A lecture note on performance-driven routing

Mads Jepsen · Niels Peter Meyn Milthers · Simon Spoorendonk

November 29, 2004

Contents

1 Introduction **2**
1.1 LAST 4

2 The LAST algorithms **8**
2.1 BRBC Algorithm 8
2.2 KRY Algorithm 11
2.3 ALG1 Algorithm 15
2.4 ALG2 Algorithm 18

3 Experimental Results on Single-Source algorithms **20**
3.1 Comparing Weight and Distance 20
3.2 Delay Simulations 23
3.3 Concluding Remarks 26

4 Extensions and additions **27**
4.1 Multiple Source 27
4.2 Improving the delay estimate 28

References **29**

1 Introduction

After the global and final placement of the modules, the next step of the design of a VLSI chip is to do the actually net layout.

This means we have to build some routing tree T where we interconnect the pins of the modules in each net. If our resulting routing tree T is a Rectilinear Steiner Minimal Tree (RSMT) [11], we know that the total net length of the tree is minimal.

However minimizing the total net length, does not necessary lead to a solution, where the critical signal path is minimal. The problem is that the delay between two nodes in the tree may be high.

In this lecture note we consider the Performance-Driven Routing problem, in which we both consider the length of the net and the delay between a designated driver module and the other modules.

The delay depends on the total length of T, and additionally, for some source module/driver r of a net n_j it depends on the distance from r to a module $z_i \in T$, how much load a single module has to handle (how many signals goes through), and several other factors including length of the wires and thickness of these.

A typical objective for a delay optimal tree T is a tree where the maximal delay from a driver r to all terminals z_i is minimal. If we know that some delay between a driver and some terminal is important, we can include weights on the delay between the drivers and the terminals z_i.

The computation of the exact delay in a given routing tree can be found using the simulation tool SPICE. This procedure is expensive and is therefore not very good if we are trying to do a iterative improvement of the delay in our routing tree [5]. To overcome this problem several methods to estimate the delay has been suggested.

The simplest way to estimate the delay is the linear estimate where we use the distance of the path between the driver to the terminals, i.e. we disregard any resistance the tree rooted at a given terminal may introduce, how much load a given terminal has to handle and how thick the wires are.

In the elmore delay each edge $e(v, w)$ in the routing tree T have a resistance r_e and a capacitance c_e, and each node v has a capacitance c_v. For a given node $w \in T$, let T_e be the subtree rooted at target w of the edge $e(v, w)$. The total capacitance C_e of the subtree T_e denotes the sum of node

and edge capacitance. Given a on-resistance for the driver r_0 the elmore delay is the given as:

$$ED(v_0, v_i) = r_0 C_0 + \sum_{e \in path(v_0, v_i)} r_e (\frac{c_e}{2} + C_e)$$

Example of calculating the Elmore delay On figure 1 we see a small routing tree, each edge has a pair of weights (r_e, c_e) and each node has a name/capacitance (c_i). We will denote the c_e as $c_(u, v)$. To calculate the Elmore delay from the source v_0 to any node, we start by calculating the values $C_i = C_e(i, j)$ for each node. The value is equal c_i for any leaf node so we can calculate C_e bottom up (see table 1.1).

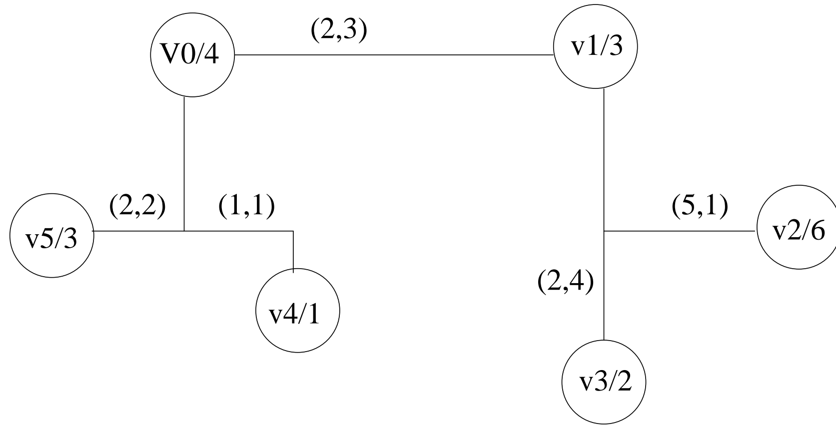


Figure 1.1: A small routing tree

$$\begin{array}{lclcr}
C_0 & = & c_0 + c_{0,1} + c_{0,4} + c_{0,5} + C_1 & = & 30 \\
C_1 & = & c_1 + c_{1,2} + c_{1,3} + C_2 + C_3 & = & 16 \\
C_2 & = & c_2 & = & 6 \\
C_3 & = & c_3 & = & 2 \\
C_4 & = & c_4 & = & 1 \\
C_5 & = & c_5 & = & 3
\end{array}$$

Table 1.1: The capacitance of the subtrees C_e

Calculating the Elmore delay can be done by evaluating top down. If we have a path $v_0, v_1, v_2, ..., v_k$ we can calculate $ED(v_0, v_k)$ based on $ED(v_0, v_{k-1})$

and C_k since:

$$\begin{aligned} ED(v_0, v_k) &= r_0 C_0 + \sum_{e \in path(v_0, v_k)} r_e \left(\frac{c_e}{2} + C_e \right) \\ &= r_0 C_0 + \sum_{e \in path(v_0, v_{k-1})} r_e \left(\frac{c_e}{2} + C_e \right) + r_{(k-1,k)} \left(\frac{c_{(k-1,k)}}{2} + C_k \right) \\ &= ED(v_0, v_{k-1}) + r_{(k-1,k)} \left(\frac{c_{(k-1,k)}}{2} + C_k \right) \end{aligned}$$

In table 1.2 we see the Elmore delay from the driver to the terminals where $r_0 = 1$.

$$\begin{array}{lcrcr} ED(v_0, v_1) &=& C_0 + 2(\frac{3}{2} + 16) &=& 65 \\ ED(v_0, v_2) &=& ED(v_0, v_1) + 5(\frac{1}{2} + 6) &=& 97.5 \\ ED(v_0, v_3) &=& ED(V_0, v_1) + 2(\frac{2}{2} + 2) &=& 70 \\ ED(v_0, v_4) &=& C_0 + \frac{1}{2} + 1 &=& 31.5 \\ ED(v_0, v_5) &=& C_0 + 2(\frac{2}{2} + 3) &=& 37 \end{array}$$

Table 1.2: Elmore delay for the example tree

Since minimizing Elmore delay when building the routing tree T is not straight forward, the heuristics in this lecture note, only use it as an estimate for how good a routing tree they produce.

1.1 LAST

Minimizing both the total net length and the linear delay between all drivers in the net seems very difficult, since we need to find a Steiner tree with additional constraint or a additional objective function. Khuller et al. [6] therefore consider combining the minimum spanning tree (MST) and the shortest path tree (SPT).

The combined tree is called a Light Approximate Shortest-path Tree (LAST). The reason for the interest of a LAST tree is that it saves some delay time between the vertices, without the use of much more wire lenght. The minimum spanning tree is atmost $\frac{4}{3}$ times as large as the Steiner tree [10]. The shortest path tree has the shortest linear estimate delay between the driver/source and all other vertices. Futhermore the shortest path tree has the property that the radius $r(T)$, defined as the longest shortest path in the tree is minimal.

Figure 1.1 on page 7 shows the delay of the MST is very large, and the wire-length of the SPT is also very large, in the combination, we can get the best of both worlds... (minimizing the delay without increasing the wire length to much.)

Definition 1.1. *For $\alpha \geq 1$ and $\beta \geq 1$, a spanning tree T of G meeting the two following requirements is called a (α, β)-LAST rooted at r.*

- *(Distance)For every vertex v, the distance between r and v in$T$ is at most α times the shortest distance from r to v in G.*
- *(Weight) The weight of T is at most β times the weight of a minimum spanning tree of G.*

(α, β)-LAST's are good, since they ensures that no delay to any vertex is longer than α times the minimal delay possible, and the total usage of wire is no longer than β times the minimal.

Lemma 1.2. [6, Lemma 4.1] *Fix $\alpha > 1$ and $1 \leq \beta < 1+2/(\alpha-1)$. A planar graph G with a vertex r exists such that G contains no (α, β)-LAST rooted at r.*

Proof. In figure 1.1 an example of a plannar graph is given. The root r is connected to a central vertex c by a path of of weight $A = \alpha + 1$, of edges of weight some small δ. The central vertex is connected through similar paths of weight $B = \alpha + \epsilon - 1$ to the l leaves. The root is connected to each leaf with n edge of weight $C = 2$. ϵ is a arbitrarily small constant, for small enough δ the minimum spanning tree is formed using all edges except those of weight C

The shortest path from the root to any of the l leaves is the edge of length 2, all other paths weigh more than 2α. This mean that in any (α, β)-LAST all l edges of weight 2 are present. All but l of the remaining edges are present. Therefore the weight of any (α, β)-LAST is at least $2l + T_M - l\delta$ Where $T_M = (\alpha+1)+l(\alpha+\epsilon-1)$ is the weight of the minimum spanning tree. The ratio of the weight of the (α, β)-LAST to the weight of the minimum spanning tree is at least

$$1 + \frac{l(2-\delta)}{\alpha + 1 + l(\alpha - 1 + \epsilon)}$$

Since $\beta < 1 + 2/(\alpha - 1)$ the above exceeds β for sufficiently small ϵ and δ and sufficiently large l. ☐

Khuller et al. shows that the problem of deciding if there exist an (α, β)-LAST for $\alpha > 1$ and $\beta = 1$ is NP-hard by reduction from 3-SAT, and then shows that this problem can be reduced to the general problem where $1 \leq \beta < 1 + 2/(\alpha - 1)$. In the next section we present an efficient algorithm to solve the problem when $\beta \geq 1 + 2/(\alpha - 1)$ and some heuristics for finding (α, β)-LAST where we minimize both α and β.

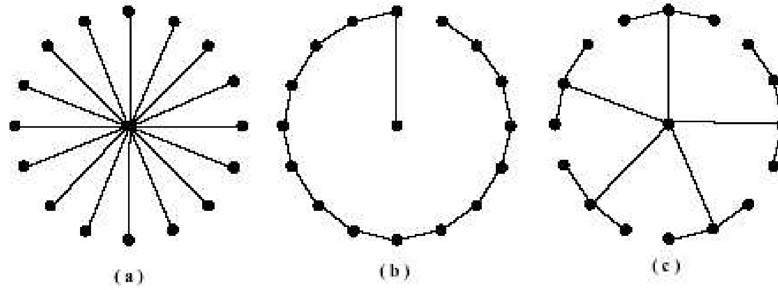


Figure 1.2: (a) A SPT rooted at the center node. (b) A MST, (c) A LAST.

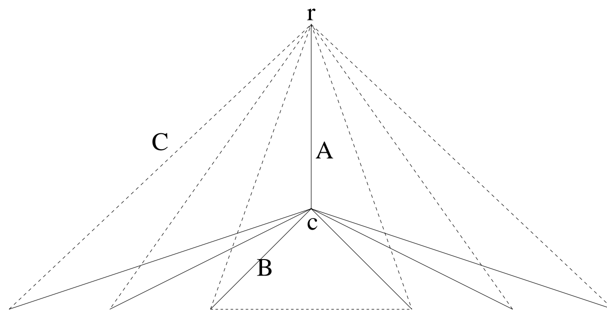


Figure 1.3: Planar Graph with no (α, β)-LAST, for $\beta < 1 + 2/(\alpha - 1)$

2 The LAST algorithms

In this section we present algorithms for finding LAST.

First we describe two algorithms that guarantees a bound on the radius r(T) and a bound on the length of the tree compared to the MST. The Bounded-Radius Bounded-Cost algorithm (BRBC) by Cong et al only have a bound on the radius of the tree, while Khuller- Raghavachari-Young algorithm (KRY) by Khuller et al. [6] find a (α, β)-LAST. Both these algorithms are based on an initial MST and then add paths from a SPT as needed.

The next two algorithms differs from this approach since they are purely based on the Prim/Dijkstra algortihms. The algorithms ALG1 and ALG2 were developed by Alpert et al. [1] and do not yield (α,β)-trees, but experimental test performed by Alpert et al. has shown that the produced trees are more useful in practice.

2.1 BRBC Algorithm

The Bounded-Radius Bounded-Cost algorithm (BRBC) is also based on Prim's algorithm and was presented by Cong et al. [2]. The algorithm gives a guaranty on both the radius $1+\epsilon$ and the cost $1+\frac{2}{\epsilon}$ of the produced tree.

In the following $minpath_G(v_i, v_j)$ is the shortest path between v_i and v_j in the graph G, and $dist_G(v_0, v_i)$ is the distance of that path. $cost(G)$ denotes the edge costs of G and $cost(v_i, v_j)$ denotes the edge cost on the path from v_i to v_j on some nodes belonging to a graph.

The BRBC algorithm builds a subgraph Q of the original routing network $G(V, E)$ where the SPT_Q named T is the routing tree fulfilling the bounds given above.

First a MST_G and a SPT_G is produced and the subgraph Q is set equal to MST_G. A inner tree walk of Q is performed so that all edges are visited exactly twice, the visited edges and nodes are described as the walk L. For each node in L a accumulating edge cost S is calculated and for a given node L_i it is tested wether $S \geq \epsilon dist_G(v_0, v_i)$ and if so the shortest path v_0-v_i is added to Q. A formal description of the algorithm can be seen below.

```
BRBC(G(V, E))
 1  compute MST_G and SPT_G
 2  Q ← MST_G
 3  L ← inner tree walk of MST_G
 4  S ← 0
 5  for i = 0 to |L| − 1
 6       do S ← S + cost(L_i, L_{i+1})
 7          if S ≥ ε dist_G(v_0, L_{i+1})
 8            then Q ← Q ∪ minpath_G(v_0, L_{i+1})
 9                 S ← 0
10  T ← SPT_Q
```

An example of BRBC can be seen in the following example.

A trial run of BRBC with $\epsilon = 1$ can be seen in example 2.1.

Theorem 2.1. [2, Theorem 2] *For any weigthed graph G and a parameter ϵ, the routing tree T constructed by BRBC has radius $d(T) \leq (i + \epsilon)D$.*

Proof. For any $v \in V$, let v_{i-1} be the last node before v_i on the MST_G traversal L for which we added $dist_G(v_0, v_{i-1})$ to Q. By the construction of BRBC we now that $dist_L(v_{i-1}, v_i) \leq \epsilon D$. We then have:

$$\begin{aligned} dist_T(v_0, v_i) &\leq dist_T(v_0, v_{i-1}) + dist_L(v_{i-1}, v_i) \\ &\leq dist_G(v_0, v_{i-1}) + \epsilon D \\ &\leq D + \epsilon D = (1 + \epsilon)D \end{aligned}$$

□

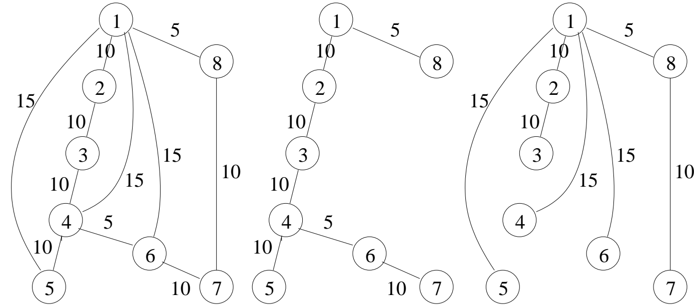
Furthermore the BRBC algorithms make guarantees on the bounding of the cost on the tree compared to the MST.

Theorem 2.2. [2, Theorem 3] *For any weigthed graph G and a parameter ϵ, the routing tree T constructed by BRBC has $cost(T) \leq (1 + \frac{2}{\epsilon})cost(MST_G)$.*

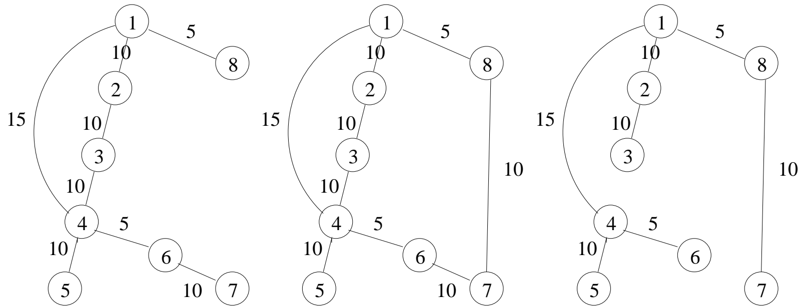
Proof. Let $v_1, v_2, \ldots, v_m$ be the nodes for which shortest paths from v_0 was added to Q, then

$$cost(T) \leq cost(MST_G) + \sum_{i=1}^{m} dist_G(v_0, v_i)$$

Example This is an example of the BRBC algorithm.



The graph, the MST and the SPT. A DFS gives the list of visited nodes $L = \{1, 2, 3, 4, 5, 4, 6, 7, 6, 4, 3, 2, 1, 8, 1\}$.



Running the BRBC algorithm until termination. The first figure shows when $i = 2$, i.e. node 3 and a minimum path to node 4 is added. The next figure shows when $i = 6$, i.e. node 6 and a minimum path to node 7 is added. The last figure shows the SPT of the tree produced in the for loop.

Figure 2.1: Trial run of BRBC with $\epsilon = 1$.

since T is a subtree of the union of the MST_G with all the added shortest paths. By the algorithm construction $dist_L(v_{i-1}, v_i) \geq \epsilon dist_G(v_0, v_i)$ we have:

$$\begin{aligned} cost(T) &\leq cost(MST_G) + \sum_{i=1}^{m} \frac{1}{\epsilon} dist_L(v_{i-1}, v_i) \\ &\leq cost(MST_G) + \frac{1}{\epsilon} cost(L) \end{aligned}$$

since $cost(L) \leq 2cost(MST_G)$ we have:

$$\begin{aligned} cost(T) &\leq cost(MST_G) + \frac{2}{\epsilon} cost(MST_G) \\ &= (1 + \frac{2}{\epsilon}) cost(MST_G) \end{aligned}$$

□

The running time of the algorithm is bounded by the time it takes to produce the initial MST_G and SPT_G and the final tree $T = SPT_Q$, i.e. a running time $O(E \log V)$ can be obtained using a binary heap.

2.2 KRY Algorithm

The kry algorithm is a modification of the BRBC algorithm, it does not only guaranty a bound on the radius, but gives a bound on all paths in the tree.

Let $G = (V, E)$ be a graph with nonnegative edge weights and a root vertex r. Let G have n vertices and m edges. Let $w(e)$ be the weight of edge$e \in E$. the *distance* $D_G(u, v)$ is the minimum weight of any path in G between them. Next pseudocode of KRY algorithm is presented as it is given in [6].

Initialise distance estimates d, and parent pointers p

INITIALISE()

```
1  for each non-root vertex v
2          do p[v] ← NIL
3             d[v] ← ∞
4  d[r] ← 0
```

Check for shorter path to v through (u, v)

RELAX(u, v)
1 **if** $d[v] > d[u] + w(u, w)$
2 **then** $d[u] \leftarrow d[u] + w(u, v)$
3 $p[v] \leftarrow u$

Relax edges along path from r to v in SPT_G.

ADD-PATH(u)
1 **if** $d[u] > \alpha D_{SPT_G}(r, u)$
2 **then** ADD-PATH$(Parent_{SPT_G}(u))$
3 RELAX$(Parent_{SPT_G}(u), u)$

Traverse the subtree of MST_G rooted at u, relaxing edges as they are traversed, end adding paths from SPT_G as needed.

DFS(u)
1 **if** $d[u] > \alpha D_{SPT_G}(r, u)$
2 **then** Add-Path(u)
3 **for** each child v of u in MST_G
4 **do** RELAX(u, v)
5 DFS(v)
6 RELAX(v, u)

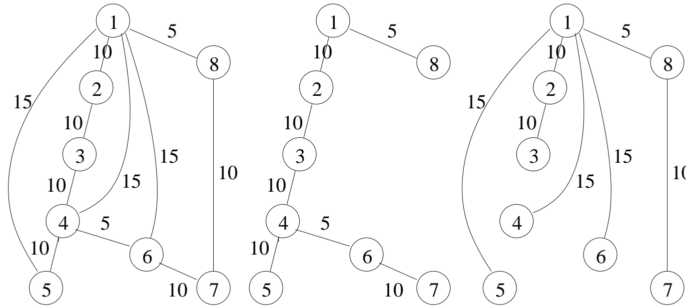
The main procedure with input: a minimum spanning tree MST_G, shortest-path tree SPT_G, a root vertex r and an $\alpha > 1$. The output of the algorithm is an $(\alpha, 1 + 2/(\alpha - 1)) - LAST$ named T rooted at the vertex r.

KRY$(MST_G, SPT_G, r, \alpha)$
1 INITIALISE()
2 DFS(r)
3 **return** $T = \{(v, p[v]) | v \in V - \{r\}\}$

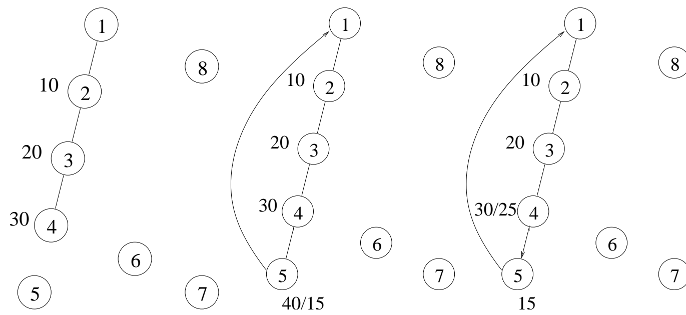
A trial run of KRY with $\alpha = 2$ can be seen in example 2.2. We now regard some properties for the KRY algorithm.

Theorem 2.3. [6, Theorem 1] *Let G be a graph with nonnegative edge weights; let r be a vertex of G, let $\alpha > 1$and $\beta \geq 1 + 2/(\alpha - 1)$. Then G contains an $(\alpha, \beta) - LAST$ rooted at r. The LAST can be found in linear time given a minimum spanning tree and a shortest-path tree, and in $O(m + n \log n)$ time otherwise.*

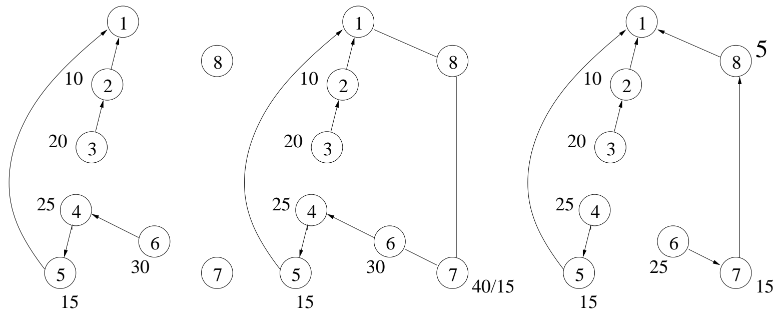
Example This is an example of the KRY algorithm.



The graph, the MST and the SPT.



The first 3 iterations.
Adding shortest path to 5, i.e edge from 1 to 5.
Relaxing edge from 4 to 5 making a shorter path to 4.



Continuing DFS from 4 to 6.
Adding shortest path to 7, i.e path from 0 to 8 to 7.
The resulting (α, β)-LAST tree, note that the edge between 4 and 6 has been removed during the last iterations.

Figure 2.2: An example of the KRY algorithm with $\alpha = 2$

This is proved from the following lemmas.

Lemma 2.4. [6, Lemma 3.1] *The distance between v and r in T is at most α times the shortest-path distance*

Proof. If $d[v] > \alpha D_{T_{SPT}}(r,u)$ when traversed in DFS, it is relaxed. In which case ADD-PATH makes it equal to the shortest path distance. in either case the distance after v is visited will be no greater than α times the shortest-path distance, and cannot increase. □

Lemma 2.5. [6, Lemma 3.2] *The weight of T is at most* $(1+2/(\alpha-1)$ *times the minimum spanning-tree weight*

Proof. Let $v_0 = r$ and let $v_1, v_2, \ldots, v_k$ be the vertices that caused shortest paths to be added during the traversal, in order they were encountered. When the shortest path from r to v_i $(i \geq 1)$ was added, the net weight of the added edges was at most $D_{T_S}(r, v_i)$. Also, the edges on the path to v_i consisting of the shortest path to v_{i-1} followed by the path in the MST from v_{i-1} to v_i had been relaxed in order, so that $d[v_i] \leq D_{T_{SPT}}(r, v_{i-1}) + D_{T_{MST}}(v_{i-1}, v_i)$. The shortest path to v_i was added because$\alpha D_{T_{SPT}}(r, v_i) < d[v_i]$ combining these:

$$\alpha D_{T_S}(r, v_i) < D_{T_S}(r, v_{i-1}) + D_{T_M}(v_{i-1}, v_i)$$

summing over i bounds the net weight of the added paths:

$$\alpha \sum_{i=1}^{k} D_{T_S}(r, v_i) < \sum_{i=1}^{k} (D_{T_S}(r, v_{i-1}) + D_{T_M}(v_{i-1}, v_i))$$

which equals

$$\alpha \sum_{i=1}^{k} D_{T_S}(r, v_i) - \sum_{i=1}^{k} D_{T_S}(r, v_{i-1}) < \sum_{i=1}^{k} D_{T_M}(v_{i-1}, v_i)$$

and since

$$\sum_{i=1}^{k} D_{T_S}(r, v_{i-1}) < \sum_{i=1}^{k} D_{T_S}(r, v_i)$$

it gives:

$$(\alpha - 1) \sum_{i=1}^{k} D_{T_S}(r, v_i) < \sum_{i=1}^{k} D_{T_M}(v_{i-1}, v_i)$$

The DFS traverses each edge exactly twice, and hence the the sum on the right hand side is at most twice the weight of T_{MST}, giving the maximum weight of the added edges, is $(2/(\alpha - 1))w(T_{MST})$. Giving the β assurance. □

The running time is proportional to the number of relaxations, this at most $O(V)$ because each edge in MST_G or SPT_G is relaxed at most twice by DFS and at most once by ADD-PATH. If these are unknown, they can be produced in $O(E + E \log V)$.

2.3 ALG1 Algorithm

First recall the Prim's MST and The Dijkstra's SPT algorithms:

Prim's MST We are given a set of nodes V and an initial tree T consisting of the source node v_0. Then edges e_{ij} and sinks $v_j \in V \backslash T$ are iteratively added to T, where v_i and v_j are chosen so

$$\min\{d_{ij} : v_i \in T, v_j \in V \backslash T\} \tag{2.1}$$

That is, the edge e_{ij} (and thereby the sink v_j) with the minimum weight d_{ij} is connected to the existing tree T.

Dijkstra's SPT We are given a set of nodes V and an initial tree T consisting of the source node v_0. Then e_{ij} and sinks $v_j \in V \backslash T$ are iteratively added to T, where v_i and v_j are chosen so

$$\min\{l_i + d_{ij} : v_i \in T, v_j \in V \backslash T\} \tag{2.2}$$

That is, the edge e_{ij} (and thereby the sink v_j) with the minimum sum of the edge weight d_{ij} and the shortest path l_i to v_i is connected to existing tree T.

Denote the length of the shortest v_0-v_i path as D_i and $D = \max\{D_i : \forall v_0\text{-}v_i,\ v_0, v_i \in V\}$ as the longest shortest path. D is also called the radius of the tree.

See Figure 2.3 (a)-(b) for an example of the results of the MST and SPT algorithms.

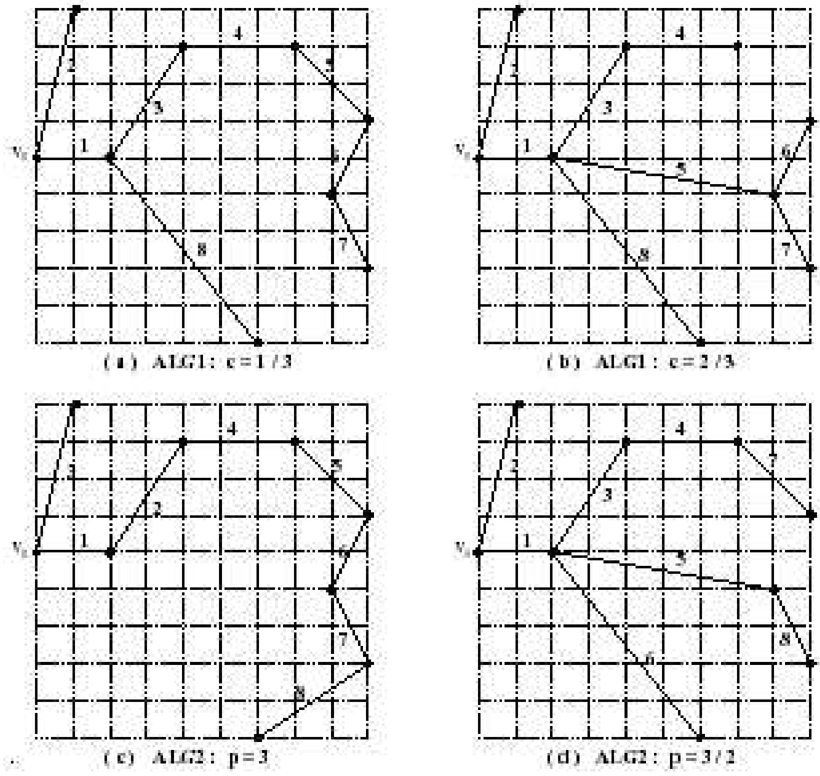


Figure 2.3: Example of the ALG1 and ALG2 algortihms.

The Prim's MST and The Dijkstra's SPT algorithms are based on a labeling method that expands a fixed source by adding edges that minimizes a key specified by the algorithms, (2.1) for the MST and (2.2) for the SPT algorithm. It therefor seems logical to combine the algorithms so that both the min-weight objective of the MST and the min-length objective of the SPT algorithms are taken into account. This leads to the two algorithms described below that are both based on the same principle.

The similarity of (2.1) and (2.2) leads to the following key of ALG1

$$\min\{(cl_i) + d_{ij} : v_i \in T, v_j \in V\backslash T\} \tag{2.3}$$

for some choice of $0 \leq c \leq 1$. See pseudocode below.

Alg1$(G(V, E), c)$
1 $T \leftarrow \{v_0\}$
2 **while** $V\backslash T \neq \emptyset$
3 **do** $e_{ij} \leftarrow \min\{(cl_i) + d_{ij} : v_i \in T, v_j \in V\backslash T\}$
4 $T \leftarrow T \cup path_G(v_i, v_j, e_{ij})$
5 **return** T

From this we get the observations:

Observation 2.6. [1, Observation 1] *When $c = 0$, ALG1 is identical to Prim's MST algorithm, i.e. (2.3) with $c = 0$ equals (2.1).*

Observation 2.7. [1, Observation 2] *When $c = 1$, ALG1 is identical to Dijkstra's SPT algorithm, i.e. (2.3) with $c = 1$ equals (2.2).*

Hence both the MST and the SPT algorithms is contained in ALG1. Clearly the running time is $O(E \log V)$ using a binary heap since Alg1 is basically Prim's or Dijkstra's algorithm. See Figure 2.3 (c)-(d) for an example of ALG1 with $p = \frac{1}{3}$ and $p = \frac{2}{3}$.

It can be observed that any v_0-v_i path build using ALG1 is maximal a factor $\frac{1}{c}$ times larger than the optimal shortest path.

Proposition 2.8. [1, Observation 3] *ALG1 constructs a tree T with $cl_j \leq D_j$ for all sinks v_j.*

The proof is by induction [1].

2.4 ALG2 Algorithm

In the second algorithm the key is not straight forward as in ALG1.

Let l_i^p equal the following expression based on the edge weights of the path v_0-v_i:

$$l_i^p = \Big(\sum_{k=1}^{i} d_{k-1,k}^p \Big)^{\frac{1}{p}}$$

for a $1 \leq p \leq \infty$.

In the following we write l_i^p as $||d_{01}, \ldots, d_{i-1,i}||_p$. This leads to the key

$$\min\{||l_i^p, d_{ij}||_p : v_i \in T, v_j \in V \backslash T\} \tag{2.4}$$

The pseudocode of ALG2 is equivalent with that of ALG1, only substitute the key (2.3) in line 3 with (2.4), i.e. the pseudocode should be

Alg2$(G(V, E), c)$
1 $T \leftarrow \{v_0\}$
2 **while** $V \backslash T \neq \emptyset$
3 **do** $e_{ij} \leftarrow \min\{||l_i^p, d_{ij}||_p : v_i \in T, v_j \in V \backslash T\}$
4 $T \leftarrow T \cup path_G(v_i, v_j, e_{ij})$
5 **return** T

Again the running time is noted to be $O(E \log V)$. See Figure 2.3 (e)-(f) for an example of ALG1 with $p = 3$ and $p = \frac{2}{3}$.

Observe that l_i^1 is the sum of edge weights on the path v_0-v_i. That is $||l_i^1, d_{ij}||_1 = \sum_{k=1}^{i} d_{k-1,k} + d_{ij}$ for the source v_0 and the sink v_j. This leads to the following observation:

Observation 2.9. [1, Observation 4] *When $p = 1$, ALG2 is identical to Dijkstra's SPT algorithm, i.e. (2.4) with $p = 1$ equals (2.1).*

Let $|l_i|$ denote the edge with largest weight on the v_0-v_i path. When $p = \infty$ the objective of (2.4) is equivalent with $\max\{|l_i|, d_{ij}\}$, i.e. the heaviest edge on a possible path from v_0 to v_j. This is also known as the "bottleneck" SPT. Salowe et al. [9] has previously solved this problem and independently discovered ALG2 by reducing from the above objective.

This objective does not yield a unique solution, since the presence of a heavy edge on some shortest path will make it possible to add any edge with

less weight. In order get a unique solution ties is broken by choosing the sink v_j. This leads to:

Proposition 2.10. [1, Observation 5] *When $p = \infty$, ALG2 is equivalent to Prim's MST algorithm, i.e. (2.4) with $c = \infty$ is equivalent to (2.2).*

The proof is by induction [1]. From the above it follows that ALG2 also contains both the Prim's MST and the Dijkstra's SPT algorithm.

3 Experimental Results on Single-Source algorithms

In this section we present experimental results from Alpert et al. [1] based on the algorithms by Alpert et al., Khuller et al. [6], and Cong et al. [2] presented in Section 2.

The test is divided into two parts, first the parameters are tuned for each algorithm in order to find the best settings of the algorithm specific parameter, then a sinal delay performance is done using the tuned parameters in a simulation of different wire technologies. This is done on both tree outputed by the algorithms and on a Steiner Tree induced from the output tree. Alpert et al. use a greedy edge overlapping method to build the Steiner Trees.

3.1 Comparing Weight and Distance

Based on a signal net, 51 tress for ALG1 was generated with c going from 0 to 1 in 0.02 intervals. Based on c for ALG1 a corresponding parameter value for the others algorithms where chosen as shown in Table 3.1 in order to produce a similar number of output trees. For BRBC a range from 0 to 1.5 with intervals 0.03 was used since the trees showed to near identical when $\epsilon \geq 1.5$.

	ALG1	ALG2	BRBC	KRY
User parameter	c	p	ϵ	α
Yields a MST when	$c = 0$	$p = \infty$	$\epsilon = \infty$	$\alpha = \infty$
Yields a SPT when	$c = 1$	$p = 1$	$\epsilon = 0$	$\alpha = 1$
Relation to c	c	$p = \frac{1}{c}$	$\epsilon = \frac{1-c}{c}$	$\alpha = \frac{1}{c}$

Table 3.1: Equivalence of algorithm parameters.

Each algorithm were run with it's specified parameters on signal nets of 16 sinks randomly distributed in a 1cm by 1cm Manhatten square. Each set of parameters were run 250 times which is represented by each point on Figure 3.2 in a cost/distance ratio.

The algorithms all have smooth tradeoff between cost and weight, maybe except for BRBC which seems benefit when the ratio of the radius of the

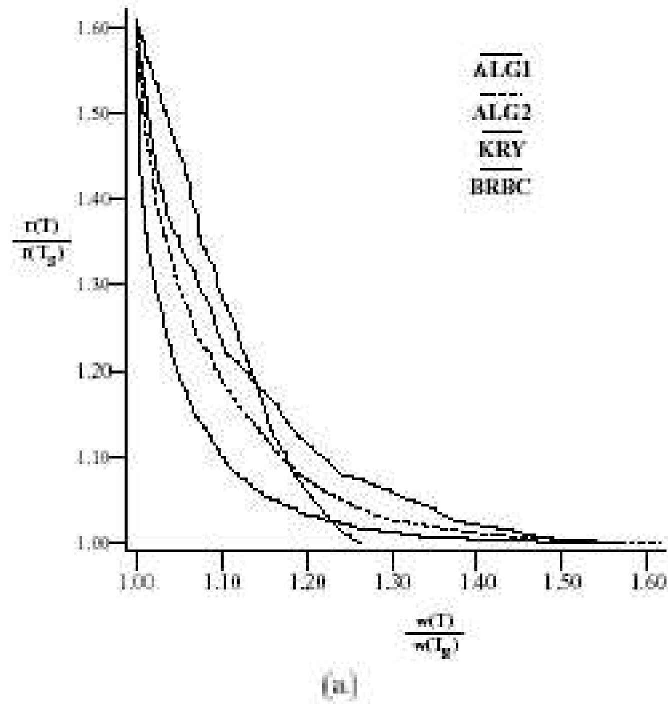


Figure 3.1: Graphs of radius ratio $\frac{r(T)}{r(SPT)}$ versus cost ratio $\frac{w(T)}{w(MST)}$ for ALG1, ALG2, KRY and BRBC.

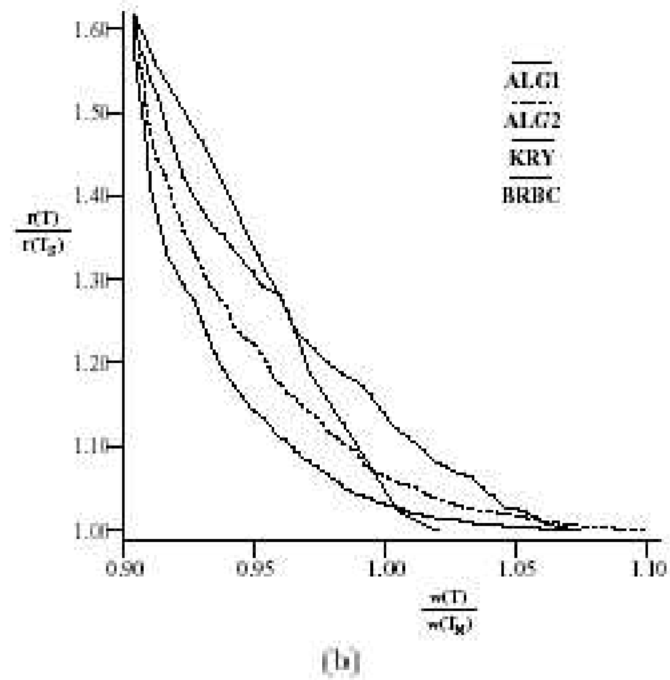


Figure 3.2: Graphs of radius ratio $\frac{r(T)}{r(SPT)}$ versus cost ratio $\frac{w(T)}{w(MST)}$ for ALG1, ALG2, KRY and BRBC for the induced Steiner Tree.

trees drop to near 1. ALG1 is clearly superiour, especially in the practical interesting region where the tree cost is only 10-20 % more than the optimal.

The picture is almost equivalent when plotting the induced Steiner Trees. The edge-overlapping method produces trees where radius can be reduced compared to the original tree since extra cost can be used on the weight of the tree, this implies the shift to right on weight axis on Figure 3.2 (b) compared to (a).

3.2 Delay Simulations

The next test was done on random signal nets of 4, 8 and 16 sinks. Alpert et al. used a Two-Pole circuit simulator by Zhou et al. [12] to simulate the delay on the signal nets for the four different technologies described see [1]

Fot each instance of sinks in the signal net all parameters settings where tried and the lowest delay was recorded. This was done on 250 instances for each technology and both average delay on all sinks, the maximum delay (the sink furthest away) and the average parameter setting (in parentheses) is reported in Figure 3.3.

The ALG1 algorithm performs best yeilding the lowest delayes in 27 of 30 cases compared to the suprisingly good runner-up KRY. Suprisingly since the cost/radius test showed he KRY algorithm to significantly worse than ALG1. Alpert et al. suggests this has to do with the KRY trees often being self-intersecting where the ALG1 trees do not.

The best parameter setting clearly shows how this is depending on both net size, i.e. number of sinks and technology. Alpert et al. suggest that if only spanning trees were constructed good parameters would go towards low delay trees.

Regarding the induced Steiner Trees the ALG1 algorithm is again superiour with KRY as the closest competitor, se Figuer 3.4.

On average the Steiner Trees imposes lower delayes than the original output trees, but for some best fixed parameters the orignal output tree can outperform the induced Steiner Tree. Note however that the induced Steiner Trees are build from a greedy method, i.e. minimum Steiner Trees possibly gives lower delayes.

Spanning Trees		Max sink delay vs. MST (best parameter)			
#sinks	ALG.	IC1	IC2	IC3	MCM1
4	ALG1	0.896 (0.13)	0.859 (0.28)	0.849 (0.30)	0.759 (0.39)
	ALG2	0.900 (23.62)	0.861 (19.25)	0.851 (18.29)	0.759 (14.84)
	KRY	0.897 (22.88)	0.859 (19.17)	0.850 (18.08)	0.759 (14.64)
	BRBC	0.906 (0.09)	0.872 (0.06)	0.863 (0.07)	0.786 (0.04)
8	ALG1	0.808 (0.19)	0.746 (0.45)	0.732 (0.46)	0.584 (0.62)
	ALG2	0.823 (11.24)	0.760 (7.66)	0.745 (6.80)	0.590 (3.46)
	KRY	0.815 (9.83)	0.752 (6.62)	0.736 (6.47)	0.584 (2.93)
	BRBC	0.850 (0.13)	0.796 (0.08)	0.784 (0.09)	0.657 (0.06)
16	ALG1	0.742 (0.23)	0.666 (0.49)	0.648 (0.52)	0.458 (0.73)
	ALG2	0.772 (4.78)	0.696 (3.39)	0.678 (3.09)	0.484 (1.40)
	KRY	0.752 (3.52)	0.671 (2.38)	0.648 (2.07)	0.456 (1.29)
	BRBC	0.831 (0.17)	0.772 (0.11)	0.758 (0.12)	0.615 (0.07)

Spanning Trees		Avg sink delay vs. MST (best parameter)			
#sinks	ALG.	IC1	IC2	IC3	MCM1
4	ALG1	0.911 (0.10)	0.866 (0.32)	0.854 (0.34)	0.712 (0.55)
	ALG2	0.916 (21.29)	0.871 (17.28)	0.858 (16.27)	0.714 (8.59)
	KRY	0.912 (19.56)	0.866 (16.22)	0.854 (15.83)	0.712 (8.10)
	BRBC	0.928 (0.09)	0.891 (0.10)	0.880 (0.10)	0.768 (0.11)
8	ALG1	0.808 (0.15)	0.778 (0.47)	0.759 (0.49)	0.540 (0.75)
	ALG2	0.861 (11.45)	0.794 (6.31)	0.774 (5.85)	0.551 (2.00)
	KRY	0.850 (9.42)	0.781 (4.83)	0.760 (3.96)	0.540 (1.79)
	BRBC	0.899 (0.08)	0.848 (0.06)	0.834 (0.05)	0.678 (0.04)
16	ALG1	0.800 (0.20)	0.720 (0.48)	0.697 (0.50)	0.429 (0.82)
	ALG2	0.829 (5.22)	0.749 (2.90)	0.726 (2.58)	0.452 (1.24)
	KRY	0.808 (3.57)	0.723 (1.99)	0.696 (1.87)	0.424 (1.19)
	BRBC	0.893 (0.12)	0.839 (0.13)	0.824 (0.13)	0.648 (0.12)

Figure 3.3: Maximum source-sink dealy and average source-sink delay in the best tree for each algorithm.

Steiner Trees		Max sink delay vs. MST (best parameter)			
#sinks	ALG.	IC1	IC2	IC3	MCM1
4	ALG1	0.812 (0.17)	0.789 (0.21)	0.780 (0.24)	0.748 (0.30)
	ALG2	0.837 (32.56)	0.813 (30.46)	0.804 (29.14)	0.778 (25.33)
	KRY	0.812 (31.77)	0.789 (28.85)	0.780 (27.86)	0.747 (23.79)
	BRBC	0.816 (0.03)	0.794 (0.05)	0.785 (0.06)	0.758 (0.04)
8	ALG1	0.727 (0.34)	0.684 (0.46)	0.672 (0.49)	0.587 (0.57)
	ALG2	0.797 (16.53)	0.755 (9.81)	0.742 (9.25)	0.671 (6.77)
	KRY	0.728 (14.45)	0.684 (8.97)	0.672 (8.72)	0.586 (6.15)
	BRBC	0.744 (0.10)	0.707 (0.10)	0.694 (0.10)	0.628 (0.10)
16	ALG1	0.665 (0.44)	0.606 (0.52)	0.590 (0.54)	0.463 (0.68)
	ALG2	0.758 (11.52)	0.699 (5.63)	0.682 (3.72)	0.567 (1.93)
	KRY	0.671 (5.66)	0.611 (3.73)	0.596 (3.85)	0.466 (1.76)
	BRBC	0.713 (0.16)	0.664 (0.16)	0.651 (0.15)	0.556 (0.12)

Steiner Trees		Avg sink delay vs. MST (best parameter)			
#sinks	ALG.	IC1	IC2	IC3	MCM1
4	ALG1	0.807 (0.25)	0.775 (0.27)	0.763 (0.28)	0.694 (0.32)
	ALG2	0.821 (27.45)	0.787 (26.24)	0.773 (27.04)	0.706 (22.88)
	KRY	0.807 (26.12)	0.776 (24.37)	0.763 (24.56)	0.694 (21.85)
	BRBC	0.817 (0.06)	0.787 (0.05)	0.775 (0.06)	0.716 (0.06)
8	ALG1	0.749 (0.50)	0.696 (0.55)	0.680 (0.56)	0.551 (0.64)
	ALG2	0.808 (11.63)	0.754 (8.93)	0.737 (7.34)	0.617 (4.27)
	KRY	0.751 (9.86)	0.698 (6.15)	0.682 (5.55)	0.550 (3.38)
	BRBC	0.777 (0.18)	0.735 (0.15)	0.720 (0.17)	0.625 (0.13)
16	ALG1	0.711 (0.52)	0.644 (0.60)	0.624 (0.62)	0.443 (0.75)
	ALG2	0.791 (12.83)	0.719 (5.03)	0.697 (3.38)	0.526 (1.56)
	KRY	0.715 (3.99)	0.647 (2.48)	0.628 (2.28)	0.445 (1.29)
	BRBC	0.765 (0.24)	0.719 (0.25)	0.702 (0.25)	0.579 (0.29)

Figure 3.4: Maximum source-sink dealy and average source-sink delay in the best tree for each algorithm.

3.3 Concluding Remarks

The ALG1 algorithm is the best performing algorithm presented in this lecture note, both concerning the cost/radius benefit and the delay in the IC and MCM wire technologies.

However we should remember that so far we have only consider trees that were tradeoffs between MST and SPT. We did get a little closer by examinating the induced Steiner Trees, but it is still unclear how a Minimum Steiner Tree would perform in these test.

Also we have only consider routing from a single source to some sinks in relative small networks. It is also interesting to consider multiple sources routed to multiple sinks in the same network. Possibly this gives a complete different view of the connection between the radius/cost and the average delay in the output tree.

4 Extensions and additions

In this section we will briefly consider additional work on the performance routing problem. We will not describe the work in detail, but hopefully the reader will be able to use the articles to future work.

4.1 Multiple Source

While the algorithms described in section 2 are only concerned with one source driving a net, Cong and Madden [3] consider the more general problem where multiple sources can drive a single net. The problem is that minimizing the delay with respect to one source may have a bad influence on the delay from another source.

To illustrate the above problem we consider the Tri-State gate, see figure 4.1. When we minimize the delay with respect to p_1 driving the net, we get the solution shown on the left hand side of figure 4.1, which both minimizes the maximum delay and the average delay. But as we can see the solution is very poor when p_2 drives the net. An optimal solution which minimizes the delay with respect to both p_1 and p_2 is seen on the right side of figure 4.1, here we have the minimal average delay when we consider both p_1 and p_2 as a driver. It should be noted that the optimal solution does not fall

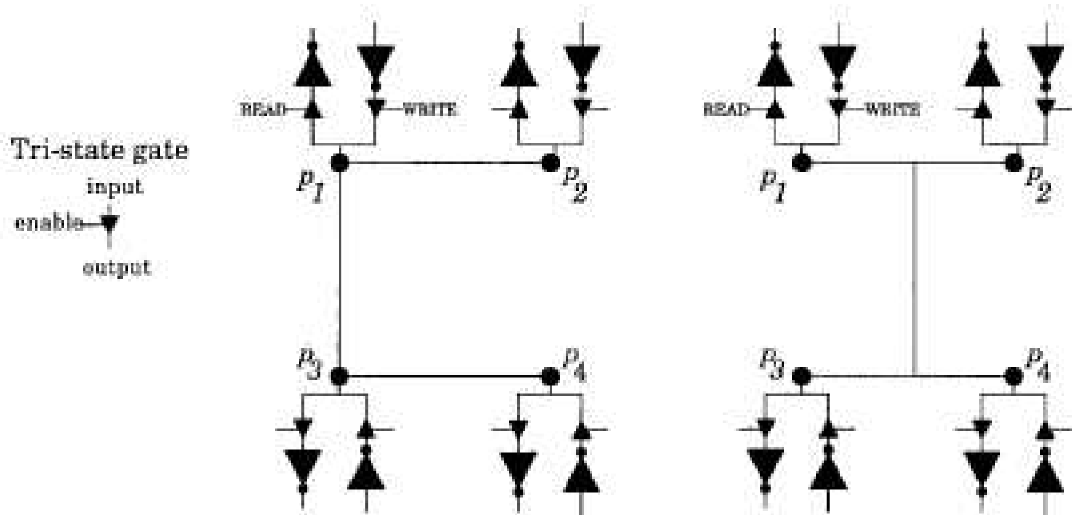


Figure 4.1: The tri state example

on the Hanan grid, this means that a canonical fulsome RSMT [11], which minimizes the total net length may not include the optimal solution to the multiple source problem. Furthermore an algorithm for solving the problem cannot be restricted only to consider points on the hanan grid.

The problem can be defined as: Given n points $\{p_1, ..., p_n\}$ in the Manhatten plane, the distance function $d(p_i, p_j) = |y_j - y_i| + |x_j - x_i|$ and the delay function $delay(p_i, p_j)$, we wish to find a Steiner tree T spanning all nodes with the following properties:

- Minimal total delay $WD(T) = \sum_{p_i, p_j} delay(p_i, p_j)$
- Minimal total net length $L(T) = \sum_{e(p_i, p_j) \in T} d(p_i, p_j)$

The two objectives is obtained through Minimum Diameter trees. Where the diameter of a routing $D_T(P)$ is the longest path between a pair of points in T.

4.2 Improving the delay estimate

Lillis and Buch [7] introduce a table look-up method and show some promising results when embedding the model into the P-tree algorithm [8]. The insight of the purposed model stems from the observation that the two main error contributions in the Elmore delay model is:

- It's inability to account for resistive shielding
- It's inability to account for the effect of signal slew rate τ

Resistive shielding is a phenomenon that arises when the interconnected wires become thinner and longer [4] and the slew rate can be thought of as the "speed" or "strength" of a signal.

For a driver v_0 the signal slew at node $v \in T$ can be calculated as $\tau_A = 2.197ED(v_0, A)$ under the assumption that there is a single dominant time constant exist. The approach to calculate the actual delay is based on dynamic programming, i.e. we calculate it bottom up. The new delay model is embedded in the P-tree algorithm [3].

References

[1] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, and D. Karger. Prim-dijkstra tradeoffs for improved performance-driven global routing. *IEEE Transactions on CAD*, 14(7):890–896, 1995.

[2] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh, and C. K. Wong. Provably good performance-driven global routing. *IEEE Trans. on CAD*, 11(6):739–752, 1992.

[3] Jason Cong and Patrick H. Madden. Performance-driven routing with multiple sources.

[4] Jiang Hu and Sachin S. Sapatnekar. Far-ds: Full-plane awe routuing with driver sizing. Technical report, Department of Electrical and Computer Engineering, ???

[5] Andrew B kahng and Sudhakar Mudda. An analytical delay model for rcl interconnections. Technical report, UCLA Computer Science Department Los Angeles, ????

[6] S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14:305–321, 1995.

[7] John Lellis and Premal Buch. Table-lookup methods for improved performance-driven routing.

[8] John Lellis, Chung-Kuan Cheng, Ting-Ting Y. Lin, and Ching-Yen Ho. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing.

[9] J. S. Salowe, D. S. Richards, and D. Wrege. Mixed spanning trees. *Proc. Great Lakes Symp. on VLSI*, pages 62–66, March 1993.

[10] Vijay V. Vazirani. *Aproximation Algorithms*. Springer-Verlag, 2003.

[11] Martin Zachariasen. The rectilinear steiner tree problem: A tutorial. In *Steiner Trees in Industries*, pages 467–507. Kluwer Academic Publishers, 2001.

[12] D. Zhou, S. Su, F. Tsui, D. S. Gao, and J. Cong. Analysis of treesof transmission lines. *Technical report UCLA CSD-920010*, 1992.