Electronic Notes in Theoretical Computer Science 42 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume42.html 17 pages

Improved Shortest Path Algorithms for Nearly Acyclic Graphs

Shane Saunders[1] and Tadao Takaoka[2]

Department of Computer Science
University of Canterbury, Christchurch, New Zealand

Abstract

Dijkstra's algorithm solves the single-source shortest path problem on any directed graph in $O(m+n\log n)$ time when a Fibonacci heap is used as the frontier set data structure. Here n is the number of vertices and m is the number of edges in the graph. If the graph is nearly acyclic, other algorithms can achieve a time complexity lower than that of Dijkstra's algorithm. Abuaiadh and Kingston gave an single source shortest path algorithm for nearly acyclic graphs with $O(m + n\log t)$ time complexity, where the new parameter, t, is the number of delete-min operations performed in priority queue manipulation. If the graph is nearly acyclic, then t is expected to be small, and the algorithm out-performs Dijkstra's algorithm. Takaoka, using a different definition for acyclicity, gave an algorithm with $O(m + n\log k)$ time complexity. In this algorithm, the new parameter, k, is the maximum cardinality of the strongly connected components in the graph.

This paper presents two new shortest path algorithms for nearly acyclic graphs. The first is a generalised single source (GSS) algorithm for nearly acyclic graphs, which has a time complexity of $O(m + r\log r)$, where r is the number of *trigger* vertices, with trigger vertices defined as roots of trees that result when the graph is decomposed into trees. The second is a new all-pairs algorithm for nearly acyclic graphs, with $O(mn + nr^2)$ worst case time complexity, where r is the number of vertices in a pre-calculated feedback vertex set for the nearly acyclic graph. For certain graphs, these new algorithms offer an improvement on the time complexity of the previous algorithms. The new GSS algorithm can be used in Takaoka's algorithm, giving an improved hybrid algorithm.

1 Introduction

Dijkstra's algorithm [2] is used as the basis for many shortest path algorithms, and can solve the single-source shortest path problem in $O(m+n\log n)$ worst

[1] E-mail: sas59@cosc.canterbury.ac.nz
[2] E-mail: tad@cosc.canterbury.ac.nz

case time if a Fibonacci heap [3] is used as the frontier set data structure. Here n is the number of vertices and m is the number of edges in the directed graph. Variations and improvements on Dijkstra's algorithm, have seen algorithms better suited to certain classes of graphs. These new algorithms improve the time complexity by introducing a parameter related to the graph structure. One such class of algorithms offer improvement for *nearly acyclic* graphs. Abuaiadh and Kingston [1] gave a single source shortest path algorithm for nearly acyclic graphs with $O(m + n \log t)$ time complexity, where the new parameter, t, is the number of delete-min operations performed in priority queue manipulation. If the graph is nearly acyclic, then t is expected to be small, and the algorithm out-performs Dijkstra's algorithm. For this algorithm, the definition of t is not directly related to the graph structure. Takaoka [4], using a different definition for acyclicity, gave an algorithm with $O(m + n \log k)$ time complexity. In this algorithm, the new parameter, k, is the maximum cardinality of the strongly connected components in the graph. The definition of k is directly related to the graph structure. Takaoka also gave a hybrid form of this new algorithm, which combined the new approach with that of Abuaiadh and Kingston.

These improved algorithms have shown that for nearly acyclic graphs, the number of delete-min operations performed in priority queue manipulation can be reduced. Using Dijkstra's algorithm to calculate the single-source shortest path problem will always involve n delete-min operations, giving a total time complexity of $O(m + n \log n)$. In contrast, the single-source shortest path problem over a directed acyclic graph with positive edge weights involves no delete-min operations, allowing a total time complexity of $O(m + n)$. The delete-min operations introduce an additional factor into the time complexity, which can increase the running time of the algorithm. If the structure of the graph allows a reduction in the number of delete-min operations, then the improved algorithms offer a better time complexity. These improved algorithms offer a better understanding of how to calculate shortest path problems more efficiently in terms of graph structure and the time complexity.

This paper introduces two new shortest path algorithms for nearly acyclic graphs. Section 4 presents a new generalised single-source (GSS) shortest path algorithm for nearly acyclic graphs with time complexity $O(m + r \log r)$, where r is the number of *trigger vertices*. Trigger vertices are the roots of trees that result when the graph is decomposed into trees. Section 5 then generalises the selection of trigger vertices to any feedback vertex set, for which a new all-pairs shortest path algorithm with worst case time complexity $O(mn + nr^2)$ is possible. For many nearly acyclic graphs, r is much less than n, allowing this new all-pairs algorithm to perform with $O(mn)$ time complexity. For an introduction to these new algorithms, Section 2 begins with a description of Dijkstra's algorithm, then gives an overview of existing shortest path algorithms for nearly acyclic graphs. Section 3 describes the potential for improvement to existing GSS algorithms. Concluding remarks are given in Section 6.

2 An Overview of Existing Shortest Path Algorithms

This section begins with a description of Dijkstra's algorithm, which is used as a basis for the more specialised shortest path algorithms; refer to Algorithm 1. Dijkstra's algorithm computes the shortest paths from a starting vertex to all other vertices in a directed graph, $G = (V, E)$, where V is the set of vertices in the graph, and E is the set of edges. Here V is given by the set integers $\{1, 2, \ldots, n\}$. In the following description of Dijkstra's algorithm, $OUT(v)$ is defined as the set of all vertices w such that there is a directed edge from vertex v to vertex w. The cost function $c(v, w)$ gives the edge cost from vertex v to vertex w.

Dijkstra's algorithm maintains three sets for keeping track of vertices: the solution set, S, the frontier set, F, and the set of vertices not in S or F (i.e. unexplored vertices). The set S stores vertices for which the shortest distance has been computed. The set F holds vertices v that have an associated tentative shortest path distance, $d[v]$, but do not have a finalised shortest path distance. This tentative distance is the distance of the shortest path that involves only v and vertices in S. Any vertex in F is directly connected to some vertex in S. We assume that all vertices in the graph are reachable from the source vertex, s.

Algorithm 1 Dijkstra's Algorithm

```
$S = \{s\}$;
$F = \emptyset$;
for each $w$ in $OUT(s)$ do {
  add $w$ to $F$ with $d[w] = c(s, w)$;
}
while $F$ is not empty do {
  select $v$ such that $d[v]$ is minimum among $v$ in $F$;
  remove $v$ from $F$; /* delete_min */
  add $v$ to $S$;
  for each $w$ in $OUT(v)$ and not in $S$ do {
    if $w$ is not in $F$ then {
      $d[w] = d[v] + c(v, w)$;
      add $w$ to $F$; /* insert */
    }
    else {
      $d[w] = min(d[w], d[v] + c(v, w))$; /* decrease_key */
    }
  }
}
```

There are three operations used on F: *insert*, *delete_min*, and *decrease_key*.

The Fibonacci heap [3], 2-3 heap [5], and trinomial heap [6] support *insert* and *decrease_key* in $O(1)$ time, and *delete_min* in $O(\log n)$ time. Since every vertex is visited there are n *insert* and n *delete_min* operations. The number of *decrease_key* operations is $O(m)$ since this corresponds to the number of edges in the graph. Thus, the overall time complexity when a Fibonacci heap, 2-3 heap, or trinomial heap, is used for F is $O(m + n \log n)$.

The time complexity for the single source shortest path problem can be reduced for specific graph types. If the graph is acyclic, the shortest path problem requires just $O(m + n)$ time to solve. Abuaiadh and Kingston [1] improved Dijkstra's algorithm by defining 'easy' vertices which are not pointed to by any edges from outside of S. Vertices which are pointed to by edges from outside of S are called 'difficult' vertices. If a vertex in F is an easy vertex, it is deleted from F. When there are no easy vertices in F, a *delete_min* operation is required. If t such *delete_min* operations are required, then overall the algorithm executes n *insert*, t *find_min*, and n *delete* operations on the frontier set. With these heap operations and the use of a modified Fibonacci heap for the frontier set data structure, the algorithm's time complexity is $O(m + n \log t)$. For a given graph, if the value of t is small compared to n, Abuaiadh and Kingston's algorithm will out-perform Dijkstra's algorithm.

Takaoka [4] gave a single source shortest path algorithm for nearly acyclic directed graphs based on the strongly connected (SC) components of the graph. In Takaoka's algorithm, a graph is decomposed into SC components and the acyclic structure linking them. This requires an initial scan of the graph using Tarjan's algorithm [7] to determine the strongly connected components. The shortest path calculation proceeds efficiently through the acyclic structure linking SC components. The shortest paths within an SC component are computed using a generalised single source (GSS) shortest path algorithm. If the number of vertices in the largest strongly connected component is k, then Takaoka's algorithm solves the single source shortest path problem in $O(m + n \log k)$ time. For given graphs, if the value of k is small compared to n, Takaoka's algorithm will out-perform Dijkstra's algorithm. Takaoka showed that this new algorithm could be combined with that by Abuaiadh and Kingston into a hybrid algorithm which incorporates the merits of each.

The generalised single source (GSS) shortest path problem, defined by Takaoka [4], specifies initial distances $d_0[v]$ for each vertex v in the graph. The algorithm for the GSS problem is the same as Dijkstra's algorithm, except it begins with all vertices in the frontier set. [3] For this purpose, the GSS initial distances for a given SC component arise from shortest paths through the acyclic structure to the SC component. The GSS algorithm of Takaoka [4] is given below, but presented similarly to Dijkstra's algorithm for comparison. Also, only vertices with a non-infinite initial distance are initially placed in

[3] This is not strictly necessary since only vertices with $d_0[v] \neq \infty$ are required to be in the frontier set initially. Thus, if only some vertices have a non-infinite initial distance, it is possible to avoid a large frontier set.

the frontier set.

Algorithm 2 GSS Algorithm

```
$S = \emptyset$;
$F = \emptyset$;
for each $v$ in $V$ do {
    if $d[v] \neq \infty$ then add $v$ to $F$ with $d[v] = d_0[v]$;
}
while $F$ is not empty do {
    select $v$ such that $d[v]$ is minimum among $v$ in $F$;
    remove $v$ from $F$; /* delete_min */
    add $v$ to $S$;
    for each $w$ in $OUT(v)$ and not in $S$ do {
        if $w$ is not in $F$ then {
            $d[w] = d[v] + c(v, w)$;
            add $w$ to $F$; /* insert */
        }
        else {
            $d[w] = min(d[w], d[v] + c(v, w))$; /* decrease_key */
        }
    }
}
```

The use of GSS is not restricted only to Takaoka's algorithm for nearly acyclic graphs. The conventional single source shortest path problem has $d_0[s] = 0$ and $d_0[v] = \infty$, and as a result all shortest paths must originate from vertex s. If we had other source vertices u with $d_0[u] = 0$, a resulting shortest path distance $d[v]$ would be for the shortest path from the closest source to v.

3 Introducing Improvements to the GSS Algorithm

The hybrid algorithm described by Takaoka [4] uses Abuaiadh and Kingston's method for the GSS algorithm. The new GSS algorithm, presented in Section 4, improves on the existing GSS algorithms, so that a wider range of nearly acyclic graphs can benefit from improved time complexity. This is done by introducing a new parameter, r, relating to the graph structure. As will be shown, the GSS problem can be solved in $O(m + r \log r)$ time.

Nearly acyclic graph structures are possible for which Abuaiadh and Kingston's method cannot guarantee improved time complexity over Dijkstra's algorithm. For the same graphs, the new algorithm can guarantee an improvement in time complexity. Consider solving a shortest path problem where the whole graph is strongly connected. Then Takaoka's algorithm cannot give improved time complexity over Dijkstra's algorithm. Similarly, if a graph (or SC component for GSS) does not result in any easy vertices there can be no improvement in time complexity when running Abuaiadh and Kingston's algorithm. Consider using Abuaiadh and Kingston's method to solve the GSS problem on some graph or SC component. The time complexity of Abuaiadh and Kingston's algorithm, $O(m + n \log t)$, is defined in terms of the number of *delete_min* operations, t, and not in terms of the graph structural properties. Consider the a vertex chain structure in part of a graph, defined as follows:

- The chain consists of the vertices $v_0, v_1, \ldots, v_j$.
- Edges in the chain are of the form (v_i, v_{i+1}) for $i = 0, 1, \ldots, j-1$.
- Vertices $v_1,\ v_2,\ \ldots\ v_j$ have only one incoming edge.
- Assume for a GSS problem, some arbitrary initial distance for each vertex on the chain.

Now consider what happens if when calculating GSS problem using Abuaiadh and Kingston's method, a vertex v_i on this vertex chain is moved to the solution set as a result of a *delete_min* operation. Following this *delete_min* all vertices, v_k for $i < k \leq j$ will subsequently be moved to the solution set as each becomes 'easy'. During this process, the tentative distance is updated from one vertex to the next on the chain by *decrease_key* operations. The worst case occurs if there is an initial distance distribution such that *delete_min* operations remove vertices from F in the order $v_j, v_{j-1}, \ldots, v_0$. In this case, no vertex on the chain ever becomes easy, and j *delete_min* operations occur as they would have if Dijkstra's algorithm were used. In the best case, one *delete_min* operation would remove vertex v_0 (the head vertex on the chain) from F, and the subsequent *delete* operations would remove all remaining vertices on the chain from F. If it is possible to limit *delete_min* operations to only head vertices of chains in the graph, then the worst case time complexity can be reduced.

4 An Improved GSS Algorithm for Nearly Acyclic Graphs

This section introduces a new GSS algorithm for nearly acyclic directed graphs. For certain kinds of graphs, this algorithm improves on Abuaiadh and Kingston's algorithm [1] (when used for solving GSS problems), and introduces improvement to Takaoka's algorithm [4]. The basic form of the new algorithm is presented. More complicated variants of the new algorithm, which are not presented, can improve time efficiency by a constant factor.

Section 3 used vertex chains for introducing the potential for improving on the time complexity of Abuaiadh and Kingston's method in GSS problems. The same concept can be generalised from chains of vertices to trees of vertices. Define $IN(v)$ as the set of vertices u such that there is an edge (u, v) in the graph. Then tree structures in a graph can be identified as follows:

- A root vertex, v, in a tree structure has $|IN(v)| > 1$.

- A non-root vertex, v, in a tree structure has $|IN(v)| = 1$.

Denote the tree structure in the graph rooted at vertex v by $tree(v)$. If there is a directed edge from a vertex in a tree, T, to a root vertex w of some other tree, then T is a *neighbouring tree* of w. Suppose a *delete_min* operation occurred on a root vertex, v, of a tree structure $tree(v)$ within the graph. Then all other vertices in $tree(v)$ would subsequently be moved to S as each becomes easy. The moving of vertices to S propagates through the entire tree structure. This is the best case for Abuaiadh and Kingston's method. However, within a tree of size j, the worst case for Abuaiadh and Kingston's method is j *delete_min* operations.

Figure 1 illustrates a graph viewed as a set of tree structures. Edges which point to a root vertex are shown as dashed lines. In the simplified view, such edges with the same source tree and destination root vertex are represented using a single pseudo-edge. From the simplified view, it is easily seen that in general only 1 *delete_min* operation per tree structure is necessary. The

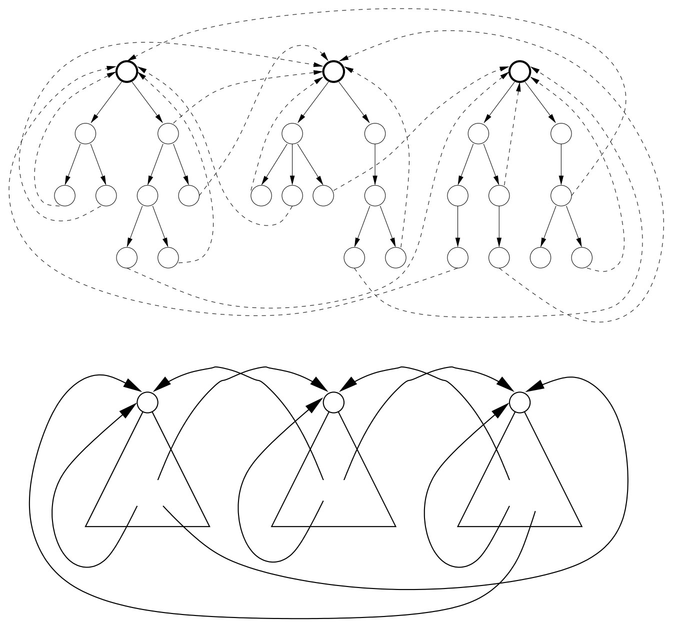

Fig. 1. Example of a graph viewed as linked tree structures.

first step of the new algorithm is to scan each vertex, v, in the graph to determine root and non-root vertices, according to the value of $|IN(v)|$. In this description, a root vertex is called a 'trigger' vertex. A trigger vertex *triggers* shortest path distance updates into other vertices in the tree. Note that this algorithm assumes that all non-trigger vertices in the graph will be a descendant of some tree root. This will always be the case if the GSS problem is over a strongly connected graph, or an SC component of some graph.[4] For

[4] For the special case, where a strongly connected graph is a ring of vertices, such that every vertex v has $IN(v) = 1$, any one vertex can be chosen at random for the trigger.

a graph which is not strongly connected, if there exist non-triggers that are not descendant of some root vertex, the algorithm can be easily altered by identifying vertices v with $|IN(v)| = 0$. The rest of the algorithm consists of two updating passes through the graph.

Algorithm 3 gives the first updating pass of the algorithm. This calculates first-tentative shortest path distances $d_1[v]$ for vertices in each tree. No *delete_min* operations are performed during this first updating pass. At the beginning of the algorithm, each vertex v has a GSS initial distance, $d_0[v]$. The updating of vertices in a tree requires a queue, Q, to be maintained. If the queue is maintained first-in first-out, then vertices in a tree will be updated in a breadth first search. If the queue is maintained last-in first-out, then vertices in a tree will be updated in a depth first search. Note that if depth first search is used, that part of the algorithm could be implemented recursively, eliminating the need for the algorithm to maintain a queue.

Algorithm 3 First Stage of the New GSS Algorithm

```
/* assume trigger vertices are known */
$Q = \emptyset$;
for each vertex $v$ do $d_1[v] = d_0[v]$;
for each trigger vertex $u$ do {
  add non-trigger vertices in $OUT(u)$ to $Q$;
  while there is a vertex $v$ in $Q$ do {
    remove $v$ from $Q$;
    for each vertex $w$ in $OUT(v)$ do {
      $d_1[w] = min(d_1[w], d_1[v] + c(v, w))$;
      if $w$ is not a trigger vertex then add $w$ to $Q$;
    }
  }
}
```

The distance updates in Algorithm 3 are restricted from propagating between trees. Even though this is not strictly necessary for the algorithm to work, for now it makes the explanation simpler. A more efficient version of this algorithm, which is not presented, allows the distance updates to be less restrictive, which can reduce the number of distance updates during the second updating pass.

A first-tentative shortest path distance $d_1[v]$ is the shortest distance resulting from the initial distance $d_0[v]$ or paths of the form:

$$(v_1, v_2, \ldots, v_k, v), \quad k \geq 1$$

for which:

$$d_1[v] = d_0[v_1] + c(v_1, v_2) + \ldots + c(v_k, v)$$

With path length defined in terms of the number of edges traversed by the path, this path has length k. The properties of such a path of length k are:

- Each v_i $(1 \leq i \leq k)$, lies on the same tree, T; that is, $v_i \epsilon T$ for $1 \leq i \leq k$.
- If vertex v is a non-trigger, then it is on the same tree as vertices v_i $(1 \leq i \leq k)$.
- If vertex v is a trigger vertex, then vertices v_i $(1 \leq i \leq k)$ are on a neighbouring tree of v.

Note that in this restricted algorithm no trigger vertex will be involved in the first-tentative shortest path of another trigger vertex. A trigger vertex can only be updated from as far away as non-trigger vertices in neighbouring trees. At the end of the first updating pass, the following assertions hold:

- For each trigger vertex u, the shortest path to u that can result from non-trigger vertices in neighbouring trees of u has been calculated. This distance is given in $d_1[u]$.
- Any improvements on $d_1[u]$ for any trigger vertex, u, must involve a path from another trigger vertex.

Algorithm 4 gives the second updating pass algorithm.

Algorithm 4 Second Stage of the New GSS Algorithm (Continues from Algorithm 3)

```
1.  $S = \emptyset$;
2.  insert all trigger vertices into $F$;
3.  **for** each vertex $v$ **do** $d[v] = d_1[v]$;
4.  **while** $F$ is not empty **do {**
5.     select $u$ such that $d[u]$ is the minimum among $u$ in $F$; /* *delete_min* */
6.     remove $u$ from $F$;
7.     add $u$ to $S$;
8.     add $u$ to $Q$;
9.     **while** there is a vertex $v$ in $Q$ **do {**
10.       remove $v$ from $Q$;
11.       **for** each vertex $w$ in $OUT(v)$ **do {**
12.          $d[w] = min(d[w], d[v] + c(v, w))$;
                /* *If $w$ is a trigger vertex a decrease_key operation may occur.* */
13.          **if** $w$ is not a trigger vertex **then** add $w$ to $Q$;
14.       **}**
15.    **}**
16. **}**
```

For the second updating pass, only trigger vertices are involved in the frontier set, F, and solution set, S. At lines 5 and 6, the trigger vertex, u, which has minimum $d[u]$, is selected and removed from F. Call this the *minimum trigger vertex*. This vertex is then added to the solution set, S.

Before the ith iteration at line 5, let the state of the solution set, S, be:

$$S = \{u_1, u_2, \ldots, u_{i-1}\} \quad \text{(added in this order)}$$

Then we have the following theorem:

Theorem 4.1

(i) *for trigger vertices $u_k \,\epsilon\, S$ $(1 \leq k \leq i-1)$, $d[u_k]$ is the shortest distance to vertex u_k.*

(ii) *for all vertices $v \,\epsilon\, tree(u_k)$ and all u_k $(1 \leq k \leq i-1)$, $d[v]$ is the shortest distance to vertex v.*

(iii) *for trigger vertices $u \,\epsilon\, F$, $d[u]$ is the distance of the shortest path to u that is allowed to go though only non-triggers, trigger vertices in S, and u.*

Proof (by induction)

Basis $i = 1$: Assertions 1 and 2 above are automatically true since S is empty for $i = 1$. For assertion 3 above, $d[u]$ is correctly computed by Algorithm 3 since S is empty.

Induction step. Assume the theorem is true for $S = \{u_1, u_1, \ldots, u_{i-1}\}$. If u_i is the minimum among trigger vertices in F, then $d[u_i]$ is the shortest distance to u_i since the distance for a path through any other trigger vertex in F will be longer. Also, for $v \,\epsilon\, tree(u_i)$, the shortest distance $d[v]$ is correctly computed since there is no shorter path to v that goes through other triggers. Finally, for trigger vertices u remaining in F, $d[u]$ will be updated if $tree(u_i)$ is a neighbouring tree of u. Therefore for triggers u remaining in F, the distance of the shortest path that goes through trigger vertices in $u_1, u_2, \ldots, u_i$ is correctly computed since u_i and $tree(u_i)$ will be the latest possible trigger and tree structure to go through to reach u. Hence the theorem is true for $S = \{u_1, u_2, \ldots, u_i\}$. □

Let there be a total of n vertices and m edges in the graph. The first updating pass through the graph takes $O(m)$ time. Now assume a Fibonacci heap is used for F. Suppose there are r trigger vertices in the graph, then there will be r *delete_min* operations in the second updating pass, each taking at most $O(\log r)$ time, giving a combined worst case time complexity $O(r \log r)$. The second updating pass also has an $O(m)$ time component, which accounts for each edge traversed, and any *decrease_key* operations. Combining these times, the worst case time complexity of the entire algorithm is $O(m + r \log r)$.

For the conventional single-source problem all initial distances are infinite except for the source vertex. This allows a simplified first updating pass when the source vertex is a non-trigger, and no first updating pass when the source vertex is a trigger. The second updating pass and $O(m + r \log r)$ time complexity remains the same as for the GSS algorithm. The $O(m + n \log t)$ time complexity of Abuaiadh and Kingston's algorithm, depends on the number of *delete_min* operations, t. For GSS, the worst case value for t is $O(n)$, but for the conventional single source problem, the worst case value of t is only $O(r)$. For the conventional single source problem, the possible improvement over Abuaiadh and Kingston's method is diminished.

The amount of improvement offered by the new algorithm depends highly on the structure of the graph, since this determines the value of r compared to t. For a strongly connected graph made up of r tree structures, Abuaiadh and Kingston's algorithm, will take $O(m + n \log n)$ time to solve the GSS problem if t is $O(n)$. Under the same circumstances, the new algorithm can have $r < n$ and its worst case time will be $O(m + r \log r)$. Although the worst case time complexity is improved, in special circumstances it is possible for a run of Abuaiadh and Kingston's algorithm to perform better. This will occur when *delete_min* occurs on a small number of trigger vertices which in turn cause all the remaining trigger vertices to eventually become easy vertices. For now, the important result is that the worst case time complexity of this algorithm is guaranteed to be as good or better than that of Abuaiadh and Kingston's algorithm for GSS. This new GSS algorithm can offer further improvement to Takaoka's single source algorithm for acyclic graphs, by using them in hybrid form. Another improvement is to take Abuaiadh and Kingston's concept of 'easy' vertices, and extend this to 'easy' trigger vertices. Then easy trigger vertices could be identified and deleted from F in $O(1)$ time, thus reducing the number of *delete_min* operations required.

As was mentioned, the updates for shortest path distances in the new algorithm were deliberately limited to make the description simpler. An improved version of the algorithm can allow distance updates to propagate between trigger vertices during the first updating pass, without changing the correctness of the algorithm. Then, if during the second updating pass, the distance to a vertex, v, does not update, the algorithm does not need to continue distance updates past v. By terminating the search at vertices which do not update, on average there may be a slight gain in time efficiency, even though the worst case time complexity will not change.

Further work for possible improvements to this algorithm includes generalising from tree decomposition to a special form of acyclic decompositions. For an acyclic part A resulting from decomposition of the graph, there must be only one trigger vertex ancestor, u, of vertices in A. Thus, now a trigger vertex u triggers updates into its acyclic part instead of a tree structure. This allows the selection of trigger vertices to be less restrictive, reducing the number of trigger vertices, and number of *delete_min* operations that must occur.

5 An Improved All-Pairs Algorithm

The GSS algorithm, presented in the previous section, selected trigger vertices according to tree structures in the graph. Because tree structures are acyclic, shortest path distances through tree structures could be computed efficiently. This section extends the concept of trigger vertices to any selection of vertices that cause the remainder of the graph to become acyclic. As will be shown, this allows for a more efficient all-pairs algorithm, but, at present, does not provide an improved single-source algorithm.

Let G be the overall graph, and V be the set of vertices of G. Using the same notation as before, n is the total number of vertices, m is the number of edges, and r is the number of trigger vertices. Suppose a selection of trigger vertices is obtained through some efficient algorithm. A set of trigger vertices, T, must satisfy the following property:

- If all vertices in T are removed from the graph, the remaining vertices, $\overline{T}$, induce a graph which is acyclic. Note that the graph formed by vertices in $\overline{T}$ is allowed to be disconnected.

That is, a selection of trigger vertices corresponds to a feedback vertex set. Figure 2 shows an example graph to illustrate this concept. The lower illustration shows a generalised view of this concept for a selection of r trigger vertices $u_1,\ u_2,\ \ldots\ u_r$. The view of edges into and out of the acyclic structure has been simplified using copies of each trigger vertex, and pseudo-edges to represent many edges to or from the same trigger vertex.

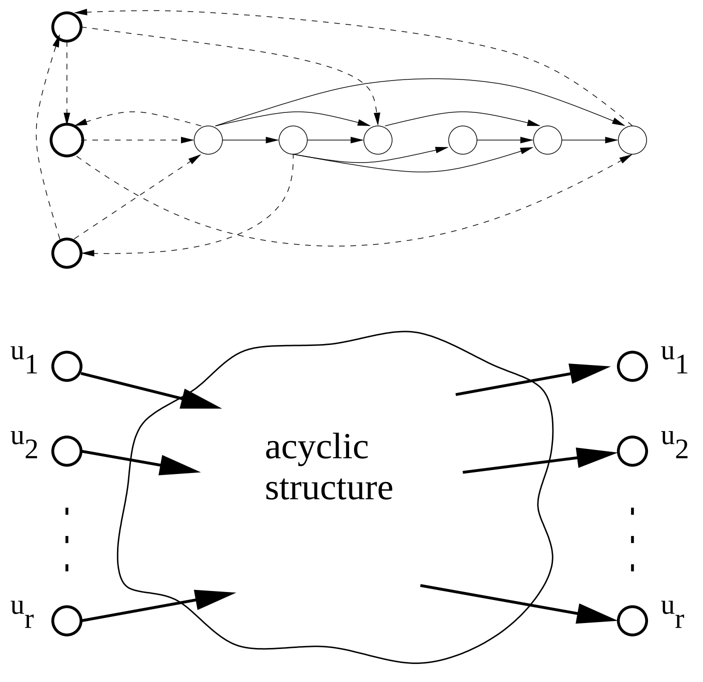

Fig. 2. Example of identifying the graph structure as consisting of trigger vertices and an acyclic part.

The new all-pairs algorithm consists of two stages. Algorithm 5 shows the first stage, and Algorithm 6 shows the second stage. The algorithm uses a two dimensional array, D, to hold shortest path distances as the computation proceeds. At the end of the algorithm, array D holds the shortest path distance between any pair of vertices. In the algorithm, the reference array, d, is used for referring to a row in D. Updating the shortest path calculation through vertices in $\overline{T}$, can be done efficiently, since the graph induced by $\overline{T}$ is acyclic. The algorithm uses a topological ordering of vertices in $\overline{T}$, stored in an ordered set, L, which can be obtained in $O(m+n)$ time. A graph, P, whose vertices correspond to triggers, is constructed by the first stage of the algorithm, and used by GSS for calculating shortest path distances through vertices in T.

The first stage of the new all-pairs algorithm calculates first-tentative distances $d_1[v_0, v]$. The notation $d_1[v_0, v]$ is used to clarify this description, and corresponds to the state of $D[v_0, v]$ at the end of Algorithm 5. This involves performing the first stage of several single-source problems. For each $v_0 \in V$:

- First-tentative shortest path distances $d_1[v_0, v]$, from v_0 to each vertex $v \in V$ are computed.[5] A distance $d_1[v_0, v]$ corresponds to the shortest path from paths of the form:

$$(v_0, v_1, v_2, \ldots, v_k, v), \quad k \geq 0$$

  where each $v_i \in \overline{T}$ for $1 \leq i \leq k$. The calculation of first-tentative distances from a source vertex v_0 takes $O(m)$ time.

As a by-product of the first stage of the algorithm, a reduced graph, P, is computed from G. Each vertex in P corresponds to a trigger vertex. The edge costs of edges in P (called pseudo-edges) are defined as follows:

- The cost of pseudo edge (u, w), where $u \in T$ and $w \in T$, corresponds to the shortest path from paths of the form:

$$(u, v_1, v_2, \ldots, v_k, w), \quad k \geq 0$$

  where each $v_i \in \overline{T}$ for $1 \leq i \leq k$. That is, the path goes through only vertices in $\overline{T}$ except for end points. If there is no such path, the edge (u, w) does not exist in graph P.

The first stage, including the calculation of edges distances for P, takes $O(mn)$ time. For the rest of this explanation, m' will denote the number of edges in P.

Algorithm 5 First Stage of the New All-Pairs Algorithm

```
1.  Topologically sort vertices in $\overline{T}$, placing the result into the ordered set $L$.
2.  for each vertex $v_0$ in $V$ do {
3.      let $d$ be a reference to row $v_0$ of array $D$;
4.      for each vertex $v$ in $V$ do $d[v] = \infty$;
5.      $d[v_0] = 0$;
6.      if $v_0$ is in $T$ then for each vertex $w$ in $OUT(v_0)$ do $d[w] = c(v_0, w)$;
7.      for each vertex $v$ in order from $L$ do {
8.          for each vertex $w$ in $OUT(v)$ do {
```

[5] Only the first-tentative distances $d_1[v_0, u]$, for vertices $u \in T$, and $d_1[v_0, v_0] = 0$ are important for the correctness of the second stage of the algorithm (see Algorithm 6). Other first-tentative distances are not important since the same computation can occur during Algorithm 6.

```
9.            $d[w] = min(d[w], d[v] + c(v, w));$
10.         }
11.     }
12.     if $v_0$ is in $T$ then {
13.         for each vertex $u$ in $T$ with $d[u] \neq \infty$ do {
14.             add edge $(v_0, u)$ with cost $d[u]$ to $P$;
15.         }
16.     }
17. }
```

In this first stage of the algorithm, there are no *delete_min* operations. Within the outermost loop (lines 2 to 16) of Algorithm 5 $O(m)$ total time will be taken up for updating distances through the topological ordering of vertices, and for adding edges to P. Any $O(r)$ part is contained within the $O(m)$ time bound, so the time to complete one loop is $O(m)$. With the outermost loop repeated n times, the total time taken is $O(mn)$. Upon completion of one cycle of the outermost loop, the shortest path distance through $\overline{T}$ from the source vertex, v_0, to all other vertices will have been computed. Upon completion of the first stage of the algorithm, the shortest path distance through $\overline{T}$ between any pair of vertices (u, v) is in $D[u, v]$; that is $D[u, v]$ is equal to the first-tentative shortest path from u to v. Also, for any pair of vertices $u \in T$ and $v \in T$:

- If $D[u, v] \neq \infty$, then the edge from u to v in P has an edge cost equal to $D[u, v]$.

Although this method is efficient for all-pairs, it is not efficient for a single-source problem since it would take $O(rm)$ time to calculate the pseudo edges of P, which exceeds the $O(m + n \log n)$ time complexity of Dijkstra's algorithm.

The second stage of the new all-pairs algorithm (refer to Algorithm 6) completes the all-pairs shortest path computation. Note that distance values from Algorithm 5 are retained in D and used in Algorithm 6. This is important in the correctness of Algorithm 6. The second stage of this all-pairs algorithm can be viewed as repeating the second stage of each single-source problem. For each $v_0 \in V$:

(i) Let $d_1[v_0, u]$ correspond to the value of $D[v_0, u]$ at the end of Algorithm 5. For vertices $u \in T$, distances $d_1[v_0, u]$ are used as the initial distances $d_0[u]$ for a GSS problem on graph P. Algorithm 2, or some other efficient GSS algorithm, is then used for computing the GSS shortest path distances over P. A distance $d[u]$, for $u \in T$, computed from the GSS problem on P, corresponds to the distance of the shortest path from paths of the form:

$$(v_0 \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_k \rightsquigarrow u), \quad k \geq 0$$

for which each $u_i \in T$ (for $1 \leq i \leq k$) is a unique trigger vertex on the path, and the symbol $\rightsquigarrow$ denotes a path of the form:

$$(v_1, v_2, \ldots, v_j), \quad j \geq 0$$

where $v_i \;\epsilon\; \overline{T}$ for $1 \leq i \leq j$. This represents all possible paths from v_0 to vertex u. Hence the distances $d[u]$ for $u \;\epsilon\; T$ computed from the GSS problem is the final shortest path distance $D[v_0, u]$ in the all-pairs problem. The correctness of this assertion follows from the definition of the GSS problem; see Section 2 and Takaoka [4].

(ii) The finalised shortest path distances of the form $D[v_0, u]$, where $u \;\epsilon\; T$, are then used in calculating shortest path distances of the form $D[v_0, v]$ for vertices $v \;\epsilon\; \overline{T}$. A distance $d[v]$, for $v \;\epsilon\; \overline{T}$, at the end of the single-source computation from v_0, corresponds to the distance of the shortest path from paths of the form:

$$(v_0 \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_k \rightsquigarrow v), \quad k \geq 0$$

for which each $u_i \;\epsilon\; T$ (for $1 \leq i \leq k$) is a unique trigger vertex on the path. Hence the distances $d[v]$, referring to $D[v_0, v]$, for $v \;\epsilon\; \overline{T}$ are final in the all-pairs problem.

A single-source part in the second stage takes $O(m + m' + r \log r)$ time. This is repeated n times to cover all source vertices, so the total time for the second stage is $O(mn + m'n + nr \log r)$.

Algorithm 6 Second Stage of the New All-Pairs Algorithm

```
18. for each vertex $v_0$ in $V$ do {
19.     let $d$ be a reference to row $v_0$ of array $D$;
20.     for each vertex $v$ in $T$ do set GSS initial distance for $v$ to $d[v]$;
21.     Solve GSS problem on $P$;
            /* This finalises distances $d[v]$ (that is $D[v_0, v]$) for $v$ in $T$; */
22.     for each vertex $u$ in $T$ do {
23.         for each vertex $w$ in $OUT(u)$ do $d[w] = min(d[w], d[u] + c(u, w))$;
24.     }
25.     for each vertex $v$ in order from $L$ do {
26.         for each vertex $w$ in $OUT(v)$ do {
27.             $d[w] = min(d[w], d[v] + c(v, w))$;
28.         }
29.     }
30. }
```

Each outer loop of Algorithm 6 completes the single-source shortest path calculation from the source vertex v_0 to all other vertices; lines 19 to 29. At line 21 the GSS problem is solved, and $d[v]$ holds the shortest path distance to vertices $v \;\epsilon\; T$ from vertex v_0. It takes $O(m' + r \log r)$ time to solve the GSS problem on P. During the entire second stage of the algorithm, *delete_min* and other heap operations only occur within the GSS algorithm. At the start of line 22, the shortest path distance from v_0 to trigger vertices is known. To complete the single-source computation, the shortest path from v_0 to non-trigger vertices must be determined. Lines 22 to 29 do this by scanning shortest path distance updates through the topological ordering of vertices in L. These updates take $O(m)$ time. After line 29, the single-source problem from vertex v_0 has been computed. The total time for the second stage to complete a single-source computation is:

$$O(m' + r\log r) + O(m) = O(m + m' + r\log r)$$

The completion of the single-source computation is repeated for each $v_0 \epsilon V$, so a total of n single source problems are completed. Therefore the overall time complexity of the second stage is $O(mn + m'n + nr\log r)$. Taking the combined time of the first and second stages of the algorithm, the overall time complexity is:

$$O(mn) + O(mn + m'n + nr\log r) = O(mn + m'n + nr\log r)$$

Accounting for the worst case, where m' is $O(r^2)$, the time complexity becomes $O(mn + nr^2)$. For most nearly acyclic graphs we expect r to be much smaller than n, and the time complexity of the algorithm becomes $O(mn)$. Alternatively, if $m' \leq m$, the time complexity will be $O(mn + nr\log r)$.

If for a given graph, k is large and r is small, the new algorithm can give significant improvement over the previous shortest path algorithms [4]. Other implementations of this algorithm are possible which are more efficient by a constant factor. More efficient implementations can avoid distance updates from a vertex, v, when $d[v]$ is still infinite. One such algorithm uses two separate depth first search (DFS) like functions, where one of the DFS functions only traverses edges and does not update shortest path distances.

6 Concluding Remarks

For nearly acyclic graphs, it is possible to solve the generalised single source problem in $O(m + r\log r)$ time, where r is the number of trigger vertices, with trigger vertices defined as roots of trees that result when the graph is decomposed into trees. This gives an improvement on existing shortest path algorithms for nearly acyclic graphs from Abuaiadh and Kingston [1] and Takaoka [4]. It is possible to combine this new algorithm and the previous algorithms into a hybrid algorithm which incorporates the properties of each. Future work involves generalising from tree decomposition to an acyclic decomposition, in order to allow a reduced number of trigger vertices.

For the all-pairs shortest path problem, a new algorithm was given with $O(mn+nr^2)$ worst case time complexity, where r is the number of vertices in a pre-calculated feedback vertex set for the nearly acyclic graph. In most cases of nearly acyclic graphs, r is much smaller than n, and the time complexity of the algorithm becomes $O(mn)$. This is a significant improvement when the feedback vertex set is known in advance. Given that the minimum feedback vertex set problem is NP-complete, if the feedback vertex set is not known in advance, we need to be satisfied with a non-optimal feedback vertex set for triggers. Our ultimate goal is to find a good heuristic algorithm, within the time complexity of our shortest path algorithm, which computes a feedback vertex set of a size within a constant times the optimal size. Another consideration is whether an efficient single-source algorithm is possible for which any feedback vertex set can be used as trigger vertices.

References

[1] D. Abuaiadh and J.H. Kingston. Are Fibonacci heaps optimal? In *ISAAC '94*, Lecture Notes in Computer Science, pages 41–50. 1994.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimisation algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[4] T. Takaoka. Shortest path algorithms for nearly acyclic directed graphs. *Theoretical Computer Science*, 203(1):143–150, August 1998.

[5] T. Takaoka. Theory of 2-3 heaps. In *Proc. COCOON '99*, volume 1627 of *Lecture Notes in Computer Science*, pages 41–50. July 1999.

[6] T. Takaoka. Theory of trinomial heaps. In *COCOON '00*, volume 1858 of *Lecture Notes in Computer Science*, pages 362–372. July 2000.

[7] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.