

Shortest Path Algorithms for Nearly Acyclic Directed Graphs

Tadao TAKAOKA

Department of Computer Science

University of Canterbury

Christchurch, New Zealand

E-mail: tad@cosc.canterbury.ac.nz

Phone: +64 3 364 2362

Fax: +64 3 364 2569

September, 1996, revised October, 1997

Abstract

Abuaiadh and Kingston gave an efficient algorithm for the single source shortest path problem for a nearly acyclic graph with $O(m + n \log t)$ computing time, where m and n are the numbers of edges and vertices of the given directed graph and t is the number of delete-min operations in the priority queue manipulation. They use the Fibonacci heap for the priority queue. If the graph is acyclic, we have $t = 0$ and the time complexity becomes $O(m + n)$ which is linear and optimal. They claim that if the graph is nearly acyclic, t is expected to be small and the algorithm runs fast.

In the present paper, we take another definition of acyclicity. The degree of cyclicity, $cyc(G)$, of graph G is defined by the maximum cardinality of the strongly connected components of G . When $cyc(G) = k$, we give an algorithm for the single source problem with $O(m + n \log k)$ time complexity. Finally we give a hybrid algorithm that incorporates the merits of the above two algorithms.

Keywords: Algorithm, shortest paths, acyclic graph, nearly acyclic graph, priority queue, Fibonacci heap, Dijkstra's algorithm

1 Introduction

Since the classical algorithms for shortest paths were published by Dijkstra [3] for the single source problem and Floyd [4] for the all pairs problem, there have been many variations and improvements in either analysis or special classes of the given graphs. To name just a few, Moffat and Takaoka [7] gave an $O(n^2 \log n)$ expected time algorithm for the all pairs shortest path (APSP) problem, which is a considerable improvement from $O(n^3)$ of Floyd, where n is the number of vertices of the given graph. Fredman and Tarjan [6] gave an $O(m + n \log n)$ time algorithm for the single source problem using the data-structure, called a Fibonacci heap, where m is the number of edges of the given graph. This is an improvement of $O(n^2)$ of Dijkstra, when m is small. If the given graph is planar, Frederickson [5] gave an $O(n\sqrt{\log n})$ time algorithm for the single source problem and $O(n^2)$ time algorithm for the APSP problem. If the given graph is nearly tree, Chaudhuri and Zaroligas [2] gave an algorithm for the APSP problem with $O(t^4 n)$ time where t is the tree width of the graph.

Recently Abuaiadh and Kingston [1] gave an interesting result by restricting the given graph to being nearly acyclic. When they solve the single source problem, they distinguish between two kinds of vertices in $V - S$ where V is the set of vertices and S is the solution set, that is, the set of vertices to which the shortest distances from the source have been established by the algorithm. One is the set of vertices, “easy” ones, to which there are no edges from $V - S$, e.g., only edges from S . The other is the set of vertices, “difficult” ones, to which there are edges from $V - S$. To expand S , if there are easy vertices, those are included in S and distances to other vertices in $V - S$ are updated. If there are no easy vertices, the vertex with minimum tentative distance is chosen to be included in S . If the number of such delete-minimum operations is t , the authors show that the single source problem can be solved in $O(m + n \log t)$ time with use of a Fibonacci heap. If the graph is acyclic, $t = 0$ and we have $O(m + n)$ time. Since we have $O(m + n \log n)$ when $t = n$, the result is an improvement of Fredman and Tarjan with use of the new parameter t . The authors claim that if the given graph is nearly acyclic, t is expected to be small and thus we can have a speed up.

The definition of near acyclicity and the estimate of t under it is not clear, however. Take up an example graph $G = (V, E)$ such that $V = \{v_1, v_2, \dots, v_n\}$ for even n and E is defined by

$$E = \{(v_i, v_{i+1}) \mid i = 1, 2, \dots, n-1\} \cup \{(v_i, v_{i-1}) \mid i = 2, 4, \dots, n\} \\ \cup \{(v_1, v_i) \mid i = 3, 4, \dots, n\}.$$

We give non-negative real numbers as edge costs to edges in such a way that v_1, \dots, v_n are included in S in this order. Then the graph is nearly acyclic in the sense given below, but $t = n/2$ and hence the complexity is $O(m + n \log n)$.

In the present paper, we give a new definition of near acyclicity by the maximum size k of strongly connected components of the given directed graph. Under this definition, we give an $O(m + k^2n)$ time algorithm for the single source problem and an $O(mn + kn^2)$ time algorithm for the APSP problem in Section 2. Specifically for the above example graph our time is $O(m + n)$, that is linear, whereas the time by [1] is $O(m + n \log n)$. On the other hand, the efficiency of our algorithm worsens for a circular graph with $E = \{(v_i, v_{i+1}) \mid i = 1, \dots, n-1\} \cup \{(v_n, v_1)\}$ since k becomes n , whereas the algorithm in [1] performs well since $t = 0$. In Section 4, we improve the above results by establishing $O(m + n \log k)$ time for the single source problem and $(mn + n^2 \log k)$ time for the APSP problem. In Section 5, we give a hybrid algorithm that inherits merits from the algorithms in this paper and in [1].

2 Simple algorithms

Let $G = (V, E)$ be a directed graph where $V = \{v_1, \dots, v_n\}$ and $E \subseteq V \times V$. The non-negative cost of edge (v_i, v_j) is denoted by $c(v_i, v_j)$. Let Tarjan's algorithm [8] compute strongly connected components (sc-components) V_r, V_{r-1}, \dots, V_1 in this order. Let graph $\tilde{G}(\tilde{V}, \tilde{E})$ be defined by $\tilde{V} = \{V_1, \dots, V_r\}$ and \tilde{E} , where there is an edge from V_i to V_j in \tilde{E} if there is an edge (v, w) in E such that $v \in V_i$ and $w \in V_j$. That is, \tilde{G} is the degenerated acyclic graph of G such that V_i 's are degenerated into single vertices and edges from V_i to V_j are degenerated into a single edge. The set $\tilde{V} = \{V_1, \dots, V_r\}$ is

topologically sorted in this order.

Let the graph $G_i = (V_i, E_i)$ be defined by $E_i = \{(v, w) \mid v \in V_i \text{ and } w \in V_i\}$. We solve the all pairs shortest path problem for each G_i . Let $D(v, w)$ be the shortest distance from v to w in G_i . Using this information, we solve the single source problem from source $v_0 \in V_1$ to all other vertices along the degenerated edges. We start with an algorithm for the simpler case of an acyclic graph.

Algorithm 1 $\{G = (V, E)$ is an acyclic graph. $\}$

```

1 Topologically sort  $V$  and assume without loss of generality  $V = \{v_1, \dots, v_n\}$ 
   where  $(v_i, v_j) \in E \Leftrightarrow i < j$ ;
2  $d[v_1] := 0$ ;  $\{v_1$  is the source $\}$ 
3 for  $i := 2$  to  $n$  do  $d[v_i] := \infty$ ;
4 for  $i := 1$  to  $n$  do
5   for  $v_j$  such that  $(v_i, v_j) \in E$  do
6      $d[v_j] := \min\{d[v_j], d[v_i] + c(v_i, v_j)\}$ .
```

We expand the above algorithm to Algorithm 2 with line numbers expanded with dots.

Algorithm 2 $\{\text{Solve the single source problem for graph } G = (V, E) \text{ and source } v_0\}$

```

1.1 Compute sc-components  $V_r, V_{r-1}, \dots, V_1$ 
1.2 Solve the APSP problem for  $G_1, G_2, \dots, G_r$ ;  $\{D$  computed $\}$ 
2.1 for  $v \in V_1$  do  $d[v] := \infty$ ;
2.2  $d[v_0] := 0$ ;  $\{\text{For source } v_0 \text{ let } v_0 \in V_1 \text{ without loss of generality}\}$ 
3   for  $i := 2$  to  $r$  do for  $v \in V_i$  do  $d[v] := \infty$ ;
4.1 for  $i := 1$  to  $r$  do begin
4.2   for  $v \in V_i$  do for  $w \in V_i$  do
4.3      $d[w] := \min\{d[w], d[v] + D[v, w]\}$ ;
5   for  $V_j$  such that  $(V_i, V_j) \in \tilde{E}$  do
6.1     for  $v \in V_i$  and  $w \in V_j$  such that  $(v, w) \in E$  do
6.2        $d[w] := \min\{d[w], d[v] + c(v, w)\}$ 
7   end.
```

Lines 4.2 and 4.3 are to obtain shortest distances within sc-components whereas lines 6.1 and 6.2 are to update distances through edges between sc-components.

Lemma 1 *At the beginning of Line 5 in Algorithm 1, the shortest distances from v_1 to v_j ($j \leq i$) are computed. Also at the beginning of line 5, distances computed in $d[v_j]$ ($j \geq i$) are those of shortest paths that lie in $\{v_1, \dots, v_{i-1}\}$ except for v_j .*

Proof. By induction. When $i = 1$, $d[v_1] = 0$ and the set of v_1, \dots, v_{i-1} is empty, and thus the lemma trivially holds. Assume the lemma is true for i . Since the paths from v_1 to v_i only go through the set $\{v_1, \dots, v_{i-1}\}$, the shortest distance to v_i is already computed in $d[v_i]$. At lines 5 and 6, the shortest distances to v_j ($j > i$) are updated through v_i . □

Lemma 2 *At the beginning of line 5 in Algorithm 2, the shortest distances from v_0 to $v \in V_j$ ($j \leq i$) are computed. Also at the beginning of line 5, the distances computed in $d[v]$ for $v \in V_j$ ($j \geq i$) are those of shortest paths that lie in $V_i \cup \dots \cup V_{i-1}$ except for vertices in V_j .*

Proof. Similar to the proof of Lemma 1. When $i = 1$, the shortest distances from v_0 to $v \in V_1$ are computed at lines 4.2 and 4.3. The second statement of the lemma is true since $V_1 \cup \dots \cup V_{i-1}$ is empty.

Assume the lemma is true for i . Since the paths from v_0 to $v \in V_i$ only go through the set $V_1 \cup \dots \cup V_{i-1} \cup V_i$ and the shortest distances to $v \in V_i$ through $V_1 \cup \dots \cup V_{i-1}$ are computed, the shortest distances to $v \in V_i$ are computed at lines 4.2 and 4.3. At lines 5, 6.1 and 6.2, the shortest distances to $w \in V_j$ ($j > i$) are updated through $v \in V_i$. □

The all pairs version of Algorithm 1 is to compute shortest distances by changing the source from v_1 to v_{n-1} . The all pairs version of Algorithm 2 is similar. We change v_0 in V_1 and solve the single path problems. Then we take V_2 and choose all v_0 in V_2 and so on.

3 Analysis

To analyze the algorithm, we first establish the following lemma.

Lemma 3 *Let non-negative integer variables x_1, x_2, \dots, x_n satisfy the following conditions for constant integers k and x such that $0 \leq k \leq x$ and $x \leq kn$.*

- (1) $x_1 + x_2 + \dots + x_n = x$
- (2) $x_i \leq k$ ($i = 1, \dots, n$).

Also let the maximum of the objective function $\sum_{i=1}^n f(x_i)$ be denoted by $\phi_n(x)$ where $f(x)$ is such that $f(x) = xg(x)$ and $g(x)$ is a monotone non-decreasing function. Then we have $\phi_n(x) \leq ng(k)$.

Proof.

$$\phi_n(x) = \sum_{i=1}^n k_i g(x_i) \leq \sum_{i=1}^n k_i g(k) = ng(k)$$

The value of $\phi_n(x)$ is obtained by setting as many x_i 's as possible to k . □

Now we analyze the algorithms. The computing time of Algorithm 1 is obviously $O(m + n)$. Its all pairs version takes $O(mn + n^2)$ time.

At line 1.2 of Algorithm 2, we use Floyd's algorithm. Then the time becomes $O(k_1^3 + \dots + k_r^3)$ where k_i ($1 \leq i \leq r$) is the size of V_i . Let us assume that $k_i \leq k$ ($i = 1, \dots, r$). From Lemma 3, we see that $O(k_1^3 + \dots + k_r^3) \leq O(k^2n)$, since $k_1 + \dots + k_r = n$ and x^3 satisfies the condition for $f(x)$ in Lemma 3. The overall time for lines 4.2 and 4.3 is $O(k_1^2 + \dots + k_r^2)$, which is $O(kn)$ from Lemma 3. The overall time for lines 5, 6.1 and 6.2 is $O(m)$. Hence the total time is $O(m + k^2n)$.

When we apply Algorithm 2 to n sources, we note that we can perform lines 1.1 and 1.2 only once. Thus the total time becomes $O(n(m + kn)) = O(mn + kn^2)$, since $k^2n \leq kn^2$. To summarize, we have the following definition and theorem.

Definition 1 The degree of cyclicity of graph G , denoted by $cyc(G)$, is defined to be the maximum cardinality of the strongly connected components of G .

Theorem 1 *Let $k = cyc(G)$. Then we can solve the single source problem and the APSP problem for G in $O(m + k^2n)$ time and $O(mn + kn^2)$ time respectively.*

We can say that the given directed graph is nearly acyclic, if $cyc(G)$ is small.

4 More efficient algorithms

In this section we improve Algorithm 2 by not solving APSP problems for G_1, G_2, \dots, G_r . We use a modified version of Fredman and Tarjan's algorithm [6] for the single source problem. Here we generalize the single source shortest path problem in the following way. We omit "shortest path" for simplicity.

Definition 2 The generalized single source (GSS) problem for a directed graph $G = (V, E)$ with the non-negative cost function $c(v, w)$ for edge (v, w) and the initial distances $d_0[v] \geq 0$ for $v \in V$ is to compute the shortest distances $d[w]$ for all $w \in V$. The shortest distance $d[w]$ is defined by

$$d[w] = \min_v \{d_0[v] + D(v, w)\},$$

where $D[v, w]$ is the shortest distance from v to w . The conventional single source problem has $d_0[v_1]=0$ and $d_0[v] = \infty$ for all other $v \in V$.

To solve the GSS problem, we have the following algorithm in which we do not actually compute D .

Algorithm 3

- 1 **for** $v \in V$ **do** $d[v] := d_0[v]$;
- 2 Organize V in a priority queue Q with $d[v]$ as key;
- 3 $S := \emptyset$;
- 4 **while** $S \neq V$ **do begin**
- 5 Find v from Q with minimum key and delete v from Q ;
- 6 $S := S \cup \{v\}$;
- 7 **for** $w \in V - S$ **do begin**

```

8      $d[w] := \min\{d[w], d[v] + c(v, w)\};$ 
9     Reorganize  $Q$  with new  $d[w]$ ;
10  end
11  end.

```

The correctness of this algorithm follows from that of Dijkstra's algorithm if we attach a hypothetical source vertex v_0 and edges (v_0, v) with costs $d_0[v]$. If we use a Fibonacci heap, we can solve the GSS problem in $O(m + n \log n)$ time. Note that $O(n)$ time for make-heap is absorbed in the main complexity.

Now we use Algorithm 3 for lines 4.2 and 4.3 in Algorithm 2. Then the computing time becomes $O(m + \Sigma(m_i + k_i \log k_i))$. Since $x \log x$ satisfies the condition for $f(x)$ in Lemma 3, we have the computing time given by $O(m + n \log k)$. For the APSP problem we can use this new version n times. To summarize we have Theorem 2.

Theorem 2 *The single source and APSP problems for $G = (V, E)$ can be solved in $O(m + n \log k)$ time and $O(mn + n^2 \log k)$ time respectively where $k = \text{cyc}(G)$.*

5 Further improvement

Our algorithm in the previous section and that in [1] work well for different kinds of nearly acyclic graphs. They are, however, not incompatible. In place of Algorithm 3 used at lines 4.2 and 4.3 in Algorithm 2, we use the following algorithm, Algorithm 4, which is slightly modified from that used in [1] to adjust to the GSS. This way the two algorithms can compensate for each other. Let $\text{out}(v) = \{w \mid (v, w) \in E\}$.

Algorithm 4

```

1  for  $v \in V$  do  $d[v] := d_0[v]$ ;
2  Organize  $V$  in a priority queue  $Q$  with  $d[v]$  as key;
3   $S := \emptyset$ ;
4  while  $S \neq V$  do begin
5    if there is a vertex  $v$  in  $V - S$  with no incoming edge from  $V - S$  then
6      Choose  $v$ 

```

```

7  else
8      Choose  $v$  from  $V - S$  such that  $d[v]$  is minimum;
9      Delete  $v$  from  $Q$ ;
10      $S := S \cup \{v\}$  ;
11     for  $w \in out(v) \cap (V - S)$  do  $d[w] := \min\{d[w], d[v] + c(v, w)\}$ 
12 end.

```

The priority queue Q is slightly modified in [1] from the Fibonacci heap in such a way that there is no delete-min operation, and no pointer to the minimum element in Q . Only delete operation is defined and the minimum is found when the trees of equal rank are linked. It is shown in [1] that a sequence of n delete, m decrease-key and t find-min operations is processed in $O(m + n \log t)$ time. The readers are referred to [1] for details. Now we give the following final algorithm.

Algorithm 5

```

1  Compute sc-components  $V_r, V_{r-1}, \dots, V_1$ ;
2.1 for  $v \in V_1$  do  $d[v] := \infty$  ;
2.2  $d[v_0] := 0$  ; {Let  $v_0 \in V_1$  without loss of generality}
3  for  $i := 2$  to  $r$  do for  $v \in V_i$  do  $d[v] := \infty$ ;
4.1 for  $i := 1$  to  $r$  do begin
4.2   Use Algorithm 4 to solve the GSS for  $G_i$ ;
5     for  $V_j$  such that  $(V_i, V_j) \in \tilde{E}$  do
6.1     for  $v \in V_i$  do for  $w \in V_j$  do
6.2        $d[w] := \min\{d[w], d[v] + c(v, w)\}$ 
7  end.

```

Suppose we use Algorithm 4 for the whole graph with the initial condition that $d_0[v_0] = 0$ and $d_0[v] = \infty$ for $v \neq v_0$ and the number of v 's chosen at line 8 in (hypothetical) V_i is denoted by t_i . Denote also by t'_i the number of v 's chosen at line 8 of Algorithm 4 used at line 4.2 in Algorithm 5. Let us call these vertices "min-vertices." Although the orders in which vertices are included in S in the above two computations are different in

general, we can show that $t'_i \leq t_i$ since if min-vertices $v, w \in V_i$ are included in S in this order, i.e., v first, in the latter computation, then they are included in S in the same relative order in the former computation. Let $s = \max\{t'_i\}$ and $t = \sum t_i$. Then the time for Algorithm 5 is bounded by

$$O(m + \sum(m_i + k_i \log t'_i)) \leq O(m + n \log s).$$

Since $s \leq t$ and $s \leq k$, this algorithm is an improvement over those in the previous section and in [1]. Note that this algorithm runs in linear time for the two example graphs in Section 1.

6 Concluding Remarks

If we use a simple version of Dijkstra's algorithm where the priority queue is organized in a one-dimensional array, we have $O(m + kn)$ time for the single source problem where $k = \text{cyc}(G)$. When k is small, however, this version will be faster in practice.

When the degenerated acyclic graph \tilde{G} is a tree, we can solve the single source problem in $O(k^2n)$ time since $|\tilde{E}| = O(n)$ and hence $m = O(k^2n)$. Although this is optimal in terms of n , we conjecture that the complexity is smaller. Whether it is $O(kn)$ is open.

Since we no longer follow Dijkstra's thesis "Compute shortest paths from shorter to longer," it will be hard to obtain a lower bound for the problem in this paper, based on the lower bound on sorting. We conjecture, however, our algorithm in Section 4 is optimal for the single source problem with $k = \text{cyc}(G)$.

References

- [1] Abuaiadh, D. and J.H. Kingston, Are Fibonacci heaps optimal? ISAAC'94, LNCS, pp. 442–450 (1994).
- [2] Chaudhuri, S. and C.D. Zaroliagis, Shortest path queries in digraphs of small treewidth, Proc. of 22nd Inter. Colloq., ICALP95, Szeged, Hungary, July 1995, in Lecture Notes on Computer Science Vol. 944, pp. 244–255, Springer-Verlag (1995).

- [3] Dijkstra, E.W., A note on two problems in connection with graphs, Numer. Math. Vol. 1, pp. 269–271 (1959).
- [4] Floyd, R.W., Algorithm 97: Shortest path, CACM, Vol. 5, No. , p. 345 (1962).
- [5] Frederickson, G.N., Fast algorithms for shortest paths in planar graphs, with applications, SIAM Jour. Comp., Vol. 16, No. 6, pp. 1004–1022 (1987).
- [6] Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their use in improved network optimization problems, JACM, Vol. 34, No. 3, pp. 596–615 (1987).
- [7] Moffat, A. and T. Takaoka, An all pairs shortest path algorithm with expected time $O(n^2 \log n)$, SIAM Jour. Comp., Vol. 16, No. 6, pp. 1023–1031 (1987).
- [8] Tarjan, R.E., Depth first search and linear graph algorithms, SIAM Jour. Comp., Vol. 1, No. 2, pp. 146–160 (1972).
- [9] Tarjan, R.E., Data Structures and Network Algorithms, Regional Conference Series in Applied math. 44, 1983.