Finding Common Ancestors and Disjoint Paths in DAGs

David Eppstein*

Department of Information and Computer Science
University of California, Irvine, CA 92717
http://www.ics.uci.edu/∼eppstein/

Tech. Report 95-52

December 15, 1995

Abstract

We consider the problem of finding pairs of vertex-disjoint paths in a DAG, either connecting two given nodes to a common ancestor, or connecting two given pairs of terminals. It was known how to find a single pair of paths, for either type of input, in polynomial time. We show how to find the k pairs with shortest combined length, in time $O(mn + k)$. We also show how to count all such pairs of paths in $O(mn)$ arithmetic operations. These results can be extended to finding or counting tuples of d disjoint paths, in time $O(mn^{d-1} + k)$ or $O(mn^{d-1})$. We give further results on finding the subset of the DAG involved in pairs of disjoint paths, and on finding disjoint paths in linear space.

*Work supported in part by NSF grant CCR-9258355 and by matching funds from Xerox Corp.

1 Introduction

We are interested in two problems in directed acyclic graphs (DAGs), both involving pairs of vertex-disjoint paths. In the first problem, we are given two nodes in the DAG, and we wish to find pairs of paths connecting those nodes to common ancestors. In the second, we are given two pairs of terminals (s_1, t_1) and (s_2, t_2), and the task is to connect each pair with a path.

The first problem, finding common ancestors, arises in genealogy: if one has a database of family relations, one may often wish to determine how some two individuals in the database are related to each other. Formalizing this, one may draw a DAG in which nodes represent people, and an arc connects a parent to each of his or her children. Then each different type of relationship (such as that of being a half-brother, great-aunt, or third cousin twice removed) can be represented as a pair of paths from a common ancestor (or couple forming a pair of common ancestors) to the two related individuals, with the specific type of relationship being a function of the numbers of edges in each path, and of whether the paths begin at a couple or at a single common ancestor. In most families, the DAG one forms in this way has a tree-like structure, and relationships are easy to find. However in more complicated families with large amounts of intermarriage, one can be quickly overwhelmed with many different relationships. For instance, in the British royal family, Queen Elizabeth and her husband Prince Philip are related in many ways, the closest few being second cousins once removed (through King Christian IX of Denmark and his wife Louise), third cousins (through Queen Victoria of England and her husband Albert), and fourth cousins (through Duke Ludwig Friedrich Alexander of Württemberg and his wife Henriette). A program I and my wife Diana wrote, Gene [3], is capable of finding these relationships quickly using a backtracking search with heuristic pruning, but Gene starts bogging down when asked to produce larger numbers of relationships in the same database, hence my interest in worst-case bounds for the problem.

The problem of finding pairs of disjoint paths between specified pairs of terminals has been much more well studied. For the DAG version considered here, the main result is due to Perl and Shiloach [7], who show how to find such a pair of paths, if one exists, in time $O(mn)$. Their method is easily generalized to finding a shortest pair of paths (measured by total path length), or to finding tuples of d disjoint paths between distinct specified terminals; in the latter case the running time would become $O(mn^{d-1})$. If d is not a constant, the problem of finding multiple disjoint paths becomes

NP-complete [8]. (Actually this paper considers edge-disjoint paths but the edge-disjoint and vertex-disjoint problems are easily transformed into one another.) Li et al [6] give a pseudo-polynomial algorithm for an optimization version of the two-path problem in which the length of the longer path must be minimized.

1.1 New Results

We show the following results, all of which assume a directed acyclic graph with n vertices and m arcs, with real-valued arc lengths.

- In $O(mn)$ time we can construct a data structure such that, given any two nodes u and v in the DAG, we can list (an implicit representation of) the k shortest pairs of vertex-disjoint paths from a common ancestor to u and v, in time $O(k)$. The same bound holds for listing all pairs with length less than a given bound (where k is the number of such paths). Alternately, the pairs of paths can be output in order by total length, in time $O(\log i)$ to list the ith pair. We can find each pair of paths explicitly from its implicit representation in time proportional to the number of edges in the pair. Our representation also allows computation of some simple functions (such as the length or number of edges of each path in the pair) in constant time.

- In $O(mn)$ arithmetic operations we can count, for each two nodes in the graph, the number of pairs of disjoint paths from a common ancestor to those two nodes. Each operation combines numbers of (pairs of) paths, and hence involves arithmetic on numbers of $O(n)$ bits. (It is easily shown that any DAG has at most 2^n paths.)

- Given any d pairs (s_i, t_i) and (s_2, t_2) of terminals, we can find in time $O(mn^{d-1} + k)$ an implicit representation of the k shortest d-tuples of vertex-disjoint paths connecting those terminals. As above, the tuples can be output in order in time $O(\log i)$ for the ith path, we can compute explicit representations in linear time, and we can compute simple functions on the paths in constant time. In $O(mn^{d-1})$ arithmetic operations on $O(dn)$ bit numbers we can compute the number of tuples of vertex-disjoint paths from s_i to t_i.

As in the algorithm of Perl and Shiloach for finding a single pair of disjoint paths between specified terminals [7], our method involves constructing a DAG D' with $O(n^2)$ or $O(n^d)$ nodes and $O(mn)$ or $O(mn^{d-1})$ arcs, in

which each tuple of paths in the original DAG D is represented by a single path between a certain pair of nodes in D'. However the method in that paper allows a tuple of paths in D to be represented by more than one path in D'; we modify the construction so that the representation is unique. We then apply dynamic programming to count paths in D', or use our previous algorithm [2] to find the k shortest paths.

The time bound of our algorithm is quite reasonable, but its quadratic space requirement makes it unsuitable for practical implementation. To ease this problem, we provide the following further results:

- Given any two nodes u and v in a DAG, we can in linear time find the subset of the DAG consisting of only those nodes and arcs involved in some pair of disjoint paths from a common ancestor to u and v.
- Given any two nodes u and v in a DAG with nonnegative edge lengths, we can find the set of pairs of disjoint paths from a common ancestor to u and v, having length less than some given bound, in time $O(kmn)$ and space $O(m+n)$, where k represents the number of path pairs satisfying the length bound. If we desire the k shortest pairs without a length bound, we can solve the problem again in linear space, with time either $O(k^2mn)$ or $O(kmn \log x)$ where in the second bound the edge lengths are assumed to be integers, and x denotes the length of the longest pair of paths found.

2 Representing Path Pairs by Paths

We first consider the version of the problem in which we wish to find common ancestors of a pair (u, v) of nodes. Given a DAG D, we construct a larger DAG D_1 as follows. We first find some topological ordering of D, and let $f(x)$ represent the position of vertex x in this ordering.

We then construct one vertex of D_1 for each ordered pair of vertices (x, y) (not necessarily distinct) in D. We also add one additional vertex s in D_1. We connect (x, y) to (x, z) in D_1 if (y, z) is an arc of D and $f(z) > \max(f(x), f(y))$. Symmetrically, we connect (x, y) to (z, y) if (x, z) is an arc of D and $f(z) > \max(f(x), f(y))$. We connect s to all vertices in D_1 of the form (v, v).

Lemma 1. *Let vertices u and v be given. Then the pairs of disjoint paths in D from a common ancestor a to u and v are in one-for-one correspondence with the paths in D_1 from s through (a, a) to (u, v).*

Proof: If we have such a path in D_1 we can find two paths from a to u and v simply by choosing the left and right sides respectively of each ordered pair in the path. These two paths must be vertex disjoint, since after the first time some vertex x appears on one or the other side of an ordered pair, every succeeding vertex y has $f(y) > f(x)$.

Conversely, suppose we have a pair of disjoint paths from a to u and v. We form a sequence of ordered pairs, starting from (a, a), by sorting the vertices of both paths according to their topological ordering and successively replacing one or the other side of each ordered pair by the next vertex in that order. This produces a path in D_1 according to the lemma.

We thus have two maps, one from paths in D_1 to pairs of paths in D, and one in the other direction. To show that these objects are in one-to-one correspondence, it suffices to show that composing these two maps in either order gives the identity mapping.

Starting from a path in D_1, each arc replaces one vertex of each ordered pair; the replaced vertices must be already in sorted order according to the topological ordering, and the two maps preserve the information about which vertex goes on which side of each ordered pair, so sorting the vertices and placing them back into ordered pairs recovers the original path.

Starting from a pair of paths, each individual path must again be sorted according to the topological ordering, so sorting the vertices to make a single path in D_1 simply corresponds to shuffling the two original paths. The map from a path in D_1 back to two paths in D simply undoes that shuffling, so again the composition is the identity. □

Thus we can represent paths from a common ancestor in D by single paths in D_1. We now describe a similar construction for the problem of finding a collection of disjoint paths between terminals (s_i, t_i). We assume for simplicity that all terminals are distinct; our construction is easily modified to handle non-distinct terminals as we describe later.

Given a collection of d tuples (s_i, t_i) in D, augment D by adding two vertices s and t, and arcs (s, s_i) and (t_i, t) for each terminal t. Form a graph D_2 as follows. Form a topological ordering of D, and let $f(v)$ represent the position of v in this ordering. Let the vertices of D_2 consist of ordered d-tuples of vertices of D.

Connect a tuple $v = (v_1, v_2, \ldots v_d)$ with another tuple $w = (w_1, w_2, \ldots w_d)$ exactly when the following conditions hold: (1) some arc connects some pair (v_i, w_i), (2) if $v_i = s$ then $w_i = s_i$ and if $w_i = t$ then $v_i = t_i$, (3) for each $j \neq i$, $v_j = w_j$, and (4) either $w_i = t$ or $f(w_i) > \max f(v_j)$. If terminals are

non-distinct, the construction must be modified by weakening condition (4) to allow $f(w_i) = f(v_j)$ when $i > j$ and w_i and v_j are terminals of paths i and j.

Lemma 2. *Given D and (s_i, t_i), the construction above produces a DAG D_2 such that d-tuples of disjoint paths connecting the terminals in D correspond one-for-one with paths from $(s, s, \ldots s)$ to $(t, t, \ldots t)$ in D_2.*

The proof is essentially the same as that for Lemma 1: we map paths in D_2 to sets of paths in D by keeping only one position in each d-tuple in D_2; we map sets of paths in D to paths in D_2 by sorting the vertices and using the sorted order to change d-tuples of vertices in D_2 one position at a time; the composition of these maps in either order is the identity mapping for the same reasons as before.

Note that the additional vertices s and t by which we augmented D are necessary, as there does not always exist a path in D_2 from $(s_1, s_2, \ldots s_i)$ to $(t_1, t_2, \ldots t_i)$. For instance, if $f(t_i) < f(s_j)$ for some i and j, such a path can never exist because we would be unable to change the ith position of the tuple to t_i without violating the condition that $f(w_i) > f(v_j)$.

3 Comparison with Perl and Shiloach

The constructions of the previous section are similar in some respects to that appearing in a paper by Perl and Shiloach [7], which uses similar ideas to solve in $O(mn)$ time the problem of finding a single pair of vertex disjoint paths connecting a specified pair of terminals. However there are some important differences which we now discuss.

The construction of Perl and Shiloach again forms tuples of vertices from D, connected by arcs corresponding to changing a single vertex in a tuple.

Instead of a topological ordering, Perl and Shiloach use level numbers measuring the length of the longest path from each vertex. (Note that the ordering of these is opposite that of our topological ordering positions.) Instead of introducing extra vertices s and t, Perl and Shiloach remove edges out of each terminal t_i, so that the level numbers of the t_i are all zero and the situation discussed earlier for which we added s and t does not arise. And instead of our ordering condition that $f(w_i) > \max(f(v_j))$, Perl and Shiloach use the condition that $\ell(v_i) \geq \max(\ell(w_j))$.

The first change, of using level numbers, causes Perl and Shiloach to use a $\geq$ test in the ordering condition, where we use a $>$ test. The fact that

our test reverses the roles of v and w is insignificant (equivalently one could reverse the edges in the input DAG), but the change from strict inequality to possible equality means that their construction forms a one-to-many rather than one-to-one representation of the tuples of disjoint paths. Thus it is unsatisfactory for counting or listing more than one path. The remaining change, of removing edges rather than our solution of adding extra vertices s and t, makes it difficult for Perl and Shiloach's algorithm to be generalized to allow non-distinct terminals. In particular, it cannot allow some $s_i = t_j$, and it cannot allow the situation which arises in our genealogical application, in which in one relation one person is an ancestor of another, but in which the two people have a third common ancestor in other relations.

4 Finding and Counting Disjoint Paths

We use the following, which is a specialization to DAGs of the main result of our previous paper [2], and is proved in that paper.

Lemma 3. *Let D be a DAG with a specified vertex s. Then in time $O(m+n)$ we can construct a data structure from D, such that an implicit representation of the k shortest paths from s to any vertex t can be found in time $O(k)$. From this implicit representation we can construct each path in time proportional to its number of edges. We can compute along with each implicitly represented path, in constant time per path, the value of any function represented by a monoid combining values at the edges of D (for example the length, number of edges, or heaviest edge in a path). We can list all paths with length less than a given bound in the same time bound above, and we can list the paths in order taking time $O(\log i)$ to list the ith path.*

By applying this to the graphs D_1 and D_2 constructed earlier, we get the following results.

Theorem 1. *Given a DAG D, in $O(mn)$ time we can construct a data structure such that, for any two nodes u and v in D, we can list an implicit representation of the k shortest pairs of vertex-disjoint paths from a common ancestor to u and v, in time $O(k)$. The same bound holds for listing all pairs with length less than a given bound (where k is the number of such paths). Alternately, the pairs of paths can be output in order by total length, in time $O(\log i)$ to list the ith pair. We can find each pair of paths explicitly*

from its implicit representation in time proportional to the number of edges in the pair. Our representation also allows computation of some simple functions (such as the length or number of edges of each path in the pair) in in constant time per pair of paths.

Theorem 2. *Given any d pairs (s_i, t_i) and (s_2, t_2) of terminals in a DAG, we can find in time $O(mn^{d-1}+k)$ an implicit representation of the k shortest d-tuples of vertex-disjoint paths connecting those terminals. As above, the tuples can be output in order in time $O(\log i)$ for the ith path, we can compute explicit representations in linear time, and we can compute simple functions on the paths in constant time.*

For our other results on these problems, we count paths using a standard dynamic programming technique in acyclic graphs.

Lemma 4. *Let D be a DAG with a specified vertex s. Then in $O(m+n)$ arithmetic operations we can count all paths from s to each other vertex in D. Each operation involves integers with at most $\log_2 x$ bits, where x is the maximum number of paths from s to any other vertex.*

Proof: We process the vertices in order by a topological numbering. For each vertex the number of paths from s is simply the sum of the corresponding numbers for its immediate predecessors. □

Again, we apply this lemma to the DAGs D_1 and D_2 constructed earlier. Note that (assuming D has no multiple edges or self-loops) the number of paths in our original DAG D is at most 2^n, since any path can be represented uniquely by a subsequence of some fixed topological ordering of D. Hence the number of paths in D_1 or D_2 is at most 2^{2n} or 2^{dn} respectively.

Theorem 3. *Given a DAG, in $O(mn)$ arithmetic operations we can count, for each two nodes in the graph, the number of pairs of disjoint paths from a common ancestor to those two nodes. Each operation involves arithmetic on numbers of $O(n)$ bits.*

Theorem 4. *Given any d pairs (s_i, t_i) and (s_2, t_2) of terminals in a DAG D, In $O(mn^{d-1})$ arithmetic operations on $O(dn)$ bit numbers we can compute the number of tuples of vertex-disjoint paths from s_i to t_i.*

5 Alternate Methods

As discussed in the introduction, in practice the search for disjoint path pairs may be limited more by memory availability than time. The algorithms described earlier use quadratic space, which even after the pruning described in the previous section may be too much. We describe here methods requiring only linear space. However, to achieve this we must spend more computation time.

Lemma 5. *Given a DAG D with nonnegative edge lengths and a pair u, v of nodes, we can in time $O(m+n)$ find the shortest pair of vertex-disjoint paths from a common ancestor to u and v.*

Proof: Construct a DAG D' consisting of a copy of D itself, a second copy of D with all arcs reversed, and an arc from each node in the second copy to the corresponding node in the first copy. Then pairs of paths from an ancestor to two nodes u and v in D correspond one-for-one with paths from the second copy of u to the first copy of v in D'. This method does not constrain paths to be vertex-disjoint, but it is easily seen that the shortest pair of paths in D (corresponding to a shortest path in D') must be disjoint, since if any pair of paths share a vertex one can find a shorter pair starting from that shared vertex. □

One possible practical heuristic derived from this idea would be simply to search for short paths in this graph D', and filter out the ones corresponding to non-disjoint path pairs in D. In fact, if the method used to find short paths in D' is that of Byers and Waterman [1], this would resemble a more sophisticated version of the backtracking search already implemented in Gene. However the worst case time of such a heuristic would still be exponential; we are interested here in algorithms with polynomial worst case behavior and linear space.

We quickly describe such an algorithm, based on Lawler's [5] method of partitioning a solution space. First suppose that we know a bound ℓ on the length of the path pairs we wish to find, and simply intend to return all pairs shorter than ℓ. If we are given a bound k on the number of paths to find, the actual value of ℓ can be found by binary search, multiplying the time by a factor of $O(\log \ell)$, or by increasing ℓ at each step to the next larger value found in the previous search, instead multiplying the time by $O(k)$. (The second approach is essentially the same as the standard method of depth first iterative deepening search [4].)

We then use the lemma above to find the best pair in D. If this is already worse than ℓ, we are done. And if $u = v$, there can only be the trivial pair of paths (no others are disjoint). Otherwise, we let (u, w) be the first edge in one of the paths, and call the algorithm recursively twice: once to find all path pairs connecting w and v within distance at most $\ell - d(u, w)$, in graph $D - u$, and secondly to find all path pairs connecting u and v within distance ℓ in graph $D - (u, w)$.

Theorem 5. *The method above generates all disjoint pairs of paths to u and v, of length at most ℓ, in time $O(mnk)$ and space $O(m + n)$.*

Proof: The space bound is clear. The time bound follows since each output path pair causes $O(n)$ recursive calls, each taking linear time. □

6 Pruning the Input

As one further step towards practicality, we show how to determine those vertices that are actually part of some pair of disjoint paths. The remaining vertices can then be removed from the graph, speeding up the construction of the k shortest disjoint path pairs in practice if not necessarily in theory.

Given a DAG D with nonnegative edge lengths and a pair u, v of terminal nodes, we first determine for each other vertex w which of the two terminals can be reached by paths from w. Obviously, we can eliminate all vertices not able to reach one or the other terminal. From now on we assume that all nodes can reach at least one terminal.

Next, we construct a bottom-up topological numbering $N(w)$ of the vertices of D, so that the children of any node have smaller numbers than the node itself. By abuse of notation we identify $N(w)$ with w itself. Define another number $R(w)$ recursively as follows. First, if w can only reach u, let $R(w) = u$, or if w can only reach v, let $R(w) = v$. Second, if all children of w have the same value of R, let $R(w)$ be that same value. Finally, if the children of w have more than one value of R, but w itself can reach both u and v, let $R(w) = w$.

Lemma 6. *$R(w)$ is the topological number of the lowest node through which all paths from w to u and v go.*

Proof: First, suppose there are two such nodes. Then there is a path between them (any path from w to u or v) so the lowest of the two is well

defined. If w can only reach one terminal, say u, $R(w) = u$ and the result is clear. If all children of w have the same value of R, the result clearly follows by induction: all paths from w to a terminal must go through a child, and hence through R; but for any node lower than R we can get from w to a child of w and from there to a terminal by a path avoiding that node. Finally, suppose two children x and y have $R(x) \neq R(y)$. Then we wish to show that for any node $z \neq w$, there is a path from w to a terminal avoiding z. Suppose not; then all paths from both x and y go through z. But then all such paths would also go through $R(z)$, and there would be paths from x and y through z and $R(z)$ but avoiding any lower node, so $R(x) = R(y) = R(z)$ contradicting the assumption that $R(x) \neq R(y)$. □

Lemma 7. *There is a pair of disjoint paths from w to u and v if and only if $R(w) = w$ and w can reach both terminals.*

Proof: Any pair of paths to both terminals must both contain $R(w)$, so if $R(w) \neq w$ the paths could not be disjoint. In the other direction, choose some pair of paths p_1 and p_2 from w that coincide for some number k of edges, and are disjoint below that point, with k chosen as small as possible. (Certainly some such pair exists, since from any pair of paths from w to each terminal we can simply choose the lowest point in common, and use a common path above that point.) If $k > 0$, let x be the point where p_1 and p_2 diverge. Then by the previous lemma, there is a path p_3 from w to some terminal (say u) that avoids x. Since it avoids x, it must have a minimal subpath connecting some vertex on the common portion of p_1 and p_2 to another vertex on one of the two paths below x (say on p_1). By replacing a portion of p_1 with this subpath, we find a pair of paths with a smaller value of k; but k was chosen as small as possible, therefore $k = 0$. □

Lemma 8. *A node w is part of some pair of disjoint paths to u and v if and only if it can reach u or v and some ancestor w' of w has $R(w') = w'$.*

Proof: These conditions are clearly necessary. The proof of the other direction is similar to the previous lemma. Suppose $R(w) \neq w$ (else the previous lemma applies) and choose some pair of paths p_1 and p_2 from some ancestor w' with $R(w') = w'$ that pass through w, coincide for some number k of edges, and are disjoint below that point, with k chosen as small as possible. If w is on the disjoint portion of the paths, we are done. Otherwise, let x be the point where p_1 and p_2 diverge; there is a path p_3

from w' to some terminal that avoids x. By splicing a minimal subpath of p_3 into one path we get a pair of paths with a smaller value of k; but the other path is unchanged and still contains w. Thus again we can always reduce k, proving that it must be zero. □

Theorem 6. *In $O(m+n)$ time we can find the set of nodes in D that are part of some pair of disjoint paths to u and v.*

Proof: The topological sorting of D and the computation of R can clearly be done in linear time, as can the determination of the set of descendants of nodes with $R(w) = w$. The desired set of nodes can then be identified according to the lemmas above. □

References

[1] T. H. Byers and M. S. Waterman. Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Oper. Res.* 32, 1984, pp. 1381–1384.

[2] D. Eppstein. Finding the k shortest paths. 35th IEEE Symp. Foundations of Computer Science, 1994, pp. 154–165.

[3] D. Eppstein. Gene 4.1 User Guide. HTML document, 1995, available online at http://www.ics.uci.edu/∼eppstein/gene/UserGuide.html.

[4] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence* 27, 1985, pp. 97–109.

[5] E. L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science* 18, 1972, pp. 401–405.

[6] C.-L. Li, S. T. McCormick, and D. Simchi-Levi. The complexity of finding two disjoint paths with min-max objective function. *Discrete Applied Math.* 26, 1990, pp. 105–115.

[7] Y. Perl and Y. Shiloach. Finding two disjoint paths between two pairs of vertices in a graph. *J. ACM* 25, 1978, pp. 1–9.

[8] J. Vygen. NP-completeness of some edge-disjoint paths problems. *Discrete Applied Math.* 61, 1995, pp. 83–90.