An efficient algorithm for minimum-weight bibranching

J. Keijsper and R. Pendavingh
Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde
Universiteit van Amsterdam

Abstract

Given a directed graph $D = (V, A)$ and a set $S \subseteq V$, a bibranching is a set of arcs $B \subseteq A$ that contains a v–$(V \setminus S)$ path for every $v \in S$ and an S–v path for every $v \in V \setminus S$. In this paper, we describe a primal-dual algorithm that determines a minimum weight bibranching in a weighted digraph. It has running time $O(n'(m + n \log n))$, where $m = |A|$, $n = |V|$ and $n' = \min\{|S|, |V \setminus S|\}$. Thus, our algorithm obtains the best known bounds for two important special cases of the problem: bipartite edge cover and r-branching.

1 Introduction

Let $D = (V, A)$ be a directed graph, S a subset of its vertices, and T the complement of S in V. A *bibranching* in D (with respect to S) is a set B of arcs such that

for each $v \in S$, B contains a directed path from v to a vertex in T,

for each $v \in T$, B contains a directed path from a vertex in S to v.

Bibranchings were introduced in [11].

There are two well-known special cases. First, for $S = \{r\}$, a minimal bibranching is exactly an r-branching, i.e. a directed tree rooted at r. Second, if S is one of the colour classes of a bipartite graph, and all the edges are given an orientation away from S, then a bibranching in the resulting digraph corresponds to an edge cover in the original graph. The way bibranchings generalize branchings and bipartite edge covers may be compared to the way matching forests in mixed graphs (cf. [7], [8], [9]) generalize branchings in directed and matchings in undirected graphs. However, there does not seem to exist a direct reduction of one structure to the other.

It is not difficult to see that the following algorithm finds a minimum cardinality bibranching in a digraph: determine a minimum edge cover on the bipartite subgraph induced by all S–T arcs, add a branching of the subgraph induced by T (where the 'root' is the set of vertices in T that have a neighbour in S and are therefore already reached by the edge cover), and similarly add an 'upside down branching' of the subgraph induced by S.

In this paper, we consider the more general *minimum-weight bibranching problem:*

Given: $D = (V, A)$, $S \subseteq V$, and a weight function $w : A \to \mathbb{Z}_+$

Find: a bibranching $B \subseteq A$ of minimum weight $w(B) = \sum_{b \in B} w(b)$.

The special case where either S or T is a singleton will be referred to as the *minimum-weight branching problem*; if D is bipartite with bipartition (S, T) the problem is called the *minimum-weight edge cover problem.*

Let n and m denote the number of vertices and the number of arcs of D, respectively. For the minimum-weight branching problem, a polynomial-time algorithm was first described in [1] and [2]. Presently, an $O(m+n\log n)$-algorithm is known (cf. [5]).

The minimum-weight edge cover problem can be reduced to the maximum-weight matching problem in $O(m)$ time as follows ([6]). For every vertex v, let $\mu(v) := \min\{w(a) \mid a \text{ incident with } v\}$, and let a_v be an arc incident with v for which $w(a_v) = \mu(v)$. Now, define a new weight function $\tilde{w}$ by $\tilde{w}(a) = \mu(u) + \mu(v) - w(a)$, where $a = (u, v) \in A$. If M is a maximum-weight matching with respect to $\tilde{w}$, then $M \cup \{a_v \mid v \text{ not covered by } M\}$ is a minimum-weight edge cover with respect to w. The maximum-weight matching problem in a bipartite graph can be solved by the Hungarian method [10]. The performance is $O(n'(m+n\log n))$, where $n' = \min\{|S|, |T|\}$, since it suffices to do n' shortest-path searches in a graph with nonnegative weights (cf. [3], [12]), while Dijkstra's shortest path algorithm sped up with Fibonacci heaps (cf. [4]) takes $O(m + n\log n)$ time.

Schrijver [11] showed that the minimum-weight bibranching problem is solvable in polynomial time, using the ellipsoid method. A purely combinatorial algorithm for this problem is given in section 2. It consists of three phases. The first phase together with the third phase is essentially a minimum-weight branching algorithm, while the second phase (after a simplified first phase) can be regarded as an extension of a direct bipartite edge cover algorithm. In section 3 we argue that our algorithm can be implemented so that it runs in $O(n'(m + n\log n))$ time, generalizing the complexity bounds mentioned for the special cases.

Some preliminaries are necessary before we can proceed to the description of the algorithm. Let $\mathcal{C} := (\{U \mid U \subseteq S\} \cup \{U \mid U \subseteq T\}) \setminus \{\emptyset\}$. For $a \in A$ and $U \in \mathcal{C}$, we say that *a covers* U if and only if $U \subseteq S$ and a leaves U, or $U \subseteq T$ and a enters U. Let M be the $\mathcal{C} \times A$ matrix with $(M)_{U,a} = 1$ if a covers U, and 0 otherwise. Consider the linear programming duality equation

$$\min\{wx \mid x \geq 0,\ Mx \geq \mathbf{1}\} = \max\{y\mathbf{1} \mid y \geq 0,\ yM \leq w\}. \tag{1}$$

It is easily verified that an integral optimal solution to the minimization problem in (1) is the characteristic vector of a bibranching of minimum weight. It was proved in [11] that both optima in (1) have integral optimal solutions for an integral weight function w. The correctness of our algorithm yields an alternative proof of this fact.

By complementary slackness, a feasible primal solution x and a feasible dual solution y are optimal solutions if and only if the following conditions are satisfied.

1. if $y(U) > 0$, we have equality in $Mx \geq \mathbf{1}$ at the row indexed by U and

2. if $x(a) > 0$, we have equality in $yM \leq w$ at the column indexed by a.

Define the function $w' : A \to \mathbb{Z}_+$ as follows:

$$w'(a) := w(a) - y(\{U \mid U \in \mathcal{C},\ a \text{ covers } U\}),$$

where for $X \subseteq \mathcal{C}$, $y(X)$ denotes $\sum_{U \in X} y(U)$. One verifies that for a bibranching B and a feasible dual solution y the above conditions translate to (taking for x the incidence vector of B):

1. if $y(U) > 0$ then U is covered by exactly one arc of B, and

2. if $a \in B$ then $w'(a) = 0$.

Note that 'y is feasible' is equivalent to '$y \geq 0$ and $w' \geq 0$'.

For $V' \subseteq V$ and $A' \subseteq A$, let $D(V', A')$ denote the subgraph of D with vertex set V' and arc set $\{a \in A' \mid a \text{ has both endpoints in } V'\}$.

Finally, let $\mathcal{L}$ be a collection of sets. $\mathcal{L}$ is said to be *laminar* if $U \subseteq W$ or $W \subseteq U$ or $U \cap W = \emptyset$ for any U, $W \in \mathcal{L}$. Let $\mathcal{L}$ be laminar. Then $U \in \mathcal{L}$ is called the *parent* of $U' \in \mathcal{L}$ (and U' is called a *child* of U) if U' is properly contained in U and is maximal in $\mathcal{L}$ with respect to that property.

2 Algorithm

We will now describe an algorithm for the minimum-weight bibranching problem, assuming that the given digraph $D = (V, A)$ contains at least one bibranching (with respect to the given subset $S \subseteq V$).

As variables we use the set $B \subseteq A$, a laminar collection $\mathcal{L} \subseteq \mathcal{C}$, and the function $y : \mathcal{L} \to \mathbb{Z}_+$. (Note that y can be extended to a function $\mathcal{C} \to \mathbb{Z}_+$ by defining $y(U) = 0$ for $U \in \mathcal{C} \setminus \mathcal{L}$.) The function w' is defined in terms of y and we will assume that all operations on y affect w' instantly.

Starting with the initial values $B = \emptyset$, $y = 0$ and $\mathcal{L} := \{\{v\} \mid v \in V\}$, the values of the variables are adjusted in the course of the algorithm in such a way that B develops into a bibranching, while the following properties are preserved:

(P0) y is feasible.

(P1) (a) If U is maximal in $\mathcal{L}$ then U is covered by at most one arc of B, or U is a singleton with $y(U) = 0$, covered by S–T arcs of B only.

(b) If $U \in \mathcal{L}$ is not maximal, then there is a unique $b \in B$ covering U but not its parent.

(c) If $U \in \mathcal{L}$, then $D(U, B)$ is strongly connected.

(P2) if $a \in B$ then $w'(a) = 0$.

In the first part of the algorithm, called Phase 1, B is augmented so that it covers all nonempty subsets of S. In Phase 2, B is improved in such a way that all nonempty subsets of T are covered by B-arcs, while all subsets of S remain covered. So after Phase 2, B is a bibranching. It may contain redundant arcs, though. These redundant arcs are removed in Phase 3 in such a way that for the feasible B and y the above conditions transform into the complementary slackness conditions.

Let $D_c = (V_c, A_c)$ be the digraph that is obtained from D by contracting the (inclusionwise) maximal sets in $\mathcal{L}$ and removing loops. Vertices of D_c correspond to sets in $\mathcal{L}$, so it makes sense to speak of $y(u)$ and of 'covering u by B' for a vertex $u \in V_c$. Since $\mathcal{L}$ contains only S-subsets and T-subsets, it is clear how to partition V_c into S_c and T_c. The set A_c is viewed as a subset of A: it is the set of elements of A with head and tail in distinct maximal elements of $\mathcal{L}$. Let B_c denote $B \cap A_c$.

All additions to and deletions from $\mathcal{L}$ in the course of the algorithm correspond exactly to contractions and decontractions in D_c. In fact, $\mathcal{L}$ can be seen as a way to record the 'contraction history' of D_c. We will often discuss the operations on $\mathcal{L}$ in terms of contraction and decontraction in D_c. Decontraction of $u \in V_c$ is the removal of the corresponding maximal element U of $\mathcal{L}$ from $\mathcal{L}$. After this operation, the new vertices u' appearing in V_c (corresponding to the children U' of U) are referred to as the children of u. Contraction is adding the union of a selection of maximal sets of $\mathcal{L}$ to $\mathcal{L}$. Clearly, $\mathcal{L}$ remains laminar after every such operation.

Now, starting with the initial values $B = \emptyset$, $y = 0$ and $\mathcal{L} := \{\{v\} \mid v \in V\}$ (so $w' = w$ and $D_c = D$), the first phase is as follows.

Phase 1

Repeat the following until every vertex of S_c is covered by B:

Loop 1

1. Find a vertex $u \in S_c$ that is not yet covered by B, and an arc $a \in A_c$ covering u of minimal weight $w'(a)$.
2. Increase $y(u)$ by $w'(a)$.
3. Add a to B.

4. If D_c now contains a directed cycle with arcs in B, contract this cycle in D_c.

Note that a cycle arising in step 4 is always contained in S_c, for only arcs with tail in S are added to B in step 3.

Claim 1 *After every iteration of Loop 1, B covers an extra element of $\mathcal{L}$ and the properties (P0), (P1), and (P2) hold. Phase 1 terminates after at most $2|S|-1$ iterations of Loop 1, and then B covers all nonempty subsets of S.*

Proof. Properties (P0), (P1) and (P2) obviously hold for the initial values of B, y and $\mathcal{L}$. Now assume that they are satisfied before a pass of Loop 1. We show that they still hold after the pass.

In step 1, we select a vertex u of S_c, corresponding to a maximal element $U \in \mathcal{L}$, and an arc a covering U. In step 2, feasibility of y is preserved, since y and w' remain nonnegative: w' is decreased by $w'(a) = \min\{w'(a) \mid a \text{ covers } U\}$ on each arc covering U. In particular, for the arc a that is inserted in B in step 3, $w'(a)$ has become zero, so (P2) also remains valid. Moreover, if adding a to B causes a vertex of D_c to get covered by more than one arc of B, then that must be a vertex in T_c (since u was uncovered before adding a). Vertices in T_c all correspond to singletons, have y-value zero and are only covered by S–T arcs throughout Phase 1. In step 4, a union of maximal elements U_i of $\mathcal{L}$ is only added to $\mathcal{L}$ when each of the U_i is covered by B. The (by assumption unique) B-arcs covering the U_i do not cover the union, since they form a cycle in D_c. Since the U_i are strongly connected by assumption, the union is also strongly connected because of this cycle. It follows by induction that (P0), (P1) and (P2) hold after every iteration.

Clearly, after every iteration in Phase 1, B covers an extra element of $\mathcal{L}$ that is a subset of S. Because $\mathcal{L}$ is laminar, it can contain at most $2|S|-1$ subsets of S. Therefore, Phase 1 terminates after at most $2|S|-1$ iterations. Suppose that some nonempty subset W of S is not covered by B when Phase 1 terminates. Then for every maximal U in $\mathcal{L}$ either $W \cap U = U$ or $W \cap U = \emptyset$, since every such U is strongly connected by B-arcs (by (P1)(c)). So W is a union of maximal elements from $\mathcal{L}$ and hence corresponds to a subset of the current S_c. But when Phase 1 terminates, every vertex of S_c is covered by B. So since W is uncovered, the B-arcs covering the vertices of S_c 'contained' in W must form at least one cycle in D_c. This contradicts the fact that in the last step of Phase 1 (step 4) any remaining cycle was contracted. It follows that when Phase 1 terminates, every nonempty subset of S is covered.

□

Phase 2 of the algorithm is similar to the first, in the sense that in every iteration a current B is improved to cover one more vertex of D_c, while the vertices covered before remain covered. Let $P \subseteq A_c$ be an undirected path in D_c (that is, P may traverse arcs of D_c backwards) with arcs alternatingly in

B_c and in $A_c \setminus B_c$. Suppose that the symmetric difference $B \triangle P$ covers all vertices of D_c covered by B-arcs, and in addition covers one vertex of D_c not covered by B. Then replacing B by $B \triangle P$ is an improvement of B in the above sense. To find such 'improving paths' P in D_c, we will use an auxiliary digraph $H = (V(H), A(H))$ with vertex set $V_c \cup \{r, s\}$ and the following arcs *corresponding to arcs* of D_c:

> for every arc $a \in A_c$ from $u \in S_c$ to $v \in T_c$ there is an arc (u, v) of length $l(u, v) = w'(a)$ in H,
>
> for every arc $b \in B_c$ from $u \in S_c$ to $v \in T_c$ there is an arc (v, u) of length $l(v, u) = -w'(b)$ in H,
>
> for every arc $a \in A_c$ from $u \in T_c$ to $v \in T_c$ there is an arc (r, v) of length $l(r, v) = w'(a)$ in H, unless v is covered by more than one arc of B_c,
>
> for every arc $b \in B_c$ from $u \in S_c$ to $v \in S_c$ there is an arc (r, u) of length $l(r, u) = -w'(b)$ in H.

In addition, there are arcs *representing the status of vertices* of D_c:

> for every vertex $v \in T_c$ covered by more than one arc of B_c there is an arc (r, v) of length $l(r, v) = 0$ in H,
>
> for every vertex $v \in S_c$ corresponding to a singleton in $\mathcal{L}$ there is an arc (r, v) of length $l(r, v) = y(v)$ in H, unless v is covered by an S_c–S_c arc in B_c,
>
> finally, for every vertex $v \in T_c$ not yet covered by B, there is an arc (v, s) of length $l(v, s) = 0$ in H.

For a path P in H, let $A(P)$ denote the set of arcs in A_c corresponding to (as defined above) the arcs of P.

Lemma 1 *If P is a directed r–s path in H, then $B \triangle A(P)$ covers every vertex of D_c that is covered by B, and in addition covers one vertex of D_c not covered by B. Moreover, if $l(P) = 0$ and the properties (P0),(P1) and (P2) hold for $(B, y, \mathcal{L})$, then they also hold for $(B \triangle A(P), y, \mathcal{L})$.*

Proof. Inspection of possible r–s paths shows that $A(P)$ is a path in D_c starting either with a T_c–T_c arc (traversed in forward direction) in $A_c \setminus B_c$, or with an S_c–S_c arc (traversed in backward direction) in B_c, or in a vertex $v \in T_c$ covered by at least two S–T arcs of B_c, or in a vertex $v \in S_c$ corresponding to a singleton in $\mathcal{L}$, then traversing a sequence of S–T arcs alternatingly in B_c (backward) and in $A_c \setminus B_c$ (forward), and finally ending in an uncovered vertex $v \in T_c$. But then all B-covered vertices of V_c are also $(B \triangle A(P))$-covered while the end-vertex in T_c is covered by $B \triangle A(P)$ but not by B. In fact, no vertex covered by a unique B-arc is covered by more than one arc of $B \triangle A(P)$, except that $v \in S_c$

corresponding to a singleton might be covered by two S–T arcs in $B \triangle A(P)$ if P starts with an arc (r, v) representing the y-value of v.

By feasibility of y ($w' \geq 0$), (P2) ($-w'(b) = 0$ for $b \in B$) and the definition of l, we have $l \geq 0$, so $l(P) = 0$ implies that every arc of P has length zero. Hence, every arc b of $B \triangle A(P)$ has weight $w'(b) = 0$. In particular, the first arc (r, v) of P has length zero, so if a singleton-vertex $v \in S_c$ covered by a unique B-arc is covered by two arcs of $(B \triangle A(P))$, then $y(v) = 0$. This proves (P2) and (P1)(a).

Clearly, (P0) and (P1)(b), (c) are not violated when B is replaced by $(B \triangle A(P))$.

□

In Phase 2 of the algorithm, we repeatedly search for a shortest r–s path in H, adjust the dual variable y so that the new length of the path becomes zero, and improve B using this zero-length path. Note that H is defined in terms of $\mathcal{L}$ (through D_c), B and y (also through w'), so changes in any of the variables affect H.

When an r–s path in H starts with an arc corresponding to a T_c–T_c arc, and B is improved using this path, then a T–T arc is inserted in B. So cycles of B-arcs may arise in T_c in Phase 2. As in Phase 1, such cycles are contracted in D_c. Cycles of B-arcs in S_c do not arise in Phase 2, for no S–S arcs are inserted in B during this phase.

Starting with B, y and $\mathcal{L}$ obtained in Phase 1, the description of Phase 2 of the algorithm is as follows.

Phase 2

Repeat the following until every vertex of T_c is covered by B:

Loop 2

1. Find a shortest r–s path P in H and compute $d(v)$ for each $v \in V_c$, where $d(v)$ denotes the distance from r to v in H with respect to the length function l.
2. For all $v \in S_c$: subtract $d(v)$ from $y(v)$. For all $v \in T_c$: add $d(v)$ to $y(v)$.
3. Replace B by $B \triangle A(P)$.
4. If D_c now contains a cycle with arcs in B, contract this cycle in D_c.

Step 1 of Loop 2 needs a more detailed description. To find a shortest path from r to s in H with respect to the length function l, Dijkstra's algorithm is used. This shortest path algorithm in fact determines the distance from r to every vertex of H.

While executing Loop 2 we want to preserve feasibility of y. In step 2, y-values of vertices in S_c are decreased. To retain $y \geq 0$ anyway, we decontract vertices during the execution of the shortest path routine in step 1. Vertices in S_c are decontracted when it is certain that their distance from r will be greater than their current y value. This also deals with the possibility that although not all vertices in T_c are covered by B, no r–s path exists in the current H: then vertices are decontracted until such a path appears.

Dijkstra's algorithm uses a tentative distance function $d : V_c \cup \{r, s\} \to \mathbb{Z}_+ \cup \{\infty\}$ and a tentative predecessor function $p : V_c \cup \{s\} \to A(H)$. In the following complete description of step 1, the steps marked by • ensure that vertices are decontracted when necessary; the other steps are the same as in the usual Dijkstra algorithm.

Step 1 (Shortest Path)

Set $d(r) = 0$ and $d(v) = \infty$ for all vertices $v \neq r$.

Repeat the following loop until every vertex of H is scanned:

(i) find an unscanned vertex $\bar{v} \in V(H)$ with $d(\bar{v})$ minimum;

•(ii) find an unscanned vertex $\bar{u} \in S_c$ with $y(\bar{u})$ minimum;

(iii) if $d(\bar{v}) \leq y(\bar{u})$:

for each arc $(\bar{v}, w) \in A(H)$: if $d(\bar{v}) + l(\bar{v}, w) < d(w)$, put $d(w) = d(\bar{v}) + l(\bar{v}, w)$ and $p(w) = (\bar{v}, w)$,

mark $\bar{v}$ scanned;

• else:

put $\bar{y} = y(\bar{u})$ and then $y(\bar{u}) = 0$,

expand$(\bar{u})$,

for every child u' of $\bar{u}$:

add $\bar{y}$ to $y(u')$,

if u' is covered by the B-arc that covered $\bar{u}$ then set

$d(u') = y(u')$ if u' corresponds to a singleton of $\mathcal{L}$, and $d(u') = \infty$ otherwise,

else set $d(u') = \bar{y}$;

Expansion of a vertex $u \in V_c$ covered by a unique B-arc is the following.

Expand(u)

Decontract u.

The B-arc covering u now covers one of its children, say u'; remove the unique B-arc covering u' but not u from B.

Note that *expand*(u) presumes that (P1) is valid and that u is covered by a (unique) B-arc and has children. In Claim 2, we will establish that, throughout Phase 2, (P1) holds, and all vertices of S_c are covered by B. Furthermore, in the execution of the shortest path routine, *expand* is only applied to vertices $\bar{u} \in S_c$ with children (so, by (P1), $\bar{u}$ is covered by a unique B-arc). For suppose $\bar{u}$ corresponds to a singleton in $\mathcal{L}$. If no $S-S$ arc of B covers $\bar{u}$, the arc $(r, \bar{u})$ of length $y(\bar{u})$ exists in H. If on the other hand an $S-S$ arc $b \in B$ covers $\bar{u}$, the arc $(r, \bar{u})$ of length $-w'(b) = -w(b) + y(\bar{u}) \leq y(\bar{u})$ exists in H. In both cases we have $d(\bar{v}) \leq d(\bar{u}) \leq y(\bar{u})$ for such a vertex as soon as r is scanned and $\bar{u}$ is not yet scanned. Hence $\bar{u}$ does not satisfy $y(\bar{u}) < d(\bar{v})$. These observations show that expansion is well-defined for our purposes.

We need to verify that expanding vertices does not interfere with Dijkstra's shortest path algorithm.

Lemma 2 *After every iteration of the loop in the shortest path algorithm,*

(A) all arcs of H have nonnegative length,

(B) scanned vertices v have $d(v)$ equal to the length of a shortest path from r to v in H,

(C) unscanned vertices have $d(v) = min\{d(x) + l(x, v) \mid x \text{ scanned}\}$.

Proof. Obviously, (A), (B) and (C) are valid when the shortest path routine is initiated. Now suppose that (A), (B) and (C) hold at the start of a loop. Then the vertex $\bar{v}$ selected in step (i) has $d(\bar{v})$ equal to the length of a shortest path from r to $\bar{v}$ in H, by (A), (B) and (C). If there is no expansion in step (iii), (B) and (C) are maintained as usual, and lengths in H are unaffected. In an expansion step, only arcs with length at least $y(\bar{u})$ are introduced, and scanned vertices x have $d(x) \leq y(\bar{u})$, since $\bar{u}$ was not expanded earlier. Therefore (A) and (B) are still valid at the end of the loop.

It remains to show that the value $d(u')$ assigned to a child u' of $\bar{u}$ after expansion of $\bar{u}$ satisfies $d(u') = \min\{d(x) + l(x, u') \mid x \text{ scanned }\}$. After expansion, (r, u') may be an arc of H. If not, the assigned distance $d(u')$ is ∞. We claim that if x is a scanned vertex such that (x, u') is an arc of H after expansion, then $x = r$. Suppose not, then x is a scanned vertex not equal to r. Then inspection of the definition of H shows that $x \in T_c$ and that there exists an arc from u' to x in B_c. But then there was an arc from $\bar{u}$ to x in B_c before expansion, and hence an arc $(x, \bar{u})$ of length 0 in H. Because x is scanned, $\bar{u}$ was an unscanned vertex with $d(\bar{u}) = d(x) \leq y(\bar{u}) < d(\bar{v})$, contradicting the minimality of $d(\bar{v})$. So, if (r, u') is an arc of H after expansion, then $\min\{d(x) + l(x, u') \mid x \text{ scanned}\} = d(r) + l(r, u') = l(r, u')$. Moreover, the assigned distance $d(u')$ is then equal to the length of (r, u'). This proves (C).

$\square$

In step 2 of Loop 2, we need the distances $d(v)$ for $v \in V_c$ to be finite. But this is the case after the execution of the shortest path algorithm in step 1. For suppose that the shortest path routine returns $d(v) = \infty$ for some $v \in V_c$. Then necessarily every vertex in S_c is either scanned or entirely decontracted, so in H there is a path from r to every vertex in S_c. Since $d(v) = \infty$, v is in T_c and is reached by no arc of H and hence by no arc of D_c. This means that there is no bibranching at all in D, contrary to our assumption. By the same argument, there exists an r–s path in H by the time step 1 is done: as long as $d(s) = \infty$, unscanned vertices are decontracted. The shortest r–s path P is found by tracing the predecessor function back from s.

Claim 2 *After every iteration of Loop 2, B covers an extra element of $\mathcal{L}$ and the properties (P0), (P1), and (P2) hold. Phase 2 terminates after at most $2|T| - 1$ iterations of Loop 2, and then B covers all nonempty subsets of S and T.*

Proof. By Claim 1, the properties (P0), (P1) and (P2) hold after Phase 1. We assume that they hold before a pass of Loop 2 and prove that they still hold after the pass.

In step 1, expansion is such that property (P1) remains valid and S-subsets in $\mathcal{L}$ remain covered. In step 2, property (P1) is also preserved, since any vertex v of T_c that is covered by more than one arc of B has distance $d(v) = 0$, so adding $d(v)$ to $y(v)$ does not make $y(v)$ positive (note that by assumption v is a singleton-vertex with $y(v) = 0$).

Next, we verify that (P0) and (P2) hold after steps 1 and 2. (P0) is equivalent to '$y \geq 0$ and $w' \geq 0$', and negative y-values do not occur after step 2 because of the expansion steps. For $S - S$ arcs $a \in A_c$, $w'(a)$ may become negative in expansion steps since we increase $y(u')$ for $u' \in S_c$. However, $d(u')$ exceeds the increment of $y(u')$ when the shortest path routine is done, so in step 2, $w'(a)$ becomes nonnegative again. Any remaining arc a of A_c is represented in H by an arc (u, v) of length $l(u, v) = w'(a)$. Since d is a distance function, we have $d(u) + l(u, v) \geq d(v)$ after step 1, and hence $w'(a) = l(u, v) + d(u) - d(v) \geq 0$ after step 2. Equality holds for arcs on a shortest path from r to any vertex. So after step 2, $w'(a)$ is nonnegative, and $w'(b) = 0$ for $b \in B_c$, since all arcs of B_c correspond to an arc on a shortest path from r in H. Hence (P0) and (P2) hold, and $l(P) = 0$.

Lemma 1 shows that after step 3, B covers an extra vertex of T_c and hence an extra element of $\mathcal{L}$, while every covered vertex remains covered. So after at most $2|T| - 1$ iterations Phase 2 terminates. Also by Lemma 1, step 3 does not interfere with the properties (P0), (P1) and (P2). Step 4 is identical to step 4 of Phase 1. The proof that B covers all nonempty subsets of S and T (or equivalently, that B is a bibranching) when Phase 2 terminates is also similar to the proof for Phase 1.

□

So we have that after Phase 2:

(P0*) y is feasible and B is a bibranching.

(P1*) (a) if $y(U) > 0$ then U is covered by exactly one arc of B, or $U \in \mathcal{L}$ is not maximal in $\mathcal{L}$.

(b) If $U \in \mathcal{L}$ is not maximal in $\mathcal{L}$, then there is a unique $b \in B$ covering U but not its parent.

(c) if $U \in \mathcal{L}$ then $D(U, B)$ is strongly connected.

(P2) if $a \in B$ then $w'(a) = 0$.

The properties (P0*), (P1*) and (P2) form our induction hypothesis in Phase 3. In this phase, elements of $\mathcal{L}$ are deleted. However, we do not mean to make their y-value equal to zero (y is feasible and need not be changed anymore). Therefore, from now on, we view y as a function defined on $\mathcal{C}$ instead of $\mathcal{L}$, and keep it fixed.

Phase 3

Repeat the following until $\mathcal{L} = \{\{v\} \mid v \in V\}$:

Loop 3

Select a vertex $u \in V_c$ not corresponding to a singleton in $\mathcal{L}$.

Expand(u).

Claim 3 *After every iteration of Loop 3, one element of $\mathcal{L}$ is removed, and the properties (P0*), (P1*), and (P2) hold. Phase 3 terminates after at most $|V| - 1$ iterations of Loop 3, and then $\mathcal{L}$ consists of singletons.*

□

When the algorithm terminates, all elements of $\mathcal{L}$ are maximal since $\mathcal{L}$ consists of singletons. Now (P1*) and (P2) imply that the feasible B and y satisfy complementary slackness. Hence, B is optimal.

3 Complexity

In this section, we show that by using techniques and data structures from [4] and [5], the running time of the algorithm described in the previous section can be bounded by $O(n'(m + n \log n))$, where $n = |V|, m = |A|$, and $n' = \min\{|S|, |T|\}$. The roles of S and T can be exchanged by reversing all arcs in D, so we may assume $n' = |T|$. Since we demand that D contains at least one bibranching, n is bounded by $O(m)$.

We store $\mathcal{L}$ by means of a *contraction forest*. The node set of this directed forest is (indexed by) $\mathcal{L}$, and there are arcs (U, U') whenever U' is a child of U. So the leaves of the contraction forest are singletons, and the roots are the maximal sets of $\mathcal{L}$. It takes $O(k)$ time to add the union of k maximal elements of $\mathcal{L}$ to $\mathcal{L}$ or to remove a maximal element of $\mathcal{L}$ having k children. We can list the vertices in in a given set $U \in \mathcal{L}$ by finding all leaves of the forest reachable from U by a directed path, taking $O(|U|)$ time.

At each node U of the contraction forest, $y(U)$ is stored. Also, $b(U)$ is kept at each node U, where $b(U)$ is the set of arcs of B covering U but not a possible parent of U. At each arc $a \in A$, we store membership of A_c and B, and $w'(a)$.

Phase 1 of our algorithm is essentially equal to the first phase of the branching algorithm, for which an implementation is given in [5]. If we view T as a single prescribed root node and reverse all arcs in D, the algorithm in [5] can be applied to compute B as in Phase 1, taking $O(m + n \log n)$ time. Since we want the data structures described above for use in Phase 2, we build the contraction forest (ignoring contractions involving T) while the branching algorithm runs. This takes no extra time.

Having thus built the contraction forest, computing w' is not time-consuming, since we can discard all $S - S$ arcs not in B after Phase 1. So we only need to compute $w'(u, v)$ for $u \in S$ and $v \in T$, which is at this point equal to $w(u, v) - \sum_{U \ni u} y(U)$. It takes $O(n)$ time to compute $\sum_{U \ni u} y(U)$ for all $u \in S$ by using the contraction forest containing $y(U)$ at each node U. Thus, Phase 1 can be done in $O(m + n \log n)$ time.

In Phase 2, Loop 2 is repeated until all subsets of T are covered. Since in each iteration an extra subset of T in $\mathcal{L}$ is covered by B, and since $\mathcal{L}$ is laminar, there are at most $O(n')$ iterations. We will argue that each iteration of Loop 2 takes at most $O(m + n \log n)$ time.

Steps 2, 3 and 4 take $O(m)$ time. The remaining part is step 1: finding a shortest path in H, and performing expansion steps.

When we want to apply the shortest path algorithm to the graph H, there are two main difficulties. First, D_c is coded in D and $\mathcal{L}$, i.e. it is given as a contracted graph. Second, to transform D_c into H requires reversing arcs, relocating arcs, etcetera.

First, we focus on applying the shortest path algorithm to a contraction of a general digraph $\Gamma = (V(\Gamma), A(\Gamma))$ with nonnegative length function $\lambda : A(\Gamma) \to \mathbb{Z}_+$, the contraction given by a partition $\mathcal{P}$ of the vertices.

We will modify the implementation of Dijkstra's algorithm given in [4] to meet our purpose. A *Fibonacci heap* or *F-heap* is a data structure to manipulate a number of *items*, each having a real number as its *key*. Manipulating at most t items, and under the condition that one starts and ends with empty F-heaps it takes

> $O(1)$ time to create an empty F-heap, insert an item with given key, decrease the key of an item in a heap, or find the item of minimum key.

$O(\log t)$ to delete an item of minimum key from the heap and return it.

Let $v_0 \in V(\Gamma)$ be given. The routine *Shortest Path*$(\Gamma, v_0, \mathcal{P}, \lambda)$ below computes shortest paths in the contracted graph determined by Γ and $\mathcal{P}$ from the vertex U_0 with $v_0 \in U_0 \in \mathcal{P}$ to all other vertices. It uses an F-heap with vertices $v \in V(\Gamma)$ as its items. The key of a vertex v is $d(v)$, the tentative distance from v_0 to v. The distance function is defined on original vertices, not on contracted ones. When the routine terminates, $d(v) = d(u)$ for any u and v in the same element U of the partition $\mathcal{P}$. Vertices are inserted into the F-heap when a value is assigned to $d(v)$ for the first time. After that, each time a value is assigned to $d(v)$ a 'decrease key' operation is performed. Once a vertex v is removed from the heap by a 'delete min' operation, no more assignments are made to $d(v)$. The vertex is then marked *scanned*. Besides the function d, a predecessor function $p : V(\Gamma) \to A(\Gamma)$ is computed. When the routine terminates, $p(u)$ is an arc by which a shortest path enters U with $u \in U \in \mathcal{P}$. So a shortest path in the contracted graph is obtained by tracing the function p back in $O(|V(\Gamma)|)$ time.

Shortest path$(\Gamma, v_0, \mathcal{P}, \lambda)$

Mark all vertices 'not scanned'.

Create an empty F-heap F.

Set $d(v_0) = 0$.

While the F-heap is not empty:

delete a vertex v of minimum key $d(v)$ from F;

if v is not scanned:

$\star$ compute the set U such that $v \in U \in \mathcal{P}$;

for each $u \in U$:

put $d(u) = d(v)$ and $p(u) = p(v)$;

$\star\star$ for all $(u, x) \in A(\Gamma)$: if $d(u) + \lambda(u, x) < d(x)$, put $d(x) = d(u) + \lambda(u, x)$ and $p(x) = (u, x)$;

mark u 'scanned'.

There is at most one 'delete min' operation for each element of $V(\Gamma)$, and at most one 'insert' or 'decrease key' for each element of $A(\Gamma)$. This gives a bound of $O(|A(\Gamma)| + |V(\Gamma)| \log |V(\Gamma)|)$ for the time taken by operations on the F-heap F.

Now we want to apply the routine *Shortest Path* to the graph H defined in the previous section. This can be done by taking $V \cup \{r, s\}$ for the vertex set of Γ, the set of maximal elements of $\mathcal{L}$ (together with the singletons $\{r\}$ and $\{s\}$) for the partition $\mathcal{P}$ and the length function l for λ. The arc set of Γ is in one-to-one correspondence with the arc set of H, except that Γ may have more arcs entering s. Thus, Γ has $O(m)$ arcs. It follows from the above analysis that operations involving the F-heap F take time $O(m + n \log n)$.

Given our coding of $\mathcal{L}$, in the step marked by $\star$, it takes $O(|U|)$ to find all elements of the maximal $U \in \mathcal{L}$ that contains a given v. One simply finds all leaves in the same tree of the contraction forest as node $\{v\}$.

In the step marked by $\star\star$, we need to enumerate the arcs leaving u in Γ. If $u = r$, we need to list all the $S - S$ arcs of B_c, $T - T$ arcs of A_c, the maximal sets of $\mathcal{L}$ in S that are singletons, and the maximal sets of $\mathcal{L}$ in T covered by more than one B-arc. This takes $O(m)$ time, since membership of B and A_c is stored at each arc and $b(U)$ is stored at each $U \in \mathcal{L}$. If $u \in S$, the arcs leaving u are exactly the arcs $(u, t) \in A$ with $t \in T$. If $u \in T$, there is an arc to s if U is not covered (where U is the maximal element of $\mathcal{L}$ containing u) and an arc to $x \in S$ if there is an arc $(x, u) \in B$. Running through all arcs leaving vertices of S, and all arcs entering vertices of T takes $O(m)$ time. It takes $O(n)$ time to find for all $u \in T$ the maximal $U \in \mathcal{L}$ containing u, and to check whether $b(U) = \emptyset$. No arc leaves s.

So we have a bound of $O(m)$ for operations not involving the F-heap, giving a total time bound of $O(m + n \log n)$ for one run of the shortest path routine in step 1 of Loop 2.

During the execution of the shortest path routine, expansion steps are performed. The minimum $d(v)$ must be compared with the minimum $y(U)$, where U ranges over the maximal sets of $\mathcal{L}$ that are subsets of S. To find the latter minimum, we use a second F-heap F' that has nodes U of the contraction forest as its entries, each with key $y(U)$. Before the shortest path algorithm starts, F' is created and the root nodes U with $U \subseteq S$ are inserted. It then takes $O(1)$ time to compare the minimum key $y(U)$ of F' with the minimum key $d(v)$ of F in the shortest path routine. When $y(U) < d(v)$, the following heap operations on F' are performed:

> delete the node U of minimum key $y(U)$ from F';
>
> for each child U' of U: add $y(U)$ to $y(U')$ and insert U' in F' with key $y(U')$.

Next, the algorithm performs **expand**(U):

> suppose $b(U) = \{b\}$; select the child U' of U that is covered by b; delete the elements of $b(U')$ from B; put $b(U') = \{b\}$;
>
> delete U from the contraction forest.

And finally values of d (and p) are defined (requiring 'insert' in F), but not more than once for each child, so not more than $2|S| - 1$ times.

At most $2|S| - 1$ items are inserted in F', and each of them is deleted as the minimum once (including emptying F' by deletemins at the end of step 1).

So the operations on F and F' in expansion steps take only $O(n \log n)$ time for one iteration of Loop 2.

Phase 3 just consists of the application of *expand*(U) on maximal sets $U \in \mathcal{L}$ that are not singletons, until none are left. We will obtain a bound for the time taken by *expand* in Phase 2 and 3 together. Note that $b(U)$ always contains a single arc, because we never apply *expand*(U) to a singleton U. The proper subsets $U' \in \mathcal{L}$ of U that are covered by b, when $b(U) = \{b\}$, are called the *heirs* of U. To be able to select the child that is heir in constant time in expansion steps, we prepare the contraction forest before each application of the shortest path algorithm and before Phase 3, by simply indicating whether a node U is heir. This takes $O(n)$ time by applying the following recursive routine *heir(U)* to each root U of the contraction forest.

heir(U)

unmark U.

if $b(U) = \{b\}$, and u is the endvertex of b in U, walk from $\{u\}$ to U in the contraction forest marking each node, until a child U' of U is reached.

for each child U'' of U not equal to the heir U', do *heir*(U'')

When *expand*(U) is applied, the heir U' of U becomes a root of the contraction forest and is hence not an heir anymore, but nothing else changes about heirness. We can simply unmark U'. Now, *expand*(U) takes $O(k)$ time, where k is the number of children of U. A node of the contraction forest meets *expand*(U) as a child of U at most once. There are at most $2|S| - 1$ nodes in the contraction forest for subsets of S, and $2|T| - 1$ nodes for subsets of T. Hence, there are at most $O(n)$ possible children. So the applications of *expand* in Phases 2 and 3 together take at most $O(n)$ time.

Thus, all operations in the three phases together can be performed in at most $O(n'(m + n \log n))$ time.

Acknowledgement

We thank Lex Schrijver for introducing us to the subject and for carefully reading the manuscript.

References

[1] Y.J. Chu and T.H. Liu, On the shortest arborescence of a directed graph, *Sci. Sinica* **14** (1965), 1396-1400.

[2] J. Edmonds, Optimum branchings, *J. Res. Nat. Bur. Standards* **71B** (1967), 233-240.

[3] J. Edmonds and R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19** (1972), 248-264.

[4] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34** (1987), 596-615.

[5] H.N. Gabow, Z. Galil, T. Spencer and R.E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica* **6** (2) (1986), 109-122.

[6] J.F. Geelen, personal communication.

[7] R. Giles, Optimum matching forests I: special weights, *Math. Programming* **22** (1982), 1-11.

[8] R. Giles, Optimum matching forests II: general weights, *Math. Programming* **22** (1982), 12-38.

[9] R. Giles, Optimum matching forests III: facets of matching forest polyhedra, *Math. Programming* **22** (1982), 39-51.

[10] H.W. Kuhn, The Hungarian method for the assignment problem, *Naval Res. Logist. Quart.* **2** (1955), 83-97.

[11] A. Schrijver, Min-max relations for directed graphs, *Ann. Discrete Math.* **16** (1982), 261-280.

[12] N.Tomizawa, On some techniques useful for solution of transportation network problems, *Networks* **1** (1972), 173-194.