1994

Edge Weight Reduction Problems in Directed Acyclic Graphs

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Hung-Yi Tu

Report Number:

94-014

EDGE WEIGHT REDUCTION PROBLEMS IN DIRECTED, ACYCLIC GRAPHS

Susanne E. Hambrusch
Hung-Yi Tu

CSD TR 94-014
March 1994

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 Information Sciences
Providence University
Taichung, Taiwan, ROC

March 1, 1994

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 ARPA under contract DABT63-92-C-0022.

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 [3, 4]. In some applications the weight of an edge is not a static quantity, but can be reduced for a certain cost. The 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 different classes of graphs, including 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 such edge reductions arise. Linear reductions model, for example, physical performance optimizations of circuits through gate resizing and buffer insertions [1, 2, 6, 7]. 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 [5, 8]. 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 problem in in-trees. Section 3 presents $O(m^2)$ algorithms for the linear reduction problems in series-parallel graphs. We also show that linear edge reduction problems in general dags can be solved in polynomial time by formulating them as linear programs. 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 $O(m^2h)$ time, where h is the height of a bounded-degree decomposition tree of the series-parallel graph. For in-trees in which the degree of every node is bonded by a constant, the time bounds reduce to $O(nh)$, where h is the height of the in-tree. In Section 5 we show that the 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 is at most 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 $O(n^2)$ 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).

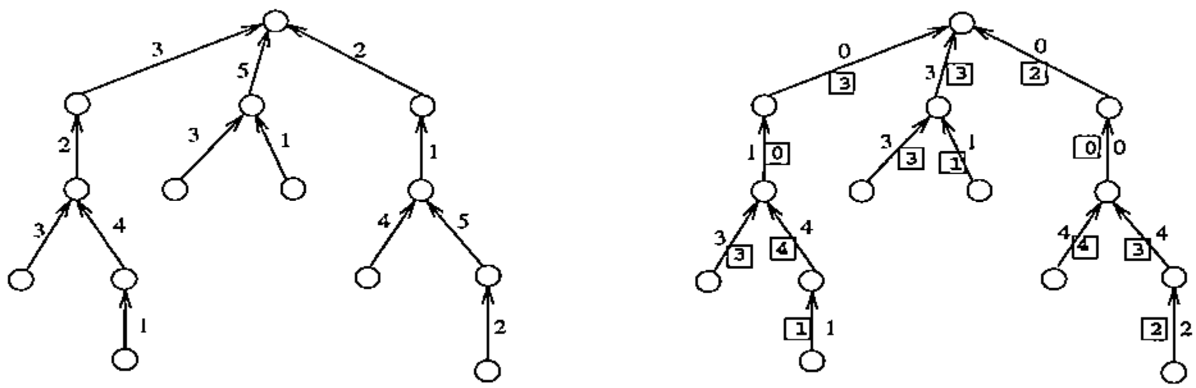
Let v_n be the root of the in-tree. For convenience, we add an artificial source v_0 and edges (v_0, v_i) with $d(v_0, v_i) = 0$ for every leaf v_i. Even though the resulting graph is no longer an in-tree, the structure crucial to the algorithm is preserved and we refer to it as an in-tree.

2.1 Finding an optimal reduction for a given L

In the (G, L)-problem we generate an optimal reduction R^* satisfying $L(G_{R^*}) \leq L$ and minimizing $M(G_{R^*})$. Reduction R^* generated by our algorithm satisfies a property, which we call the canonical property, and which is defined next. Let R be an optimal reduction. R is *canonical* if for any other optimal reduction R' the length of the path from v_i to root v_n in G_R is not longer than its length in $G_{R'}$. Stated in terms of reductions, in a canonical reduction the reductions occur as close to the root as possible. See Figure 1(b) for an example of a canonical reduction. Optimal canonical reductions for in-trees can also be characterized as stated in Lemma 2.1. We refer to an edge e with $r(e) = d(e)$ (resp. $r(e) = 0$) as an edge with *full* (resp. *zero*) reduction. An edge e with $0 < r(e) < d(e)$ is called an edge with *partial* reduction.

Lemma 2.1 *Let R be an optimal reduction and let P be any path from v_0 to v_n in G_R. Then, R is canonical if and only if path P contains an edge (v_i, v_j) with $0 \leq r(v_i, v_j) \leq d(v_i, v_j)$ such that every edge on the path from v_j to v_n has full reduction and every edge on the path from v_0 to v_i has zero reduction.*

Proof: The lemma implies that path P contains at most one edge with partial reduction. Assume every path P in G_R can be characterized as stated. Since edges closest to the root are reduced first, there cannot exist another reduction R' (with the same L- and M-values) such



(a)

A tree with a longest path length of 10

(b)

Edge weights of two optimal reductions for L=6 achieving M=10
Edge weights in boxes correspond to an optimal reduction that is not canonical

Figure 1: A tree with two optimal reductions for $L = 6$

that the length of the path from v_i to v_n in $G_{R'}$ is smaller than its length in G_R. Hence, R is canonical.

Assume now that R is a canonical reduction and G_R contains a path P not satisfying the characterization. Let (v_i, v_j) and (v_a, v_b) be two distinct edges on P such that every edge on P from v_b to v_n has full reduction, edge (v_a, v_b) has either partial or zero reduction, every edge on P from v_0 to v_i has zero reduction, and edge (v_i, v_j) has either partial or full reduction. This implies that P contains the vertices $v_0, \ldots, v_i, v_j, \ldots, v_a, v_b, \ldots, v_n$, with possibly $v_j = v_a$. Let R' be the reduction obtained from R by setting:

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

Clearly, $M(G_R) = M(G_{R'})$. The length of path P in R' is as in R. Furthermore, every path from v_0 to v_n via edge (v_i, v_j) goes through edge (v_a, v_b), and thus the length of any other path from v_0 to v_n could only have decreased. Hence, $L(G_R) \le L(G_{R'})$ and $M(G_R) = M(G_{R'})$. Let v_k be a vertex on path P between (and including) v_j and v_b. The length of the path from v_k to v_n is smaller in $G_{R'}$ than in G_R. This implies that R is not a canonical reduction and the lemma follows. $\square$

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

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

The $O(n)$ running time of the algorithm follows trivially. Clearly, $L(G_{R^*}) = L$. Optimality of R^* is established in the following lemma.

Lemma 2.2 *Let R^* be the reduction generated by our algorithm. Then, R^* is an optimal canonical reduction.*

Proof: From the way R^* is constructed it is clear that it satisfies the canonical property. More precisely, consider next any path from v_0 to v_n in G_{R^*}. This path contains at most one edge, say (v_a, v_b), with partial reduction. All edges from v_0 to v_a have zero reduction and all edges from v_b to v_n have full reduction.

Assume that R^* is not an optimal reduction. Then there exists another optimal canonical reduction R' with $M(G_{R^*}) > M(G_{R'})$ and $L(G_{R'}) \leq L$. This implies that in graph G_{R^*} there exists a path from v_0 to v_n containing an edge e with $r^*(e) > r'(e)$. Let $e = (v_i, v_j)$ be such an edge. Consider first the case when edge (v_i, v_j) has full reduction in R^*. Observe that this implies $L \leq L(v_i)$. R' is a canonical reduction and edge (v_i, v_j) has either partial or zero reduction in R'. Hence, any edge on a path from v_0 to v_i has zero reduction in R'. The length of the longest path from v_0 to v_j in $G_{R'}$ is $L(v_i)+d_{r'}(v_i, v_j) > L(v_i)+d_{r^*}(v_i, v_j) \geq L$, contradicting our assumption of $L(G_{R'}) \leq L$.

Assume now that edge (v_i, v_j) has partial reduction in R^*. This implies $L(v_i) < L < L(v_i) + d(v_i, v_j)$. From the way R^* is constructed, it follows that any edge on a path from v_0 to v_i in G_{R^*} has zero reduction. The length of the longest path from v_0 to v_j in G_{R^*} is equal to $L(v_i) + d_{r^*}(v_i, v_j) = L$. Since $r'(v_i, v_j) < r^*(v_i, v_j) < d(v_i, v_j)$ and R' is a canonical reduction, any edge on a path from v_0 to v_i in $G_{R'}$ has zero reduction. Thus, the longest path length from v_0 to v_j in $G_{R'}$ is equal to $L(v_i) + d_{r'}(v_i, v_j) > L(v_i) + d_{r^*}(v_i, v_j) = L$, giving a contradiction. 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 are to 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 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 $M(G_{R^*}) \leq M$. $M(G_R)$ is a piecewise-linear, non-increasing function of $L(G_R)$. This 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 $L(v_i)$ be again the length of the longest path from v_0 to v_i in G. Edge (v_i, v_j) creates the entry $L(v_i) + d(v_i, v_j)$ and let $\mathcal{L} =< L_1, L_2, \ldots, L_n >$ be the list containing these entries in non-decreasing order. Since $L_{i-1} \leq L_i$, we have $M(G_{R_{i-1}}) \geq M(G_{R_i})$. Assume invoking algorithm OPT_L(G, L_i) generates reduction R_i. Let k be the index such that

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

By using algorithm OPT_L, index k can determined in $O(n \log n)$ time. If $M(G_{R_{k-1}}) = M$, then $R^* = R_{k-1}$ (and $L^* = L_{k-1}$). Assume thus that $M(G_{R_{k-1}}) > M > M(G_{R_k})$. We next describe how to generate reduction R^* from the canonical reductions R_{k-1} and R_k. Clearly, an edge having full reduction in R_k has full reduction in R_{k-1}. Such an edge will receive full reduction in R^*. An edge having zero reduction in R_{k-1} has zero reduction in R_k. It will receive zero reduction in R^*. Consider now all the edges of G whose reduction in R_{k-1} is larger than in R_k (no edge can have a smaller reduction in R_k). Let E_p be the set containing these edges. Let $L_{k-1} + \delta = L_k$, $\delta > 0$. The following characterization of the edges in E_p is used in determining their reduction in R^*.

Lemma 2.3 *Let P be a path from v_0 to v_n in G. Then, P contains at most one edge belonging to set E_p. For any edge e in E_p, we have $r_{k-1}(e) - r_k(e) = \delta$.*

Proof: Assume there exists a path P containing two or more edges in set E_p. Let (v, w) be the edge on P in set E_p closest to root v_n. In R_k, edge (v, w) has either partial or zero reduction.

We only give the argument for the case when (v, w) has partial reduction (zero reduction is handled in a similar way). Since R_k and R_{k-1} are canonical reductions, the following holds. Edge (v, w) has full reduction in R_{k-1} (if it had partial reduction, path P could not contain two edges belonging to E_p). In addition, edge (u, v) on path P has zero reduction in R_k and either full or partial reduction in R_{k-1}. This also implies that the two edges of P belonging to E_p are adjacent. Hence, we have

$$L_{k-1} < L(u) + d(u, v) < L_k.$$

The left side of the inequality holds since edge (u, v) is reduced in R_{k-1}. The right side holds since edge (v, w) is partially reduced in R_k (the relation $L(u) + d(u, v) \leq L_k$ would allow full reduction on edge (v, w) in R_k). The above inequality implies that entry $L(u) + d(u, v)$ is not in list $\mathcal{L}$. Hence, path P cannot contain two edges belonging to set E_p.

Let edge $e = (v, w)$ be an edge in E_p. Assume e has partial reduction in both R_k and R_{k-1}. The other three cases of possible reductions on edge e in R_k and R_{k-1} are handled in a similar manner. From algorithm OPT_L is follows that $r_k(v, w) = L(v) + d(v, w) - L_k$ and $r_{k-1}(v, w) = L(v) + d(v, w) - L_{k-1} = L(v) + d(v, w) - L_k + \delta$. Hence, $r_{k-1}(e) - r_k(e) = \delta$ follows. □

We can now state how R^* is generated from reductions 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 amount is evenly distributed among the edges in E_p. Lemma 2.3 implies $M(G_{k-1}) - M(G_{R_k}) = \delta|E_p|$. Since $M(G_{k-1}) \geq M > M(G_k)$, we have $\frac{M - M(G_{R_k})}{|E_p|} \leq \delta$ and thus $r^*(v_i, v_j) \leq d(v_i, v_j)$. From (1), (2), and (3) is follows that R^* has the canonical property. Given index k, the optimal reduction R^* can be generated in $O(n)$ time and thus the $O(n \log n)$ overall time bound 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 it reduces the amount of relevant data by a constant fraction after each search. Let $\mathcal{L}$ now be the unsorted list containing the entries $L(v_i) + d(v_i, v_j)$. 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 $L_a = 0$ and $L_b = L_n$. The edges of G are partitioned into 4 sets, E_z, E_u, E_p, and E_f. Set E_z contains the edges having zero reduction in both R_a and R_b. Set E_f contains the edges having full reduction in both R_a and R_b. As already argued in the $O(n \log n)$ algorithm, edges in E_z receive zero reduction in R^* and edges in E_f receive full reduction in R^*. Set E_p contains edges for which it has already been determined that they receive partial reduction in R^*. An edge $e = (u, v)$ that has partial reduction in both R_a and R_b clearly belongs to E_p. In addition, (u, v) belongs to E_p if it has full reduction in R_a, partial reduction in R_b, and every edge (x, u) has zero reduction in R_a. The amount of reduction on edge $e = (u, v)$ in R^* is not yet known. However, for any L_k with $L_a + \delta = L_k < L_b$, we have $r_k(e) = r_a(e) - \delta$. Set E_u contains all remaining edges. Their type and amount of reduction remains to be decided.

Let M_f be the total reduction on the edges in set E_f; i.e., $M_f = \sum_{(u,v) \in E_f} d(u, v)$. Let $M_{p,a}$ be the total reduction made on the edges in set E_p in reduction R_a. Let $\mathcal{L}_{a,b}$ be the sublist of $\mathcal{L}$ containing the entries L_j with $L_a < L_j < L_b$, with n_{ab} being the number of elements in list $\mathcal{L}_{a,b}$. If $n_{ab} \leq 4$, we check each one of the entries in $\mathcal{L}_{a,b}$ as to whether it is L_{k-1}. Otherwise, we determine the $\frac{n_{ab}}{4}$–th smallest element in list $\mathcal{L}_{a,b}$. Let L_q be this element and let $L_a + \delta = L_q$. The reduction on the edges in E_u in R_q is determined by using the method described in Section 2.1. Doing so partitions E_u into 3 sets, $E_{u,z}$, $E_{u,p}$, and $E_{u,f}$, depending on whether an edge of E_u receives zero, partial, or full reduction in R_q, respectively. The selection of L_q implies that $|E_{u,z}| = n_{ab}/4$ and $|E_{u,p}| + |E_{u,f}| = 3n_{ab}/4$. We then have

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

with $M_{u,p} = \sum_{(u,v) \in E_{u,p}} r_q(u, v)$ and $M_{u,f} = \sum_{(u,v) \in E_{u,f}} d_q(u, v)$.

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

- The edges in $E_{u,z}$ are added to E_z and are deleted from E_u.

- 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 (i.e., the new $M_{p,a}$) is computed.

Assume now that $M(G_{R_q}) < M$. In this case we have found a new upper bound and continue with L_a and L_q (after updating the edge sets). The updating involves:

- The edges in $E_{u,f}$ are added to E_f and are deleted from E_u. M_f is updated.
- 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.

Clearly, the work done in an iteration is $O(|E_u|)$. In order to establish the $O(n)$ overall time, we show that the size of set E_u reduces by a constant fraction each iteration. If $M(G_{R_q}) > M$, the edges belonging to $E_{u,z}$ are deleted from E_u. Since $|E_{u,z}| = n_{ab}/4$, the size of E_u is at most $3n_{ab}/4$ in the next iteration.

Consider now the case when $M(G_{R_q}) < M$. The deletion of the edges belonging to set $E_{u,f}$ from E_u does not imply that the size of set E_u decreases by a constant fraction. Indeed, $E_{u,f}$ could be the empty set. However, the following argument shows that the size of E_u is reduced by at least one half. Let (u,v) be an edge that has full reduction in R_a, partial reduction in R_q, and which does not qualify for set E_p. It does not qualify since there exists at least one edge (x,u) that has full or partial reduction in R_a and zero reduction in R_q. Edge (x,u) belongs to set $E_{u,z}$ and (u,v) belongs to set $E_{u,p}$. Any such edge (u,v) can thus be assigned a unique edge (x,u) belonging to $E_{u,z}$. Both (u,v) and (x,u) remain in set E_u. Since $|E_{u,z}| = n_{ab}/4$, there can be at most $n_{ab}/4$ edges like (u,v). This implies that at least $n_{ab}/2$ edges of set E_u either belong to set $E_{u,f}$ or qualify for inclusion into set E_p (after being in set $E_{u,p}$). Thus, the size of E_u is reduced by at least one half.

The $O(n)$ time bound for the (G,M)-problem now follows easily. We first determine index k such that $M(G_{R_{k-1}}) \geq M > M(G_{R_k})$ using the algorithm described above. We then generate reduction R^* from R_k and R_{k-1} in $O(n)$ time as described earlier.

2.3 Optimal reduction for the tradeoff problem

The approach used for the (G, M)-problem in the last section can also be used to solve the tradeoff problem in in-trees in $O(n)$ time. Recall that in the tradeoff problem we are to determine a reduction R^* minimizing $f(G_R) = L(G_R) + \gamma \cdot M(G_R)$. Let $\mathcal{M}(L)$ represent the minimum total reduction needed to reduce the longest path length to L. It can be shown that $\mathcal{M}(L)$ is piecewise-linear, non-increasing and concave-up. We can thus represent $\mathcal{M}(L)$ by a sequence of linear functions, $a_1 \times L + b_1, a_2 \times L + b_2, \ldots, a_{n-1} \times L + b_{n-1}$, with all a_j's being negative. Function $a_i \times L + b_i$ is associated with interval, $[L_i, L_{i+1}]$, $1 \leq i \leq n-1$, where the L_i-values are as defined in the previous section. In interval $[L_i, L_{i+1}]$, $\mathcal{M}(L)$ is described by $a_i \times L + b_i$. Since $\mathcal{M}(L)$ is concave-up, we have $a_1 \leq a_2 \leq \cdots \leq a_{n-1} < 0$.

Function $f(G_R)$ can be re-written as a function of the longest path length L; i.e., $\mathcal{F}(L) = L + \gamma \cdot \mathcal{M}(L)$. Minimizing $f(G_R)$ is equivalent to minimizing $\mathcal{F}(L)$. We distinguish between the following four cases.

Case 1. $1 + \gamma \cdot a_{n-1} < 0$.
In this case the minimum of $\mathcal{M}(L)$ occurs at $L = L_n$.

Case 2. $1 + \gamma \cdot a_1 > 0$.
In this case the minimum of $\mathcal{M}(L)$ 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 $\mathcal{M}(L)$ 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 $\mathcal{M}(L)$ occurs at $L = L_{j+1}$.

The heart of the algorithm is the search for index j in Cases 3 and 4. Index j can be determined in $O(n)$ time by using an approach similar to the one used for the (G, M)-problem described in the previous section. We determine j by using a binary search combined with prune and search. 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. We omit the details of the $O(n)$ time search algorithm.

3 Linear reduction for series-parallel graphs

In this section we describe $O(m^2)$ time algorithms for the linear edge reduction problems in series-parallel graphs. The graphs can have multiple edges between two vertices and thus $m = \Omega(n^2)$ is possible. Our algorithms use a greedy strategy and they decide which edges to reduce by using information generated by a minimum cut and longest path computations. We start by giving the necessary definitions regarding series-parallel graphs. In Section 3.1 we then describe our algorithms. Section 3.2 addresses their correctness.

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 a series-parallel graph.

2. Given a set of series-parallel graph $G_1, G_2, \ldots, G_k$, the dag G obtained by identifying the k sources with each other and by identifying the k sinks with each other is a series-parallel graph. This type of operation is called a *parallel composition.*

3. Given a set of series-parallel graph $G_1, G_2, \ldots, G_k$, the dag G obtained by identifying the source of G_i with the sink of G_{i+1}, for $1 \leq i \leq k-1$, is a series-parallel 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. If N is a leaf, G_N corresponds to the edge of G represented by the leaf. 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. In case of a parallel composition, we refer to internal node N as p-node, otherwise as an s-node. W.l.o.g., we make the following assumptions about decomposition tree D (this simplifies our correctness argument). We assume that a node and a child of this node are of different type (i.e., they cannot be both s- or p-nodes). An internal node has degree 1 iff its only child is a leaf. Both of these assumption result in a decomposition tree having a minimum number of nodes. Finally, we assume that the children of an s-node are ordered such that if child N_1 is immediately to the left of child N_2, then the sink of G_{N_1} is identified with the source of G_{N_2}.

Testing whether a given dag G on n vertices and m edges is a series-parallel graph can be done in $O(m)$ time [9]. Furthermore, the decomposition tree D for a given sp-graph G can be constructed in $O(m)$ time by using the recognition algorithm in [9].

3.1 The algorithms for sp-graphs

We first describe an $O(m^2)$ time algorithm for the (G, L)-problem. Minor modifications in the termination of the algorithm solve the (G, M)-problem and the tradeoff problem, respectively.

Our algorithm for solving the (G, L)-problem generates an optimal reduction R^* minimizing $M(G_{R^*})$ and satisfying $L(G_{R^*}) \leq L$ over a number of iterations, with each iteration generating a new reduction. Let R_{i+1} be the reduction generated by the i-th iteration. The length of the longest paths decreases during the iterations, while the total reduction increases; i.e., $L(G_{R_i}) > L(G_{R_{i+1}})$ and $M(G_{R_i}) < M(G_{R_{i+1}})$.

Let R_i be the reduction available at the beginning of the i-th iteration, where R_1 corresponds to a reduction of 0 on every edge of G. We use $r_i(e)$ to denote the reduction done on edge e in R_i. The reduced weight $d_i(e)$ is defined by $d(e) - r_i(e)$. Clearly, in order to reduce the length of the longest paths in G_{R_i}, edges on the longest paths from v_0 to v_n have to receive a reduction. Let S_i be the subgraph of G_{R_i} containing only edges on a longest path from v_0 to v_n. Let a minimum cut in an sp-graph be a cut containing a minimum number of edges without containing an edge of weight 0. Let $C(S_i)$ be the set of edges corresponding to such a minimum cut in graph S_i. To generate R_{i+1}, we increase the reductions for the edges in cut $C(S_i)$ by the same amount. The amount is determined by two conditions:

(i) the weight of an edge in $C(S_i)$ cannot be negative, and

(ii) reduction on the edges in $C(S_i)$ has to stop as soon as a path *not* in S_i becomes a longest path.

More precisely, let $\gamma_i = \min_{e \in C(S_i)}\{d_i(e)\}$. Assume temporarily that in R_{i+1} every edge in $C(S_i)$ receives an additional reduction of γ_i. This temporary reduction may reduce the edges on the cut by too much (i.e., it can violate condition (ii)). We use this R_{i+1} to compute $L(G_{R_{i+1}})$, the length of the longest paths in $G_{R_{i+1}}$. If $L(G_{R_{i+1}}) > L$, the i-th iteration is not the last one and we set (and possibly decrease) $r_{i+1}(e) = r_i(e) + (L(G_{R_i}) - L(G_{R_{i+1}}))$ for every edge $e \in C(S_i)$ and $r_{i+1}(e) = r_i(e)$ otherwise. If $L(G_{R_{i+1}}) \leq L$, we set $r_{i+1}(e) = r_i(e) + (L(G_{R_i}) - L)$ for every edge $e \in C(S_i)$ and $r_{i+1}(e) = r_i(e)$ otherwise, and the algorithm terminates.

To complete the description of one iteration, we describe how to generate a minimum cut in a series-parallel graph G in $O(m)$ time, where m is the number of edges in G. Let D be the decomposition tree of G. For every node N of D, we compute a set $cut(N)$ which contains the edges in a minimum cut in G_N (recall that G_N is the subgraph of G corresponding to node N of D). Assume N is a leaf of D and e is the edge of G corresponding to this leaf. If the weight of e is 0, $cut(N)$ corresponds to a set having cardinality $+\infty$. Otherwise, we set $cut(N) = \{e\}$. When N is an internal node, we set

$$cut(N) = \begin{cases} cut(N_c) \text{ such that } |cut(N_c)| \leq |cut(N_d)|, & \\ \text{where } N_c \text{ and } N_d \text{ are children of } N & \text{if } N \text{ is an } s\text{-node} \\ \bigcup_{N_c \text{ is a child of } N} cut(N_c) & \text{if } N \text{ is a } p\text{-node.} \end{cases}$$

Clearly, by traversing tree D from the leaves towards the root, a minimum cut can be determined in $O(m)$ time.

From the above discussion it follows that the time of one iteration is bounded by $O(m)$. To bound the number of iterations, we first show that if an edge is included in S_i, then it is also in S_{i+1}. Let P be a longest path in S_i. Since S_i is a series-parallel graph, P contains exactly one edge belonging to cut $C(S_i)$. So, if we increase the reduction for each edge on $C(S_i)$ by $L(G_{R_i}) - L(G_{R_{i+1}})$, the length of path P decreases by $L(G_{R_i}) - L(G_{R_{i+1}})$. Thus, P is still a longest path in $G_{R_{i+1}}$ and every edge on P is in S_{i+1}. After the i-th iteration either one edge in S_i receives a weight of 0 or graph S_{i+1} contains at least one edge not in S_i. Graph G contains a total of m edges and thus the algorithm terminates after at most $2m$ iterations. The $O(m^2)$ overall time follows.

We next describe the changes to be made to the above algorithm in order to solve the (G, M)-problem and the tradeoff problem, respectively. Consider first the (G, M)-problem. Assume we have determined the minimum cut $C(S_i)$ and the temporary value of reduction R_{i+1}. If $M(G_{R_{i+1}}) < M$, the i-th iteration is not the last one and we set R_{i+1} as done in the above algorithm. If $M(G_{R_{i+1}}) \geq M$, we set $r_{i+1}(e) = r_i(e) + \frac{M - M(G_{R_i})}{|C(S_i)|}$ for every edge $e \in C(S_i)$ and $r_{i+1}(e) = r_i(e)$ otherwise and terminate the algorithm. Consider now the tradeoff problem with $f(G_R) = L(G_R) + \gamma \times M(G_R)$ as the tradeoff function. An increase in the total reduction results in a decrease in the longest path length. We now terminate the algorithm when $f(G_{R_i}) \le f(G_{R_{i+1}})$ and output reduction R_i as the reduction minimizing the tradeoff function.

Before addressing the correctness of the above algorithms, we point out that, not surprisingly, the approach of repeatedly finding a minimum cut fails for general dags. However, the linear reduction problems can be solved in polynomial time for general dags. All three versions can be phrased as linear programs. For the (G, L)-problem the formulation is 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,

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

3.2 Correctness of the series-parallel algorithm

In this section we show the correctness of the algorithm for the (G, L)-problem. The correctness arguments for the other two algorithms are almost identical. Assume now that the algorithm terminates after k iterations generating reduction R_{k+1}. We prove that R_{k+1} is an optimal reduction by showing that there exists an optimal solution R^* with $r_{k+1}(e) = r^*(e)$ for every edge e. In order for this to be true we need that at the beginning of the i-th iteration we have $r_i(e) \le r^*(e)$, $1 \le i \le k$, which is satisfied for the first iteration. For notation, if $r_i(e) \le r^*(e)$ for every edge e, we say $R_i \le R^*$. If this does not hold for one edge, we say $R_i \not\le R^*$.

Assume $R_i \le R^*$ holds at the beginning of the i-th iteration and that after the i-th iteration we have $R_{i+1} \not\le R^*$. We call an edge e with $r_{i+1}(e) < r^*(e)$ a *surplus edge*. We next describe how to generate, from R^*, another optimal reduction R' which satisfies $R_{i+1} \le R'$. Reduction R' is obtained from R^* by reduction re-allocations; i.e., by moving reduction from surplus edges to the edges in $C(S_i)$ which violate $R_{i+1} \le R^*$. Recall that in the i-th iteration only the edges in cut $C(S_i)$ encounter a change in their reduction. The optimal reduction R' with $R_{i+1} \le R'$ is possibly obtained over a number of reduction re-allocations, with each re-allocation generating another optimal reduction. The re-allocations are guided by a labeling process in the decomposition tree of graph S_i. We next describe this labeling process.

Let D_i be the decomposition tree of the series-parallel graph S_i. Initially, only the leaves of

D_i corresponding to an edge in cut $C(S_i)$ are labeled. The leaf corresponding to an edge e in the cut is labeled "g" if $r_{i+1}(e) \leq r^*(e)$, and it is labeled "b" if $r_{i+1}(e) > r^*(e)$. Only nodes on the path from a labeled leaf to the root of D_i are labeled, with each such node labeled either "g" or "b". A node can only be labeled once all its children that can receive a label have been labeled. Labeling a node N of D_i "g" means that every edge e of subgraph G_N belonging to cut $C(S_i)$ satisfies $r_{i+1}(e) \leq r^*(e)$. Labeling a node N of D_i "b" means that

(i) there exists an edge e of G_N belonging to cut $C(S_i)$ which satisfies $r_{i+1}(e) > r^*(e)$, and

(ii) there exists no other cut in G_N for which reductions done in R^* can be re-allocated to edges in G_N belonging to cut $C(S_i)$.

The labeling process stops when the root of D_i is labeled. We will show that the root is always labeled "g".

We next give the rules for labeling s- and p-nodes and describe how re-allocations are done. Let R^* be the current optimal reduction and let N be an unlabeled node in D_i. Assume first that N is a p-node. In this case no reduction re-allocations take place and the labeling is done as follows. If all children of p-node N are labeled "g", we label N "g". If all children of N are labeled "b", we label N "b". We will show in Lemma 3.2 that it is not possible for a p-node to have one child labeled "b" and another labeled "g".

Assume now that N is an s-node. Since $C(S_i)$ is a minimum cut, at most one child of N can be labeled. Assume that N has a labeled child. If this child is labeled "g", we label N "g". Assume now that N has a child, say node N_b, labeled "b". Before N is labeled, we check whether reduction re-allocations can be done. Re-allocations could result in changing all "b" labels of nodes in the subtree rooted at N_b to "g" labels. A cut C is a *surplus cut* if C is a cut containing only surplus edges. If G_N contains a surplus cut, we perform a re-allocation of reductions. Let N_u be a child of N so that G_{N_u} contains a surplus cut C. We choose N_u such that no sibling between N_u and N_b is associated with a subgraph containing a surplus cut. W.l.o.g. assume that N_u is to the left of N_b. Let c_b be the number of edges in G_{N_b} belonging to cut $C(S_i)$ and let $|C| = c_u$. Since the algorithm finds minimum cuts in S_i and R^* is an optimal reduction, it follows that $c_b = c_u$. Let e be *any* edge belonging to $C(S_i)$ and G_{N_b}. Define

$$\delta = \min\{\min_{\hat{e} \in C}\{r^*(\hat{e}) - r_{i+1}(\hat{e})\}, r_{i+1}(e) - r^*(e)\}.$$

In Lemma 3.1 we show that $r_{i+1}(e) - r^*(e) = r_{i+1}(e') - r^*(e')$ for any two edges e and e' belonging to cut $C(S_i)$ and to subgraph G_{N_b}. This property allows us to choose any edge e when determining δ. Since $r_{i+1}(e) - r^*(e) > 0$ for any edge in $C(S_i)$ and G_{N_b} and C is a surplus cut, we have $\delta > 0$. Let R' be the reduction obtained as follows:

$$r'(e) \leftarrow \begin{cases} r^*(e) + \delta & \text{if } e \text{ is in } C(S_i) \text{ and in } G_{N_b} \\ r^*(e) - \delta & \text{if } e \text{ is in } C \\ r^*(e) & \text{otherwise} \end{cases}$$

In Lemma 3.3 we will show that R' is another optimal reduction. In our labeling process we now have a new optimal solution and set $R^* = R'$. If in the new R^* we have $r_{i+1}(e) \leq r^*(e)$ for every edge in subgraph G_N, every labeled node in the subtree rooted at node N_b and node N are labeled "g". Otherwise, we continue looking for surplus cuts to perform re-allocations. If no further surplus cut exists in G_N, we label node N "b". This completes the description of how an s-node is handled. We next prove crucial properties regarding the labels and establish the optimality of reduction R'.

Lemma 3.1 *Let N be an internal node of D_i labeled "b". Then, any two labeled leaf nodes belonging to the subtree rooted at N and corresponding to edges e and e', are labeled "b" and $r_{i+1}(e) - r^*(e) = r_{i+1}(e') - r^*(e')$.*

Proof: From the rules given for labeling internal nodes it follows that, if N is labeled "b", no node in the subtree rooted at node N is labeled "g". Hence, all leaves in this subtree are labeled "b". We next show that the edges corresponding to these leaves require the same amount of reduction to be re-allocated to them. Assume $e = (a, b)$ and $e' = (a', b')$ are two such edges with $r_{i+1}(e) - r^*(e) \neq r_{i+1}(e') - r^*(e')$. Let $req(e) = r_{i+1}(e) - r^*(e)$ for any edge e. W.l.o.g. assume $req(e) > req(e')$. Let node N' be the lowest common ancestor of the nodes corresponding to e and e' in decomposition tree D_i. Observe that N' is a p-node labeled "b". Let x be the source and y be the sink of sp-graph $G_{N'}$. We use $\|u \rightsquigarrow v\|_G$ to denote the length of a longest path from a vertex u to vertex v in graph G. The length of the longest path from u to v in G going through subgraph H is denoted by $\|u \overset{H}{\rightsquigarrow} v\|_G$. From the construction of S_i and S_{i+1} in the algorithm we have

$$\|x \overset{e}{\rightsquigarrow} y\|_{S_i} = \|x \overset{e'}{\rightsquigarrow} y\|_{S_i} \text{ and } \|x \overset{e}{\rightsquigarrow} y\|_{S_{i+1}} = \|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}}.$$

Since there exist no surplus cuts in $G_{N'}$ that could decrease $req(e)$ we have

$$\|x \rightsquigarrow a\|_{G_{R^*}} = \|x \rightsquigarrow a\|_{S_{i+1}} = \|x \rightsquigarrow a\|_{S_i}.$$

Similar equalities hold for the length of the longest paths from x to a', from b to y, and from b' to y. Observe that $d^*(e) = d_{i+1}(e) + req(e)$ and $d^*(e') = d_{i+1}(e') + req(e')$. We thus have

$$\|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}} < \|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}} + req(e') = \|x \overset{e'}{\rightsquigarrow} y\|_{G_{R^*}} < \|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}} + req(e) = \|x \overset{e}{\rightsquigarrow} y\|_{G_{R^*}}.$$

Let $\hat{R}$ be the reduction obtained from R^* by decreasing the reduction on edge e' by $req(e) - req(e')$. This corresponds to increasing the weight on edge e' to $\hat{d}(e') = d_{i+1}(e') + req(e)$. The total reduction in $\hat{R}$ is smaller than the one in R^*. The length of the longest path from x to y via edge e' increases in $\hat{R}$ by $req(e) - req(e')$, resulting in a total length of $\|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}} + req(e)$. Since this equals the length of the longest path from x to y via edge e, the longest path length is not increased. Hence, R^* is not an optimal reduction and we have $req(e) = req(e')$. □

Lemma 3.2 *No p-node N of D_i has children with different labels.*

Proof: Assume N has a child N_g labeled "g" and another child N_b labeled "b". Then, for every edge e in G_{N_b} and in cut $C(S_i)$ we have $r_{i+1}(e) - r^*(e) > 0$. For every edge e' in G_{N_g} and in cut $C(S_i)$ we have $r_{i+1}(e') - r^*(e') \leq 0$. Observe that the length of any path in G_{R^*} does not exceed the length of the same path in S_i. In addition, we have $\|x \overset{e'}{\rightsquigarrow} y\|_{G_{R^*}} \leq \|x \overset{e'}{\rightsquigarrow} y\|_{S_{i+1}}$. Consider the reduction $\hat{R}$ obtained from R^* by decreasing the reduction on every edge e' by $r_{i+1}(e) - r^*(e)$. Using an argument similar to the one used in Lemma 3.1 one can show that the existence of $\hat{R}$ contradicts the optimality of R^*. □

Lemma 3.3 *Reduction R' is an optimal reduction.*

Proof: Recall that R' is obtained from optimal reduction R^* by decreasing the reduction done on surplus cut C' in graph G_{N_u} by δ and by increasing the reduction on the edges in cut $C(S_i)$ belonging to G_{N_b} by δ. We already argued that the total reduction of R' and R^* is the same. We next show that R' does not contain a path whose length exceeds L.

Let x and y be source and sink in subgraph G_N, let x_b and y_b (resp. x_u and y_u) be source and sink of G_{N_b} (resp. G_{N_u}). Recall that G_{N_u} was chosen so that there exists no surplus cut

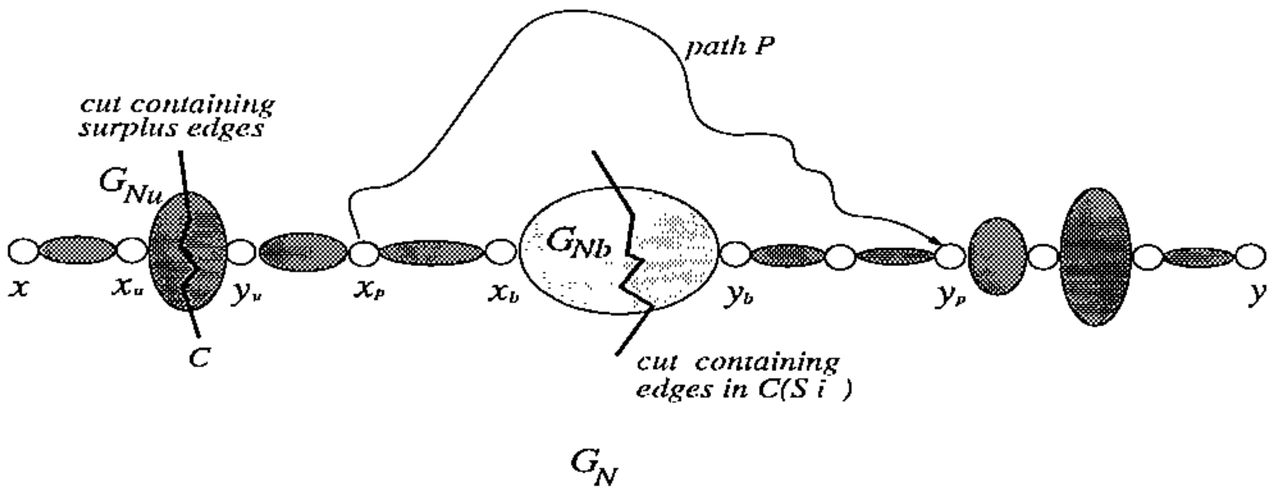


Figure 2: Graph G_N and its subgraphs G_{N_b}, G_{N_u} and path P.

between y_u and x_b in G_N. The length of any path from x to y going through G_{N_u} and G_{N_b} is the same in G_{R^*} and $G_{R'}$. However, the length of a path going through G_{N_u}, but not through G_{N_b} increases by δ. Assume there exists such a path in G_{R^*}. Let P be the path "by-passing" G_{N_b}, with x_p being the first and y_p being the last vertex on the path. Figure 2 shows such a path P and the subgraphs employed in our argument. We assume there exists no surplus cut between y_b and y_p. If one would exist, we make the leftmost such surplus cut the cut involved in the re-allocation.

Let $\|x_p \overset{G_{N_b}}{\rightsquigarrow} y_p\|_{S_{i+1}} = h_1$. Then, $\|x_p \overset{G_{N_b}}{\rightsquigarrow} y_p\|_{G_{R^*}} = h_1 + \delta + \gamma$, where $r_{i+1}(e) - r^*(e) = \gamma + \delta$, for any edge e in $C(S_i)$ and G_{N_b}, $\gamma \geq 0$. (The quantity γ corresponds to the amount of reductions still to be re-allocated in the future.) The last equation holds since there are no surplus cuts between x_p and y_p. Path P is not in S_i and thus every edge on P has a reduction of 0 in R_{i+1}. Hence,

$$\|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R'}} = \|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R^*}} \leq \|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R_{i+1}}} = \|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R_i}} \leq h_1.$$

Observe that we are again using the fact that the length of any path in G_{R^*} does not exceed the length of this path in G_{R_i}. Consider now, in R', the longest path from v_0 to v_n which contains path P. Its length is

$$\begin{aligned}
\|v_0 \rightsquigarrow x_p\|_{G_{R'}} + \|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R'}} + \|y_p \rightsquigarrow v_n\|_{G_{R'}} &= \\
\|v_0 \rightsquigarrow x_p\|_{G_{R^*}} + \delta + \|x_p \overset{P}{\rightsquigarrow} y_p\|_{G_{R'}} + \|y_p \rightsquigarrow v_n\|_{G_{R^*}} &\leq \\
\|v_0 \rightsquigarrow x_p\|_{G_{R^*}} + \delta + h_1 + \|y_p \rightsquigarrow v_n\|_{G_{R^*}} &\leq \\
\|v_0 \rightsquigarrow x_p\|_{G_{R^*}} + \delta + h_1 + \gamma + \|y_p \rightsquigarrow v_n\|_{G_{R^*}} &= \\
\|v_0 \rightsquigarrow x_p\|_{G_{R^*}} + \|x_p \overset{G_{N_b}}{\rightsquigarrow} y_p\|_{G_{R^*}} + \|y_p \rightsquigarrow v_n\|_{G_{R^*}} &\leq L.
\end{aligned}$$

Hence, the re-allocation of reduction cannot create a path whose length exceeds L and R' is another optimal solution. □

Lemma 3.4 *The root of decomposition tree D_i is labeled "g" and $R_{i+1} \leq R^*$.*

Proof: Let N be the root of D_i. Assume first that N is an s-node with label "b" for which child N_b is also labeled "b". Since there exist no surplus cuts in the subgraphs associated with the siblings of N_b, as well as in G_{N_b}, we have $\|v_0 \rightsquigarrow v_n\|_{G_{R^*}} > \|v_0 \rightsquigarrow v_n\|_{S_{i+1}} \geq L$. This contradicts the assumption that R^* is an optimal reduction. When N is a p-node labeled "b", each child of N is labeled "b". Applying the above argument to each child given a contradiction to assuming that N is labeled "b". Hence, the root is labeled "g" and, by the definition of the label "g", it follows that $R_{i+1} \leq R^*$. □.

From the above lemmas it follows that, if the algorithm terminates after k iterations, then reduction R_{k+1} generated by the algorithm is an optimal reduction.

4 0/1 reduction for series-parallel graphs and in-trees

We now turn to 0/1 edge reductions. The weight of a reduced edge is now $\epsilon \times d(v_i, v_j)$, where ϵ is given, $0 \leq \epsilon < 1$. Let G be an sp-graph containing m edges, and let h be the height of a decomposition tree D of G. In this section we present an $O(m^2 h)$ time algorithm for the reduction problems when the degree of every node in D is bounded by constant. Clearly, any decomposition tree can be turned into one of bounded degree by increasing its height. Our algorithm allows multiple edges between two vertices of G. We start by describing the approach used.

Let N_i be a node of D. Let G_i be the subgraph of G corresponding to the subtree of D rooted at vertex N_i. Assume that G_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 length of the longest path in G_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 in a bottom-up fashion, with a node using the arrays generated for its children. The final answer is determined from the array generated for the root N_{root} of D.

If node N_i is a leaf of decomposition tree D, G_i is an 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 is a p-node. Let N_l and N_r be the left child and right child of N_i, respectively. Assume T_l and T_r have already been determined. The entries in T_i can be defined by using T_r and T_l 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.

Assume now that N_i is an s-node. The entries in T_i can be defined by

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

We construct T_i by enumerating the values of $T_r[p] + T_l[q]$ for all pairs of (p,q), $0 \leq p, q \leq m_i$. This takes $O({m_i}^2)$ time.

Determining array T_{root} associated with the root of decomposition tree D takes $O(\sum_{i=1}^{m} {m_i}^2) = O(m^2 h)$ time. Recall that D contains $m-1$ interior vertices. The three reduction problems, the (G,L)-problem, (G,M)-problem and the tradeoff problem, can now be solved in $O(m^2 h)$ 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.

The approach described above can be used as follows for in-trees. Assume G is an in-tree of height h in which the in-degree of every node is bounded by a constant. All three 0/1 reduction problems can then be solved in $O(nh)$ time. We determine for every vertex v_i an array T_i defined as above. Let n_i be the number of vertices of G in the subtree rooted at v_i. Array T_i can be determined in $O(n_i)$ time from the arrays associated with v_i's children by using the method described for handling p-nodes. When the vertices of G do not have constant in-degree, the time bound of this approach is $O(\min\{nh \log c, n^2\})$, where c is the maximum in-degree of a vertex.

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.

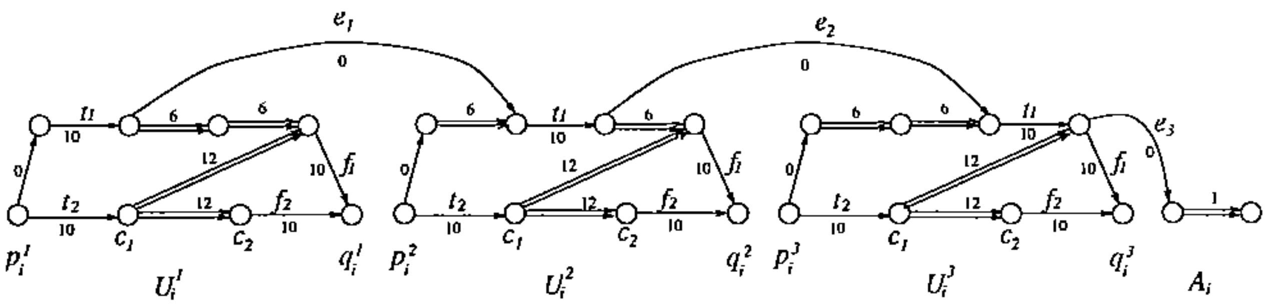
Theorem 5.1 *Given a weighted dag G and two positive reals M and L, it is NP-complete to decide whether there exists a $0/1$ reduction R with $\epsilon = 0$ such that $M(G_R) \leq M$ and $L(G_R) \leq L$.*

Proof: The problem is easily shown to be in NP. NP-completeness follows by a reduction from monotone 3-SAT [4]. 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 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.

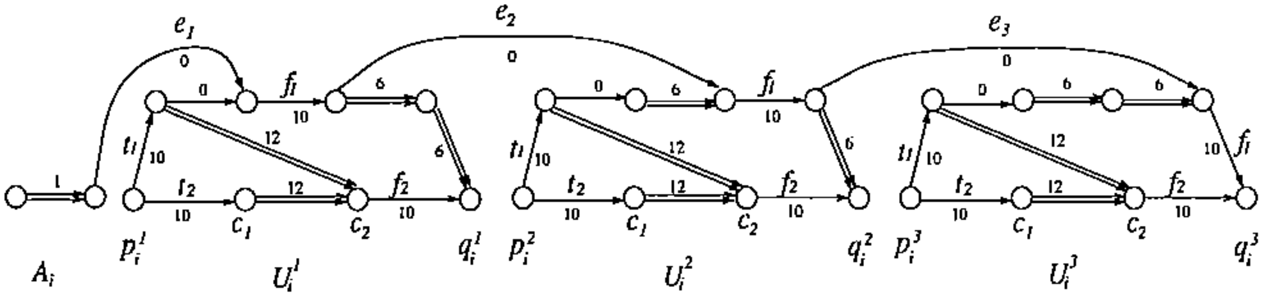
Graph G contains k *clause graph*, $G_1, G_2, \ldots, G_k$, which are connected by consistency edges. Clause graph G_i corresponds clause C_i, and we distinguish between positive and negative clause graphs (depending on the type of the corresponding clause). Each clause graph is made up of 3 components 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 3(a) shows a positive and Figure 3(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 3: 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 3. 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, $i < j$,



(a)



(b)

Figure 3: 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.

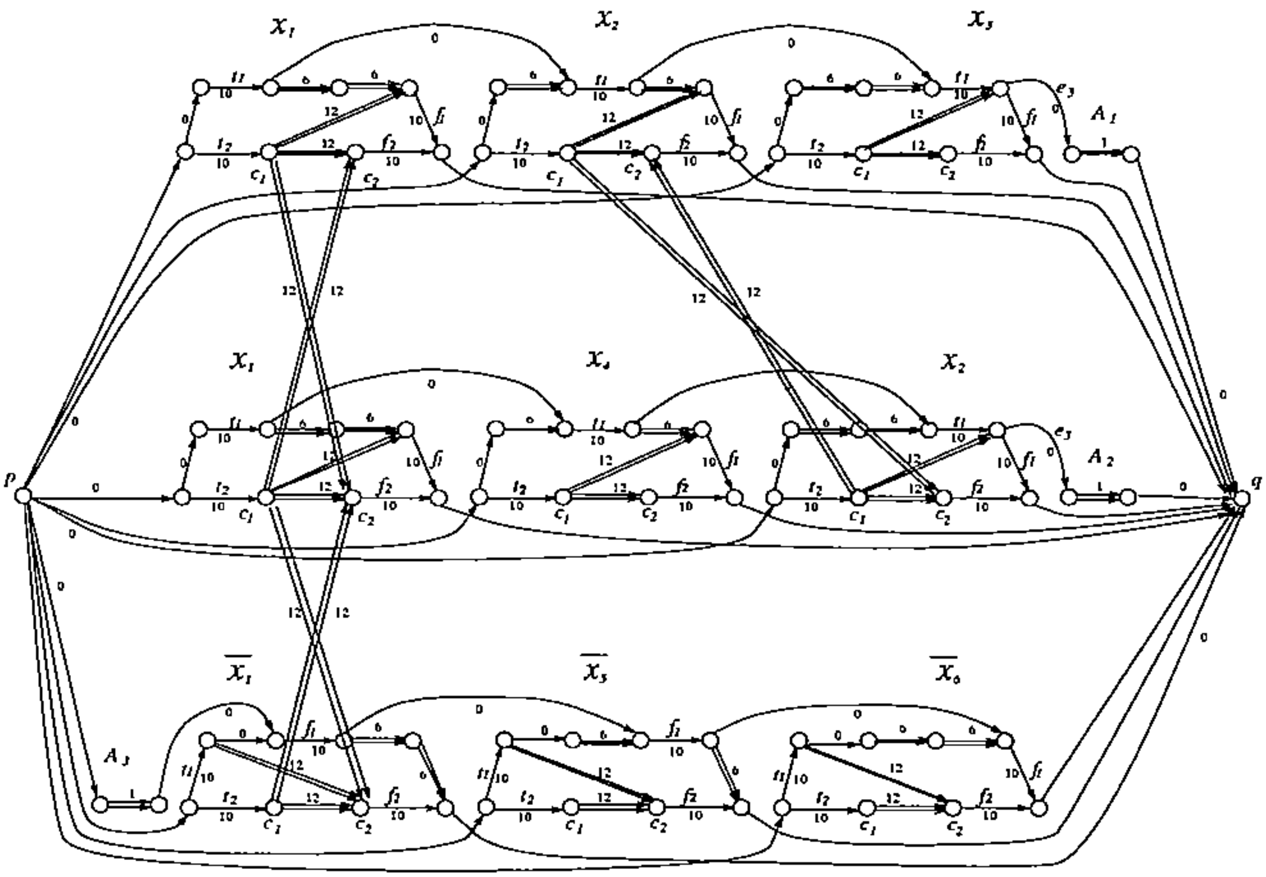


Figure 4: Graph G for formula $C = \{(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_5} \vee \overline{x_6})\}$.

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 sink vertex p and a source vertex q and edges of weight 0 from p to every p_i^j and from every q_i^j to q. Figure 4 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 5(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. Furthermore, 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 5(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 5(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 5(c). The

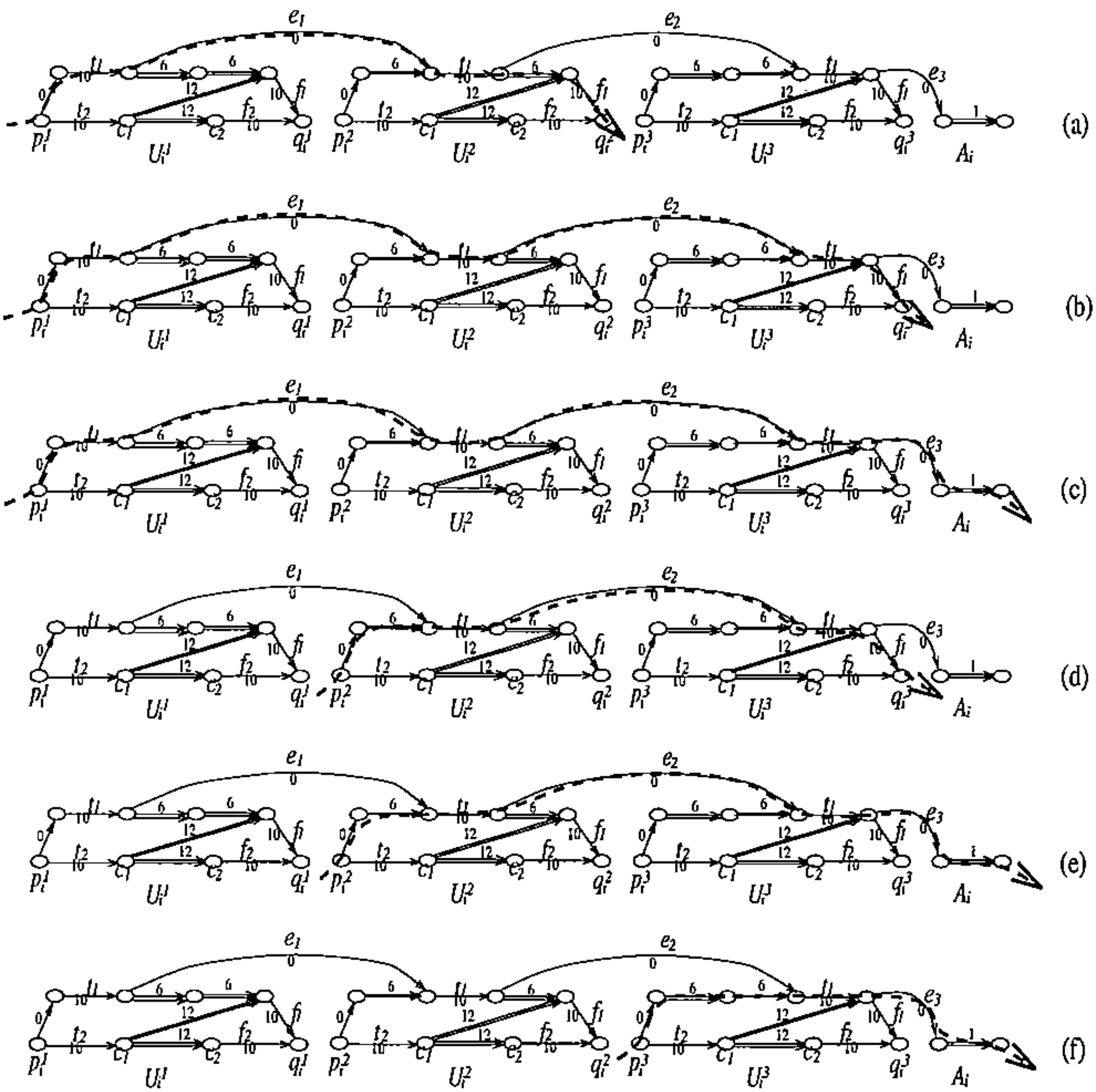


Figure 5: Paths in positive clause graph G_i going through different components and/or the attachment.

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 5(d). The length of P in G is 36 and its length in G_R is at most 26.

(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 5(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 5(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. □.

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. □.

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. □

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.

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.-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.

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

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

[5] 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.

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

[7] 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.

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

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