Edge Weight Reduction Problems in Directed, Acyclic Graphs

Susanne E. Hambrusch [*]
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Hung-Yi Tu
Department of Computer Science and Information Management
Providence University
Taichung Hsien, Taiwan, ROC

January 6, 1997

Abstract

Let G be a weighted, directed, acyclic graph in which each edge weight is not a static quantity, but can be reduced for a certain cost. In this paper we consider the problem of determining which edges to reduce so that the length of the longest paths is minimized and the total cost associated with the reductions does not exceed a given cost. We consider two types of edge reductions, linear reductions and 0/1 reductions, which model different applications. We present efficient algorithms for different classes of graphs, including trees, series-parallel graphs, and directed acyclic graphs, and we show other edge reduction problems to be NP-hard.

Keywords: Analysis of algorithms; directed, acyclic graphs; longest path computations; series-parallel graphs; trees.

---

[*] Research supported in part by DARPA under contract DABT63-92-C-0022ONR. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing official policies, expressed or implied, of the U.S. government.

1 Introduction

Determining the longest path in a directed graph G is a problem with applications in scheduling task graphs, circuit layout compaction, and performance optimization of circuits. The problem can be solved in linear time when G is a directed, acyclic graph and it is NP-hard for general graphs [4, 5]. Consider the situation when the weight of an edge is not a static quantity, but can be reduced for a certain cost. The longest path problem arising is that of determining reductions on edge weights so that the length of the longest paths is minimized and the total cost associated with the reductions does not exceed a given cost. In this paper we consider two types of edge reductions, linear reductions and 0/1 reductions, which model different applications. We present efficient algorithms for determining edge reductions in trees, series-parallel graphs, and directed acyclic graphs, and we show other edge reduction problems to be NP-hard.

Let $G = (V, E)$ be a weighted, directed, and acyclic graph (dag) with $n+1$ vertices, $v_0, v_1, v_2, \ldots, v_n$, and m edges. Edge (v_i, v_j) has weight $d(v_i, v_j)$ with $d(v_i, v_j) \geq 0$. If not stated otherwise, we assume that G contains only one source v_0 and one sink v_n. An *edge reduction* R assigns to every edge (v_i, v_j) a non-negative quantity $r(v_i, v_j)$. The *reduced weight* $d_r(v_i, v_j)$ of edge (v_i, v_j) is a function of the edge's weight and its reduction. An edge reduction R is called a *linear reduction* if for every edge (v_i, v_j), $r(v_i, v_j)$ is a non-negative real and

$$d_r(v_i, v_j) = d(v_i, v_j) - r(v_i, v_j).$$

An edge reduction is called a *0/1 reduction* if for every edge (v_i, v_j), $r(v_i, v_j)$ is either 0 or 1 and

$$d_r(v_i, v_j) = \begin{cases} d(v_i, v_j) & \text{if } r(v_i, v_j) = 0 \\ \epsilon \times d(v_i, v_j) & \text{if } r(v_i, v_j) = 1 \end{cases}$$

where ϵ is a given real with $0 \leq \epsilon < 1$. For both reductions we require $d_r(v_i, v_j) \geq 0$.

We briefly comment on where edge reductions arise. Linear reductions model, for example, physical performance optimizations of circuits through gate resizing and buffer insertions [1, 3, 7, 8]. Such optimizations do not change the topology of the circuit and result in circuits having a smaller delay. At the same time, circuit size and power consumption increase. 0/1 reductions with $\epsilon = 0$ are a basic operation in clustering heuristics for mapping task graphs to multiprocessors [6, 9]. In a task graph, the edge weights represent the communication cost and vertices mapped to the same processor experience no communication cost. For $\epsilon > 0$, 0/1 reductions can model scenarios in which there exist fast and slow buses for communication. Reducing an edge is then equivalent to assigning the corresponding communication to a fast bus.

Given a reduction R for graph G, the *reduced graph* G_R is obtained from G by replacing each edge weight $d(v_i, v_j)$ by its reduced weight $d_r(v_i, v_j)$. Throughout, $L(G_R)$ denotes the length of the longest path in G_R and $M(G_R)$ denotes the *total reduction*; i.e., $M(G_R) = \sum_{(v_i,v_j)\in E} r(v_i, v_j)$. In this paper we investigate the following three edge reduction problems:

- *(G, L)-problem*
  Given L, find an edge reduction R^* such that $L(G_{R^*}) \leq L$ and $M(G_{R^*})$ is a minimum; i.e., for any edge reduction R' with $L(G_{R'}) \leq L$, we have $M(G_{R^*}) \leq M(G_{R'})$.
- *(G, M)-problem*
  Given M, find an edge reduction R^* such that $M(G_{R^*}) \leq M$ and $L(G_{R^*})$ is a minimum; i.e., for any edge reduction R' with $M(G_{R'}) \leq M$ we have $L(G_{R^*}) \leq L(G_{R'})$.
- *Tradeoff problem*
  Given a tradeoff function $f(G_R) = L(G_R) + \gamma \cdot M(G_R)$ defined for every edge reduction R, with γ being a constant, find an edge reduction R^* minimizing the tradeoff function.

In Section 2 we consider linear reductions in in-trees. An in-tree is a tree in which the out-degree of every vertex is at most 1. We present $O(n)$ time algorithms for solving the (G, L)-, (G, M)- and the tradeoff problems in in-trees. Section 3 presents $O(m \log m)$ time algorithms for the linear reduction problems in series-parallel graphs. Sections 4 and 5 consider 0/1 reductions. We show that for series-parallel graphs each one of the three 0/1 reductions problems can be solved in $O(m^2)$ time and that 0/1 reduction problems are NP-hard for general dags.

2 Linear reduction for in-trees

A directed tree is an *in-tree* if the out-degree of every vertex, except the root, is 1. In this section we present $O(n)$ time algorithms for the three different versions of linear edge reduction in in-trees. Clearly, our results also hold for out-trees. We point out that the algorithms for series-parallel graphs given in the next section result in $O(n \log n)$ time algorithms for in-trees.

However, the algorithms given for series-parallel graphs can handle multiple edges between two vertices (which the algorithms given below cannot).

Let v_n be the root of the in-tree. For convenience, we add an artificial source v_0 and edges (v_0, v_i) with $d(v_0, v_i) = 0$ for every leaf v_i. Even though the resulting graph is no longer an in-tree, the structure crucial to the algorithm is preserved and we refer to it as an in-tree.

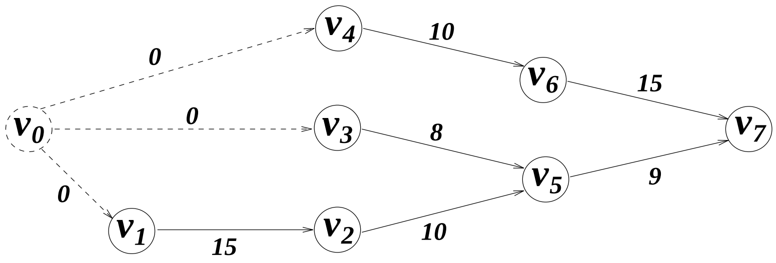
2.1 Finding an optimal reduction for a given L

In the (G, L)–problem we generate a reduction R^* satisfying $L(G_{R^*}) \leq L$ and minimizing $M(G_{R^*})$. The optimal reduction R^* generated by our algorithm satisfies the following canonical property. Let R be a reduction. R is *canonical* if for any other reduction R' with $M(G_R) = M(G_{R'})$ the length of the path from v_i to root v_n in G_R is not longer than its length in $G_{R'}$, for each vertex v_i. Stated in terms of reductions, in a canonical reduction the reductions occur as close to the root as possible. See Figure 1 for an example of two optimal reductions, one canonical and one not. Let $L_R(v_i, v_n)$ and $L_R(v_0, v_i)$ be the length of the path from v_i to v_n and the length of the longest path from v_0 to v_i in G_R, respectively. Furthermore, we refer to an edge (v_i, v_j) with $r(v_i, v_j) = d(v_i, v_j)$ (resp. $r(v_i, v_j) = 0$) as an edge with *full* (resp. *zero*) reduction. An edge (v_i, v_j) with $0 < r(v_i, v_j) < d(v_i, v_j)$ is called an edge with *partial* reduction. Lemma 2.1 gives a characterization of edge reductions in optimal canonical reductions.

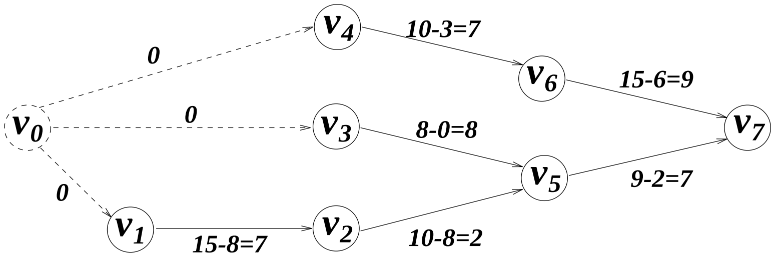
Lemma 2.1 *Let R be an optimal reduction. Then, R is the optimal canonical reduction if and only if for every path P from v_0 to v_n, if P contains reduced edges, then there exists one edge (v_i, v_j) on P such that each edge on P from v_j to v_n has full reduction and each edge on P from v_0 to v_i has zero reduction.*

Proof: Assume first that R is an optimal canonical reduction and that G_R contains a path $P = < v_0, \ldots, v_i, v_j, \ldots, v_a, v_b, \ldots, v_n >$ not satisfying the characterization. Let (v_i, v_j) and (v_a, v_b) be two distinct edges on P such that edge (v_a, v_b) has either partial or zero reduction, and edge (v_i, v_j) has either partial or full reduction. Let R' be a reduction generated from R by setting:

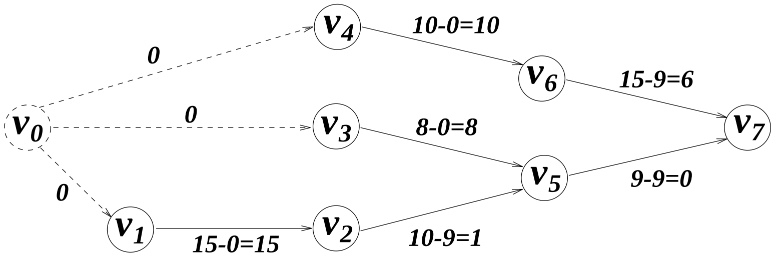
$$\begin{aligned} r'(v_a, v_b) &= \min\{d(v_a, v_b), r(v_a, v_b) + r(v_i, v_j)\} \\ r'(v_i, v_j) &= \max\{r(v_i, v_j) - d_r(v_a, v_b), 0\} \\ r'(v_p, v_q) &= r(v_p, v_q) \text{ for any other edge } (v_p, v_q). \end{aligned}$$



An intree with 8 nodes.



A non-canonical reduction.



The canonical reduction.

Figure 1: An intree with an optimal and an optimal canonical reduction, both achieving $L = 16$ and $M(G_R) = 27$.

The total reduction in R is identical to that in R'; i.e., $M(G_R) = M(G_{R'})$. R' is obtained from R by moving as much reduction as possible from edge (v_i, v_j) to edge (v_a, v_b). Thus, the length of path P in $G_{R'}$ is as in G_R. The length of every other path from v_0 to v_n is either unchanged or has been reduced. Hence, we have $L(G_R) \geq L(G_{R'})$ which implies that R' is also an optimal reduction. However, the length of the path from v_a to v_n in $G_{R'}$ now is smaller than that in G_R. This implies that R is not a canonical reduction, a contradiction.

Assume now that R is an optimal reduction and that every path from v_0 to v_n has the property stated. Assume reduction R is not canonical. This implies that there exists another optimal reduction R' and a vertex v_i on a path P from v_0 to v_n such that the length of the path from v_i to v_n in G_R is larger than that in $G_{R'}$; i.e., $L_R(v_i, v_n) > L_{R'}(v_i, v_n)$. Choose v_i as close to root v_n as possible. Let v_j be the next vertex on path P from v_i to v_n.

If edge (v_i, v_j) had zero reduction, $L_R(v_j, v_n) > L_{R'}(v_j, v_n)$ would follow and vertex v_j would be chosen instead. Hence, (v_i, v_j) has either partial or full reduction in R. Since R is an optimal reduction, there exists a longest path from v_0 to v_n in G_R which goes through vertex v_i (otherwise edge (v_i, v_j) would not need a reduction). If edge (v_i, v_j) has partial reduction in R, every edge in G_R on a path from v_0 to v_i has zero reduction and we have $L_R(v_0, v_i) \geq L_{R'}(v_0, v_i)$. Hence,

$$L(G_R) = L_R(v_0, v_i) + L_R(v_i, v_n) > L_{R'}(v_0, v_i) + L_{R'}(v_i, v_n) = L(G_{R'}),$$

contradicting the assumption that both R and R' are optimal reductions. If edge (v_i, v_j) has full reduction in R, all edges on the path from v_i to v_n also have full reduction in R. Thus, we have $L_R(v_i, v_n) = 0 \leq L_{R'}(v_i, v_n)$, contradicting our assumption $L_R(v_i, v_n) > L_{R'}(v_i, v_n)$. The lemma follows. □

While there can exist many optimal reductions, there exists only one optimal canonical reduction. We next describe how to find the optimal canonical reduction R^* in $O(n)$ time. Let $L(v_0, v_i)$ be the length of the longest path from v_0 to v_i in G. When $L(v_0, v_n) \leq L$, no edges need to be reduced and we have $r^*(v_i, v_j) = 0$ for every edge (v_i, v_j). Assume that $L(v_0, v_n) > L$. We determine R^* by setting, for every edge (v_i, v_j),

$$r^*(v_i, v_j) = \begin{cases} d(v_i, v_j) & \text{if } L \leq L(v_0, v_i) \\ L(v_0, v_i) + d(v_i, v_j) - L & \text{if } L(v_0, v_i) < L < L(v_0, v_i) + d(v_i, v_j) \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, reduction R^* gives $L(G_{R^*}) = L$. The $O(n)$ running time of the algorithm follows trivially. The following theorem completes the optimality argument of R^*.

Theorem 2.1 *Let R^* be the reduction generated by the above algorithm. Then, R^* is an optimal canonical reduction.*

Proof: From the way R^* is constructed it follows that $L(G_{R^*}) \leq L$ and that R^* is canonical (i.e., reductions occur as close to the root as possible). Assume that R^* is not optimal and let R' be the optimal canonical reduction with $M(G_{R^*}) > M(G_{R'})$ and $L(G_{R'}) \leq L$. Then, there exists an edge (v_i, v_j) with $d(v_i, v_j) \geq r^*(v_i, v_j) > r'(v_i, v_j)$. Choose edge (v_i, v_j) so that no edge on the path from v_0 to v_i qualifies. Edge (v_i, v_j) has either partial or full reduction in R^*.

From the way R^* is determined, it follows that every edge on a path from v_0 to v_i has zero reduction in R^*. Since R' is a canonical reduction, every edge on a path from v_0 to v_i in R' also has zero reduction. In R^* as well as R', every edge on the path from v_j to v_n has full reduction and there exists a longest path from v_0 to v_n containing edge (v_i, v_j). We thus have

$$L(G_{R'}) = L(v_0, v_i) + d_{r'}(v_i, v_j) > L(v_0, v_i) + d_{r^*}(v_i, v_j) = L,$$

contradicting the assumption $L(G_{R'}) \leq L$. It thus follows that R^* is an optimal canonical reduction. □

2.2 Finding an optimal reduction for a given M

We now turn to the (G, M)–problem in which we are given M and determine a reduction R^* with $M(G_{R^*}) \leq M$ minimizing the length of the longest path from v_0 to v_n. We first describe an $O(n \log n)$ time algorithm and then describe how to improve its running time to $O(n)$.

Let OPT_L(G, L) be the $O(n)$ time algorithm for solving the (G, L)–problem described in the previous section. In the (G, M)–problem we are searching for the smallest L^* such that OPT_L(G, L^*) generates a reduction R^* with $M(G_{R^*}) \leq M$. Let

$$\mathcal{M}_{min}(L) = \min\{M(G_R) | R \text{ is a reduction with } L(G_R) \leq L\}.$$

Among all reductions inducing the value of $\mathcal{M}_{min}(L)$, we only consider the optimal canonical reduction. For an optimal canonical reduction, according to Lemma 2.1, the edges close to the root receive reduction first. The number of edges receiving partial reduction is between 0 and the number of leaves in the tree. Further, for any L' and L'' with $L'' < L'$, the number of edges receiving a reduction (total or partial) in the canonical reduction inducing the value of $\mathcal{M}_{min}(L')$ is no larger than the number of edges receiving a reduction in the canonical reduction achieving L''. From the way optimal canonical reductions for a given L are determined, it thus follows that $\mathcal{M}_{min}(L)$ is piecewise linear and decreasing. In addition, $\mathcal{M}_{min}(L)$ is concave-up (i.e., the slopes are increasing in L). Figure 2(b) shows function $\mathcal{M}_{\mathrm{min}}(L)$ for the in-tree shown in Figure 2(a).

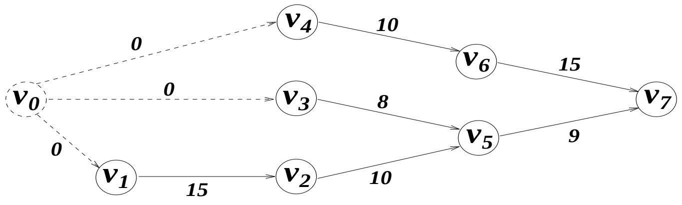
Function $\mathcal{M}_{min}(L)$ allows us to perform a binary search for L^*. Actually, the binary search we perform may not produce L^*, but a value close to it. Let $L(v_0, v_i)$ be again the length of the longest path from v_0 to v_i in G. For every vertex v_i, except the root and virtual source v_0, edge (v_i, v_j) induces the entry $L(v_0, v_i) + d(v_i, v_j)$. Let $\mathcal{L} = < L_1, L_2, \ldots, L_{n-1} >$ be the list containing these entries in non-decreasing order. List $\mathcal{L}$ is built in $O(n \log n)$ time. Assume invoking algorithm OPT_L(G, L_i) generates optimal canonical reduction R_i. Since $L_{i-1} \le L_i$, we have $M(G_{R_{i-1}}) \ge M(G_{R_i})$. Let k be the index such that

$$M(G_{R_{k-1}}) \ge M > M(G_{R_k}).$$

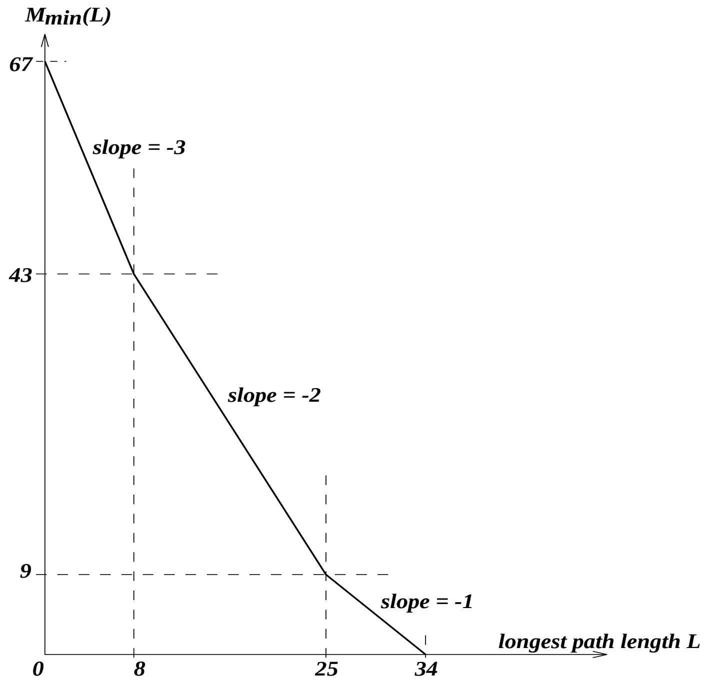
By using algorithm OPT_L and binary searching index k on list $\mathcal{L}$, index k can be determined in $O(n \log n)$ time. If $M(G_{R_{k-1}}) = M$, then R_{k-1} is the optimal reduction which we are searching for. Assume thus that $M(G_{R_{k-1}}) > M > M(G_{R_k})$. We next describe how to generate the optimal reduction R^* from the optimal canonical reductions R_{k-1} and R_k.

Since R_{k-1} and R_k are optimal canonical reductions, an edge (v_i, v_j) having full reduction in R_k also has full reduction in R_{k-1}, and thus (v_i, v_j) has full reduction in R^*. An edge (v_i, v_j) having zero reduction in R_{k-1} also has zero reduction in R_k, and thus (v_i, v_j) receives zero reduction in R^*. Let E_p be the set containing the remaining edges for which the reduction is not yet defined. Let $L_k - L_{k-1} = \delta$, $\delta > 0$. The following characterization of the edges in E_p is used in determining their reductions in R^*.

Lemma 2.2 *For every path P from v_0 to v_n in G, P contains at most one edge belonging to set E_p. In addition, for every edge (v_i, v_j) in E_p, we have $r_{k-1}(v_i, v_j) - r_k(v_i, v_j) = L_k - L_{k-1} = \delta$.*



(a) In-tree.



(b) $\mathcal{M}_{min}(L)$

Figure 2: $Function\mathcal{M}_{min}(L)$ for an in-tree.

Proof: Assume there exists a path P containing two or more edges in set E_p. Let (v_a, v_b) be the edge on P in set E_p closest to root v_n. In R_k, edge (v_a, v_b) has either partial reduction or zero reduction. (If (v_a, v_b) received full reduction in R_k, then (v_a, v_b) would have full reduction in R_{k-1}, and the edge would not be in E_p.) We only give the argument for the case when (v_a, v_b) has partial reduction in R_k. The case when (v_a, v_b) has zero reduction is handled in a similar way.

Since R_k and R_{k-1} are optimal canonical reduction, we know the following. Edge (v_a, v_b) has full reduction in R_{k-1}. (If it had partial or zero reduction, (v_a, v_b) would be the only edge on P in set E_p.) Let (v_c, v_a) be the in-coming edge on path P incident to v_a. Edge (v_c, v_a) has zero reduction in R_k and it has either full or partial reduction in R_{k-1}. (If (v_c, v_a) had zero reduction in R_{k-1}, all edges on path P from v_0 to v_c would have zero reduction and P would not contain two edges belonging to E_p.)

Since R_{k-1} is generated by invoking OPT_L(G, L_{k-1}) and edge (v_c, v_a) has either full or partial reduction in R_{k-1}, according to the reduction-setting rules of algorithm OPT_L we have

$$L_{k-1} < L(v_0, v_c) + d(v_c, v_a).$$

By the similar arguments, since edge (v_c, v_a) has zero reduction in R_k, we have

$$L(v_0, v_c) + d(v_c, v_a) \leq L_k.$$

The quantity $L(v_0, v_c) + d(v_c, v_a)$ induces an entry, say L_q, in list $\mathcal{L}$. We thus have $L_{k-1} < L_q \leq L_k$, contradicting our assumption that L_{k-1} and L_k are consecutive entries in list $\mathcal{L}$. Hence, path P contains at most one edge belonging to set E_p.

Now, we prove that for every edge (v_i, v_j) in E_p, we have $r_{k-1}(v_i, v_j) - r_k(v_i, v_j) = L_k - L_{k-1} = \delta$. Let edge (v_a, v_b) be an edge in E_p. When (v_a, v_b) has partial reduction in both R_k and R_{k-1}, we have

$$r_k(v_a, v_b) = L(v_0, v_a) + d(v_a, v_b) - L_k$$

and

$$r_{k-1}(v_a, v_b) = L(v_0, v_a) + d(v_a, v_b) - L_{k-1}.$$

Since $L_k - L_{k-1} = \delta$, we have $r_{k-1}(v_a, v_b) = L(v_0, v_a) + d(v_a, v_b) - L_k + \delta = r_k(v_a, v_b) - \delta$.

Hence, $r_{k-1}(v_a.v_b) - r_k(v_a, v_b) = \delta$ follows. The other three cases of possible reductions on edge (v_a, v_b) in R_k and R_{k-1} are handled in a similar manner. $\square$

We can now state how R^* is generated from R_k and R_{k-1}. We set

$$r^*(v_i, v_j) = \begin{cases} d(v_i, v_j) & \text{if } r_k(v_i, v_j) = d(v_i, v_j) \quad (1) \\ 0 & \text{if } r_{k-1}(v_i, v_j) = 0 \quad (2) \\ r_k(v_i, v_j) + \frac{M - M(G_{R_k})}{|E_p|} & (v_i, v_j) \in E_p \quad (3) \end{cases}$$

The justifications for (1) and (2) have already been given. $M - M(G_{R_k})$ represents the amount of reduction that can be spent in addition to $M(G_{R_k})$. This remaining amount is evenly distributed among the edges in E_p. It remains to show that using (3) gives $r^*(v_i, v_j) \leq d(v_i, v_j)$. Lemma 2.2 implies $M(G_{R_{k-1}}) - M(G_{R_k}) = \delta \times |E_p|$, where $L_{k-1} + \delta = L_k$ and $\delta > 0$. Since $M(G_{R_{k-1}}) > M$, we have $\frac{M - M(G_{R_k})}{|E_p|} < \delta$ and thus

$$\begin{aligned} r^*(v_i, v_j) &= r_k(v_i, v_j) + \frac{M - M(G_{R_k})}{|E_p|} \\ &< r_k(v_i, v_j) + \delta \\ &= r_{k-1}(v_i, v_j) \\ &\leq d(v_i, v_j). \end{aligned}$$

In summary, given index k with $M(G_{R_{k-1}}) > M > M(G_{R_k})$, the optimal reduction R^* for a (G, M)–problem can be generated in $O(n)$ time. An $O(n \log n)$ overall time bound for our algorithm for solving the (G, M)–problem follows. The remainder of this section describes how to reduce the running time to $O(n)$ by using prune-and-search. Our improved algorithm also performs $O(\log n)$ searches to determine index k, but each search reduces an upper bound on the size of the relevant data by half.

Let $\mathcal{L}$ now be the unsorted list containing the entries $L(v_0, v_i) + d(v_i, v_j)$. Assume that at the beginning of each iteration we have identified in list $\mathcal{L}$ two entries L_a and L_b with $L_a < L_k < L_b$. For the first iteration we set $L_a = -\infty$ and $L_b = +\infty$. Let R^* be the optimal reduction. In the beginning of each iteration, the edges of G are partitioned into four sets, E_z, E_u, E_p, and E_f:

- Set E_z contains the edges which have zero reduction in both R_a and R_b. These edges will receive zero reduction in R^*.

- Set E_f contains the edges which have full reduction in both R_a and R_b. These edges will receive full reduction in R^*.

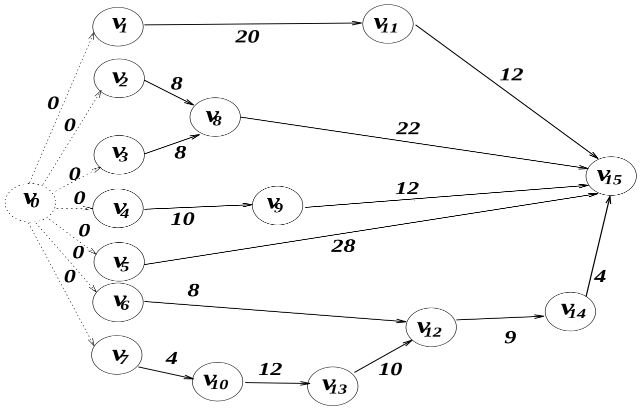
- Set E_p contains edges for which it has already been determined that they have partial reduction in R^*. This includes the edges having partial reduction in both R_a and R_b. In addition, edge (v_i, v_j) belongs to E_p if (i) (v_i, v_j) has full reduction in R_a and partial reduction in R_b, and (ii) every edge going to vertex v_i has zero reduction in R_a.
- Set E_u contains all edges not included in set E_z, E_p, and E_f. For each edge in E_u, the type and the amount of reduction remains to be decided.

Figure 3 gives an example on how edges are partitioned. In Figure 3(b) we drew the in-tree so that associations to edge sets can be seen more easily. Edges completely to the left of vertical line $L_a = 8$ belong to E_z and edges completely to the right of the vertical line $L_b = 26$ belong to E_f. Thus, $E_z = \{(v_2, v_8), (v_3, v_8), (v_6, v_{12}), (v_7, v_{10})\}$ and $E_f = \{(v_{12}, v_{14}), (v_{14}, v_{15})\}$. Using the rules stated above to partition the remaining edges, we get $E_p = \{(v_5, v_{15}), (v_8, v_{15})\}$, and $E_u = \{(v_1, v_{11}), (v_{11}, v_{15}), (v_4, v_9), (v_9, v_{15}), (v_{10}, v_{13}), (v_{13}, v_{12})\}$.

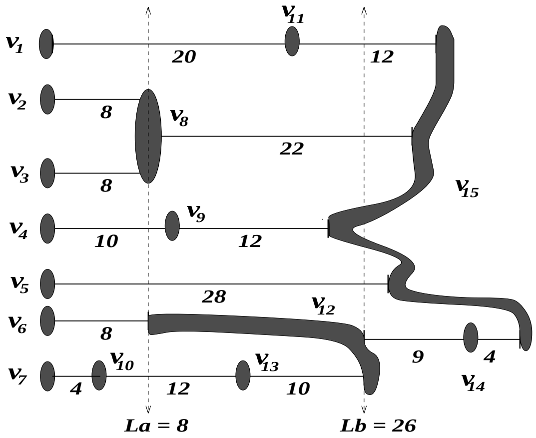
Next we describe how to perform binary search so that the size of the relevant data reduces by half in each iteration. Let $\mathcal{L}_{ab}$ be the sublist of $\mathcal{L}$ containing the entries L_j with $L_a < L_j < L_b$, $n_{ab} = |\mathcal{L}_{ab}|$. The relationship between n_{ab} and $|E_u|$ is crucial. Observe that not every edge in set E_u induces an entry in sublist $\mathcal{L}_{ab}$. For example, edge $(v_{11}, v_{15}) \in E_u$ induces the value 32 which is not between $L_a = 8$ and $L_b = 26$. However, an edge $(v_i, v_j) \in E_u$ satisfying one of the two following conditions induces an entry in list $\mathcal{L}_{ab}$:

1. (v_i, v_j) has partial reduction in R_a and zero reduction in R_b, or
2. (v_i, v_j) has full reduction in R_a and zero reduction in R_b.

The first condition applies, for example, to edge $(v_4, v_9) \in E_u$ which induces the entry 10, and the second condition applies, for example, to edge $(v_9, v_{15}) \in E_u$ which induces the entry 22. Only an edge in E_u that has full reduction in R_a and partial reduction in R_b does not induce an entry to list $\mathcal{L}_{ab}$. Such an edge in E_u is incident to at least one other edge in E_u which induces an entry in $\mathcal{L}_{ab}$. Observe that each path from v_0 to v_n contains at most one edge from E_u not inducing an entry in $\mathcal{L}_{ab}$ and that it is always the edge in E_u closest to the root. Hence, the number of edges in E_u can at most double the number of entries in $\mathcal{L}_{ab}$; i.e., $|E_u| \leq 2n_{ab}$.



(a) In-tree.



(b) In-tree drawn to indicate partition into sets when $L_a = 8$ and $L_b = 26$.

Figure 3: Partitioning the edges.

We next describe our procedure for searching for index k. Let M_f be the total reduction spent on the edges in set E_f; i.e., $M_f = \sum_{(v_i,v_j)\in E_f} d(u,v)$. Let $M_{p,a}$ be the total reduction made on the edges of set E_p in reduction R_a. Let L_q be the $\frac{n_{ab}}{2}$–th smallest element in list $\mathcal{L}_{ab}$. Recall that this list contains all entries L_i with $L_a < L_i < L_b$. Let $\delta = L_q - L_a$. We next determine the reduction on each edge of E_u in R_q using the method described earlier. Depending on whether an edge of E_u receives zero reduction, partial reduction, or full reduction in R_q, we partition E_u into three sets, $E_{u,z}$, $E_{u,p}$, and $E_{u,f}$, respectively. The total reduction of reduction R_q is determined as follows:

$$M(G_{R_q}) = M_f + (M_{p,a} - \delta \times |E_p|) + M_{u,f} + M_{u,p},$$

with

$$M_{u,f} = \sum_{(v_i,v_j)\in E_{u,f}} d(u,v) \text{ and } M_{u,p} = \sum_{(v_i,v_j)\in E_{u,p}} r_q(v_i,v_j).$$

If now $M(G_{R_q}) = M$, then we have $R^* = R_q$ and the algorithm terminates. Consider first the case when $M(G_{R_q}) > M$. L_q is a new lower bound (since $L_q < L_k < L_b$ holds) and the next iteration continues with L_q and L_b. The edge sets and reductions are updated as follows.

1. The edges in $E_{u,z}$ are added to E_z and are deleted from E_u.
2. Edges from $E_{u,p}$ and $E_{u,f}$ that qualify for E_p are moved from set E_u to E_p. The total reduction made on the edges in the new set E_p in reduction R_q is computed.

Assume now that $M(G_{R_q}) < M$. In this case we have found a new upper bound and continue the next iteration with L_a and L_q. The edge sets and reductions are now updated as follows.

1. The edges in $E_{u,f}$ are added to E_f and are deleted from E_u. M_f is updated.
2. Edges from $E_{u,p}$ that qualify for E_p are moved from set E_u to E_p. The total reduction made on the edges in the new set E_p in reduction R_q is computed.

It is easy to see that the work done in an iteration is bounded by $O(|E_u|)$. That the upper bound on the number of edges in E_u reduces by half from one iteration to the next is seen as follows. First, an edge $(u,v) \in E_u$ with $L(v_0,u) + d(u,v) = L_q$ is no longer in E_u by the end of the iteration. Hence, when $L_q < L_k < L_b$, we have $n_{qb} \leq \frac{n_{ab}}{2}$ and when $L_a < L_k < L_q$ we have $n_{aq} \leq \frac{n_{ab}}{2}$. We already argued that $|E_u| \leq 2n_{ab}$. This implies that the size of the new set $|E_u|$ is bounded by n_{ab} and the upper bound on $|E_u|$ reduces by half. Hence, searching for index k takes $O(n)$ time. After having determined index k, reduction R^* is generated from R_k and R_{k-1} in $O(n)$ time as described earlier. The $O(n)$ time bound for the (G, M)–problem follows.

2.3 Optimal reduction for the tradeoff problem

The approach used for the (G, M)–problem leads to an $O(n)$ time solution for the tradeoff problem in in-trees. Recall that in the tradeoff problem we are to determine a reduction R^* minimizing the tradeoff function $f(G_R) = L(G_R) + \gamma \cdot M(G_R)$. As stated in the previous section, $\mathcal{M}_{min}(L)$ represents the minimum total reduction needed to reduce the longest path length to L, and $\mathcal{M}_{min}(L)$ is a piecewise linear, decreasing and concave-up function. We can thus represent $\mathcal{M}_{min}(L)$ by a sequence of linear functions of L, $a_1 \times L + b_1, a_2 \times L + b_2, \ldots, a_{n-2} \times L + b_{n-2}$, with all a_j's being negative. Function $a_i \times L + b_i$ is associated with interval, $[L_i, L_{i+1}]$, $1 \leq i \leq n-2$, where the L_i-values are as defined in the previous section. In interval $[L_i, L_{i+1}]$, $\mathcal{M}_{min}(L)$ is described by $a_i \times L + b_i$. Since $\mathcal{M}_{min}(L)$ is concave-up, we have $a_1 \leq a_2 \leq \cdots \leq a_{n-2} < 0$. Function $f(G_R)$ can be re-written as a function of the longest path length L; i.e., $\mathcal{F}(L) = L + \gamma \cdot \mathcal{M}_{min}(L)$. Minimizing $f(G_R)$ is equivalent to minimize $\mathcal{F}(L)$. We distinguish between the following four cases.

Case 1. $1 + \gamma \cdot a_{n-2} < 0$.

In this case the minimum of $\mathcal{F}(L)$ occurs at $L = L_{n-1}$.

Case 2. $1 + \gamma \cdot a_1 > 0$.

In this case the minimum of $\mathcal{F}(L)$ occurs at $L = L_1$.

Case 3. There exists an a_j such that $1 + \gamma \cdot a_j = 0$.

In this case the minimum of $\mathcal{F}(L)$ occurs at $L = L_j$.

Case 4. There exists an a_j such that $1 + \gamma \cdot a_j < 0$ and $1 + \gamma \cdot a_{j+1} > 0$.

In this case the minimum of $\mathcal{F}(L)$ occurs at $L = L_{j+1}$.

The heart of the algorithm is the search for index j in Cases 3 and 4 without generating the whole $\mathcal{M}_{min}(L)$ function. Index j can be determined in $O(n)$ time by using an approach similar to the one used for the (G, M)–problem. In each iteration we again have a lower bound L_a, an upper bound L_b, and a new value L_q. The value of a_q can be determined in $O(|E_u|)$ time and the upper bound on $|E_u|$ is reduced by half in each iteration. We omit the details of the $O(n)$ time search algorithm.

3 Linear reduction for series-parallel graphs

In this section we present $O(m \log m)$ time algorithms for performing linear edge reduction in series-parallel graphs. The graphs can now have multiple edges between two vertices (thus m could be arbitrarily larger than n). We start by giving the necessary definitions regarding series-parallel graphs and outline a dynamic programming solution. We first give an $O(m^2)$ time algorithm and then describe how to improve the running time to $O(m \log m)$.

A *series-parallel graph* (sp-graph for short) G is a dag with exactly one source v_0 and one sink v_n, recursively defined as follows:

1. A dag consisting of a single edge from v_0 to v_n is an sp-graph.
2. Given two sp-graphs G_1 and G_2, the dag G_3 obtained by identifying the sources of G_1 and G_2 with each other and by identifying the sinks of G_1 and G_2 with each other is an sp-graph. This type of operation is called a *parallel composition.*
3. Given two sp-graphs G_1 and G_2, the dag G_3 obtained by identifying the source of G_1 with the sink of G_2 is an sp-graph. This type of operation is called a *series composition.*

An sp-graph G can be represented by its *decomposition tree* D. Each node N of decomposition tree D corresponds to a subgraph G_N of G. A leaf of D corresponds to a single edge of G. If N is an internal node of D, then G_N corresponds to the subgraph of G obtained by either a parallel or a series composition of the subgraphs associated with the children of N. Testing whether a given dag G on n vertices and m edges is an sp-graph can be done in $O(m)$ time [10]. Furthermore, the decomposition tree D for a given sp-graph G can be constructed in $O(m)$ time by using the recognition algorithm in [10].

Let N be a node in the decomposition tree and let G_N be the associated subgraph of G. Let $\mathcal{M}_N(L)$ be the minimum edge reduction reducing the length of the longest path in G_N to L. In the following we show how to determine function $\mathcal{M}_N(L)$ for the root of the decomposition tree. All three reduction problems can be solved using the function associated with the root. The $\mathcal{M}_N(L)$ functions are computed in a bottom-up fashion from the decomposition tree. Let (v_i, v_j) be an edge of G corresponding to leaf N of the decomposition tree. Then, function $\mathcal{M}_N(L)$ is defined in the interval $[0, d(v_i, v_j)]$ by the linear segment $a \times L + b$ with $a = -1$ and $b = d(v_i, v_j)$. For $L > d(v_i, v_j)$ the value of the function is 0. Consider now an internal node N of the decomposition tree which has two children, N_1 and N_2. Assume the functions $\mathcal{M}_{N_1}(L)$ and $\mathcal{M}_{N_2}(L)$ associated with these children have already been determined. If node N represents a parallel composition of graphs G_{N_1} and G_{N_2}, then we have

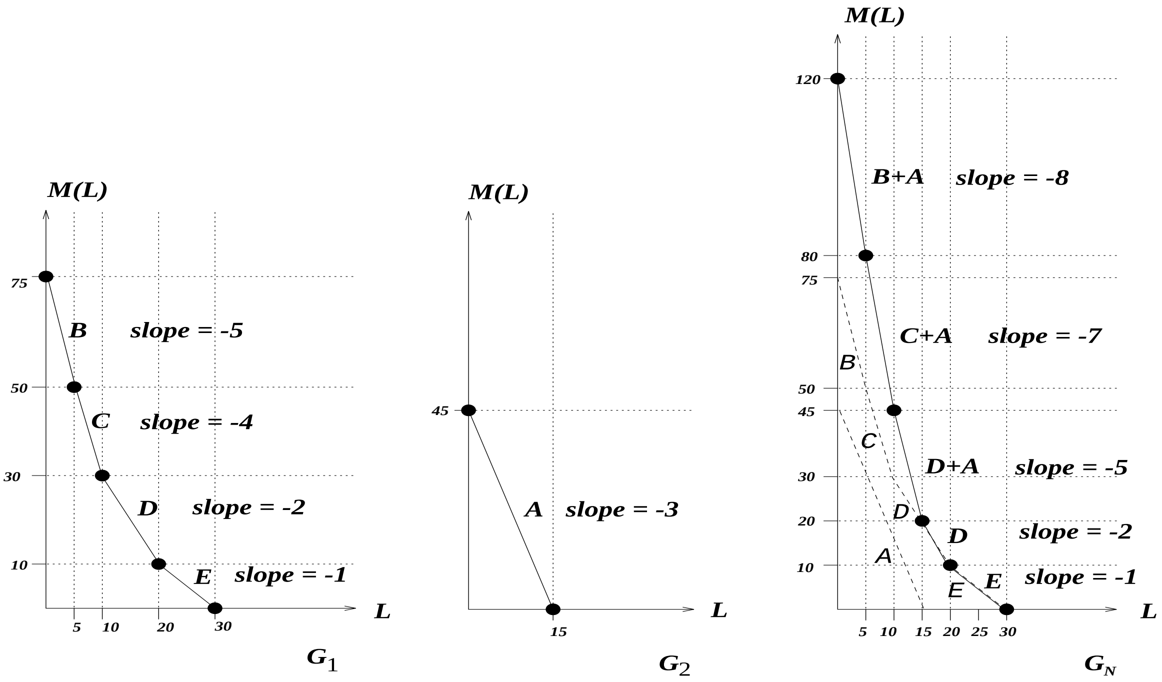
$$\mathcal{M}_N(L) = \mathcal{M}_{N_1}(L) + \mathcal{M}_{N_2}(L).$$

If node N represents a series composition of graphs G_{N_1} and G_{N_2}, function $\mathcal{M}_N(L)$ is defined as

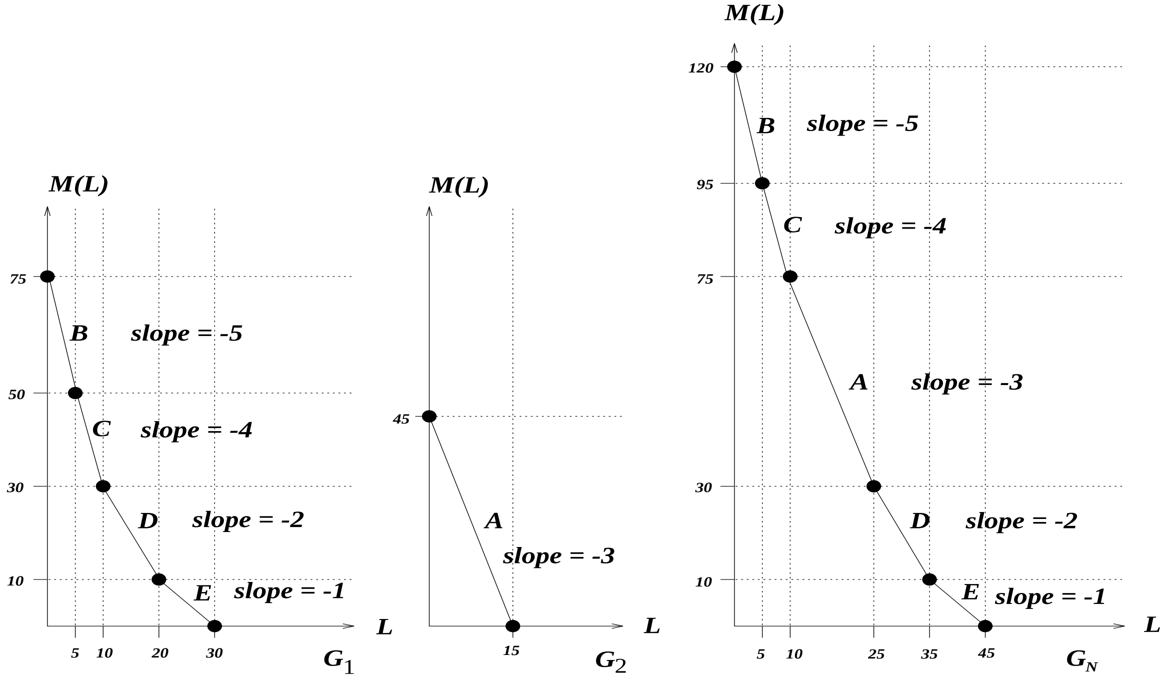
$$\mathcal{M}_N(L) = \min\{\mathcal{M}_{N_1}(L_1) + \mathcal{M}_{N_2}(L_2) | L = L_1 + L_2\}.$$

Figures 4(a) and 4(b) show the result of such an operation for parallel and series composition, respectively. The time spent on computing function $\mathcal{M}_N(L)$ depends on its representation. We next show that $\mathcal{M}_N(L)$ is piecewise linear. The function associated with a leaf of the decomposition tree consists of a single linear segment and is thus piecewise linear. Assume that the functions associated with N_1 and N_2 consist of k_1 and k_2 linear segments, respectively. A parallel composition adds corresponding values and thus generates a function consisting of at most $k_1 + k_2$ linear segments. Consider a series composition. Let L' and L'' be two values so that there exists no interval endpoint between L' and L''. Then, the change in reduction occurring between L' to L'' corresponds to either a reduction in G_{N_1} or in G_{N_2}. When an endpoint is reached, this can change (the reduction may now occur in the other subgraph or in the same subgraph at a different rate). A series composition thus generates a linear function consisting of at most $k_1 + k_2$ linear segments, each coming from either $\mathcal{M}_{N_1}(L)$ or $\mathcal{M}_{N_2}(L)$.

Consider representing each function as a list of intervals, where each interval is associated with a linear function. For example, if $\mathcal{M}_N(L)$ consists of k linear functions, $a_1 \times L + b_1, a_2 \times$



(a) $\mathcal{M}(L)$-function for a parallel composition.



(b) $\mathcal{M}(L)$-function for a series composition.

Figure 4: $\mathcal{M}(L)$-functions built through parallel composition and series composition.

$L + b_2, \ldots, a_k \times L + b_k$, then $a_i \times L + b_i$ is associated with an interval $[L_i, L_{i+1}]$, $1 \leq i \leq k$. (Recall that the function is 0 for positions greater than L_{k+1}.) Using this representation, a parallel or series composition can be done in time linear in the number of intervals. The time needed to determine the $\mathcal{M}_N(L)$–function corresponding to the root of the decomposition tree is then bounded by $O(m^2)$.

We are able to reduce the time to $O(m \log m)$ by employing a different representation of the functions and making use of the following two properties. The $\mathcal{M}_N(L)$–functions are concave-up (i.e., the slopes are increasing in L with $a_1 < a_2 < \ldots < a_k$) and they are monotonically decreasing (i.e., the slopes are negative). Assume each function associated with a child of a node has increasing slopes (this trivially holds for the leaves). Then, a series composition merges sorted lists and a parallel composition adds the values of two sorted lists. Both operations result in increasing values and thus an $\mathcal{M}_N(L)$ function with increasing slopes. Monotonically decreasing follows from the definition of the $\mathcal{M}_N(L)$ functions.

Before describing a more efficient representation of the functions, we briefly discuss how $\mathcal{M}_N(L)$ is generated. In a parallel composition, we insert each interval of $\mathcal{M}_{N_2}(L)$ into $\mathcal{M}_{N_1}(L)$. Let $[L_i, L_{i+1}]$ be an interval in $\mathcal{M}_{N_2}(L)$ having slope a_i. We identify in $\mathcal{M}_{N_1}(L)$ the line segment containing position L_{i+1}. Assume it is segment $[L_j, L_{j+1}]$ with slope a_j. This segment is transformed into two new segments: $[L_j, L_{i+1}]$ whose slope is $a_i + a_j$ and segment $[L_{i+1}, L_{j+1}]$ whose slope is a_j. The slope of every segment in $\mathcal{M}_{N_1}(L)$ to the left of L_j and in the range $[L_i, L_j]$ increases by a_i. Offsets change accordingly. In Figure 4(a), $\mathcal{M}_{N_2}(L)$ corresponds to the interval $[0, 15]$ with the linear function $-3 \times L + 45$. Inserting interval $[0, 15]$ into $\mathcal{M}_{N_1}$ results in the interval $[10, 20]$ being split into intervals $[10, 15]$ and $[15, 20]$. Interval $[10, 15]$ has slope $(-2) + (-3) = -5$ and offset $50 + 45 = 95$.

In a series composition, we also create $\mathcal{M}_N(L)$ by inserting each interval of $\mathcal{M}_{N_2}(L)$ into $\mathcal{M}_{N_1}(L)$. Let $[L_i, L_{i+1}]$ be an interval in $\mathcal{M}_{N_2}(L)$ having slope a_i. We identify in $\mathcal{M}_{N_2}(L)$ the intervals $[L_j, L_{j+1}]$ with slope a_j and $[L_{j+1}, L_{j+2}]$ with slope a_{j+1} such that $a_j \leq a_i < a_{j+1}$. We create the interval $[L_{j+1}, L_{j+1} + (L_{i+1} - L_i)]$ having slope a_i and increase the position of every endpoint to the right of L_{j+1} by $L_{i+1} - L_i$. For $a_i = a_j$, we can view segment $[L_j, L_{j+1}]$ as being extended. The offset of every interval to the left of L_{j+1} increases accordingly. For example, in

Figure 4(b) the series composition inserts segment A with slope -3 between segments C and D. The intervals associated with linear segments D and E are shifted to the right by 15. The offset of linear segments A, B, and C increases by 30.

The $O(m \log m)$ time is achieved by representing each function $\mathcal{M}_N(L)$ in a balanced binary tree, called the *function tree* T_N. Balanced tree representations are also used in [2] for maximum flow problems in sp-graphs. The leaves of a function tree correspond to the endpoints of intervals arranged according to increasing positions (and thus increasing slope). For interval $[L_i, L_{i+1}]$ we thus have an entry for L_i and one for L_{i+1}. The slope and offset of interval $[L_i, L_{i+1}]$ are associated with the leaf corresponding to L_i. The function tree does not have the position of L_i, the offset and slope of segment $[L_i, L_{i+1}]$ stored explicitly at the leaf. Each one of these three values is stored in a distributed fashion on the path from the leaf to the root of T_N. Every node v of T_N contains three entries, $subp(v)$, $subs(v)$, and $subo(v)$. Let u be a leaf. Then, the position of the endpoint represented by u is the sum of the $subp(\cdot)$-values encountered on the path from u to the root of the function tree. Slope and offset are obtained by adding the $subs(\cdot)$ and $subo(\cdot)$-entries, respectively, on the same path. In addition to these entries, we maintain at every node v of the tree an entry $maxp(v)$ which contains the sum of the $subp(\cdot)$-values along the rightmost path in the subtree rooted at v. Also, we maintain at every node v an entry $maxs(v)$ which contains the sum of the $subs(\cdot)$-values on the same rightmost path.

Assume we are given two function trees T_1 and T_2 with m_1 and m_2 leaves, respectively. Assume $m_1 \geq m_2$. We can generate, in $O(m_2)$ time, the intervals represented in T_2, as well as their slope and offset. Consider first a parallel composition. We describe how an interval $[L_i, L_{i+1}]$ with slope a_i and offset b_i is inserted into T_1, assuming all intervals to the left of $[L_i, L_{i+1}]$ were already inserted. However, in the actual implementation the insertions are processed simultaneously. We insert a new leaf v representing L_{i+1}, using $maxp(\cdot)$- and $subp(\cdot)$-entries to guide the search. The difference between the sum of the $subp(\cdot)$-values from the root to the new leaf and L_{i+1} determines the value of $subp(v)$. Intervals to the left of v and greater than or equal to L_i experience an increase in the slope by a_i. Observe that there exists a leaf corresponding to endpoint L_i and thus recording this increase corresponds to updating $subs(\cdot)$-entries on a single path. The updating of the offset entries is done in a similar way. Remaining balancing issues arising in the insertion are straightforward and are omitted.

Consider next the case when T_1 and T_2 are combined through a series composition. Then, the insertion of an interval $[L_i, L_{i+1}]$ with slope a_i and offset b_i into T_1 is handled as follows. We determine the position of a new leaf v having slope a_i by using the $maxs(\cdot)$- and $subs(\cdot)$-entries. The position of each leaf to the right of v (including v) increases by $L_{i+1} - L_i$. To record this, we increase the $subp(\cdot)$-values of right children of the nodes on the path from the root to v. The insertion does not change the slope values of other nodes. Entry $subs(v)$ is set to the difference between a_i and the the sum of the $subs(\cdot)$-values from the root to v. The entries $subo(\cdot)$ are updated in a similar way.

It is clear that inserting one interval into T_1 costs $O(\log m_1)$ time, where m_1 is the current number of leaves in function tree T_1. Handling the $m_2 - 1$ intervals one after the other gives $O(m_2 \log m_1)$ time for combining two function trees. However, by handling the $m_2 - 1$ insertions simultaneously, function trees T_1 and T_2 can be combined in $O(m_2 \log \frac{m_1+m_2}{m_2})$ time.

We first insert into T_1 the $m_2 - 1$ new leaves. Then, the balancing and updating of entries proceeds level by level within T_1. Assume that the number of leaves between the j-th and $(j+1)$-st new leaf is n_j. Then, the total time needed to update and balance the new function tree is bounded by

$$O(m_2 + \log m_1 + \sum_{j=1}^{m_2-1} (1 + \log n_j))$$

which is

$$O(\log m_1 + m_2 \log \frac{m_1 + m_2}{m_2}).$$

This holds since $\sum_{j=1}^{m_2-1} n_j \leq m_1$ and the work is maximized when the newly inserted leaves are as far apart as possible. The function associated with the root of the decomposition tree of an sp-graph can thus be determined in time

$$T(m) = \max_{m_1+m_2=m, m_2 \leq m_1} \{T(m_1) + T(m_2) + O(\log m_1 + m_2 \log \frac{m_1 + m_2}{m_2})\}$$

which is $O(m \log m)$.

Once function $\mathcal{M}(L)$ associated with the root of decomposition tree D has been determined, all three reduction problems can be solved in $O(m \log m)$ time. For the (G, L)-problem we simply compute $\mathcal{M}(L)$. The reductions on the edges can be generated by traversing the tree from the root back to the leaves and using the information stored in the function tree associated with each node. This can be done within the $O(m \log m)$ time bound. For the (G, M)-problem, we determine the smallest L such that $\mathcal{M}(L) \leq M$. Again, determining the reduction on the edges is done by traversing the decomposition tree and the associated function trees once more. To find the optimal tradeoff between M and L, we build function $f(L) = L + \gamma \cdot \mathcal{M}(L)$ by using the function tree associated with the root of the decomposition tree. Then, we determine the L resulting the minimum of $f(L)$. To determine the reduction giving the minimum of $f(L)$ we again traverse the decomposition tree and its associated function trees.

We conclude this section by pointing out that the linear reduction problems can be solved in polynomial time for general dags by phrasing them as linear programs. For example, the (G, M)-problem can be formulated as follows. Let $t_0, t_1, \ldots, t_n$ and $r(v_i, v_j)$ for every edge (v_i, v_j) in G be the variables. Then,

$$\begin{array}{lll}
\text{Minimize} & t_n - t_0 & \\
\text{subject to} & t_i + d(v_i, v_j) - r(v_i, v_j) \leq t_j & \text{for every } (v_i, v_j) \in E \\
 & d(v_i, v_j) - r(v_i, v_j) \geq 0 & \text{for every } (v_i, v_j) \in E \\
 & \sum_{(v_i, v_j) \in E} r(v_i, v_j) \leq M & \\
 & t_0 = 0 \text{ and } t_i \geq 0 & \text{for } 1 \leq i \leq n
\end{array}$$

4 0/1 reduction for series-parallel graphs

We now turn to 0/1 edge reductions. The cost of a reduction now corresponds to the number of edges reduced. The weight of a reduced edge is $\epsilon \times d(v_i, v_j)$, where ϵ is given, $0 \leq \epsilon < 1$. In this section we use an approach similar to the one used for linear edge reductions for sp-graphs to solve 0/1 edge reductions for sp-graphs in $O(m^2)$ time. Our algorithms allows multiple edges between two vertices.

Let D be again the decomposition tree of sp-graph G. Let N_i be a node of D and let G_{N_i} be the subgraph of G corresponding to the subtree of D rooted at vertex N_i. Assume that G_{N_i} has m_i edges. For vertex N_i we construct an array T_i of size $m_i + 1$. Entry $T_i[j]$ represents the minimum length of the longest path in G_{N_i} when at most j edges are reduced. We thus have $T_i[0] \geq T_i[1] \geq T_i[2] \geq \ldots \geq T_i[m_i - 1] \geq T_i[m_i]$. The T_i-arrays are determined from the decomposition tree in a bottom-up fashion, with a node using the arrays associated with its children. The final answer for all three reduction problems is determined from the array generated for the root of D.

If node N_i is a leaf of decomposition tree D, G_{N_i} corresponds to a single edge. Assume this edge is (v_a, v_b). Array T_i has size two and we have $T_i[0] = d(v_a, v_b)$ and $T_i[1] = \epsilon \times d(v_a, v_b)$.

If N_i is not a leaf, T_i is constructed as follows. Assume N_i has two children, N_l and N_r, and that arrays T_l and T_r have already been determined. If node N_i represents a parallel composition of graphs G_{N_l} G_{N_r}, the entries in T_i can be defined as follows:

$$T_i[j] = \min_{p+q=j} \{\max\{T_r[p], T_l[q]\}\}.$$

By making use of the fact that the entries in arrays T_r and T_l are sorted, T_i can be constructed in $O(m_i)$ time. One possible solution is given below.

We determine T_i by scanning arrays T_l and T_r twice, each time from right to left. During the first scan of the arrays we determine the entries of T_i induced by entries in array T_r. Assume the scan in T_r is at position p. We determine the smallest q such that $T_l[q-1] > T_r[p] \geq T_l[q]$. Let $j = p + q$. Then, $T_r[p]$ is a possible solution for $T_i[j]$. If we already recorded a better solution for $T_i[j]$, we discard p and q. Otherwise, we record it as the currently best one. We then consider $T_r[p-1]$. When we now search for an entry in array T_l, we search for an index q' with $q' \leq q$. Hence, all requests made to array T_l can be satisfied by executing one right to left scan. We then scan both arrays again to determine the entries of T_i induced by entries in array T_l. Finally, a left to right scan of array T_i is performed. We may have recorded in $T_i[j+a]$ a solution that is worse than the one recorded in $T_i[j]$. (Observe that a solution recorded for $T_i[j]$ is also a solution for $T_i[j+a]$ with $a > 1$.) Hence, we propagate the solution recorded in $T_i[j]$ to the right until a better solution is encountered. In total, it takes $O(m_i)$ times to generate T_i from lists T_l and T_r.

If node N_i represents a series composition of graphs G_{N_l} G_{N_r}, the entries in T_i can be defined by

$$T_i[j] = \min_{p+q=j} \{T_r[p] + T_l[q]\}.$$

Let m_i, m_l, and m_r be the number of edges in the graphs G_{N_i}, G_{N_l} G_{N_r}, respectively, with $m_i = m_l + m_r$. We construct T_i by enumerating the values of $T_r[p] + T_l[q]$ for all pairs of (p, q), $0 \leq p \leq m_r$ and $0 \leq q \leq m_l$. This takes $O(m_l m_r)$ time.

Let $C(N_i)$ be the cost to compute table T_i for node N_i. Then we have

$$C(N_i) \leq C(N_l) + C(N_r) + m_l m_r$$

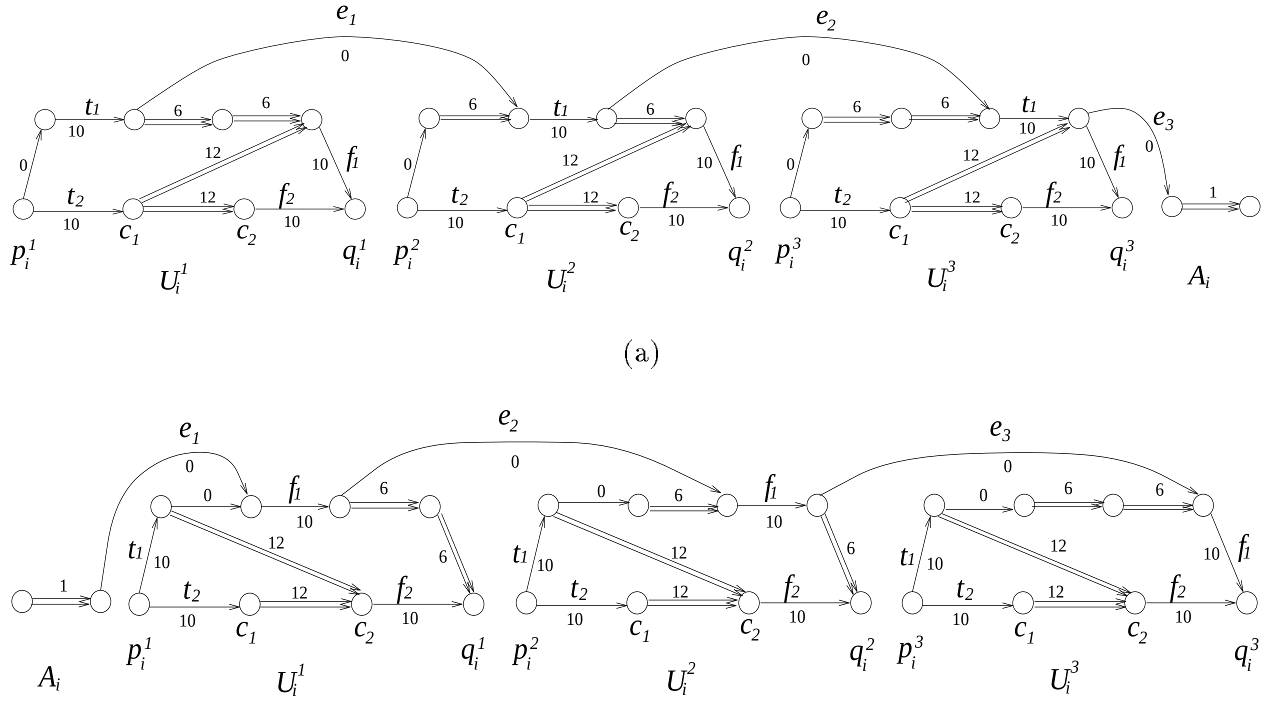
and thus $C(N_i) = O(m_i^2)$. Hence, the array T_{root} associated with the root of decomposition tree D can be determined in $O(m^2)$ time. The three reduction problems can now be solved in $O(m^2)$ time as follows. For the (G, L)-problem we determine the smallest j such that $T_{root}[j] \leq L$. Quantity j represents the minimum number of edges that need to be reduced in order to achieve the path length of at most L. By traversing the tree from the root back to the leaves and using the list associated with each vertex, the edges receiving a reduction can be determined in an additional $O(m)$ time. For (G, M)-problem, entry $T_{root}[M]$ represents the minimum longest path length that can be obtained by reducing at most M edges. Clearly, the size of the array associated with a vertex does not have to exceed M. Again, determining which edges get reduced is done by traversing the tree once more. To find the optimal tradeoff between M and L, we evaluate $T_{root}[j] + \gamma \cdot j$ for $0 \leq j \leq m$. The pair $(T_{root}[j], j)$ resulting the minimum tradeoff value gives the solution to the tradeoff problem.

5 0/1 Reduction for general dags

In this section we show that 0/1 reduction problems are NP-hard for general dags. The theorem below proves that the corresponding decision problem is NP-complete for $\epsilon = 0$. By changing the weights of the edges in the graph constructed, NP-completeness follows for other values of ϵ. We discuss the weight changes for $\epsilon = \frac{1}{2}$ at the end of this section.

Theorem 5.1 *Given a weighted dag G and two positive reals M and L, it is NP-complete to decide whether there exists a 0/1 reduction R with $\epsilon = 0$ such that $M(G_R) \leq M$ and $L(G_R) \leq L$.*

Proof: The problem is easily shown to be in NP. NP-completeness follows by a reduction from monotone 3-SAT [5]. Let $X = \{x_1, x_2, \ldots, x_n\}$ be n variables and $C = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be an instance of monotone 3-SAT. A clause containing only un-negated variables is called a *positive clause* and a clause containing only negated variables is called a *negative clause*. Let $C_i = u_i^1 \vee u_i^2 \vee u_i^3$, where u_i^j is referred to as a literal, $1 \leq j \leq 3$. We next describe how to



(a)

(b)

Figure 5: The clause graph G_i corresponding to clause $C_i = u_i^1 \vee u_i^2 \vee u_i^3$. (a) shows the clause graph corresponding to a positive clause and (b) shows the clause graph corresponding to a negative clause.

construct a weighted dag $G = (V, E)$ and determine M and L such that G has a 0/1 reduction R with $M(G_R) \leq M$ and $L(G_R) \leq L$ if and only if C is satisfiable.

Graph G contains k *clause graphs*, $G_1, G_2, \ldots, G_k$, which are connected by consistency edges. Clause graph G_i corresponds to clause C_i, and we distinguish between positive and negative clause graphs (depending on the type of the corresponding clause). Each clause graph is made up of 3 components and one attachment. Each component is an 8-vertex graph and the attachment is a 2-vertex graph. Positive and negative clause graphs are constructed somewhat differently. Figure 5(a) shows a positive and Figure 5(b) shows a negative clause graph. A clause graph contains multiple edges between some of its vertices. Multiple edges between the same pair of vertices have the same weight and thus only one weight is shown.

Let U_i^1, U_i^2, U_i^3, and A_i be the three components and the attachment of clause graph G_i, respectively. In each component U_i^j we name the following vertices and edges as shown in Figure 5: edges t_1 and t_2 are called the *true-edges*, edges f_1 and f_2 are called the *false-edges*, p_i^j is the source and q_i^j is the sink of component U_i^j, and c_1 and c_2 are the vertices incident to the consistency edges. The path from p_i^j to q_i^j containing edges t_1 and f_1 is called the *upper path*, and the one containing t_2 and f_2 is called the *lower path*. The three components and the attachment are connected by edges of weight 0 as shown in Figure 5. Positive and negative clause graphs differ in the way the upper and lower path in a component interact, in the position of edges t_1 and f_1 on the upper path, and in how the components and the attachment are connected.

As already stated, the k clause graphs are connected by consistency edges. *Consistency edges* are edges of multiplicity 2 and each such edge has a weight of 12. Let u_i^a and u_j^b, $i < j$, be two literals formed by the same variable, say x_l, and assume that x_l does not form a literal in clauses $C_{i+1}, \ldots, C_{j-1}$. Graph G contains a consistency edge from vertex c_1 in component U_i^a to vertex c_2 in component U_j^b, and one from vertex c_1 in component U_j^b to vertex c_2 in component U_i^a. To complete the construction of G, we add a source p and a sink q and edges of weight 0 from p to every p_i^j and from every q_i^j to q. Figure 6 shows the graph G created for the formula $C = \{(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_5} \vee \overline{x_6})\}$.

Clearly, given a monotone 3-SAT formula C, the corresponding graph G can be built in polynomial time. G has a total of $26k + 2$ vertices. The length of the longest path from source p to sink q is 40. G contains k such longest paths, one for every clause. For a positive clause graph G_i, this path contains vertices p and p_i^1, edge t_1 of component U_i^1, edge e_1 of G_i, edge t_1 of component U_i^2, edge e_2, edges t_1 and f_1 of component U_i^3, and vertex q_i^3. Figure 7(b) shows such a path. Finally, we set $M = 6k$ and $L = 30$. We claim that G has a 0/1 reduction in which at most $6k$ edges are reduced and the length of every path from p to q is at most 30 if and only if clause C is satisfiable.

Since there exist two edge-disjoint paths of length 32 (one is the upper path and the other is the lower path) in every one of the $3k$ components, reducing the path length to 30 without reducing more than $6k$ edges implies that we reduce exactly two edges per component. Fur-

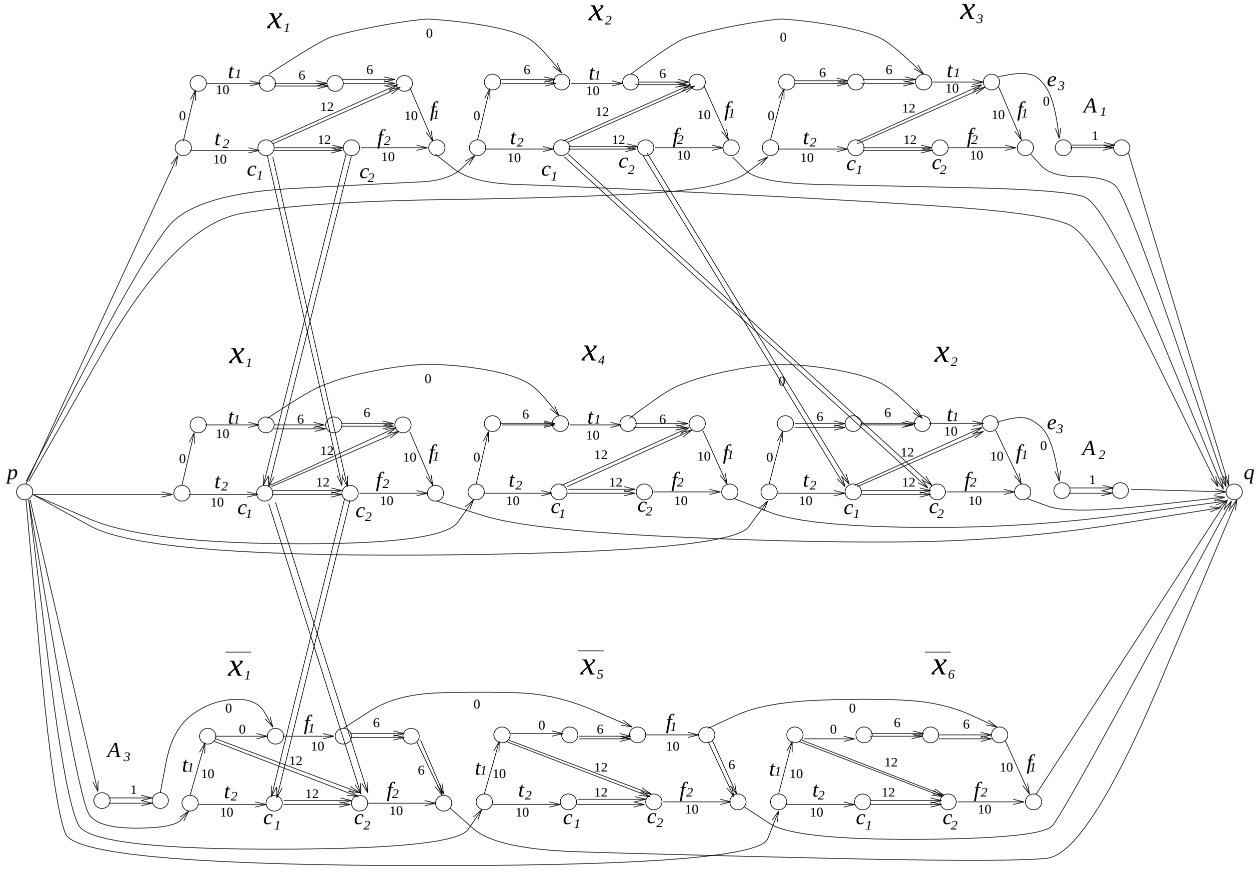


Figure 6: Graph G for formula $C = \{(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_5} \vee \overline{x_6})\}$.

thermore, no multiple edges can be reduced. Assume that $t : X \rightarrow \{T, F\}$ is a truth assignment satisfying C. We construct a 0/1 reduction R for G as follows. Let x_i be a variable with $t(x_i) = T$. Then, in every component U_j^b with $u_j^b = x_i$ or $u_j^b = \overline{x_i}$, edges t_1 and t_2 are reduced. On the other hand, if $t(x_i) = F$, then in every component U_j^b with $u_j^b = x_i$ or $u_j^b = \overline{x_i}$, edges f_1 and f_2 are reduced. We are reducing exactly two edges per component and thus reduce a total of $6k$ edges. It remains to be shown that the reduced graph G_R contains no path exceeding 30. Let P be any path from p to q. The structure of P is one of the following:

(i) Path P contains source p_i^j and sink q_i^j of some component U_i^j. Any such path has cost 32 in G. Either t_1 and t_2 or f_1 and f_2 are reduced. Hence, path P contains either one true or one false edge that is reduced, and the cost of P in G_R is 22.

(ii) Assume P contains vertices of a single clause graph G_i, with the vertices belonging to different components or the attachment. The majority of the cases described below make use of the fact that any upper path in a component has either its true- or its false-edge reduced. Assume G_i is a positive clause graph. The situation for a negative clause graphs is symmetrical and is omitted.

   (a) P goes through vertex p_i^1, edge t_1 of U_i^1, edge e_1, edges t_1 and f_1 of U_i^2 and vertex q_i^2, as shown in Figure 7(a). The length of P in G is 36 and it is at most 26 in G_R.

   (b) P goes through vertex p_i^1, edge t_1 of U_i^1, edge e_1, edge t_1 of U_i^2, edge e_2, edges t_1 and f_1 of U_i^3 and vertex q_i^3, as shown in Figure 7(b). The length of P in G is 40 and it is at most 30 in G_R.

   (c) P goes through vertex p_i^1, edge t_1 of U_i^1, edge e_1, edge t_1 of U_i^2, edge e_2, edge t_1 of U_i^3, edge e_3, and the attachment of clause graph G_i, as shown in Figure 7(c). The length of such a path in G is 31. Since at least one of the three literals of positive clause C_i is assigned "T", at least one of the three true-edges on the upper paths of the components of G_i is reduced. This implies that P is at most 21 in G_R.

   (d) P goes through vertex p_i^2, edge t_1 of U_i^2, edge e_2, edges t_1 and f_1 of U_i^3 and vertex q_i^3,as shown in Figure 7(d). The length of P in G is 36 and its length in G_R is at most 26.

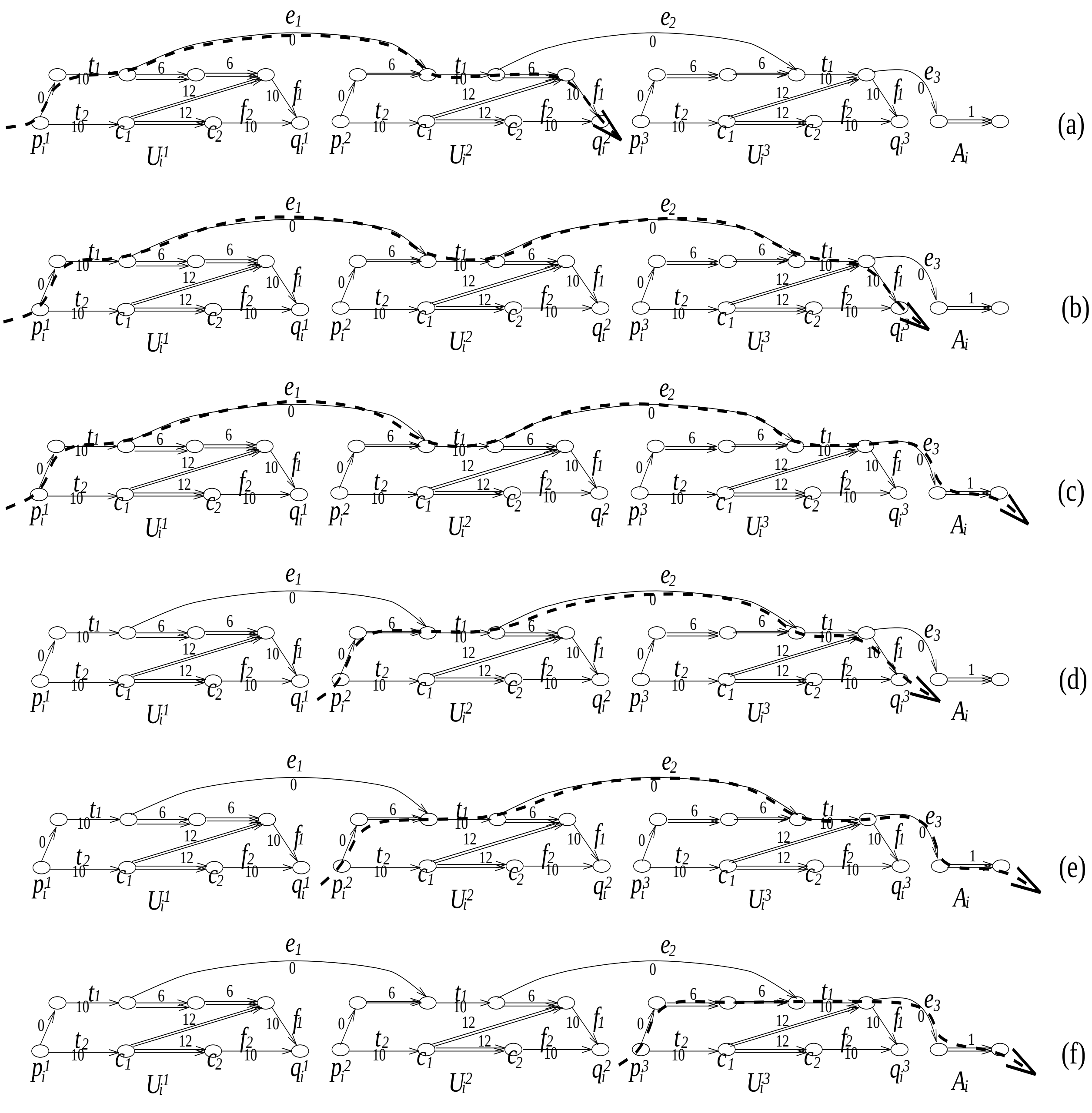


Figure 7: Paths in positive clause graph G_i going through different components and/or the attachment.

(e) P goes through vertex p_i^2, edge t_1 of U_i^2, edge e_2, edge t_1 of U_i^3, edge e_3, and the attachment of G_i, as shown in Figure 7(e). The length of P in G is 27 and does not need to be reduced.

(f) P goes through vertex p_i^3, edge t_1 of U_i^3, edge e_3, and the attachment of G_i, as shown in Figure 7(f). The length of P in G is 23 and does not need to get reduced.

(iii) Assume now that path P contains edges belonging to different clause graphs. Our construction of G allows such a path to contain edges of no more than two different clause graphs. Let P contain edges from components U_i^a and U_j^b, $i \neq j$. P either contains vertices c_1 of U_i^a and c_2 of U_j^b or vertices c_1 of U_j^b and c_2 of U_i^a. Any such path has length 32 and it contains a t_2 and an f_2 edge belonging to different components. Components U_i^a and U_j^b correspond to literals formed by the same variable. We thus have in both components either all true or all false edges reduced. This implies that any such path has a length of exactly 22 in G_R.

Hence, reducing $6k$ true- or false-edges according to the truth assignment satisfying C results in a reduced graph G_R containing no path exceeding 30. We now complete the proof by showing that if there exists a 0/1 reduction R with $M(G_R) \leq 6k$ and $L(G_R) \leq 30$, then C can be satisfied. We start by giving properties that any such reduction R must satisfy.

Property 5.1 *In a component U_i^a belonging to a positive clause graph the set of reduced edges is either $\{t_1, t_2\}$, or $\{f_1, t_2\}$, or $\{f_1, f_2\}$. In a component U_i^a belonging to a negative clause graph the set of reduced edges is either $\{t_1, t_2\}$, or $\{t_1, f_2\}$, or $\{f_1, f_2\}$.*

Proof: As already stated, in order to reduce the length of every path to 30 and reduce at most $6k$ edges, two edges per component need to get reduced. Clearly, reduction R may reduce both true-edges or both false-edges. For components belonging to a positive clause graph it is also possible that edges f_1 and t_2 are reduced. Observe that reducing edges f_2 and t_1 preserves a path length of 32 within this component. In a symmetrical way, for components belonging to a negative clause graph, it is possible that edges f_2 and t_1 are reduced. □

Property 5.2 *Let U_i^a and U_j^b be two components linked together by consistency edges. Then, either the t_2 edges of U_i^a and U_j^b are reduced or the f_2 edges of U_i^a and U_j^b are reduced.*

Proof: Assume the t_2 edge of component U_i^a is reduced, but the t_2 edge of component U_j^b is not. By Property 5.1, the f_2 edge of component U_i^a is not reduced. This would imply that G_R contains a path of length 32 containing edge t_2 and vertex c_1 of U_j^b as well as vertex c_2 and edge f_2 of U_i^a. The other situations result in similar contradictions. $\square$.

Property 5.3 *If G_i is a positive clause graph, at least one of the three t_1 edges in G_i is reduced. If G_i is a negative clause graph, at least one of the three f_1 edges in G_i is reduced.*

Proof: Let P be a path from source p to sink q going through clause graph G_i and containing edges e_1, e_2, e_3 of G_i. Such path has length 31 in G. Since the edges in the attachment cannot be reduced, at least one of the three edges having weight 10 is reduced in R. These three edges correspond to true-edges in a positive clause graph and correspond to false edges in a negative clause graph. $\square$.

Given a graph G and a reduction R, a truth assignment $t : X \rightarrow \{T, F\}$ satisfying C is constructed as follows. For every variable x_i, find a component U_j^b corresponding to a literal u_j^b formed by x_i. If the t_2 edge of component U_j^b is reduced, set $t(x_i) = T$. If the f_2 edge of U_j^b is reduced, set $t(x_i) = F$. Property 5.2 guarantees that any literal formed by x_i induces the same truth assignment. By Property 5.3, at least one literal is true in each clause, and thus $t : X \rightarrow \{T, F\}$ satisfies C. This concludes our NP-completeness proof. $\square$

The assumption $\epsilon = 0$ is not crucial to the argument used in the proof. For example, the following change in the edge weights of the multiple edges gives an NP-completeness proof for $\epsilon = \frac{1}{2}$. Multiple edges having a weight of 12 now have a weight of 16. The ones having a weight of 6 now have a weight of 8, and the edges in the attachment now have a weight of 6. The longest path length in G remains 40. An argument identical to the one already used shows that there exists a 0/1 reduction R with $M(G_R) \leq 6k$ and $L(G_R) \leq 35$ reducing at most $6k$ edges if and only if C can be satisfied.

6 Acknowledgements

We would like to thank the anonymous referees for their helpful and constructive comments which lead to an improvement in Section 3.

References

[1] A. Al-Khalili, Y. Zhu, and D. Al-Khalili. A module generator for optimized cmos buffers. In *Proceedings of 26th ACM/IEEE Design Automation Conference*, pages 245–250, 1989.

[2] H. Booth and R.T. Tarjan. Finding the minimum-cost maximum flow in a series-parallel network. *Journal of Algorithms*, 15:416–446, 1993.

[3] H.-C. Chen, D.H.-C. Du, and L.-R. Liu. Critical path selection for perfomance optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):185–195, 1993.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[6] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[7] D. Marple. Transistor size optimization in the tailor layout system. In *Proceedings of 26th ACM/IEEE Design Automation Conference*, pages 43–48, 1989.

[8] F. Obermeier and R. Katz. An electrical optimizer that considers physical layout. In *Proceedings of 25th ACM/IEEE Design Automation Conference*, pages 453–459, 1988.

[9] C.H. Papadimitriou and J.D. Ullman. A communication-time tradeoff. *SIAM Journal of Computing*, 16(4):639–646, August 1987.

[10] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recongnition of series parallel digraph. *SIAM J. Comput.*, 11(2):298–313, May 1982.