Constructing Minimal Spanning Trees With Bounded Path Length

Iksoo Pyo, Jaewon Oh and Massoud Pedram

CENG Technical Report 94-35

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

October 1994

Constructing Minimal Spanning Trees with Bounded Path Length

Iksoo Pyo, Jaewon Oh and Massoud Pedram
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089

October 15, 1994

Abstract

This report 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 $1.35 \cdot cost(BMST^*)$ 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). Extension of these techniques to the construction of MSTs with lower and upper bounded path lengths is presented as well. Empirical results demonstrate the effectiveness of these algorithms on a large benchmark set.

Contents

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. In this report, we also use wire length to approximate interconnection delay during the construction of routing trees. In practice, a subsequent iterative improvement step, based on a more accurate RC delay model, may be used to enhance the routing solution.

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.

In this report, 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]. Furthermore, the tree cost is empirically observed to be at most 1.35 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

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

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

The remainder of this report 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 presents benchmark results and comparisons. Section 7 describes extension to the lower bounded path length constraints and Section 8 concludes the report.

2 Background

On a Euclidean 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 6). 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 Two Heuristics: 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 $G(V, E)$, 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 Euclidean distances between every pair of nodes, i.e. $D[x,y] = dist_G(x,y)$. This matrix is computed from the coordinates of nodes in the Euclidean space. 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 at the beginning of the tree construction process. As the tree grows, P and r are updated by the Merge routine given below:[2]

// 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 1 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 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 $O(V^2)$.

[2] P and r are already calculated for t_u and t_v.

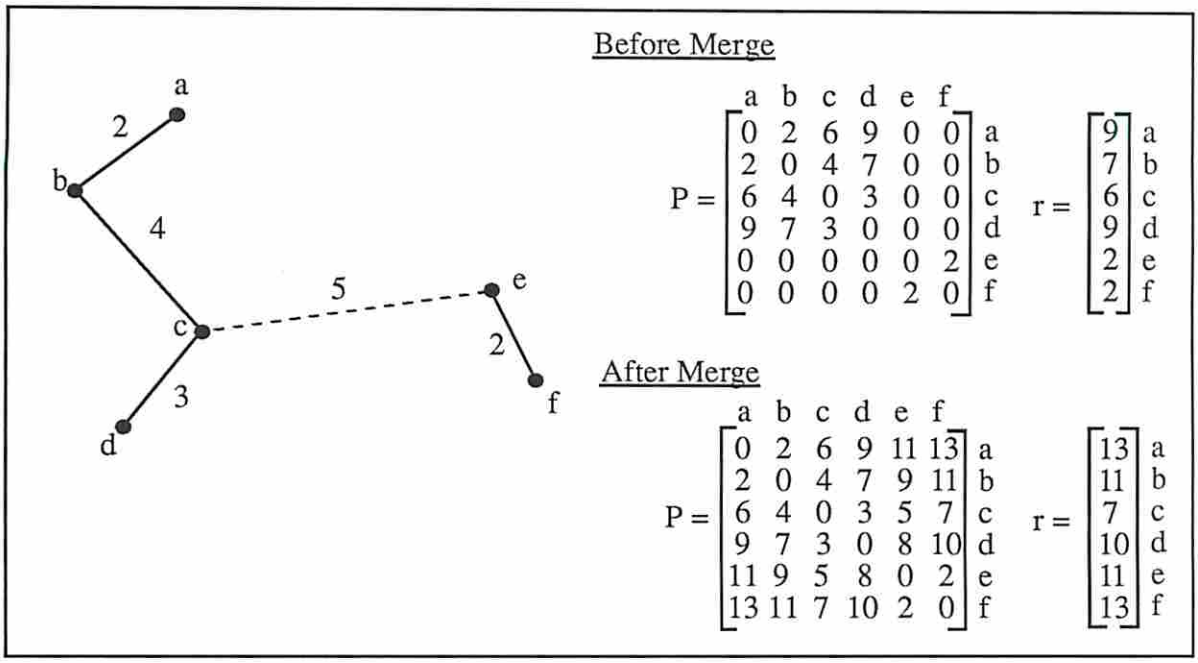

Figure 1: Example of Merging Two Partial Trees

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

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

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 **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 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.

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 $dist_{t_u}$(S,u) + $dist_G(u, v)$ + $radius_{t_v}(v)$ > 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

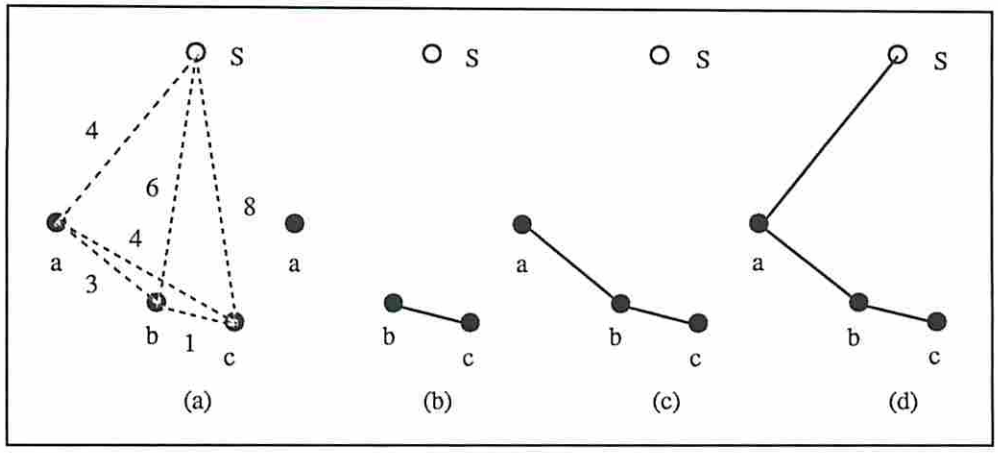

Figure 2: BKRUS Example

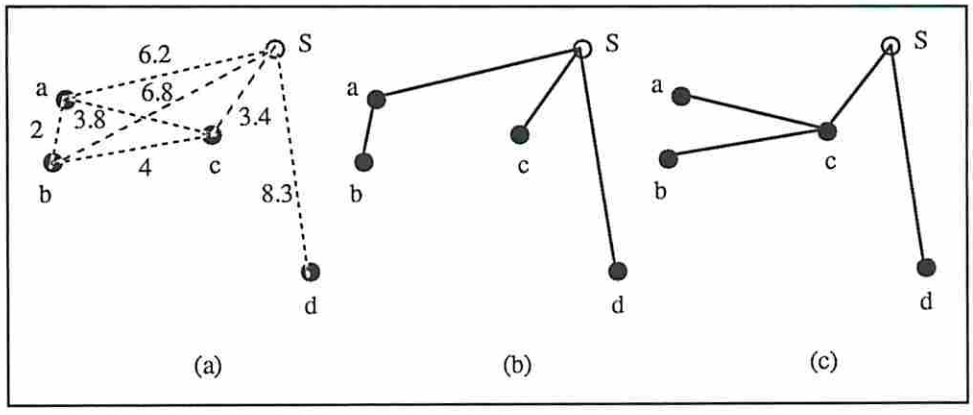

Figure 3: Example where BKRUS can not generate an optimal solution

$\notin t_u$ or t_v, then the edge (u, v) was rejected because for all $x \in t_u$, $dist_G$(S,x) + $dist_{t_u}(x, u)$ + $dist_G(u, v)$ + $radius_{t_v}(v)$ > 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 x is still not feasible in the new merged tree). Then $dist_G$(S,y) + $dist_{t_u}(y, u)$ + $dist_G(u, v)$ + $radius_{t_v}(v) \leq$ upper_bound. Note that in the path from y to u, there is an x. So the above inequality can be rewritten as $dist_G$(S,y) + $dist_{t_u}(y, x)$ + $dist_{t_u}(x, u)$ + $dist_G(u, v)$ + $radius_{t_v}(v) \leq$ upper_bound. However, we have $dist_G$(S,y) + $dist_{t_u}(y, x)$ > $dist_G$(S,x) by the triangular 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 3

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 rejected a-b, we could generate 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.

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 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[3]. 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). 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 have been able to somewhat reduce the space and time complexities by the following rules.

Lemma 4.1 *Let S, a and b be vertices of a triangle. For every triangle including source, if there are edge(S, a) and (S, b) and weight(a, b) > weight(S, a), weight(S, b), then eliminate edge(a, b).*

Lemma 4.2 *Eliminate* $\forall$ *edge* $\in E$ *if edge cannot be fesible.*

Lemma 4.3 *Include edge(S, i)* $\in E$, *for* $\forall i$, *if edge(S, i) is only possible.*

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). Lemma 4.2 is self-explanatory. 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 upperbound. Using these rules, we have used Gabow's algorithm on trees with as many as 15 sinks.

[3] We actually use the right time complexity instead of Gabow's $O(KE\alpha(E,V))$.

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 4 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 DFS_EXCHANGE (T, weight_sum)
1 FA ← make_father_array(T)
2 for each edge (x,y) in G − T do
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
```

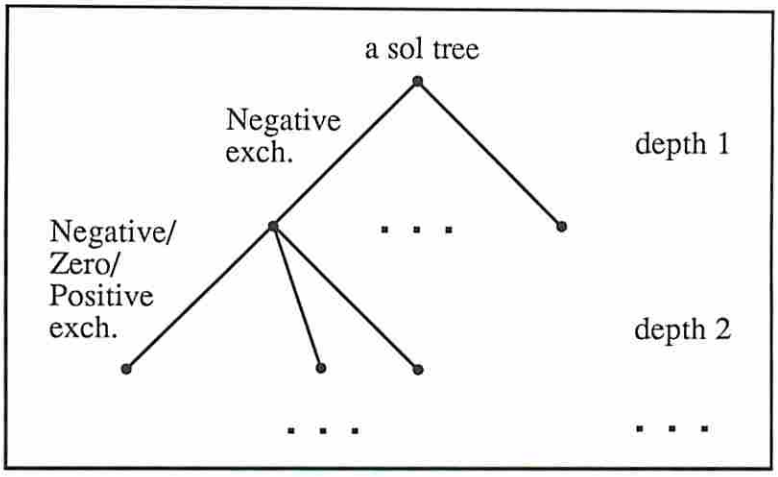

Figure 4: BKDFS Negative-sum-Exchange Search Tree

```
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

// main loop of BKEX //
Algorithm BKEX(G)
1 T ← BKRUS(G)
2 while DFS_EXCHANGE(T,0) do
```

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 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

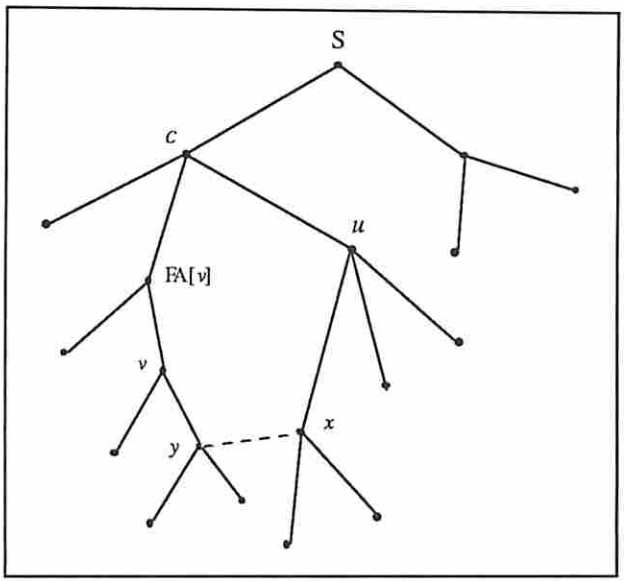

Figure 5: Example of finding T-exchanges

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 5 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 5, 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.

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 $O(E^2V^3)$.

6 Experimental Results

We implemented BKRUS, BMST_G, BKEX, and BKH2 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 11, but p2 has one more sink between the source and the group of sinks. Benchmark p3 has the same configuration as that of Figure 6. Benchmark p4 has the same configuration as that of Figure 11, but sinks are scattered around a circle of diameter 20. Without loss of generality, we add one more node as the source to the r* and primary* benchmarks because they did not come with a source. Description of all the benchmarks is given in Table 1.

BPRIM generates $4.2 \cdot cost(MST)$ and $3.3 \cdot cost(BKT)$ for benchmark p3 as shown in Figure 6.

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)[4]. The results show that the performance ratio of BKT over MST is at most 1.66 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]The runtime for BPRIM was surprisingly worse than that for BKRUS despite its better time complexity. This is likely due to the difference in our implementation style and data structures and so we decided not to include their runtimes here.

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	384.0	19.4
pr2	604	182106	693.7	14.3
r1	268	35778	41774.1	914.7
r2	599	179101	61247.4	1182.8
r3	863	371953	60725.3	1067.9
r4	1904	1811656	87933.8	227.2
r5	3102	4809651	97843.0	491.1

R: length of the shortest path from source to the farthest sink in G
r: length of the shortest path from source to the nearest sink in G

Table 1: Characteristics of Benchmarks

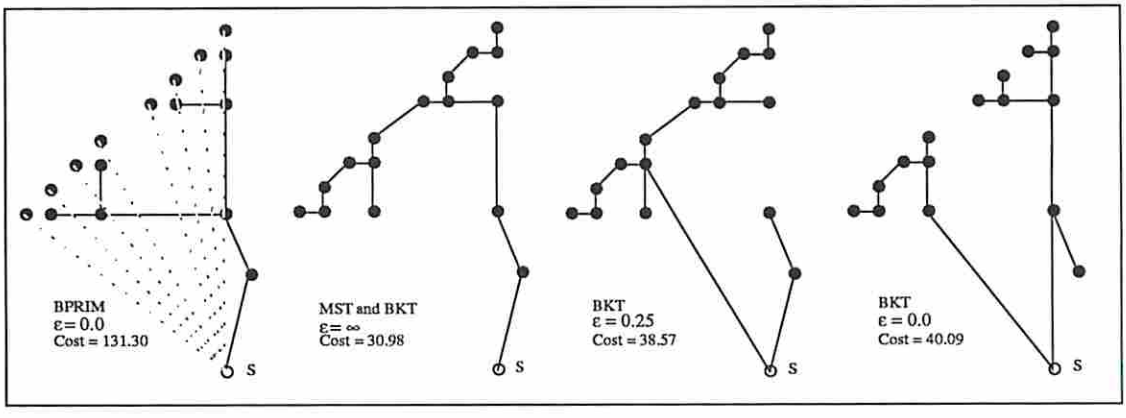

Figure 6: Example where the performance ratio of BRPIM is not bounded for any ϵ

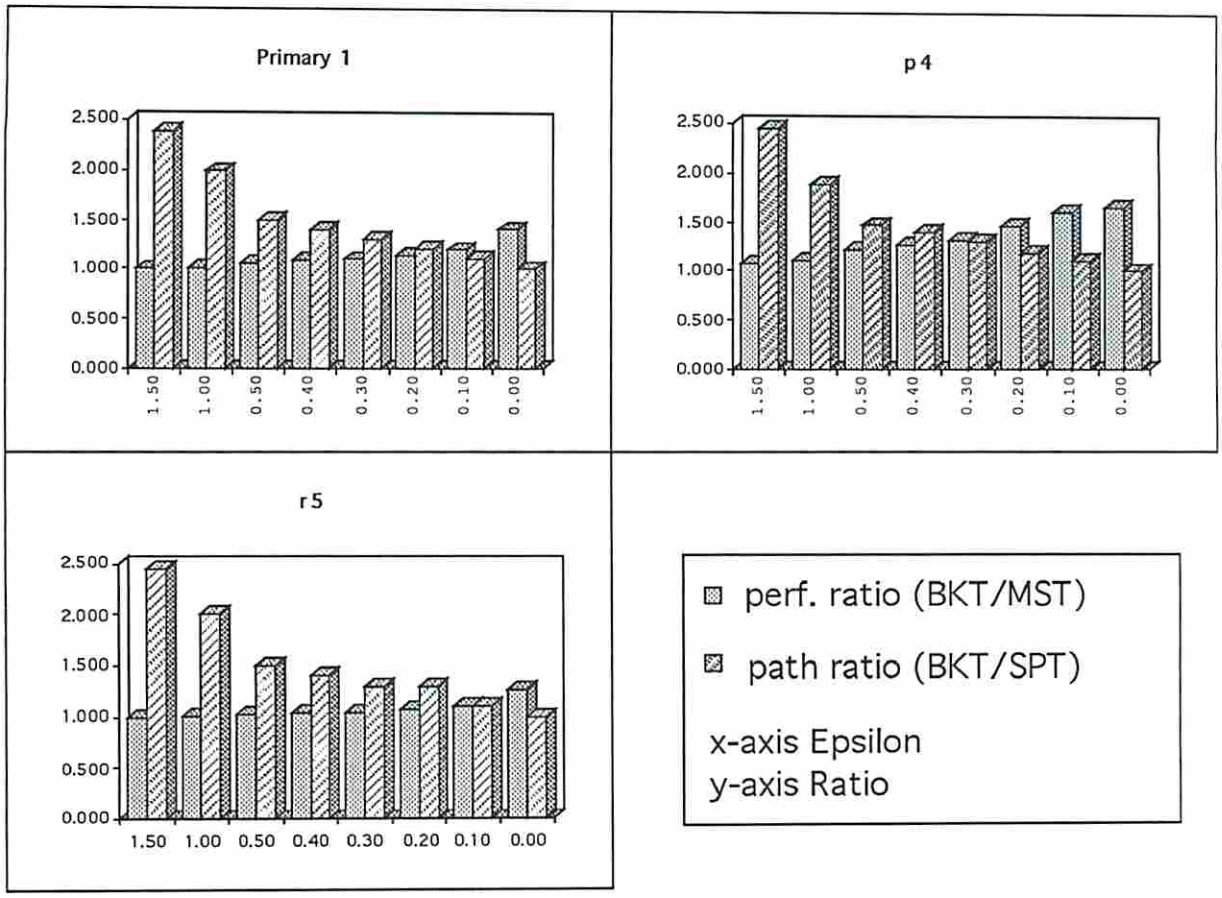

Figure 7: Tradeoff Curve

(4) benchmarks, the comparison of BPRIM, BRBC, BKRUS, BKH2 and BMST_G in terms of routing 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. Also, the longest path length comparison of them is shown in Table 5. 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 7.

In Figure 8, 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 $cost(BKH2)/cost(BKEX)$

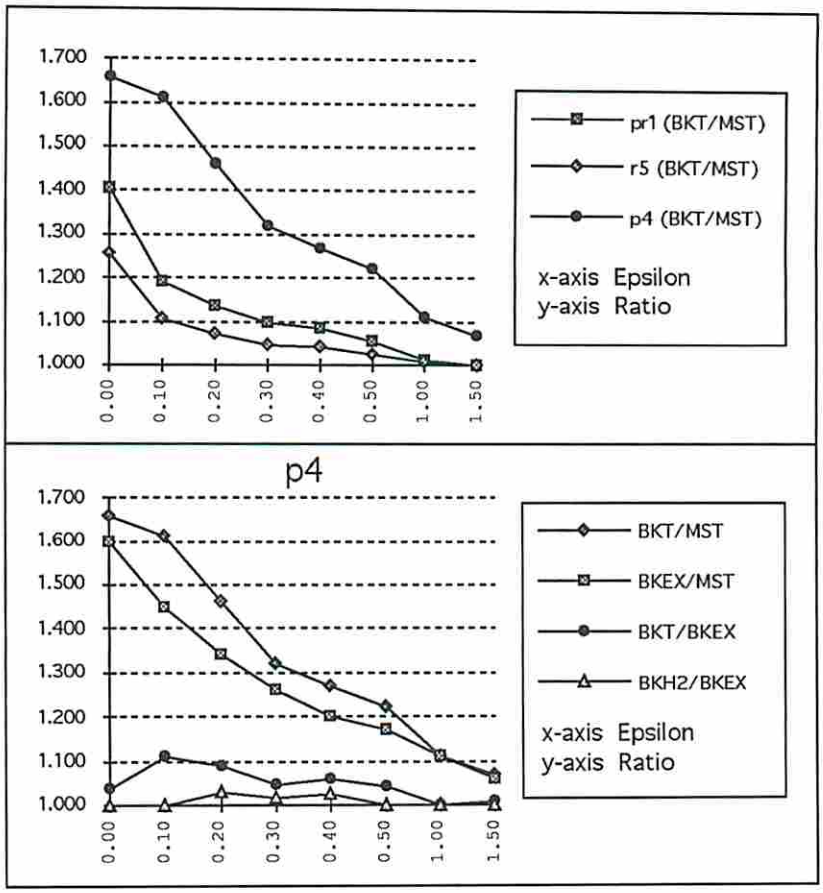

Figure 8: Ratio Curve

respectively.

From these results, the various BMST methods can be ordered by their routing costs as shown in Figure 9. This chart shows the average relative position.

7 Results of 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.

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.

bench	ϵ	BMST_G path ratio	BMST_G perf. ratio	BMST_G cpu	BKEX path ratio	BKEX perf. ratio	BKEX cpu	BKRUS path ratio	BKRUS perf. ratio	BKRUS cpu	BKH2 path ratio	BKH2 perf. ratio	BKH2 cpu	BPRIM path ratio	BPRIM 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	∞	2.39	1.000	0.98	2.39	1.000	0.96	0.0
	1.50	2.39	1.000	0.95	2.39	1.000	0.96	0.0
	1.00	1.99	1.012	0.96	1.97	1.001	1.63	1.1
	0.50	1.50	1.057	1.10	1.47	1.006	221.91	4.8
	0.40	1.40	1.085	1.03	1.40	1.010	682.49	6.9
	0.30	1.30	1.098	0.97	1.30	1.009	1436.33	8.1
	0.20	1.20	1.135	0.98	1.19	1.023	3714.85	9.9
	0.10	1.10	1.193	0.97	1.10	1.045	11068.07	12.4
	0.00	1.00	1.404	0.99	1.00	1.147	25243.41	18.3
pr2	∞	2.76	1.000	5.54	2.76	1.000	5.84	0.0
	1.50	2.47	1.007	5.51	2.49	1.000	6.79	0.7
	1.00	1.99	1.036	5.58	1.97	1.000	283.78	3.5
	0.50	1.50	1.066	5.59	1.49	1.013	55388.49	5.0
	0.40	1.40	1.064	5.60	1.40	1.034	45091.41	2.8
	0.30	1.30	1.085	5.63	1.30	1.044	52022.80	3.8
	0.20	1.20	1.130	6.00	1.20	1.079	45923.72	4.5
	0.10	1.10	1.176	5.89	1.10	1.135	47421.23	3.5
	0.00	1.00	1.431	5.80	1.00	1.385	45255.12	3.2
r1	∞	2.13	1.000	1.16	2.13	1.000	1.20	0.0
	1.50	2.13	1.000	1.15	2.13	1.000	1.17	0.0
	1.00	1.90	1.001	1.14	1.90	1.001	1.21	0.0
	0.50	1.50	1.060	1.14	1.50	1.006	102.80	5.1
	0.40	1.40	1.086	1.15	1.40	1.023	2456.76	5.8
	0.30	1.29	1.090	1.15	1.30	1.028	755.76	5.7
	0.20	1.19	1.163	1.18	1.20	1.031	3714.26	11.3
	0.10	1.10	1.235	1.24	1.10	1.088	17543.58	11.9
	0.00	1.00	1.557	1.18	1.00	1.243	44812.95	20.2
r2	∞	2.21	1.000	7.09	2.21	1.000	7.44	0.0
	1.50	2.21	1.000	7.10	2.21	1.000	7.46	0.0
	1.00	1.99	1.003	7.13	1.98	1.000	11.85	0.3
	0.50	1.50	1.034	7.34	1.50	1.006	6869.85	2.7
	0.40	1.40	1.050	7.74	1.39	1.009	18979.52	3.9
	0.30	1.30	1.075	7.87	1.30	1.071	47554.84	0.4
	0.20	1.20	1.101	7.30	1.20	1.052	45385.07	4.5
	0.10	1.10	1.159	7.63	1.10	1.104	45701.29	4.7
	0.00	1.00	1.272	7.53	1.00	1.264	47675.82	0.6
r3	∞	2.82	1.000	15.97	2.81	1.000	16.37	0.0
	1.50	2.20	1.000	15.93	2.20	1.000	17.46	0.0
	1.00	2.00	1.001	15.99	1.94	1.000	22.41	0.1
	0.50	1.49	1.029	16.06	1.50	1.004	6120.54	2.4
	0.40	1.40	1.048	15.96	1.40	1.033	46523.45	1.4
	0.30	1.30	1.067	16.01	1.30	1.040	49079.06	2.5
	0.20	1.20	1.092	15.97	1.20	1.088	46200.38	0.4
	0.10	1.10	1.170	15.97	1.10	1.170	44926.20	0.0
	0.00	1.00	1.376	15.97	1.00	1.375	47149.83	0.1
r4	∞	3.84	1.000	97.85	3.84	1.000	105.53	0.0
	1.50	2.50	1.011	97.91	2.50	1.000	1577.44	1.1
	1.00	2.00	1.007	97.69	1.99	1.001	16349.39	0.6
	0.50	1.50	1.027	104.84	1.50	1.027	43380.01	0.0
	0.40	1.40	1.036	107.60	1.40	1.036	43407.99	0.0
	0.30	1.30	1.040	108.03	1.30	1.040	48097.32	0.0
	0.20	1.20	1.082	115.45	1.20	1.082	49435.97	0.0
	0.10	1.10	1.124	113.85	1.10	1.123	46073.56	0.1
	0.00	1.00	1.223	105.78	1.00	1.223	45045.10	0.0
r5	∞	3.14	1.000	257.96	3.14	1.000	261.72	0.0
	1.50	2.45	1.000	257.78	2.45	1.000	285.45	0.0
	1.00	2.00	1.008	257.59	2.00	1.008	46185.97	0.0
	0.50	1.50	1.024	297.83	1.50	1.024	46070.31	0.0
	0.40	1.40	1.043	306.85	1.40	1.043	47657.18	0.0
	0.30	1.30	1.048	336.82	1.30	1.048	44185.97	0.0
	0.20	1.20	1.071	320.63	1.20	1.071	46412.70	0.0
	0.10	1.10	1.108	320.13	1.10	1.108	43297.40	0.0
	0.00	1.00	1.257	306.86	1.00	1.257	48552.52	0.0

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				BMST_G			
size	ϵ	min	ave	max	max	min	ave	max	cpu	min	ave	max	cpu	min	ave	max	cpu
5	0.0	1.000	1.349	1.945	2.095	1.000	1.341	1.875	0.036	1.000	1.340	1.875	0.036	1.000	1.340	1.875	0.317
5	0.1	1.000	1.189	1.945	2.095	1.000	1.168	1.534	0.036	1.000	1.166	1.534	0.039	1.000	1.166	1.534	0.313
5	0.2	1.000	1.118	1.512	1.733	1.000	1.099	1.491	0.029	1.000	1.093	1.491	0.037	1.000	1.093	1.491	0.314
5	0.3	1.000	1.070	1.491	1.707	1.000	1.059	1.325	0.032	1.000	1.051	1.325	0.034	1.000	1.051	1.325	0.310
5	0.4	1.000	1.041	1.302	1.595	1.000	1.034	1.210	0.029	1.000	1.028	1.176	0.040	1.000	1.028	1.176	0.306
5	0.5	1.000	1.033	1.315	1.717	1.000	1.020	1.176	0.030	1.000	1.018	1.176	0.033	1.000	1.018	1.176	0.311
5	0.6	1.000	1.021	1.315	1.437	1.000	1.015	1.176	0.031	1.000	1.015	1.176	0.037	1.000	1.015	1.176	0.311
5	0.7	1.000	1.025	1.356	1.443	1.000	1.011	1.120	0.028	1.000	1.010	1.120	0.038	1.000	1.010	1.120	0.313
5	0.8	1.000	1.024	1.356	1.502	1.000	1.008	1.120	0.030	1.000	1.008	1.120	0.037	1.000	1.008	1.120	0.314
5	0.9	1.000	1.010	1.342	1.502	1.000	1.002	1.085	0.034	1.000	1.002	1.058	0.035	1.000	1.002	1.058	0.305
5	1.0	1.000	1.003	1.162	1.365	1.000	1.002	1.085	0.031	1.000	1.001	1.058	0.037	1.000	1.001	1.058	0.309
8	0.0	1.105	1.466	2.083	2.519	1.105	1.436	2.056	0.039	1.105	1.428	2.056	0.044	1.105	1.428	2.056	0.316
8	0.1	1.000	1.320	2.001	2.519	1.000	1.207	1.818	0.034	1.000	1.187	1.818	0.033	1.000	1.181	1.818	0.320
8	0.2	1.000	1.220	1.952	2.224	1.000	1.127	1.493	0.034	1.000	1.098	1.493	0.040	1.000	1.098	1.493	0.314
8	0.3	1.000	1.141	1.566	2.068	1.000	1.085	1.459	0.035	1.000	1.066	1.459	0.038	1.000	1.064	1.459	0.311
8	0.4	1.000	1.111	1.726	1.917	1.000	1.056	1.321	0.032	1.000	1.040	1.247	0.045	1.000	1.040	1.247	0.315
8	0.5	1.000	1.091	1.654	1.903	1.000	1.033	1.244	0.033	1.000	1.023	1.210	0.035	1.000	1.022	1.210	0.316
8	0.6	1.000	1.069	2.033	1.633	1.000	1.022	1.244	0.033	1.000	1.014	1.210	0.045	1.000	1.014	1.210	0.312
8	0.7	1.000	1.056	1.883	1.536	1.000	1.019	1.188	0.032	1.000	1.010	1.116	0.037	1.000	1.010	1.116	0.316
8	0.8	1.000	1.041	1.932	1.594	1.000	1.010	1.171	0.034	1.000	1.008	1.105	0.038	1.000	1.008	1.105	0.313
8	0.9	1.000	1.032	1.932	1.594	1.000	1.008	1.171	0.033	1.000	1.005	1.105	0.039	1.000	1.005	1.105	0.314
8	1.0	1.000	1.014	1.292	1.488	1.000	1.004	1.121	0.030	1.000	1.004	1.105	0.038	1.000	1.004	1.105	0.314
10	0.0	1.093	1.565	2.711	2.891	1.156	1.457	2.006	0.040	1.156	1.433	2.006	0.046	1.156	1.432	2.006	0.319
10	0.1	1.000	1.343	2.335	2.760	1.000	1.192	1.670	0.036	1.000	1.154	1.676	0.038	1.000	1.153	1.676	0.353
10	0.2	1.000	1.260	2.155	2.274	1.000	1.119	1.508	0.034	1.000	1.092	1.441	0.041	1.000	1.083	1.441	0.331
10	0.3	1.000	1.184	1.987	2.320	1.000	1.066	1.292	0.034	1.000	1.050	1.292	0.040	1.000	1.050	1.292	0.325
10	0.4	1.000	1.132	1.766	1.940	1.000	1.049	1.288	0.037	1.000	1.031	1.233	0.037	1.000	1.030	1.233	0.322
10	0.5	1.000	1.090	1.522	1.793	1.000	1.030	1.303	0.028	1.000	1.019	1.231	0.040	1.000	1.019	1.214	0.314
10	0.6	1.000	1.076	1.831	1.793	1.000	1.022	1.262	0.033	1.000	1.013	1.112	0.033	1.000	1.013	1.112	0.309
10	0.7	1.000	1.072	1.678	1.831	1.000	1.019	1.262	0.035	1.000	1.007	1.063	0.037	1.000	1.007	1.063	0.306
10	0.8	1.000	1.053	1.607	1.632	1.000	1.013	1.176	0.033	1.000	1.006	1.063	0.038	1.000	1.006	1.063	0.312
10	0.9	1.000	1.021	1.252	1.566	1.000	1.007	1.091	0.033	1.000	1.004	1.063	0.038	1.000	1.004	1.063	0.307
10	1.0	1.000	1.017	1.452	1.566	1.000	1.004	1.063	0.035	1.000	1.003	1.063	0.037	1.000	1.003	1.063	0.313
12	0.0	1.151	1.585	2.168	3.277	1.151	1.516	2.034	0.037	1.151	1.455	1.910	0.039	1.151	1.452	1.910	0.534
12	0.1	1.000	1.349	1.903	2.749	1.000	1.188	1.470	0.040	1.000	1.140	1.436	0.042	1.000	1.127	1.403	1.755
12	0.2	1.000	1.260	1.970	2.593	1.000	1.115	1.466	0.032	1.000	1.070	1.294	0.043	1.000	1.067	1.294	0.475
12	0.3	1.000	1.179	1.763	2.221	1.000	1.055	1.254	0.032	1.000	1.034	1.166	0.040	1.000	1.032	1.156	0.364
12	0.4	1.000	1.137	1.554	2.078	1.000	1.041	1.202	0.042	1.000	1.022	1.117	0.039	1.000	1.021	1.117	0.326
12	0.5	1.000	1.121	1.770	1.776	1.000	1.031	1.147	0.032	1.000	1.016	1.112	0.038	1.000	1.015	1.112	0.324
12	0.6	1.000	1.073	1.770	1.715	1.000	1.017	1.125	0.037	1.000	1.010	1.067	0.040	1.000	1.010	1.067	0.321
12	0.7	1.000	1.060	2.355	1.645	1.000	1.013	1.122	0.030	1.000	1.006	1.059	0.040	1.000	1.006	1.059	0.316
12	0.8	1.000	1.046	1.590	1.592	1.000	1.008	1.073	0.030	1.000	1.004	1.059	0.038	1.000	1.003	1.059	0.311
12	0.9	1.000	1.038	2.012	1.539	1.000	1.002	1.047	0.038	1.000	1.001	1.030	0.039	1.000	1.001	1.030	0.309
12	1.0	1.000	1.033	1.943	1.461	1.000	1.002	1.047	0.041	1.000	1.001	1.022	0.037	1.000	1.001	1.022	0.314
15	0.0	1.123	1.705	2.394	3.649	1.154	1.517	2.219	0.039	1.120	1.437	1.797	0.055	1.120	1.430	1.797	36.584
15	0.1	1.000	1.455	1.910	3.389	1.000	1.223	1.495	0.036	1.000	1.144	1.368	0.052	1.000	1.135	1.368	24.258
15	0.2	1.000	1.360	2.140	2.721	1.000	1.131	1.478	0.033	1.000	1.083	1.267	0.050	1.000	1.079	1.236	15.538
15	0.3	1.000	1.316	2.135	2.481	1.000	1.087	1.290	0.039	1.000	1.050	1.145	0.048	1.000	1.048	1.145	1.988
15	0.4	1.000	1.222	2.588	2.071	1.000	1.046	1.189	0.030	1.000	1.030	1.164	0.044	1.000	1.025	1.080	0.582
15	0.5	1.000	1.175	1.871	1.992	1.000	1.031	1.165	0.036	1.000	1.016	1.097	0.043	1.000	1.015	1.065	0.406
15	0.6	1.000	1.137	2.003	1.806	1.000	1.018	1.139	0.035	1.000	1.011	1.065	0.038	1.000	1.011	1.063	0.382
15	0.7	1.000	1.083	1.684	1.758	1.000	1.018	1.186	0.030	1.000	1.008	1.056	0.042	1.000	1.008	1.056	0.339
15	0.8	1.000	1.073	1.674	1.660	1.000	1.013	1.137	0.037	1.000	1.006	1.056	0.038	1.000	1.006	1.056	0.317
15	0.9	1.000	1.044	1.404	1.645	1.000	1.009	1.137	0.038	1.000	1.005	1.056	0.046	1.000	1.005	1.056	0.316
15	1.0	1.000	1.031	1.442	1.605	1.000	1.007	1.112	0.038	1.000	1.004	1.056	0.043	1.000	1.004	1.056	0.314

50 random test cases were generated for each point

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

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

net size	ϵ	BPRIM			BKRUS			BKH2			BMST_G		
		min	ave	max	min	ave	max	min	ave	max	min	ave	max
5	0.0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
5	0.1	1.000	1.020	1.100	1.000	1.023	1.093	1.000	1.021	1.082	1.000	1.022	1.082
5	0.2	1.000	1.068	1.195	1.000	1.078	1.196	1.000	1.078	1.199	1.000	1.078	1.199
5	0.3	1.000	1.130	1.296	1.000	1.140	1.298	1.000	1.136	1.298	1.000	1.136	1.298
5	0.4	1.000	1.178	1.399	1.000	1.195	1.400	1.000	1.197	1.400	1.000	1.197	1.400
5	0.5	1.000	1.218	1.483	1.000	1.225	1.483	1.000	1.225	1.483	1.000	1.225	1.483
5	0.6	1.000	1.248	1.563	1.000	1.235	1.536	1.000	1.235	1.536	1.000	1.235	1.536
5	0.7	1.000	1.266	1.647	1.000	1.251	1.698	1.000	1.259	1.698	1.000	1.259	1.698
5	0.8	1.000	1.296	1.798	1.000	1.263	1.799	1.000	1.271	1.799	1.000	1.271	1.799
5	0.9	1.000	1.311	1.898	1.000	1.301	1.887	1.000	1.310	1.887	1.000	1.310	1.887
5	1.0	1.000	1.312	1.919	1.000	1.303	1.921	1.000	1.312	1.921	1.000	1.312	1.921
8	0.0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
8	0.1	1.000	1.044	1.100	1.000	1.046	1.099	1.000	1.048	1.099	1.000	1.046	1.099
8	0.2	1.000	1.125	1.198	1.000	1.116	1.199	1.000	1.115	1.199	1.000	1.115	1.199
8	0.3	1.000	1.201	1.299	1.000	1.178	1.300	1.000	1.180	1.300	1.000	1.182	1.300
8	0.4	1.003	1.273	1.399	1.000	1.237	1.397	1.000	1.256	1.397	1.000	1.258	1.397
8	0.5	1.003	1.334	1.497	1.003	1.329	1.495	1.003	1.336	1.495	1.003	1.337	1.495
8	0.6	1.003	1.373	1.587	1.003	1.351	1.584	1.003	1.358	1.585	1.003	1.360	1.585
8	0.7	1.003	1.408	1.700	1.003	1.397	1.697	1.003	1.381	1.697	1.003	1.382	1.697
8	0.8	1.003	1.437	1.772	1.003	1.416	1.760	1.003	1.412	1.776	1.003	1.414	1.776
8	0.9	1.003	1.445	1.876	1.003	1.425	1.878	1.003	1.422	1.878	1.003	1.424	1.878
8	1.0	1.003	1.456	1.974	1.003	1.448	1.965	1.003	1.456	1.965	1.003	1.458	1.965
10	0.0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
10	0.1	1.000	1.059	1.099	1.000	1.063	1.097	1.000	1.064	1.098	1.000	1.064	1.099
10	0.2	1.000	1.148	1.200	1.000	1.142	1.200	1.000	1.146	1.200	1.000	1.151	1.200
10	0.3	1.000	1.219	1.297	1.000	1.218	1.299	1.000	1.213	1.299	1.000	1.215	1.299
10	0.4	1.067	1.297	1.399	1.067	1.275	1.397	1.067	1.285	1.400	1.067	1.286	1.400
10	0.5	1.067	1.354	1.497	1.067	1.327	1.498	1.067	1.339	1.493	1.067	1.339	1.494
10	0.6	1.067	1.394	1.594	1.067	1.365	1.593	1.067	1.372	1.593	1.067	1.372	1.593
10	0.7	1.067	1.442	1.698	1.067	1.409	1.700	1.067	1.418	1.695	1.067	1.418	1.695
10	0.8	1.067	1.461	1.795	1.067	1.434	1.796	1.067	1.429	1.760	1.067	1.429	1.760
10	0.9	1.067	1.490	1.899	1.067	1.461	1.899	1.067	1.454	1.899	1.067	1.454	1.899
10	1.0	1.067	1.497	1.988	1.067	1.476	1.946	1.067	1.486	1.993	1.067	1.486	1.993
12	0.0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
12	0.1	1.000	1.065	1.100	1.000	1.071	1.100	1.023	1.070	1.100	1.023	1.067	1.100
12	0.2	1.000	1.157	1.200	1.023	1.142	1.199	1.023	1.144	1.199	1.023	1.146	1.199
12	0.3	1.076	1.243	1.299	1.023	1.221	1.298	1.023	1.217	1.296	1.023	1.215	1.296
12	0.4	1.076	1.318	1.397	1.054	1.267	1.394	1.072	1.270	1.394	1.072	1.268	1.394
12	0.5	1.076	1.371	1.498	1.078	1.328	1.498	1.078	1.337	1.498	1.078	1.338	1.498
12	0.6	1.076	1.427	1.598	1.078	1.394	1.595	1.078	1.385	1.595	1.078	1.393	1.595
12	0.7	1.076	1.462	1.700	1.078	1.430	1.693	1.078	1.426	1.682	1.078	1.431	1.678
12	0.8	1.076	1.479	1.800	1.078	1.451	1.786	1.078	1.454	1.786	1.078	1.470	1.786
12	0.9	1.076	1.506	1.900	1.078	1.483	1.871	1.078	1.500	1.871	1.078	1.508	1.871
12	1.0	1.076	1.512	1.995	1.078	1.490	1.925	1.078	1.506	1.979	1.078	1.514	1.979
15	0.0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
15	0.1	1.000	1.075	1.099	1.000	1.072	1.099	1.011	1.071	1.099	1.011	1.070	1.099
15	0.2	1.047	1.173	1.200	1.049	1.162	1.198	1.049	1.162	1.198	1.049	1.165	1.198
15	0.3	1.047	1.260	1.299	1.049	1.245	1.300	1.049	1.243	1.299	1.049	1.241	1.299
15	0.4	1.047	1.338	1.398	1.049	1.312	1.396	1.049	1.320	1.399	1.049	1.321	1.399
15	0.5	1.047	1.403	1.498	1.049	1.372	1.500	1.049	1.379	1.500	1.049	1.369	1.496
15	0.6	1.047	1.480	1.600	1.049	1.436	1.599	1.049	1.440	1.599	1.049	1.438	1.599
15	0.7	1.047	1.520	1.700	1.049	1.480	1.686	1.049	1.479	1.692	1.049	1.475	1.692
15	0.8	1.047	1.551	1.799	1.049	1.506	1.765	1.049	1.528	1.793	1.049	1.521	1.793
15	0.9	1.047	1.586	1.896	1.049	1.523	1.868	1.049	1.543	1.897	1.049	1.539	1.897
15	1.0	1.047	1.603	1.990	1.049	1.567	1.989	1.049	1.577	1.977	1.049	1.573	1.977

50 random test cases were generated for each point

Table 5: The Ratio of the Longest Path Length over the Longest Geometric Distance from Source to any Sink (R)

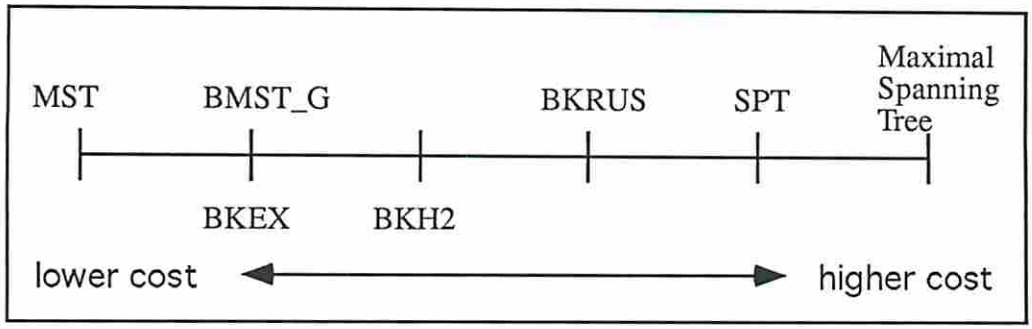

Figure 9: Routing Cost Chart

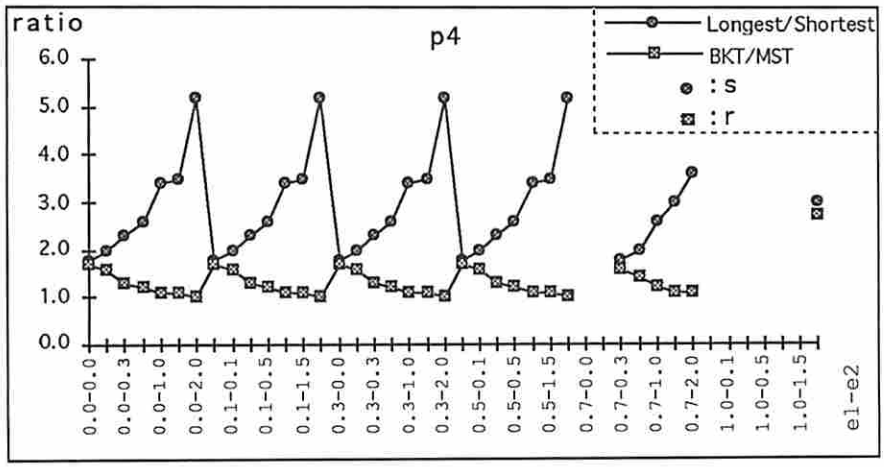

Figure 10: The ratio of Longest Path Length over Shortest Path Length and the ratio of Routing Cost over MST

BKRUS, BMST_G, BKEX, and BKH2 algorithms can be applied to the lower and upper bounded path length MST problem with the inclusion of Lemma 7.1. The inclusion of Lemma 7.1 eliminates an edge (S, i) $< \epsilon_1 \cdot R$, $\forall$ i such that the resultant tree does not violate the lower bound.

Lemma 7.1 *Eliminate edge (S, i)* $\in E$, *for* $\forall i$, *if weight(S, i)* $< \epsilon_1 \cdot R$.

We tested BKRUS method for (1), (2), (3) benchmarks as shown in Table 6. 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 10, we show a typical trade-off between routing cost and clock skew.

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

ϵ_1	ϵ_2	p1		p2		p3		p4		pr1		pr2	
		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 6: The Lower and Upper Bounded BKRUS Results

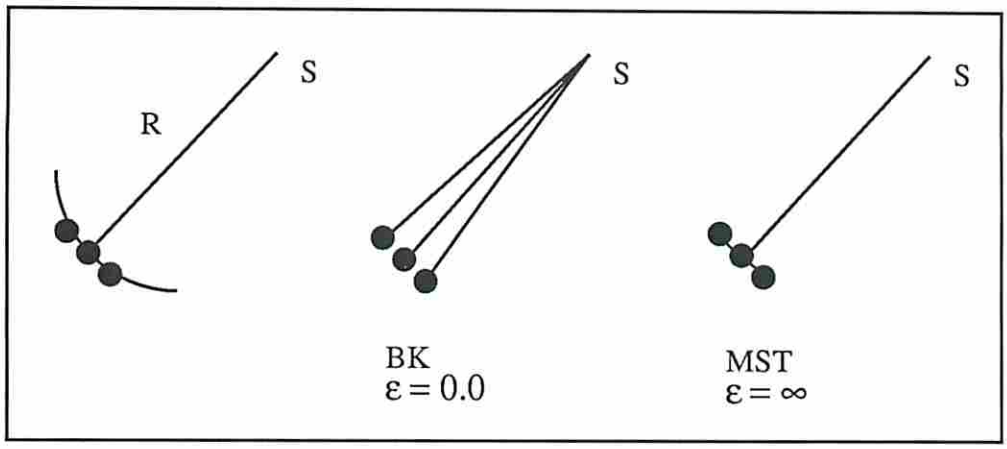

Figure 11: Example where $cost(BKT)/cost(MST)$ can be N where N is number of sinks

achieves smaller cost than that of BPRIM and BRBC. However, by the nature of the problem, our methods can generate almost $N \cdot cost(MST)$ where N is the number of sinks in Figure 11 (p1 case). This is not because of the method but the nature of problem. By a user-specified parameter, design favors are satisfied. Future research includes capturing real delay model, extending the scope to Rectilinear Stenier Tree and preserving planarity.

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, "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, et al., "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] 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.