum$ and the sum is still negative, then the new spanning tree generated by adding (x, y) and removing (v,FA[v]) has less cost than

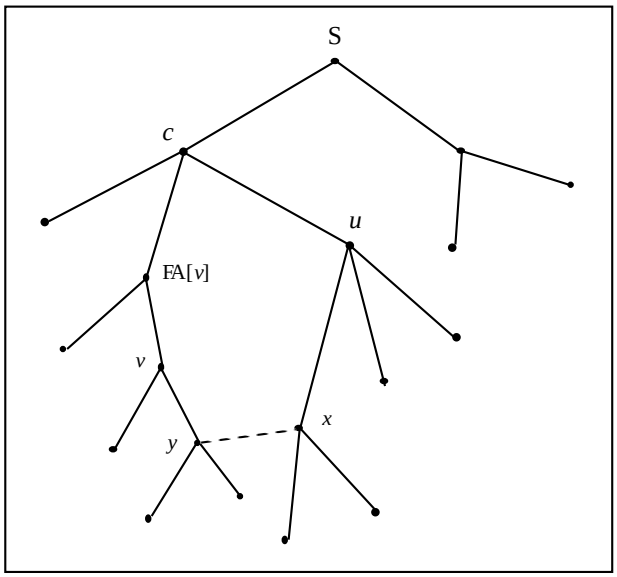

Figure 8: Example of finding T-exchanges

the root. If it is not negative, we replace v by FA[v] (line 17) and continue the above procedure until u and v meet at a common ancestor. When the new spanning tree has less cost than the root, we check if this new tree is feasible. If it is, a new iteration begins (line 2 in BKEX). If not, we recursively call DFS_EXCHANGE with the new spanning tree and the new $weight\_sum$ value (line 12). If subsequent calls to DFS_EXCHANGE still fails, we recover the original spanning tree (line 14). The iteration in BKEX is terminated when there is no feasible exchange.

Figure 8 shows an example of how edge pairs for T-exchange are found. For each (x, y), u, v start from x, y respectively and move toward the common ancestor (node c). In Figure 8, v has a higher depth compared to u, so (v,FA[v]) and (x, y) are paired up as a possible T-exchange. Suppose the exchange is rejected. Then the new v becomes FA[v]. This procedure alternates between u and v until they meet at node c.

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 in Table 2 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. BKH2 does one or two negative-sum-exchange(s) in the breadth first search manner and checks if the resultant tree is a solution. This procedure is repeated until there is no improvement.

With the help of Lemma 3.1, 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 deep local optimum with respect to two T-exchanges. BKH2 may not get an optimal solution because we may need three or more exchanges to improve the solution. 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 3).

6 Lower and Upper Bounded Path Length MST

In the clock routing, there are two important parameters - clock skew and routing cost - so that we would like to simultaneously control both the longest/shortest interconnection path and routing cost. Also, in case of global routing, fast sink delays should be avoided to prevent the so called "double clocking" as described next. Consider two synchronizing elements F_1, F_2 and a combinational circuit that delivers signal from F_1 to F_2. Suppose the combinational circuit delay (delay due to routing tree and logic gates) is very short. If F_2 is a slow flip-flop, then at the rising edge of the clock, the new value through the combinational circuit may arrive fast and replace the input data of F_2 before F_2 latches the previous data. A usual practice to avoid this problem is to add a buffer to slow down the fast combinational circuit. However this requires extra area and introduces additional power consumption. Instead, we can adjust the delay lower bound by wire-length control. As interconnection delays dominate gate delays these days, this method will be more area and power efficient.

A routing tree can be constructed with shortest interconnection path length at least $\epsilon_1 \cdot R$, longest interconnection path length at most $(1+\epsilon_2) \cdot R$ for any given values of parameters ϵ_1 and ϵ_2. That is, any source to sink path length is bounded by:

$$\epsilon_1 \cdot R \leq path_T(S, x) \leq (1 + \epsilon_2) \cdot R, \quad \forall sink \;\; x$$

Bounded-delay-difference Steiner heuristics were presented in [11], [12] and [13]. Our algorithm is a spanning tree heuristic, so it may give higher tree costs than the Steiner tree heuristics. However, our algorithm runs fast, and gives reliable estimation of tree cost upper bounds to the Steiner tree heuristics. Besides, our algorithm accepts explicit lower bound and upper bound on delay rather than delay differences as in [11], [12] and [13].

BKRUS, BMST_G, BKEX, and BKH2 algorithms are implemented for both the lower and the upper bounded path length with the inclusion of Lemma 6.1. The inclusion of Lemma 6.1 eliminates an edge (S, i) $< \epsilon_1 \cdot R$, $\forall$ i such that the resultant tree does not violate the lower bound.

Lemma 6.1 *Eliminate edge (S, i) $\in E$, for $\forall i$, if weight(S, i) $< \epsilon_1 \cdot R$.*

bench	# of pts.	# of edges	R	r
p1	6	15	20.4	20.0
p2	8	28	20.4	10.0
p3	17	136	16.0	6.1
p4	31	465	10.4	5.8
pr1	270	36315	542	27
pr2	604	182106	981	17
r1	268	35778	58700	1175
r2	599	179101	86554	1246
r3	863	371953	85509	1357
r4	1904	1811656	124357	564
r5	3102	4809651	138318	640

R: length of the shortest path from source to the farthest sink

r: length of the shortest path from source to the nearest sink

Table 1: Characteristics of Benchmarks

It is possible that solutions do not exist depending on the benchmark or on the lower/upper bounds combinations. One can easily generate such instances. This is unavoidable since we are restricting ourselves to spanning tree heuristics.

7 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 four sets of benchmarks: (1) the sink placements for the four benchmarks p1-p4 generated specially to test extreme results; and (2) the sink placements for MCNC Primary1 and Primary2 benchmarks used in [3] and [4], and originally provided by the authors of [3]; and (3) the sink placements for the five benchmarks r1-r5 used in [5]; and (4) five sets of 5 to 15 sinks and 50 random test cases for each set. Benchmark p1 and p2 have the same configuration as that of Figure 13, but p2 has one more sink between the source and the group of sinks. Benchmark p3 has the same configuration as that of Figure 1. Benchmark p4 has the same configuration as that of Figure 13, but sinks are scattered around a circle of diameter 20. 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. Description of all the benchmarks is given in Table 1.

A comparison of BMST_G, BKEX, BKRUS, BKH2 and BPRIM over MST is given in Table 2 and Table 3 for benchmarks (1), (2), (3) The results show that the performance ratio of BKT over MST is at most 1.263 except in p1 and p2 which have a very special configuration. In the case of p2 with $\epsilon = 0.2$, BPRIM generates poor performance ratio compared to our methods. In the case of p4 with $\epsilon = 0.3$, the cost reductions are 23%, 22% and 20% over BPRIM for BKEX, BKH2 and BKRUS respectively. For (4) benchmarks, the comparison of BPRIM, BRBC, BKRUS, BKH2, BMST_G and BKST in terms of routing

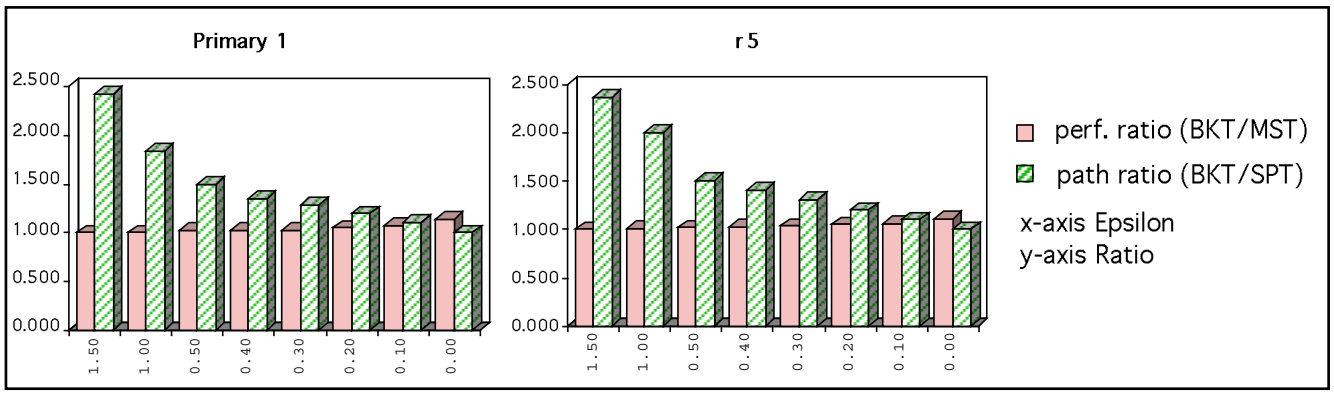

Figure 9: Tradeoff Curve

cost is shown in Table 4. The benchmark results show about worst-case performance ratios of 2.711, 2.219, 2.056 and 2.056 and average performance ratios of 1.705, 1.517, 1.455 and 1.452 for BPRIM, BKRUS, BKH2 and BMST respectively. In the case of 15 points with $\epsilon = 0.2$, the average cost reductions are 21%, 20% and 17% over BPRIM for BMST_G (BKEX), BKH2 and BKRUS respectively. The result of Bounded Kruskal Steiner Tree (BKST) on benchmark set (4) 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 and Steiner points on those paths that connect other sinks. Although BKST produces lower cost trees, we feel that spanning tree heuristics are worthwhile because they run much faster.

BKRUS method offers a continuous, smooth trade-off between the competing requirements of longest path length and total wire length in terms of ϵ as shown in Figure 9.

In Figure 10, we show ratios of $cost(BKRUS)/cost(MST)$, $cost(BKEX)/cost(MST)$, $cost(BKRUS)/cost(BKEX)$ and $cost(BKH2)/cost(BKEX)$. The reason why we compare BKEX, BKRUS and BKH2 with MST is to show that our methods generate a low cost routing tree compared to MST whatever the ϵ value is. The effectiveness of BKRUS and BKH2 is shown by $cost(BKRUS)/cost(BKEX)$ and $cost(BKH2)/cost(BKEX)$ respectively.

From these results, the various BMST methods can be ordered by their routing costs as shown in Figure 11. This chart shows the average relative position.

For lower/upper bounded delay MST, we tested BKRUS method for (1), (2), (3) benchmarks as shown in Table 5. BKRUS uses 3.9 times routing cost of MST to generate an exact zero skew tree. Note that many values of ϵ_1 and ϵ_2 lead to infeasible solutions since BKRUS uses node-branching technique. Path-branching and Steiner-branching are more desirable. In Figure 12, we show a typical trade-off between routing cost and clock skew.

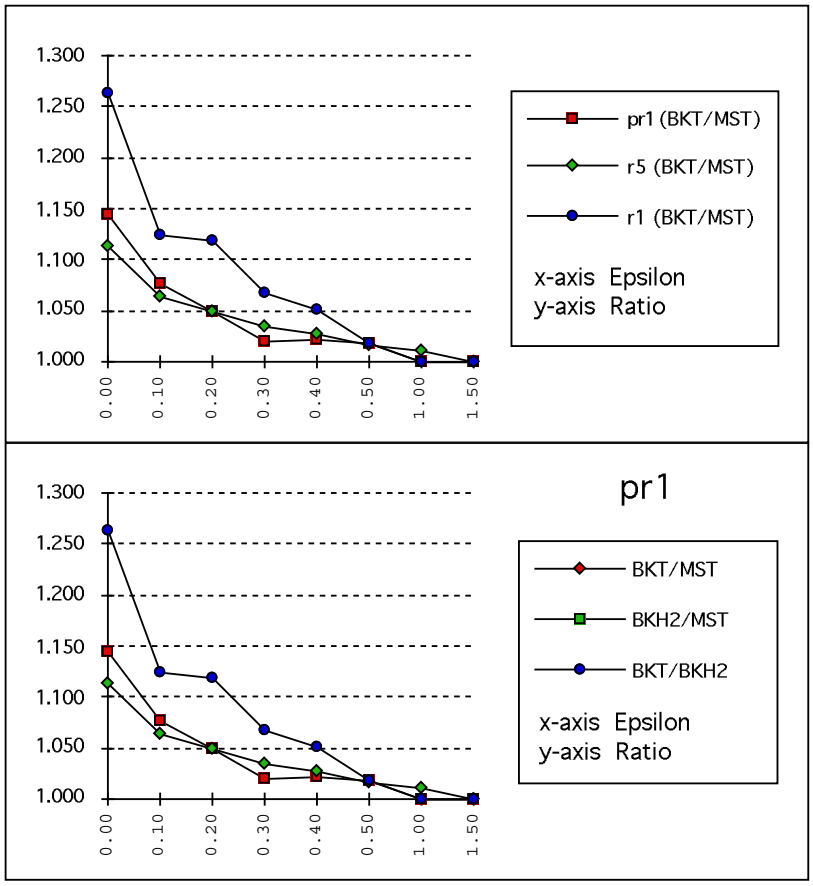

Figure 10: Ratio Curve

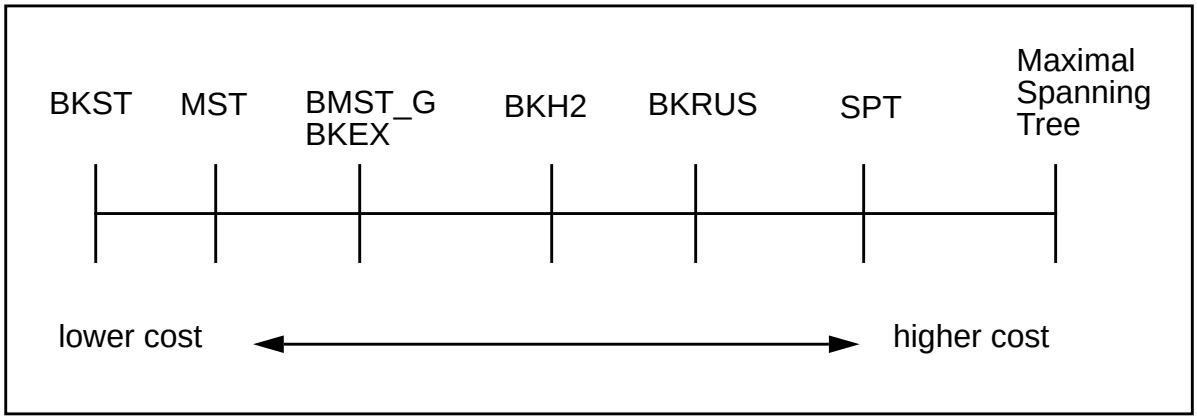

Figure 11: Routing Cost Chart

		BMST_G			BKEX			BKRUS			BKH2			BPRIM	
bench	ϵ	path ratio	perf. ratio	cpu	path ratio	perf. ratio	cpu	path ratio	perf. ratio	cpu	path ratio	perf. ratio	cpu	path ratio	perf. ratio
p1	∞	1.18	1.00	0.30	1.18	1.00	0.03	1.18	1.00	0.03	1.18	1.00	0.03	1.28	1.00
	1.5	1.18	1.00	0.30	1.18	1.00	0.03	1.18	1.00	0.01	1.18	1.00	0.02	1.28	1.00
	1.0	1.18	1.00	0.30	1.18	1.00	0.01	1.18	1.00	0.01	1.18	1.00	0.03	1.28	1.00
	0.5	1.18	1.00	0.30	1.18	1.00	0.03	1.18	1.00	0.01	1.18	1.00	0.03	1.28	1.00
	0.4	1.18	1.00	0.31	1.18	1.00	0.01	1.18	1.00	0.02	1.18	1.00	0.01	1.28	1.00
	0.3	1.18	1.00	0.30	1.18	1.00	0.02	1.18	1.00	0.01	1.18	1.00	0.02	1.28	1.00
	0.2	1.18	1.00	0.31	1.18	1.00	0.01	1.18	1.00	0.02	1.18	1.00	0.02	1.18	1.74
	0.1	1.08	1.70	0.29	1.08	1.70	0.02	1.08	1.70	0.02	1.08	1.70	0.03	1.08	1.70
	0.0	1.00	3.88	0.29	1.00	3.88	0.02	1.00	3.88	0.02	1.00	3.88	0.03	1.00	3.88
p2	∞	1.38	1.00	0.29	1.38	1.00	0.01	1.38	1.00	0.02	1.38	1.00	0.03	1.38	1.00
	1.5	1.38	1.00	0.29	1.38	1.00	0.04	1.38	1.00	0.01	1.38	1.00	0.01	1.38	1.00
	1.0	1.38	1.00	0.29	1.38	1.00	0.02	1.38	1.00	0.01	1.38	1.00	0.01	1.38	1.00
	0.5	1.38	1.00	0.30	1.38	1.00	0.03	1.38	1.00	0.01	1.38	1.00	0.01	1.38	1.00
	0.4	1.38	1.00	0.29	1.38	1.00	0.04	1.38	1.00	0.02	1.38	1.00	0.02	1.38	1.00
	0.3	1.29	1.13	0.33	1.29	1.13	0.04	1.19	1.17	0.01	1.29	1.13	0.01	1.29	1.16
	0.2	1.19	1.17	0.35	1.19	1.17	0.03	1.19	1.17	0.01	1.19	1.17	0.03	1.19	1.95
	0.1	1.10	1.30	0.32	1.10	1.30	0.03	1.10	1.32	0.01	1.10	1.30	0.02	1.08	1.35
	0.0	1.00	2.69	0.29	1.00	2.69	0.03	1.00	2.69	0.01	1.00	2.69	0.01	1.00	2.69
p3	∞	1.81	1.00	0.30	1.81	1.00	0.02	1.81	1.00	0.02	1.81	1.00	0.03	1.74	1.00
	1.5	1.81	1.00	0.31	1.81	1.00	0.02	1.81	1.00	0.01	1.81	1.00	0.03	1.74	1.00
	1.0	1.81	1.00	0.29	1.81	1.00	0.03	1.81	1.00	0.03	1.81	1.00	0.03	1.74	1.00
	0.5	1.43	1.01	0.31	1.50	1.01	0.02	1.49	1.16	0.01	1.50	1.01	0.04	1.45	1.14
	0.4	1.40	1.03	0.56	1.40	1.03	0.14	1.34	1.18	0.02	1.40	1.03	0.05	1.39	1.37
	0.3	1.23	1.07	34.37	1.23	1.07	0.48	1.30	1.25	0.01	1.23	1.07	0.05	1.28	1.26
	0.2	1.19	1.09	113.61	1.19	1.09	1.80	1.16	1.44	0.03	1.19	1.09	0.08	1.19	1.23
	0.1	1.02	1.11	564.82	1.02	1.11	2.92	1.10	1.44	0.01	1.02	1.11	0.23	1.10	1.27
	0.0	-	-	-	1.00	1.17	3.55	1.00	1.29	0.02	1.00	1.17	0.08	1.00	1.89
p4	∞	5.21	1.00	0.32	5.21	1.00	0.02	5.21	1.00	0.02	5.21	1.00	0.04	*	*
	1.5	-	-	-	2.40	1.06	20.09	2.45	1.07	0.03	2.40	1.06	0.19	2.38	1.49
	1.0	-	-	-	1.89	1.11	25.79	1.89	1.11	0.02	1.89	1.11	0.23	2.00	1.47
	0.5	-	-	-	1.47	1.17	27.66	1.47	1.22	0.02	1.47	1.17	0.55	1.50	1.50
	0.4	-	-	-	1.39	1.20	44.72	1.40	1.27	0.02	1.39	1.23	0.41	1.39	1.55
	0.3	-	-	-	1.30	1.26	27.91	1.30	1.32	0.02	1.30	1.28	0.44	1.29	1.64
	0.2	-	-	-	1.20	1.34	33.34	1.18	1.46	0.02	1.20	1.38	0.25	1.17	1.89
	0.1	-	-	-	1.10	1.45	10.03	1.10	1.61	0.02	1.10	1.45	0.12	1.10	1.88
	0.0	1.00	1.60	25.06	1.00	1.60	0.12	1.00	1.66	0.02	1.00	1.60	0.04	1.00	2.04

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

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

CPU time is measured in seconds.

-: memory overflow

*: can not get MST even with $\epsilon = 2.0$

Table 2: BMST_G, BKEX, BKRUS, BKH2 and BPRIM results for special benchmarks

bench	ϵ	BKRUS path ratio	BKRUS perf. ratio	BKRUS cpu	BKH2 path ratio	BKH2 perf. ratio	BKH2 cpu	perf. reduction %
pr1	∞	2.424	1.000	0.73	2.424	1.000	0.76	0.00
	1.50	2.424	1.000	0.75	2.424	1.000	0.77	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
	0.40	1.354	1.021	0.75	1.345	1.006	12.63	1.42
	0.30	1.279	1.020	0.75	1.292	1.008	26.61	1.21
	0.20	1.196	1.050	0.75	1.185	1.008	98.66	4.00
	0.10	1.100	1.076	0.76	1.096	1.009	139.62	6.24
	0.00	1.000	1.144	0.75	1.000	1.037	2027.01	9.37
pr2	∞	2.372	1.000	4.26	2.372	1.000	4.45	0.00
	1.50	2.372	1.000	4.27	2.372	1.000	4.44	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	108.84	0.88
	0.40	1.377	1.027	4.25	1.396	1.004	623.1	2.25
	0.30	1.292	1.030	4.27	1.300	1.007	2497.06	2.20
	0.20	1.196	1.064	4.31	1.200	1.008	9230.86	5.23
	0.10	1.100	1.073	4.30	1.100	1.034	31130.22	3.60
	0.00	1.000	1.102	4.37	1.000	1.100	31914.81	0.18
r1	∞	1.941	1.000	1.12	1.941	1.000	1.17	0.00
	1.50	1.941	1.000	1.12	1.941	1.000	1.15	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
	0.40	1.398	1.051	1.11	1.384	1.005	131.68	4.38
	0.30	1.299	1.067	1.12	1.299	1.009	177	5.44
	0.20	1.189	1.119	1.14	1.196	1.013	367.15	9.47
	0.10	1.096	1.124	1.14	1.090	1.021	2094.06	9.21
	0.00	1.000	1.263	1.18	1.000	1.072	4673.46	15.11
r2	∞	1.838	1.000	7.32	1.838	1.000	7.67	0.00
	1.50	1.838	1.000	7.36	1.838	1.000	7.69	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	322.89	1.99
	0.40	1.377	1.031	7.40	1.382	1.001	52.54	2.89
	0.30	1.299	1.026	7.43	1.300	1.006	11939.77	2.00
	0.20	1.195	1.054	7.47	1.190	1.003	23811.72	4.84
	0.10	1.098	1.084	7.51	1.097	1.034	52564.27	4.62
	0.00	1.000	1.148	7.55	1.000	1.118	32609.49	2.58
r3	∞	2.291	1.000	13.69	2.291	1.000	14.12	0.00
	1.50	2.291	1.000	13.67	2.291	1.000	14.07	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
	0.40	1.394	1.008	13.68	1.394	1.002	208.83	0.59
	0.30	1.300	1.021	13.65	1.296	1.003	5259.78	1.70
	0.20	1.200	1.034	13.62	1.189	1.014	46460.65	2.01
	0.10	1.100	1.060	13.68	1.099	1.051	21982.55	0.86
	0.00	1.000	1.156	14.27	1.000	1.137	28567.49	1.71
r4	∞	3.372	1.000	82.65	3.372	1.000	88.06	0.00
	1.50	2.500	1.001	82.79	2.409	1.000	373.65	0.08
	1.00	1.995	1.011	82.62	1.898	1.000	23543.57	1.08
	0.50	1.490	1.014	84.83	1.496	1.010	54987.24	0.41
	0.40	1.396	1.026	97.95	1.399	1.016	46892.91	1.00
	0.30	1.299	1.027	86.79	1.299	1.027	57719.58	0.00
	0.20	1.199	1.056	91.89	1.199	1.056	61761.98	0.00
	0.10	1.100	1.076	103.67	1.100	1.076	54489.35	0.00
	0.00	1.000	1.104	102.74	1.000	1.104	18689.98	0.00
r5	∞	3.151	1.000	216.37	3.151	1.000	219.88	0.00
	1.50	2.357	1.000	222.83	2.357	1.000	237.81	0.00
	1.00	1.999	1.011	218.28	1.927	1.011	9181.02	1.05
	0.50	1.500	1.017	243.27	1.500	1.017	46070.31	0.00
	0.40	1.399	1.027	258.44	1.400	1.027	47657.18	0.00
	0.30	1.300	1.034	278.06	1.300	1.034	44185.97	0.00
	0.20	1.200	1.050	259.71	1.200	1.050	46412.70	0.00
	0.10	1.100	1.064	278.65	1.100	1.064	43297.40	0.00
	0.00	1.000	1.113	262.31	1.000	1.113	48552,52	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 3: BKRUS and BKH2 results for large benchmarks

net		BPRIM		BRBC	BKRUS			BKH2		
size	ϵ	ave	max	max	ave	max	cpu	ave	max	cpu
5	0.0	1.157	1.854	1.854	1.153	1.854	0.021	1.151	1.854	0.024
5	0.1	1.108	1.514	1.715	1.092	1.514	0.021	1.088	1.514	0.022
5	0.2	1.073	1.332	1.715	1.063	1.332	0.022	1.059	1.332	0.024
5	0.3	1.040	1.332	1.715	1.035	1.332	0.023	1.033	1.332	0.021
5	0.4	1.034	1.332	1.438	1.032	1.300	0.023	1.028	1.300	0.022
5	0.5	1.024	1.209	1.595	1.020	1.206	0.021	1.018	1.168	0.023
5	1.0	1.004	1.089	1.315	1.002	1.048	0.023	1.002	1.048	0.022
8	0.0	1.254	2.043	2.494	1.239	1.938	0.024	1.202	1.938	0.022
8	0.1	1.156	1.928	2.116	1.115	1.457	0.023	1.094	1.457	0.023
8	0.2	1.120	1.933	1.908	1.077	1.318	0.021	1.057	1.294	0.024
8	0.3	1.072	1.515	1.773	1.043	1.318	0.023	1.029	1.187	0.021
8	0.4	1.056	1.351	1.767	1.032	1.205	0.024	1.023	1.156	0.020
8	0.5	1.057	1.469	1.665	1.023	1.156	0.020	1.019	1.156	0.023
8	1.0	1.003	1.145	1.361	1.002	1.112	0.011	1.000	1.025	0.013
10	0.0	1.235	2.121	2.136	1.221	1.970	0.012	1.173	1.970	0.014
10	0.1	1.168	1.621	2.136	1.127	1.495	0.011	1.094	1.369	0.014
10	0.2	1.143	1.764	1.843	1.069	1.338	0.010	1.054	1.277	0.013
10	0.3	1.110	1.906	1.803	1.031	1.242	0.014	1.029	1.242	0.012
10	0.4	1.065	1.489	1.803	1.025	1.253	0.012	1.020	1.242	0.018
10	0.5	1.056	1.591	1.535	1.022	1.247	0.022	1.016	1.242	0.023
10	1.0	1.005	1.149	1.455	1.001	1.020	0.022	1.001	1.020	0.022
12	0.0	1.242	1.611	2.129	1.226	1.558	0.024	1.155	1.517	0.029
12	0.1	1.200	1.536	2.129	1.115	1.542	0.023	1.072	1.300	0.024
12	0.2	1.136	1.407	1.707	1.073	1.408	0.024	1.041	1.246	0.024
12	0.3	1.092	1.398	1.601	1.056	1.286	0.022	1.029	1.171	0.023
12	0.4	1.068	1.272	1.525	1.038	1.254	0.022	1.020	1.133	0.024
12	0.5	1.044	1.312	1.508	1.016	1.139	0.022	1.009	1.087	0.025
12	1.0	1.001	1.052	1.346	1.001	1.033	0.023	1.001	1.030	0.022
15	0.0	1.282	1.763	2.179	1.245	1.745	0.025	1.157	1.745	0.032
15	0.1	1.204	1.705	2.037	1.126	1.298	0.025	1.077	1.298	0.030
15	0.2	1.138	1.618	1.877	1.078	1.282	0.025	1.047	1.212	0.024
15	0.3	1.141	1.794	1.802	1.056	1.282	0.024	1.031	1.148	0.025
15	0.4	1.099	1.556	1.711	1.034	1.159	0.024	1.020	1.105	0.026
15	0.5	1.067	1.345	1.708	1.024	1.133	0.025	1.012	1.092	0.024
15	1.0	1.012	1.151	1.385	1.004	1.046	0.023	1.003	1.046	0.023

net		BMST_G			BKST			
size	ϵ	ave	max	cpu	min	ave	max	cpu
5	0.0	1.150	1.854	0.219	0.802	0.953	1.366	0.308
5	0.1	1.088	1.514	0.221	0.802	0.934	1.125	0.303
5	0.2	1.059	1.332	0.224	0.802	0.925	1.114	0.307
5	0.3	1.033	1.332	0.226	0.802	0.923	1.114	0.307
5	0.4	1.028	1.300	0.202	0.802	0.924	1.114	0.306
5	0.5	1.018	1.168	0.220	0.802	0.924	1.114	0.308
5	1.0	1.002	1.048	0.218	0.802	0.914	1.000	0.303
8	0.0	1.202	1.938	0.221	0.864	0.986	1.205	0.307
8	0.1	1.094	1.457	0.227	0.853	0.982	1.205	0.308
8	0.2	1.057	1.294	0.225	0.853	0.959	1.111	0.307
8	0.3	1.029	1.187	0.221	0.853	0.951	1.111	0.307
8	0.4	1.023	1.156	0.219	0.840	0.945	1.111	0.307
8	0.5	1.019	1.156	0.216	0.840	0.944	1.111	0.306
8	1.0	1.000	1.025	0.211	0.840	0.934	1.025	0.306
10	0.0	1.169	1.970	0.249	0.861	1.009	1.224	0.311
10	0.1	1.092	1.369	0.239	0.861	0.983	1.224	0.312
10	0.2	1.052	1.277	0.231	0.861	0.946	1.136	0.312
10	0.3	1.029	1.242	0.223	0.861	0.941	1.136	0.314
10	0.4	1.019	1.242	0.229	0.861	0.939	1.136	0.313
10	0.5	1.015	1.242	0.228	0.861	0.937	1.136	0.312
10	1.0	1.001	1.016	0.218	0.861	0.926	1.007	0.310
12	0.0	1.149	1.517	0.818	0.829	1.002	1.217	0.314
12	0.1	1.072	1.300	0.313	0.829	0.980	1.256	0.314
12	0.2	1.041	1.246	0.292	0.829	0.956	1.135	0.314
12	0.3	1.029	1.171	0.240	0.829	0.947	1.129	0.315
12	0.4	1.020	1.133	0.218	0.829	0.937	1.033	0.314
12	0.5	1.009	1.087	0.222	0.829	0.932	1.028	0.314
12	1.0	1.001	1.030	0.218	0.829	0.927	0.976	0.316
15	0.0	1.148	1.686	10.465	0.926	1.032	1.266	0.326
15	0.1	1.073	1.298	22.490	0.872	0.985	1.245	0.324
15	0.2	1.043	1.212	8.518	0.872	0.971	1.245	0.329
15	0.3	1.028	1.148	4.411	0.846	0.955	1.245	0.323
15	0.4	1.018	1.092	0.683	0.846	0.938	1.044	0.323
15	0.5	1.012	1.092	0.634	0.846	0.939	1.120	0.321
15	1.0	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 $\epsilon = 0.0$ 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 4: The Ratio of the Routing Cost over MST

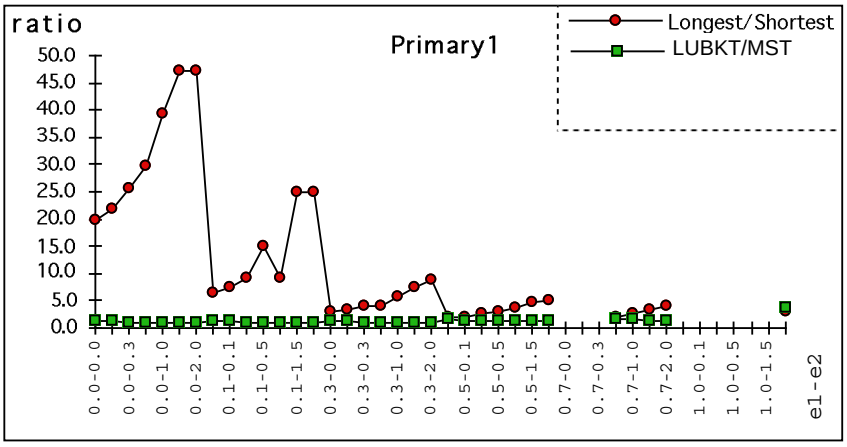

Figure 12: The ratio of Longest Path Length over Shortest Path Length and the ratio of Routing Cost over MST

8 Conclusion

We have presented bounded path length minimal spanning tree schemes which can control longest/shortest path length and routing cost. With upper bound, our method achieves smaller cost than that of BPRIM and BRBC. However, it is possible that even the optimal method can generate almost $N \cdot cost(MST)$ where N is the number of sinks in Figure 13 (p1 case).

We presented two optimal algorithms: BMST_G using the Gabow's method, BKEX using the negative T-exchange technique. When the number of sink is less than 15, as is often the case in a global routing problem, we have shown in the expriemental results that the optimal algorithms can be used. We also compared our heuristic solutions against optimal solutions to show their effectiveness.

The BKRUS solution is further enhanced with the use of BKH2 as a post processing. We also presented BKRUS method can be extended to Elmore delay model, to Steiner trees with bounded path length and to lower/upper bounded path length spanning trees. 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.

Acknowledgment

The authors would like to thank Professor Andrew Kahng and Kenneth Boese for providing us with the benchmark data and the BPRIM and BRBC programs.

References

[1] J. B. Kruskal, "On 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, et al., "Provably Good Performance-Driven Global Routing," *IEEE Transactions on Computer Aided Design* , Vol. 11, NO. 6, pp. 739-752, June, 1992.

		p1		p2		p3		p4		pr1		pr2	
$\epsilon 1$	$\epsilon 2$	s	r	s	r	s	r	s	r	s	r	s	r
0.0	0.0	1.0	3.9	2.0	2.7	2.0	1.3	1.8	1.7	19.8	1.4	48.4	1.4
0.0	0.1	1.1	1.7	2.2	1.3	1.9	1.4	2.0	1.6	21.8	1.2	53.2	1.2
0.0	0.3	1.2	1.0	1.6	1.2	1.7	1.3	2.3	1.3	25.7	1.1	62.9	1.1
0.0	0.5	1.2	1.0	2.8	1.0	2.1	1.2	2.6	1.2	29.7	1.1	72.6	1.1
0.0	1.0	1.2	1.0	2.8	1.0	4.8	1.0	3.4	1.1	39.4	1.0	96.2	1.0
0.0	1.5	1.2	1.0	2.8	1.0	4.8	1.0	3.5	1.1	47.4	1.0	119.7	1.0
0.0	2.0	1.2	1.0	2.8	1.0	4.8	1.0	5.2	1.0	47.4	1.0	133.9	1.0
0.1	0.0	1.0	3.9	2.0	2.7	2.0	1.3	1.8	1.7	6.6	1.4	5.6	1.4
0.1	0.1	1.1	1.7	2.2	1.3	1.9	1.4	2.0	1.6	7.7	1.2	6.0	1.2
0.1	0.3	1.2	1.0	1.6	1.2	1.7	1.3	2.3	1.3	9.1	1.1	6.4	1.1
0.1	0.5	1.2	1.0	2.8	1.0	2.1	1.2	2.6	1.2	15.0	1.1	5.2	1.1
0.1	1.0	1.2	1.0	2.8	1.0	4.8	1.0	3.4	1.1	9.2	1.0	7.9	1.0
0.1	1.5	1.2	1.0	2.8	1.0	4.8	1.0	3.5	1.1	24.9	1.0	9.8	1.1
0.1	2.0	1.2	1.0	2.8	1.0	4.8	1.0	5.2	1.0	24.9	1.0	23.9	1.0
0.3	0.0	1.0	3.9	2.0	2.7	2.0	1.3	1.8	1.7	3.1	1.4	3.3	1.5
0.3	0.1	1.1	1.7	2.2	1.3	1.9	1.4	2.0	1.6	3,5	1.3	3.7	1.3
0.3	0.3	1.2	1.0	1.6	1.2	1.7	1.3	2.3	1.3	4.2	1.1	4.3	1.2
0.3	0.5	1.2	1.0	2.8	1.0	2.1	1.2	2.6	1.2	4.2	1.1	4.2	1.2
0.3	1.0	1.2	1.0	2.8	1.0	4.8	1.0	3.4	1.1	5.9	1.1	6.1	1.1
0.3	1.5	1.2	1.0	2.8	1.0	4.8	1.0	3.5	1.1	7.6	1.1	8.1	1.1
0.3	2.0	1.2	1.0	2.8	1.0	4.8	1.0	5.2	1.0	8.8	1.1	9.7	1.1
0.5	0.0	1.0	3.9	-	-	2.0	1.3	1.8	1.7	2.0	1.7	2.0	2.1
0.5	0.1	1.1	1.7	-	-	1.9	1.4	2.0	1.6	2.2	1.5	2.2	1.7
0.5	0.3	1.2	1.0	1.6	1.2	1.7	1.3	2.3	1.3	2.6	1.4	2.6	1.6
0.5	0.5	1.2	1.0	2.8	1.2	2.1	1.2	2.6	1.2	3.0	1.3	3.0	1.6
0.5	1.0	1.2	1.0	2.8	1.2	3.6	1.1	3.4	1.1	3.9	1.2	3.9	1.5
0.5	1.5	1.2	1.0	2.8	1.2	3.6	1.1	3.5	1.1	4.9	1.2	4.5	1.5
0.5	2.0	1.2	1.0	2.8	1.2	3.6	1.1	5.2	1.0	5.1	1.2	5.8	1.5
0.7	0.0	1.0	3.9	-	-	-	-	-	-	-	-	-	-
0.7	0.1	1.1	1.7	-	-	-	-	-	-	-	-	-	-
0.7	0.3	1.2	1.0	1.6	1.2	1.7	1.4	1.6	1.6	-	-	-	-
0.7	0.5	1.2	1.0	2.8	1.2	1.8	1.4	1.4	1.4	2.1	1.8	2.1	2.4
0.7	1.0	1.2	1.0	2.8	1.2	2.7	1.3	1.2	1.2	2.8	1.6	2.9	2.1
0.7	1.5	1.2	1.0	2.8	1.2	2.7	1.3	1.1	1.1	3.4	1.5	3.5	2.0
0.7	2.0	1.2	1.0	2.8	1.2	2.7	1.3	1.1	1.1	4.2	1.5	4.0	2.0
1.0	0.0	-	-	-	-	-	-	-	-	-	-	-	-
1.0	0.1	-	-	-	-	-	-	-	-	-	-	-	-
1.0	0.3	1.3	1.1	-	-	-	-	-	-	-	-	-	-
1.0	0.5	1.3	1.1	-	-	-	-	-	-	-	-	-	-
1.0	1.0	1.3	1.1	2.8	1.4	1.8	2.3	-	-	-	-	-	-
1.0	1.5	1.3	1.1	2.8	1.4	2.2	2.3	-	-	-	-	-	-
1.0	2.0	1.3	1.1	2.8	1.4	2.2	2.3	2.7	2.7	3.0	3.7	3.0	5.6

		r1		r2		r3		r4		r5	
0.0	0.0	45.7	1.6	51.8	1.3	56.9	1.4	196.7	1.2	199.2	1.3
0.0	0.1	50.2	1.2	56.9	1.2	62.5	1.2	216.1	1.1	219.1	1.1
0.0	0.3	59.0	1.1	67.1	1.1	73.7	1.1	255.6	1.0	259.0	1.0
0.0	0.5	68.3	1.1	77.6	1.0	84.6	1.0	294.8	1.0	298.8	1.0
0.0	1.0	86.7	1.0	103.2	1.0	113.5	1.0	392.5	1.0	398.3	1.0
0.0	1.5	97.3	1.0	114.5	1.0	125.2	1.0	491.2	1.0	488.2	1.0
0.0	2.0	97.3	1.0	114.5	1.0	160.2	1.0	578.3	1.0	590.2	1.0
0.1	0.0	6.0	1.6	8.9	1.3	7.1	1.4	9.3	1.2	5.0	1.3
0.1	0.1	8.5	1.3	4.0	1.2	8.1	1.2	10.7	1.1	10.9	1.1
0.1	0.3	11.6	1.1	7.1	1.1	6.8	1.1	7.1	1.1	6.6	1.1
0.1	0.5	8.9	1.1	13.3	1.1	9.4	1.1	7.4	1.0	4.9	1.0
0.1	1.0	11.7	1.0	17.0	1.0	19.1	1.0	18.5	1.0	10.1	1.0
0.1	1.5	19.6	1.0	23.6	1.0	7.5	1.0	23.2	1.0	10.2	1.0
0.1	2.0	19.6	1.0	23.6	1.0	11.5	1.0	26.5	1.0	27.3	1.0
0.3	0.0	3.1	1.7	3.3	1.5	3.0	1.5	3.3	1.5	3.1	1.5
0.3	0.1	3.4	1.4	3.5	1.4	3.6	1.3	3.6	1.4	3,4	1.3
0.3	0.3	4.2	1.3	4.1	1.3	4.3	1.2	4.1	1.3	3.3	1.3
0.3	0.5	4.8	1.2	5.0	1.2	4.9	1.2	4.9	1.3	4.4	1.3
0.3	1.0	6.0	1.2	6.2	1.2	6.6	1.2	6.5	1.3	6.7	1.3
0.3	1.5	7.2	1.2	8.3	1.2	6.3	1.2	8.1	1.3	8.3	1.2
0.3	2.0	8.8	1.2	8.7	1.2	9.1	1.1	7.0	1.3	8.7	1.2
0.5	0.0	-	-	2.0	2.3	-	-	-	-	-	-
0.5	0.1	2.2	1.9	2.2	1.9	2.2	1.7	2.2	2.0	2.2	2.1
0.5	0.3	2.6	1.7	2.5	1.8	2.6	1.6	2.6	1.9	2.6	1.8
0.5	0.5	2.8	1.7	3.0	1.8	3.0	1.6	3.0	1.9	3.0	1.8
0.5	1.0	4.0	1.5	4.0	1.7	4.0	1.5	4.0	1.8	4.0	1.8
0.5	1.5	5.0	1.5	4.8	1.7	4.9	1.5	5.0	1.8	5.0	1.8
0.5	2.0	5.4	1.5	6.0	1.7	5.9	1.5	5.8	1.8	5.8	1.8
0.7	0.0	-	-	-	-	-	-	-	-	-	-
0.7	0.1	-	-	-	-	-	-	-	-	-	-
0.7	0.3	-	-	-	-	-	-	-	-	-	-
0.7	0.5	2.1	2.2	2.1	2.8	2.1	2.6	2.1	3.1	2.1	3.1
0.7	1.0	2.7	1.9	2.8	2.5	2.8	2.3	2.8	2.8	2.9	2.9
0.7	1.5	3.5	1.9	3.6	2.4	3.5	2.3	3.5	2.8	3.6	2.9
0.7	2.0	4.2	1.9	4.1	2.4	4.3	2.3	4.3	2.8	4.3	2.9
1.0	0.0	-	-	-	-	-	-	-	-	-	-
1.0	0.1	-	-	-	-	-	-	-	-	-	-
1.0	0.3	-	-	-	-	-	-	-	-	-	-
1.0	0.5	-	-	-	-	-	-	-	-	-	-
1.0	1.0	-	-	-	-	-	-	-	-	-	-
1.0	1.5	-	-	-	-	-	-	-	-	-	-
1.0	2.0	3.0	4.0	3.0	4.6	3.0	4.8	3.0	8.1	3.0	8.9

s: the ratio of longest path length over shortest path length (zero clock skew: $s = 1.0$)

r: the ratio of routing cost over MST, -: infeasible configurations

Table 5: The Lower and Upper Bounded BKRUS Results

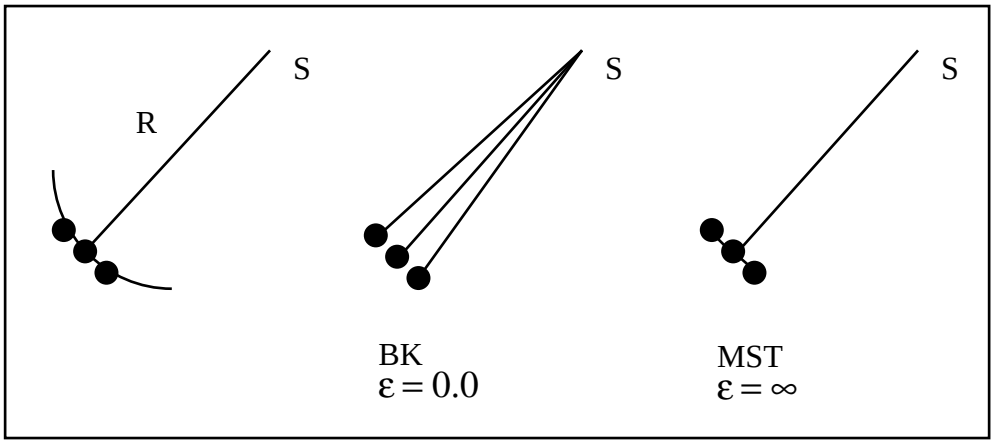

Figure 13: Example where $cost(BKT)/cost(MST)$ can be N where N is number of sinks

[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] A. Kahng, J. Cong, and G. Robins, "High-performance clock routing based on recursive geometric matching," *28th Design Automation Conference* , pp. 322-327, 1991.

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

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

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

[8] 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.

[9] 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.

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

[11] Jason Cong and Cheng-Koh Koh, "Minimum-Cost Bounded-Skew Clock Routing," *International Symposium on Circuits and Systems*, pp. 215-218, 1995.

[12] Dennis J.-H. Huang, Andrew B. Kahng and Chung-Wen Albert Tsao, "On the Bounded-Skew Clock and Steiner Routing Problems," *Proc. ACM/IEEE Design Automation Conference*, pp. 508-513, June, 1995.

[13] Jason Cong, Andrew B. Kahng, Cheng-Koh Koh and C.-W. Albert Tsao, "Bounded-Skew Clock and Steiner Routing Under Elmore Delay," *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 66-71, November, 1995.

[14] Iksoo Pyo, Jaewon Oh and Massoud Pedram, "Constructing Minimal Spanning Trees with Bounded Path Length," *Proc. SIGDA/ACM European Design and Test Conference*, pp. 244-249, March, 1996.