A Quick Method for Finding Shortest Pairs of Disjoint Paths

J. W. Suurballe
Bell Laboratories, West Long Branch, New Jersey

R. E. Tarjan
Bell Laboratories, Murray Hill, New Jersey

Let G be a directed graph containing n vertices, one of which is a distinguished source s, and m edges, each with a non-negative cost. We consider the problem of finding, for each possible sink vertex v, a pair of edge-disjoint paths from s to v of minimum total edge cost. Suurballe has given an $O(n^2 \log n)$-time algorithm for this problem. We give an implementation of Suurballe's algorithm that runs in $O(m \log_{(1+m/n)} n)$ time and $O(m)$ space. Our algorithm builds an implicit representation of the n pairs of paths; given this representation, the time necessary to explicitly construct the pair of paths for any given sink is $O(1)$ per edge on the paths.

I. INTRODUCTION

Let $G = (V, E)$ be a directed graph containing n vertices $v \in V$, one of which is a distinguished *source* vertex s, and m edges $(v, w) \in E$, each with a non-negative *length* $c(v, w)$. We extend the length function to (directed) paths by defining the length $c(p)$ of a path p to be the total length of its edges. A path from a vertex v to a vertex w is *shortest* if there is no path from v to w of shorter length. The *distance* from a vertex v to a vertex w, denoted by $d(v, w)$, is the length of the shortest path from v to w, defined only if a shortest path exists.

In this paper we consider the following *shortest pairs problem*: find, for each possible sink v, a pair of edge-disjoint paths from s to v of minimum total length. (Thus we seek n pairs of paths.) In discussing this problem we shall assume that every vertex in G is reachable from s (thus $m \geq n - 1$), and for convenience in stating time bounds that $n \geq 2$; this entails no loss of generality. Note that the reachability of a vertex v from s is not enough to imply the existence of two edge-disjoint paths from s to v; there exist two such paths if and only if there is no edge contained in all paths from s to v. We shall also assume that G is antisymmetric [if (v, w) is an edge then (w, v) is not an edge]. Removing this restriction is easy (see Section IV).

For a single sink v, finding a shortest pair of disjoint paths from s to v is a special case of minimum-cost network flow and is solvable in $O(m \log_{(1+m/n)} n)$ time using two iterations of an efficient implementation [5], [10] of Dijkstra's single-source

NETWORKS, Vol. 14 (1984) 325-336

CCC 0028-3045/84/020325-12$04.00

shortest-path algorithm [2], [7]. This gives a bound of $O(nm \log_{(1+m/n)} n)$ for all n sinks. Suurballe [8] discovered an ingenious way to combine the Dijkstra calculations for the various sinks, thus obtaining an $O(n^2 \log n)$-time* algorithm for the n-sink problem. (Suurballe considered vertex-disjoint rather than edge-disjoint paths, but his version of the problem is linear-time reducible to ours; see Section IV.) In this paper we propose an efficient implementation of Suurballe's algorithm (adapted to find edge-disjoint paths). Our version runs in $O(m \log_{(1+m/n)} n)$ time and $O(m)$ space, the same bounds as for Dijkstra's algorithm. The algorithm builds an implicit representation of the shortest pairs of paths; the time necessary to construct the pair for any given sink from this representation is $O(1)$ per edge on the paths.

In Section II we provide a description and proof of the algorithm. Suurballe's proof uses the dual simplex method for linear programming; our proof is combinatorial, relying on properties of Dijkstra's algorithm. In Section III we discuss efficient implementation of the algorithm. In Section IV we conclude with some remarks about allowing multiple edges, the relationship between the edge-disjoint and vertex-disjoint versions of the problem, and what happens when we allow edges to have negative lengths.

II. AN ALGORITHM FOR THE SHORTEST PAIRS PROBLEM

Suurballe's algorithm requires two preliminary steps (see Figure 1):

Step 1. Find a shortest-path tree T rooted at s. Such a tree contains, for every vertex v, a shortest path from s to v. Compute $d(s, v)$, the distance from s to v, for every vertex v. We call the edges in T *tree edges* and the remaining edges *nontree edges.*
Step 2. Transform the length of every edge (v, w) by defining $c'(v, w) = c(v, w) - d(s, w) + d(s, v)$.

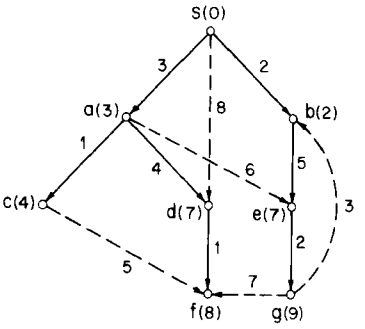
Step 1 requires $O(m \log_{(1+m/n)} n)$ time using Dijkstra's single-source shortest-path algorithm [2], [5], [10]. Step 2, which takes $O(m)$ time, is a well-known transformation based on the duality principle of linear programming and has the following property (which is true even if G contains negative-length edges but no negative cycles):

(1) For every edge (v, w), $c'(v, w) \geq 0$, with equality if (v, w) is a tree edge.

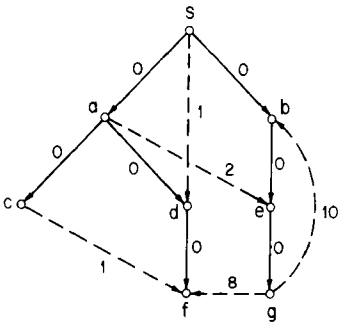
If p is any path from a vertex v to a vertex w, then $c'(p) = c(p) - d(s, w) + d(s, v)$. Thus two paths with the same start and finish vertices have their lengths transformed by the same amount, and the ordering of such paths by length is unaffected by the transformation. This means that we can solve the shortest pairs problem for the transformed lengths and obtain a solution for the original lengths. [We can compute the original length of a path from its transformed length in $O(1)$ time.] Henceforth in discussing the shortest pairs problem we shall assume that the lengths $c(v, w)$ have property (1). This implies in particular that $d(s, v) = 0$ for every vertex v.

For any vertex v, let G_v be the graph formed from G by reversing all the edges along

*All logarithms in this paper are base two unless otherwise stated.



(a)



(b)

FIG. 1. Preliminary steps of Suurballe's algorithm. (a) Shortest-path tree of a directed graph. Tree edges are solid, nontree edges are dashed. Distances from source s are in parentheses. (b) Graph after edge length transformation.

the path in T from s to v (see Fig. 2). (Without the antisymmetry restriction, G_v may have multiple edges.) The following theorem is immediate from the minimum-cost augmenting path algorithm for minimum-cost network flow [6]:

Theorem 1. For any vertex v, the total length of a shortest pair of edge-disjoint paths from s to v is $d_v(s, v)$, where d_v denotes the distance function in G_v. Such a pair of paths can be obtained from the path from s to v in T and a shortest path from s to v in G_v by taking the union of the edges in the two paths, discarding every edge in one path whose reversal appears in the other, and grouping the remaining edges into two paths (see Fig. 2).

Theorem 1 implies that we can find a shortest pair of paths for a single sink v in $O(m \log_{(1+m/n)} n)$ time by using one (more) application of Dijkstra's algorithm, and thus pairs for all sinks in $O(nm \log_{(1+m/n)} n)$ time. We obtain a faster algorithm by realizing that the graphs G_v for different vertices v are related; we can in effect run Dijkstra's algorithm in parallel on all of them, obtaining $d_v(s, v)$ for all vertices v in a single Dijkstra-like calculation.

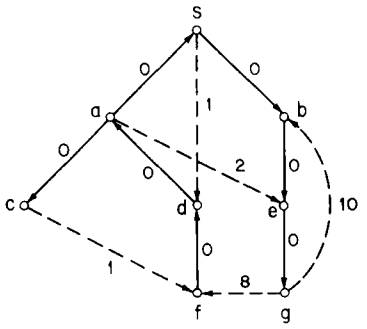


FIG. 2. Graph G_f for the graph G in Fig. 1(b). A shortest path from s to f in G_f is (s, d), (d, a), (a, c), (c, f). A shortest pair of paths from s to f in G is (s, a), (a, c), (c, f), and (s, d), (d, f).

In order to understand the method, we must first understand Dijkstra's algorithm. Let G be a graph with source s and arbitrary non-negative edge length function c. Dijkstra's algorithm is a labeling process that maintains, for each vertex v, a *tentative distance* $d(v)$ such that $d(v) \geq d(s, v)$. The algorithm also maintains a partition of the vertices into *unlabeled* vertices and *labeled* vertices; if v is a labeled vertex $d(v) = d(s, v)$. For every vertex v such that $v \neq s$ and $d(v)$ is finite, the algorithm maintains a *tentative predecessor* $p(v)$ such that there is a path from s to v of length $d(v)$ whose last edge is $(p(v), v)$. Initially $d(s) = 0$, $d(v) = \infty$ if $v \neq s$, all vertices are unlabeled, and $p(v)$ is undefined for every vertex v. The algorithm consists of repeating the following step until there is no unlabeled vertex v with $d(v)$ finite:

Labeling Step. Choose an unlabeled vertex v such that $d(v)$ is minimum. Make v labeled. For each edge (v, w), if $d(v) + c(v, w) < d(w)$, define $d(w)$ to be $d(v) + c(v, w)$ and $p(w)$ to be v.

When the algorithm terminates, $d(v) = d(s, v)$ for every vertex v reachable from s, and the set of edges $\{(p(v), v) | v \neq s \text{ and } d(v) \text{ is finite}\}$ defines a shortest-path tree rooted at s. [If v is not reachable from s, $d(v) = \infty$ and $p(v)$ is undefined on termination.] The algorithm labels vertices in nondecreasing order by distance from s. If the collection of unlabeled vertices v with $d(v)$ finite is maintained as a heap [10] implemented as a d-heap [4], [10] with $d = \theta(1 + m/n)$, then the running time of Dijkstra's algorithm, which is dominated by the heap operations, is $O(m \log_{(1+m/n)} n)$ [10].

Let us return to the shortest pairs problem. Recall that we assume c has property (1). We shall describe a variant of Dijkstra's algorithm that computes $d_v(s, v)$ for every vertex v. The algorithm maintains the same variables $d(v)$ and $p(v)$ as Dijkstra's algorithm, though their meaning is slightly different, as we discuss below. The algorithm maintains a new variable $q(v)$ for every vertex v, whose meaning we also discuss below. As in Dijkstra's algorithm, every vertex is either unlabeled or labeled. Initially $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$, all vertices are unlabeled, and $p(v)$ and $q(v)$ are undefined for every vertex v.

The algorithm needs one more concept beyond those in Dijkstra's algorithm. Removing the labeled vertices from the shortest-path tree T divides T into subtrees that span the set of unlabeled vertices. The algorithm maintains these *unlabeled subtrees*. Initially there is only one unlabeled subtree, equal to T.

The algorithm consists of repeating the following step until there is no unlabeled vertex v with $d(v)$ finite (see Fig. 3):

Labeling Step. Choose an unlabeled vertex v such that $d(v)$ is minimum. Let S be the unlabeled subtree containing v. Make v labeled. This splits S into new unlabeled subtrees; there is one new subtree for the parent of v, if it exists and is unlabeled, and one new subtree for each unlabeled child of v. For each nontree edge (u, w) such that u and w are in S and either $u = v$ or u and w are in different unlabeled subtrees after v is labeled, if $d(v) + c(u, w) < d(w)$, define $d(w)$ to be $d(v) + c(u, w)$, $p(w)$ to be u, and $q(w)$ to be v.

We say an edge (u, w) is *processed* when vertex v is labeled if u and w are in the same unlabeled subtree before v is labeled and either $u = v$ or u and w are in different unlabeled subtrees after v is labeled; thus labeling v entails testing the inequality $d(v) + c(u, w) < d(w)$. For any vertex v, if $p(v)$ is defined then $q(v)$ is the vertex whose labeling caused $(p(v), v)$ to be processed. Each edge is processed at most once; some edges may never be processed.

Theorem 2. When the shortest pairs algorithm terminates, $d(v) = d_v(s, v)$ for every vertex v. Thus $d(v)$ is the total length of a shortest pair of edge-disjoint paths from s to v [$d(v) = \infty$ if there is no such pair]. If $v \neq s$ and $d(v)$ is finite then in G_v there is a path from s to v of length $d(v)$ whose last edge is $(p(v), v)$.

Proof: Let v be any vertex. We shall run Dijkstra's algorithm on G_v in a way that simulates the behavior of the shortest pairs algorithm running in parallel on G. We can

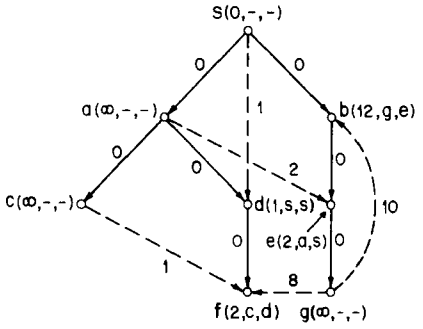


FIG. 3. Values of d, p, and q computed for graph in Fig. 1(b) by shortest pairs algorithm. Execution proceeds as follows: s labeled, (s, d), (a, e), (f, g) processed; d labeled, (c, f) processed; e labeled, (g, b) processed; f labeled; b labeled. Successive values of $d(f)$ are ∞, 8, 2.

do this so as to maintain the following invariant, where d_v and p_v denote the tentative distance and parent functions Dijkstra's algorithm computes on G_v.

(2) A vertex x is unlabeled in G_v if and only if in G it is in the same unlabeled subtree as v. If x is unlabeled in G_v, $d_v(x) = d(x)$, and $p_v(x) = p(x)$ (if either is defined). If x is labeled in G_v, $d_v(x) = d(y)$, where y is the vertex whose labeling in G removed x from the unlabeled subtree containing v; if $x = y$, $p_v(x) = p(y)$.

The invariant certainly holds initially. Suppose the invariant holds before the shortest pairs algorithm labels some vertex y in G. There are two cases:

Case 1 (y is labeled in G_v). By the invariant, y and v are in different unlabeled subtrees just before y is labeled in G. Labeling y affects none of the variables for the vertices in the unlabeled subtree containing v and thus does not affect the invariant. To simulate the labeling of y and G, Dijkstra's algorithm running on G_v does nothing.

Case 2 (y is unlabeled in G_v). By the invariant, y and v are in the same unlabeled subtree just before y is labeled in G. Let S be the set of vertices in this subtree, and let S' be the set of vertices in the labeled subtree containing v just after y is labeled (if $y = v$, S' is empty). By the invariant, $d_v(y)$ must be minimum among unlabeled vertices in G_v. To simulate the labeling of y in G, Dijkstra's algorithm running on G_v first labels y and then labels the remaining vertices in $S - S'$ in an appropriate order; every such vertex is reachable from y in G_v by a path of length zero consisting of a sequence of zero or more reversals of tree edges followed by zero or more tree edges. It is easy to see that this simulation preserves the invariant.

Thus the invariant always holds, and the theorem follows immediately. ∎

We conclude this section by describing how, given the values of d, p, and q computed by the shortest pairs algorithm, we can explicitly construct a pair of paths for a given vertex v [with $d(v)$ finite]. The algorithm constructs each path by traversing it backward. The algorithm consists of initialization and two backward traversals, one to construct each path. We begin with all vertices unmarked. To carry out the initialization, we define x to be v and repeat the following step until $x = s$.

Initialization Step. Mark x. Replace x by $q(x)$.

The initialization step must terminate at s, since for any vertex x, $q(x)$ is labeled before x, and s is labeled first. [The set of edges $\{(q(x), x) | x \neq s \text{ and } d(x) \text{ is finite}\}$ defines a tree rooted at s.] To construct one of the paths, we define x to be v, the path to be empty, and repeat the following step until $x = s$:

Traversal Step. If x is marked, unmark x, add $(p(x), x)$ to the front of the path, and replace x by $p(x)$. Otherwise add (y, x) to the front of the path and replace x by y, where y is the parent of x in T.

As an example, consider the graph of Figure 3 with $v = f$. The marked vertices are f and d. The first path generated is $(s, a), (a, c), (c, f)$. The second path generated is $(s, d), (d, f)$.

If two paths are constructed in this way, they will be edge-disjoint and have total edge length $d(v)$; together they contain $(p(v), v)$ for every vertex v marked in the initialization and in addition zero or more tree edges. This follows immediately from Theorems 1 and 2. The construction of the two paths unmarks all the marked vertices; thus an arbitrary sequence of pairs of paths can be constructed with no additional initialization. The time to construct one or more pairs of paths is $O(1)$ time per edge on the paths plus $O(n)$ time to initially unmark all the vertices; the $O(n)$ unmarking time is dominated by the time required for the calculation of d, p, and q. If we construct all n pairs of paths, the total time is $O(n^2)$ (this estimate may be pessimistic).

III. IMPLEMENTATION OF THE SHORTEST PAIRS ALGORITHM

We can implement the shortest pairs algorithm in the same way as Dijkstra's algorithm, using a d-heap with $d = \theta(1 + m/n)$ to store the unlabeled vertices. The only added complication is that we need a mechanism to determine which edges to process when a vertex is labeled. Not counting the time to generate edges for processing, the shortest pairs algorithm runs in $O(m \log_{(1+m/n)} n)$ time. (Recall that each edge is processed at most once.)

To generate edges for processing, we use a three-part data structure. The first part consists of a preorder and a postorder numbering of the vertices of T. We denote the preorder number of a vertex v by $pre(v)$ and the postorder number by $post(v)$. Computing these numberings takes $O(n)$ time [9] and allows us to test the ancestor-descendant relationship in $O(1)$ time: a vertex v is an ancestor of a vertex w if $pre(v) \leqslant pre(w)$ and $post(v) \geqslant post(w)$ [9].

The second part of our data structure represents the partition of the unlabeled vertices into unlabeled subtrees. With each unlabeled vertex v, we store a doubly linked list $children(v)$ of its unlabeled children. We also store with v its parent $p(v)$, if it is defined and unlabeled; if v has no parent (it is the root of T) or its parent is labeled, we define $p(v)$ to be a special value *null*. With this representation we can start at any unlabeled vertex and visit all vertices in its unlabeled subtree in $O(1)$ time per vertex visited. Doubly linking the lists of children allows us to delete a child in $O(1)$ time.

Initially there is one unlabeled subtree containing all the vertices; its initialization takes $O(n)$ time. When labeling a vertex v, we update the unlabeled subtrees by defining $p(w) = null$ for each vertex in $children(v)$ and deleting v from $children(p(v))$ if $p(v) \neq null$. Since each vertex is labeled at most once and occurs as a child at most once, the total time to update the unlabeled subtrees is $O(n)$.

The third part of our data structure is a doubly linked list $incident(v)$, for each vertex v, of the unprocessed nontree edges incident to v. [These are the unprocessed nontree edges of the form (v, w) or (w, v).] Within each such list, we sort the edges in nondecreasing order on the preorder number of the end other than v; for each possible other end, there are at most two edges in the list. Initializing all the incidence lists requires $O(m)$ time if we use radix sorting [1].

When labeling a vertex v, we update the unlabeled subtrees and then generate edges for processing, as follows. First we scan $incident(v)$. For each edge (u, w) encountered, we delete (u, w) from $incident(u)$ and $incident(w)$, and if $u = v$ we process (u, w). This requires $O(|incident(v)|)$ time, or $O(m)$ time when summed over all ver-

tex labelings. Next, we traverse the new unlabeled subtrees formed by labeling v. The traversal is slightly different for the subtree containing $p(v)$ than for the subtrees containing the children of v. If $p(v) \neq null$ we traverse the subtree containing $p(v)$ by visiting each vertex x in the subtree. When visiting x we scan $incident(x)$. For each edge (u, w) encountered, we test whether the end other than x is a descendant of v. If so we delete (u, w) from $incident(u)$ and $incident(w)$ and process it. If not, we skip it. [In this case we call (u, w) *wasted.*]

We traverse the subtree containing a given child y of v by visiting each vertex x in the subtree. When visiting x, we begin a scan of $incident(x)$ in the forward direction. For each edge (u, w) encountered, we test whether the end other than x is a descendant of y. If not, we delete (u, w) from $incident(u)$ and $incident(w)$ and process it. If so, we skip it. [We call (u, w) wasted.] In this case we abort the forward scan and begin a backward scan of $incident(x)$. We deal with edges encountered during the backward scan in exactly the same way as during the forward scan, except that when we encounter a wasted edge we abort the backward scan and go on to the next vertex in the subtree. The ordering of the edges in $incident(x)$ ensures that the edges that must be processed occur in two contiguous groups, at the front and at the rear of $incident(x)$, and the forward and backward scan encounter them all. (This is because the descendants of y are numbered consecutively in preorder [9].)

We carry out the traversals of the various unlabeled subtrees concurrently, by taking a step in each subtree, then another step in each subtree not yet completely traversed, and so on, until completely traversing all but one of the subtrees. A *step* consists of continuing the traversal in the current subtree until encountering a wasted edge or completing the visit to a vertex. We do not need to complete the traversal of the last subtree since every edge that needs processing has its ends in two different subtrees and occurs in an incidence list in each. When all but the last traversal is complete, every remaining unprocessed nontree edge has both its ends in the same unlabeled subtree.

It is easy to verify the correctness of this method. The only tricky point in the implementation is to make sure that the traversals of the various subtrees do not interfere with each other. To prevent such interference, when deleting an edge (u, w) from a list $incident(z)$, we must check whether there is a scan of $incident(z)$ suspended at edge (u, w). If so, before deleting (u, w) we must move the pointer to (u, w) maintained by the scan either forward or backward one edge in $incident(z)$, depending on whether the scan is proceeding forward or backward. Even with this additional requirement, deleting an edge from an incidence list requires only $O(1)$ time.

The task remaining is to analyze the time required for subtree traversals. A subtree traversal step takes $O(1)$ time plus $O(1)$ time per edge processed. Thus the total time for subtree traversals is $O(m)$ plus time proportional to the number of steps.

To count steps, let us consider the labeling of a vertex v. Let S be the unlabeled subtree containing v just before v is laveled, let S_0 be the unlabeled subtree (if any) containing $p(v)$ just after v is labeled, and let $S_1, S_2, \ldots, S_k$ be the subtrees containing the unlabeled children of v just after v is labeled, in nonincreasing order on the number of vertices in S_i. We shall count the steps that take place while traversing each of these subtrees. We can ignore one of the subtrees of our own choice, since the concurrency of the traversals implies that the number of steps taken within any single sub-

tree is at most one more than the number taken within some other subtree; thus if we ignore one subtree but add one to our count, we are off by at most a factor of 2. The extra count of one per vertex labeling sums to at most n for all vertex labelings.

Consider first the count of steps within $S_2, \ldots, S_k$. Each vertex x in such a subtree accounts for at most two steps [one to abort the forward scan and one to abort the backward scan of *incident*(x)]. labeling v causes the size of the unlabeled subtree containing x to decrease by a factor of at least 2. Thus over all vertex labelings a single vertex x can account for at most 2 log$n$ steps in such subtrees, for a total count over all vertices and vertex labelings of at most $2n \log n$.

We must still count the steps within S_0 and S_1. We consider two cases:

Case 1 (S_0 small). If $\alpha |S_0| \leqslant |S|$, where α is a positive constant to be chosen later, we ignore steps within S_1. Each vertex x in S_0 accounts for at most $m(x) + 1$ steps, where $m(x)$ is the number of edges incident to x. Labeling v causes the size of the unlabeled subtree containing x to decrease by a factor of at least α. Thus over all vertex labelings a single x can account for at most $[m(x) + 1] \log_\alpha n$ steps in such subtrees for a total count over all vertices and vertex labelings of at most $(2m + n) \log_\alpha n$ since $\Sigma_{x \in V} m(x) = 2m$.

Case 2 (S_0 large). If $\alpha |S_0| \geqslant |S|$, we ignore steps within S_0. Each vertex x in S_1 accounts for at most two steps, and labeling v causes the size of the unlabeled subtree containing x to decrease by a factor of at least $\alpha/(\alpha - 1)$. Over all vertices and vertex labelings the total count of steps in such subtrees is thus at most $2n \log_{\alpha/(\alpha-1)} n$.

Combining our estimates we find that the total count of steps is at most

$$n + 4n \log n + (4m + 2n) \log_\alpha n + 4n \log_{\alpha/(\alpha-1)} n.$$

The last term in this sum is

$$O\left(\frac{n \log n}{\log [\alpha/(\alpha - 1)]}\right) = O\left(\frac{n \log n}{\log (1 + 1/\alpha)}\right) = O(\alpha n \log n).$$

Let us choose

$$\alpha = \max\left\{2, \frac{m}{[n \log (1 + m/n)]}\right\}.$$

Then $\log \alpha = \max \{1, \log (m/n) - \log \log (1 + m/n)\} = \Omega(\log (1 + m/n))$, and the total count of steps is

$$O(n \log n + m \log_{(1+m/n)} n) = O(m \log_{(1+m/n)} n).$$

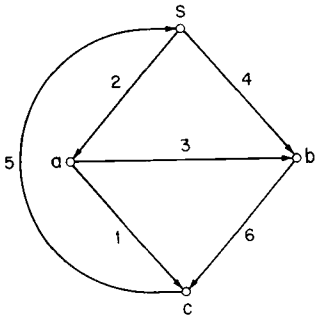
Combining our estimates for the running times of all parts of the algorithm, we obtain the following theorem:

Theorem 3. The shortest pairs algorithm runs in $O(m \log_{(1+m/n)} n)$ time, the same bound as for Dijkstra's algorithm. The space required is $O(m)$.

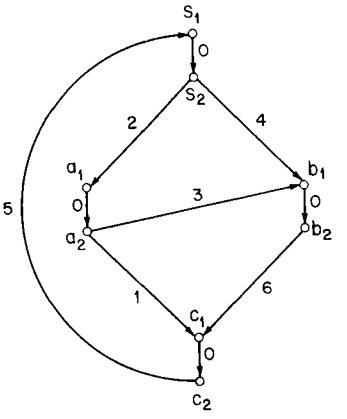
IV. REMARKS

We can easily adapt our algorithm to work on arbitrary multidigraphs; that is, directed graphs with multiple edges, without the antisymmetry restriction. We need only redefine the tentative predecessor $p(v)$ of a vertex v to be an edge entering v rather than a vertex preceding v.

Another easy adaption allows our algorithm to find shortest vertex-disjoint pairs of paths. Given an input graph G, we construct a graph G' by splitting each vertex v of G into two vertices v_1 and v_2 joined by an edge (v_1, v_2) of length zero. An edge (v, w) of G becomes an edge (v_2, w_1) of G, (see Fig. 4). We then solve the shortest pairs problem for graph G' with source s_2. For any vertex v in G, a shortest pair of vertex-disjoint paths from s to v in G corresponds to a shortest pair of edge-disjoint paths from s_2 to v_1 in G', and vice versa. G' has $2n$ vertices and $n + m$ edges and is constructible in $O(m)$ time. Thus we can solve the shortest pairs problem for vertex-disjoint paths in $O(m \log_{(1+m/n)} n)$ time and $O(m)$ space.



(a)



(b)

FIG. 4. Vertex-splitting transformation: (a) original graph G, (b) transformed graph G'.

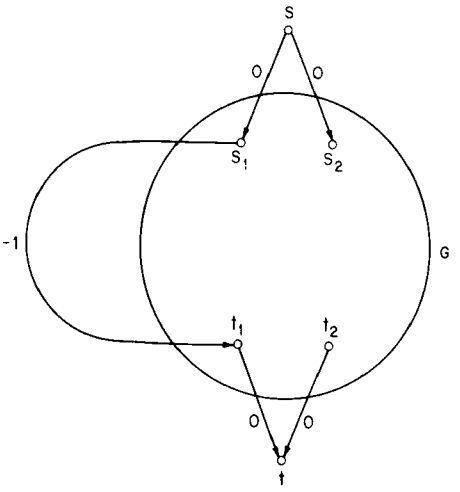


FIG. 5. Transformation of the two-paths problem to the shortest pair existence problem. All edges in G have zero cost.

A third question is what happens when G contains one or more edges of negative length. If G contains no negative cycles, we can solve the shortest pairs problem in $O(nm)$ time by finding shortest paths in G using the Ford-Bellman algorithm [6], [10] instead of Dijkstra's algorithm. If G is allowed to contain negative cycles, the problem of determining whether G contains a shortest pair of paths from a given source s to a given sink t is NP-complete. We can prove this by reducing the following *two-paths* problem, which is NP-complete [3], to the shortest pair existence problem:

Two-Paths Problem

Instance. A directed graph $G = (V, E)$ and four distinct vertices, s_1, s_2, t_1, t_2.

Question. Are there two vertex-disjoint paths, one from s_1 to t_1, the other from s_2 to t_2?

Given an instance of the two-paths problem, we construct an instance of the shortest pair existence problem consisting of a graph G' whose vertices are those of G with an extra source s and an extra sink t, and whose edges are those of G, with length zero; (s, s_1), (s, s_2), (t_1, t), (t_2, t), also with length zero; and (t_1, s_1), with length minus one (see Fig. 5). It is easy to show that G contains two vertex-disjoint paths, one from s_1 to t_1 and one from s_2 to t_2, if and only if G' does not contain a shortest pair of vertex-disjoint paths from s to t (because there are arbitrarily short pairs of paths). We can reduce the vertex-disjoint to the edge-disjoint problem by the vertex-splitting transformation discussed above.

Finally, we may ask what happens if we seek shortest sets of k edge-disjoint paths for k an arbitrary positive integer. (We have considered the special case of $k = 2$.) For any $k \geqslant 1$, the minimum-cost augmenting-path algorithm for minimum-cost network flow [6] will find a shortest set of k edge-disjoint paths from a given source s to

a given sink v in k iterations of Dijkstra's algorithm, or $O(km \log_{(1+m/n)} n)$ time. For all n possible sinks, this method takes $O(knm \log_{(1+m/n)} n)$ time. We leave as an open problem the extension of our improvement of this result for $k = 2$ to the case $k > 2$.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA (1974).

[2] E. W. Dijkstra, A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959) 269–271.

[3] S. Fortune, J. Hopcroft, and J. Wylie, The directed subgraph homeomorphism problem. *Theoretical Computer Sci.* 10 (1980) 111–121.

[4] D. B. Johnson, Priority queues with update and finding minimum spanning trees. *Information Processing Lett.* 4 (1975) 53–57.

[5] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks. *Assoc. Comput. Mach.* 24 (1977) 1–13.

[6] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York (1976).

[7] J. W. Suurballe, Disjoint paths in a network. *Networks* 4 (1974) 125–145.

[8] J. W. Suurballe, The single-source, all-terminals problem for disjoint paths. Unpublished technical memorandum, Bell Laboratories (1982).

[9] R. Tarjan, Finding dominators in directed graphs. *SIAM J. Comput.* 3 (1974) 62–89.

[10] R. E. Tarjan, Data structures and network algorithms. *Soc. Ind. Appl. Math.* (1983).