



The University of Sydney

**Efficient Shortest Path Algorithms
By Graph Decomposition**

Technical Report Number 475

January, 1994

Diab Abuaiadh and Jeffrey H Kingston

ISBN 0 86758 901 9

**Basser Department of Computer Science
University of Sydney NSW 2006**

Efficient shortest path algorithms by graph decomposition

Diab Abuaiadh and Jeffrey H. Kingston

ABSTRACT

This paper introduces a divide-and-conquer approach to the single-source shortest path problem. For an arbitrary digraph with n vertices, m edges, and c cycles, a particular division is exhibited which leads to an $O(k \log k + m)$ algorithm, where $k = \min(n, c)$, improving on previous methods for near-acyclic digraphs.

4 January, 1994

Efficient shortest path algorithms by graph decomposition

Diab Abuaiadh and Jeffrey H. Kingston

1. Introduction

The single source shortest path problem is well known. Let $G = (V, E)$ be a directed graph with a distinguished *source vertex* s . Each edge (v, w) has a non-negative integer *cost* $c(v, w)$. The cost of a path in G is the sum of the costs of its edges, and a shortest path from vertex v to vertex w is a path from v to w of minimal cost. We assume the existence of at least one path from s to every vertex; thus $m \geq n - 1$, where n and m are the number of vertices and edges respectively. The single source shortest path problem is to find a shortest path from s to v for every vertex v in G .

If no restriction is placed on the form of the digraph, the algorithm due to Dijkstra [3] is the most efficient known for this problem. The literature contains a sequence of increasingly efficient implementations of Dijkstra's algorithm [1, 7, 10], culminating in the $O(n \log n + m)$ implementation by Fredman and Tarjan using Fibonacci heaps [5]. An alternative data structure by Peterson [9] later achieved the same result. Since Dijkstra's algorithm sorts the vertices into order of increasing distance from the source vertex, and since it examines every edge, no further asymptotic improvement is possible under the decision tree model.

Further improvement within the decision tree model therefore requires a change of algorithm. Efficient alternatives do exist for restricted classes of digraphs: for example, $O(n + m)$ for acyclic digraphs [10], and $O(n\sqrt{\log n})$ for planar graphs [4].

This paper introduces a divide-and-conquer approach to the single-source shortest path problem. For an arbitrary digraph with n vertices, m edges, and c cycles, a particular application of this approach is exhibited which yields an $O(k \log k + m)$ algorithm, where $k = \min(n, c)$, improving on previous methods for near-acyclic digraphs [2].

2. Edge-disjoint partitionings

Let $G = (V, E)$ be an arbitrary directed graph. We divide G by choosing r pairwise disjoint subsets of V called V_1, V_2, \dots, V_r under the restriction that no edge (v_i, v_j) may exist with $v_i \in V_i$ and $v_j \in V_j$ such that $i \neq j$. Such a division will be called an *edge-disjoint partitioning of G* (Figure 1).

Let G_i be the subgraph induced by V_i (that is, $G_i = (V_i, E_i)$ where E_i contains those edges of G with both endpoints in V_i). For convenience we colour the vertices lying within any induced subgraph black, and the remaining vertices red. For any black vertex v we let $F(v)$ be the subgraph G_i in which v lies.

Let $I(G_i)$ be the set of incoming edges of G_i (that is, the edges (v, w) such that $v \notin G_i$ and $w \in G_i$), and let $i(G_i) = |I(G_i)|$. Similarly, let $O(G_i)$ be the set of outgoing edges of G_i , with cardinality $o(G_i)$.

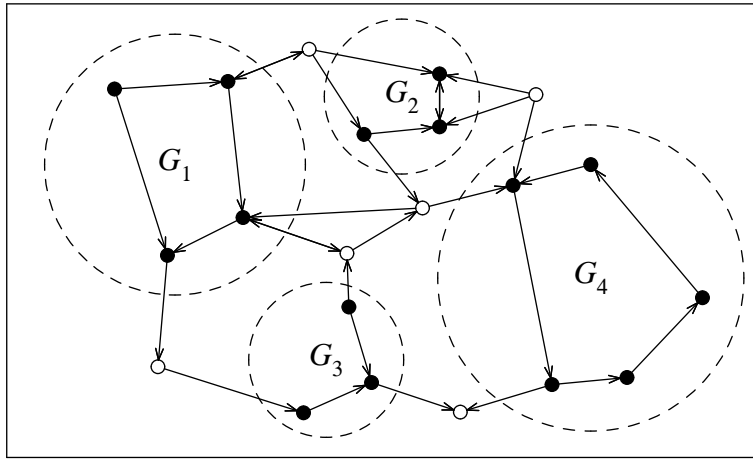


Figure 1. An edge-disjoint partitioning into four subgraphs. Black vertices are shown as filled circles, red vertices as open circles.

3. Shortest paths in partitioned graphs

In this section we present an algorithm for shortest paths in a digraph with an edge-disjoint partitioning G_1, \dots, G_r . For each G_i we will assume the existence of an algorithm $FSP(G_i)$ which solves the shortest path problem on G_i and also updates the red vertices at the ends of the outgoing edges of G_i as appropriate. If the construction of G_i forces G_i to have a special form (e.g. to be acyclic), then $FSP(G_i)$ may take advantage of this knowledge to run efficiently.

We always assume that each vertex v is initially assigned some arbitrary distance $d(v)$ from a mythical source vertex, which must be decreased to its final value by the algorithm. A shortest path in this setting will be a path v_1, v_2, \dots, v_k in which v_1 retains its initial distance and $d(v_i) + c(v_i, v_{i+1}) = d(v_{i+1})$ for $1 \leq i \leq k - 1$, such that $d(v_k)$ is minimal. The single-source shortest path problem is the special case in which the source vertex is initialized with distance 0 and the others with infinity.

The algorithm is presented in Figure 2. We first prove its correctness.

Theorem 1. The algorithm of Figure 2 assigns to each vertex v the cost $d(v)$ of a shortest path to v .

Proof. For each vertex v_k , let $SP(v_k) = v_1, \dots, v_{k-1}, v_k$ be any shortest path to v_k . The proof is by induction on the length of $SP(v_k)$.

The basis, $k = 1$, is immediate since v_k retains its initial cost. So we let $k > 1$ and assume that v_1, \dots, v_k are assigned the cost of a shortest path by the algorithm, and we prove this for v_{k+1} .

If all of v_1, \dots, v_k are black, then v_{k+1} will be assigned the cost of its shortest path during Stage 2. So we assume in the sequel that v_j is a red vertex in $SP(v_{k+1})$ such that j is maximal.

Suppose first that v_{k+1} is black. It is clear that $F(v_{j+1}) = F(v_{k+1})$ and that $FSP(F(v_{j+1}))$ will be invoked immediately after v_j is deleted from the heap. Since v_j must have been assigned the cost of its shortest path by then, $FSP(F(v_{j+1}))$ will assign to v_{k+1} the cost of its shortest path in case $j < k$, and the algorithm will assign to v_{k+1} the cost of its shortest path in case $j = k$.

Otherwise, v_{k+1} is red. We must delete v_j from the priority queue before v_{k+1} , since otherwise

```

Stage 1:
  Build  $G_1, \dots, G_r$ ;
Stage 2:
  for  $i := 1$  to  $r$  do  $FSP(G_i)$ ;
Stage 3:
  Initialize( $Q$ );
  for all red vertices  $v$  do  $Insert(New(d(v), v), Q)$ ;
  while not  $Empty(Q)$  do begin
     $v := ValueOf(DeleteMin(Q))$ ;
    for all  $x$  adjacent to  $v$  do begin
      if  $d(v) + c(v, x) < d(x)$  then begin
         $d(x) := d(v) + c(v, x)$ ;
        if  $x$  is black then  $FSP(F(x))$ 
        else (*  $x$  is red and must lie in  $Q$  *)
           $DecreaseKey(EntryOf(x), d(v) + c(v, x), Q)$ ;
      end;
    end;
  end;
end;

```

Figure 2. Shortest path algorithm for a digraph with an edge-disjoint partitioning. Q is a priority queue of entries whose keys are integers and whose values are vertices; $New(key, value)$ returns a new entry with the given key and value; $ValueOf(entry)$ returns the value field of an entry; and $EntryOf(v)$ returns the entry in Q corresponding to vertex v .

there is a shorter path to v_{k+1} than via v_j . When v_j is deleted, v_{k+1} will have the cost of its shortest path by $FSP(F(v_{j+1}))$ or by the algorithm in case $j = k$. \square

For the time complexity analysis, let us assume that we can build G_1, \dots, G_r in time $B(n, m)$. For each G_i we execute $FSP(G_i)$ exactly $i(G_i) + 1$ times. Let $T(G_i)$ be the worst-case time complexity of $FSP(G_i)$ excluding the cost of updating the endpoints of the outgoing edges. Let $h(k)$ be the total cost of all priority queue operations, where $k \leq n$ is the number of red vertices. Then the worst-case time complexity of the algorithm is

$$W(n, m) = B(n, m) + h(k) + \sum_{i=1}^r (i(G_i) + 1)[T(G_i) + o(G_i)]$$

Our aim is to find an edge-disjoint partitioning such that

$$B(n, m) + \sum_{i=1}^r (i(G_i) + 1)[T(G_i) + o(G_i)] = O(n + m)$$

with k relatively small.

4. Application to near-acyclic digraphs

In this section we present an $O(k \log k + m)$ algorithm for the shortest path problem, where $k \leq \min(n, m - n + 1)$. This is superior to previous algorithms when the digraph is very sparse.

The first step is to perform the well-known factorization of G into strongly connected components with acyclic quotient digraph [8]. It then suffices to take these components in topological order and solve the shortest path problem on each in turn, including updating the distances of the vertices at the ends of edges leading out of the component. Calculations on subsequent components may then ignore these edges completely, and so from now on we assume that G is strongly connected.

We next find an edge-disjoint partitioning of G in linear time as follows. The reader is invited to verify that in every case the black-coloured subgraphs are acyclic and every red vertex (except possibly one vertex) has indegree at least 2. These are the essential points for our subsequent analysis.

Case 1. Every vertex has indegree at least 2. Then we colour every vertex red.

Case 2. Every vertex has indegree 1. Then G consists of one simple cycle. We choose an arbitrary vertex and colour it red, then colour the remaining vertices black.

Case 3. Neither (a) nor (b) applies. Since G is strongly connected, every vertex has indegree at least 1, so in this case we can find an edge (v_1, w_1) such that v_1 has indegree at least 2, and w_1 has indegree 1. An edge with these properties will be called a *starting edge*. We now consider this case in detail.

We find an acyclic subgraph S such that w_1 is an ancestor of every vertex in S . This is easily done using the usual topological sorting procedure, by first including w_1 , then including other vertices only when all their predecessors have been included.

While performing this topological sort it is possible that we might decide that v_1 may be included. If this occurs, we claim that every vertex in G lies in S . For suppose on the contrary that there is a vertex $x \in G$ such that $x \notin S$. Now G is strongly connected, so there is a path x, \dots, v_1 . Since this path begins outside S and ends at v_1 in S , it must contain an edge (x_i, x_{i+1}) such that $x_i \notin S$ and $x_{i+1} \in S$, and this edge cannot be (v_1, w_1) , contradicting the construction of S . In this case we colour v_1 red and all the other vertices black.

In the more usual case where v_1 is not encountered before the topological sort exhausts its set of possible vertices to include, we colour all the included vertices black and v_1 red. (Alternatively, with care our algorithm can be optimized at this point by leaving v_1 uncoloured.) Then we continue the search for another starting edge (v_2, w_2) and repeat the process. We do not include any vertex already coloured red, and we cannot encounter a black vertex; thus the partitioning is edge-disjoint. Any vertices remaining uncoloured at the end are coloured red.

Since each subgraph G_i is acyclic with just one incoming edge, it is clear that

$$B(n, m) + \sum_{i=1}^r (i(G_i) + 1)[T(G_i) + o(G_i)] = O(n + m)$$

as desired. Now let r_1 be the number of vertices with indegree 1, and let r_2 be the number with indegree 2 or more. Counting vertices gives $r_1 + r_2 = n$, and counting endpoints of edges gives $r_1 + 2r_2 \leq m$. Subtraction gives $r_2 \leq m - n$, and since every red vertex except possibly one has

indegree at least 2, it follows that k , the number of red vertices, satisfies $k \leq r_2 + 1 \leq m - n + 1$. If we implement the priority queue using a Fibonacci heap, the total cost of all priority queue operations will be $O(k \log k + m)$, giving our desired time complexity bound.

In practice the efficiency of the algorithm depends in the cycle structure of the strongly connected components. In fact $m - n + 1$, the *cyclomatic number* [6], is a reasonable measure of the number of cycles in a strongly connected digraph, so it is reasonable to say that k is bounded by the number of cycles in G . Also, our algorithm is $O(m + n)$ and thus asymptotically optimal whenever m exceeds n by any fixed constant.

References

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Abuaiadh, Diab and Jeffrey H. Kingston, *An efficient algorithm for the shortest path problem*. Tech. Rep. TR93-473 (1993), Basser Department of Computer Science F09, The University of Sydney 2006, Australia.
3. Dijkstra, E.W., A note on two problems in connexion with graphs. *Numerische Mathematik* **1**, 269-271 (1959).
4. Greg N. Frederickson, Fast algorithm for shortest path in planar graph, with applications. *SIAM Journal on Computing* **16**, 1004-1022 (1987).
5. Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34**, 569-615 (1987).
6. Gondran M. and Minoux M., *Graphs and Algorithms*. John Wiley & Sons Ltd, 1984.
7. D.B. Johnson, Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM* **24**, 1-13 (1977).
8. Kingston, Jeffrey H., *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990.
9. Gary L. Peterson, *A balanced tree scheme for meldable heaps with updates*. Tech. Rep. GIT-ICS-87-23 (1987), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332..
10. Tarjan, R.E., *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.