Improved Shortest Path Algorithms for Nearly Acyclic Graphs

Shane Saunders and Tadao Takaoka

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 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. 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.

The generalised single-source (GSS) problem allows an initial distance to be defined at each vertex in the graph. Decomposing a graph into r trees allows the GSS problem to be solved within $O(m+r \log r)$ time. This paper presents a new all-pairs algorithm with a time complexity of $O(mn + nr \log r)$, where r is the number of acyclic parts resulting when the graph is decomposed into acyclic parts. The acyclic decomposition used is set-wise unique and can be computed in $O(mn)$ time. If the decomposition has been pre-calculated, then GSS can be solved within $O(m+r \log r)$ time whenever edge-costs in the graph change. A second new all-pairs algorithm is presented, 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.

Key words: shortest path algorithms, nearly acyclic graphs, graph decomposition

Email addresses: `sas59@cosc.canterbury.ac.nz` (Shane Saunders), `tad@cosc.canterbury.ac.nz` (Tadao Takaoka).

1 Introduction

Dijkstra's algorithm [1] 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 case time if a Fibonacci heap [2] 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. For an introduction to graph theory terms refer to [3]. 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 [4] 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 [5], 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 presents two new shortest path algorithms for nearly acyclic graphs. Section 2 presents an introductory generalised single-source (GSS) shortest path algorithm with time complexity $O(m+r\log r)$, where r is the number of *trigger vertices*. Here trigger vertices are defined as the roots of trees that result when the graph is decomposed into trees. A more general definition of trigger vertices is used in Section 3, where the graph is decomposed into acyclic structures, and each acyclic structure is 'dominated' by a single trigger vertex.

This gives a new all-pairs algorithm with time complexity $O(mn + nr \log r)$, where r is defined as the number of such trigger vertices in the decomposition. Section 4 then generalises the selection of trigger vertices to any feedback vertex set of size r, 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 less than $\sqrt{m}$, allowing this new all-pairs algorithm to perform with $O(mn)$ time complexity. Concluding remarks are given in Section 5.

The following subsections provide further introduction. Section 1.1 reviews the concept of solving shortest path algorithms by graph decomposition, which has appeared previously and is relevant to the algorithms presented by this paper. To introduce the new algorithms, Section 1.2 begins with a description of Dijkstra's algorithm, then gives an overview of existing shortest path algorithms for nearly acyclic graphs. Section 1.3 describes the potential for improvement to existing GSS algorithms.

1.1 Related Work

The algorithms presented in Sections 2 and 3 of this paper solve the shortest path problem efficiently by decomposing the graph into parts. Section 2 uses a tree-decomposition, and Section 3 uses an acyclic decomposition. Both of these decompositions are set-wise unique.

The concept, solving shortest path algorithms by graph decomposition, was introduced in the Ph.D. thesis of Diab Abuaiadh [6]. This work also appears in a technical report published by Abuaiadh and Kingston [7]. Abuaiadh and Kingston prove that in general an edge-disjoint decomposition can be used to break the graph into several parts in order to improve the time complexity for solving the shortest path problem. The analysis leaves parts of the time complexity with hypothetical values, which are dependent upon the algorithms chosen for decomposing and solving shortest paths on each part of the graph. Thus, the actual time complexity is not known until a specific decomposition algorithm is specified. Abuaiadh and Kingston presented one such algorithm for near acyclic directed graphs, whereby the graph was decomposed into acyclic parts. The resulting decomposition lies somewhere between the tree decomposition and acyclic decomposition methods presented in Sections 2 and 3 of this paper. Note that the decomposition presented in [6] is not set-wise unique. That is, the partitioning is not deterministic since several different decompositions can result, depending on the order that the graph is decomposed in.

Although Abuaiadh and Kingston prove that any edge-disjoint decomposition can be applied to the shortest path problem, the exact form of the edge-disjoint decomposition, how to calculate it, and the time complexity of the resulting shortest path problem remain undefined. The algorithms presented in Sections 2 and 3 of this paper contribute applications of the concept previously proven by Abuaiadh and Kingston. A significant part of this paper's contribution lies in the thorough proofs of the time complexity for each application of this concept.

The "trigger" vertices resulting from this paper's tree and acyclic decompositions are similar to the "red" vertices presented in [6]. This paper differs from [6] by enforcing set-wise uniqueness of the decompositions used. This allows the new parameter introduced into the time complexity to be well defined. That is, for both the tree and acyclic decompositions in this paper, the resulting number of "trigger" vertices depends only on the graph structure. Thus, the resulting number of trigger vertices is fixed for any given graph. In comparison, the resulting number of "red" vertices in Abuaiadh and Kingston's acyclic decomposition method [6] depends on the order in which the algorithm proceeds. Their decomposition is able to perform at least as well as this paper's tree-decomposition, with the resulting number of "red" vertices" $\leq$ "the number of tree decomposition "trigger" vertices. However, it cannot do better than this paper's acyclic decomposition, with the number of "red" vertices $\geq$ the number of acyclic decomposition "trigger" vertices. Tree decomposition can be seen as a special case of the acyclic decomposition presented in [6], as there is some similarity between the "red" vertices and tree decomposition "triggers". The tree decomposition presented in this paper serves as introducing the more advanced concept of set-wise unique acyclic decomposition.

The acyclic decomposition method does require $O(mn)$ time, as opposed to $O(m)$ time needed for the decomposition in [6]. This is needed in order to ensure that the resulting decomposition is set-wise unique and has the lowest possible number of trigger vertices. An $O(mn)$ time decomposition is still acceptable for use with all-pairs calculations. That is, once the decomposition has been computed, repeated single-source calculations can be performed efficiently, and the $O(mn)$ time complexity does not impact upon the time complexity required for solving all-pairs.

This paper builds upon the general concept presented in [6], and contributes with its acyclic decompositions applied under this general concept. Section 4 of this paper presents an all-pairs shortest path algorithm which makes use of a pre-computed feedback vertex set. It introduces a new concept where pseudo-edges are used to efficiently calculate shortest paths. This is a significant advance as it is outside the framework of the edge disjoint decomposition concept presented in [6]. There is still much research to be done in the area of solving shortest paths by graph decomposition. Particularly as to which graph decomposition is optimal for solving the shortest path problem on a given type of graph.

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 = ∅;
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: *delete_min*, *insert*, and *decrease_key*. The Fibonacci heap [2], 2-3 heap [8], and trinomial heap [9] 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 [4] 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 without effort to locate the minimum vertex. 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 in [4] will out-perform Dijkstra's algorithm. In the remainder of this paper, whenever "Abuaiadh and Kingston's" method/algorithm is referred to assume it refers to their paper, [4], unless cited otherwise.

Takaoka [5] 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 [10] 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 [5], 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.[1] 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 [5] 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 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 correspond to the shortest path from the closest source to v.

1.3 Possible Improvements to Existing Algorithms

Takaoka's algorithm can only offer improved time complexity over Dijkstra's algorithm for graphs which are not strongly connected. Abuaiadh and Kingston's

[1] 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.

algorithm only offers improvement when easy vertices result during a run of the algorithm. Nearly acyclic graph structures are possible for which no easy vertices will result during a run of Abuaiadh and Kingston's algorithm. If a single vertex, u, in the graph points to all others, then no vertex can become "easy" until u has been included into S; and it is possible that u could be the last vertex included into S. The problem here is that 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. This is especially true when solving a GSS problem where the GSS initial distance distribution causes delete-min operations to occur in an order which prevents the occurrence of easy vertices.

Consider a graph containing a tree-structure, $tree(v)$, where v is the root vertex of the tree. Suppose a *delete_min* operation selects v first. Then all other vertices in $tree(v)$ will 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. This occurs when GSS initial distances $d_0[w]$ for vertices $w \in tree(v)$ are inversely proportional to the number of edges on the path from v through $tree(v)$ to w.

2 Shortest Path Algorithms Using Tree-Decomposition

This section presents a GSS algorithm which decomposes a graph into trees in order to improve the time complexity required when solving the shortest path problem on nearly acyclic directed graphs. This serves as an introduction to the new algorithm presented in Section 3 which uses a more general acyclic decomposition. An important feature of both these algorithms is that the decomposition used is set-wise unique; refer back to Section 1.1. For certain kinds of graphs, the algorithm in this section improves on Abuaiadh and Kingston's algorithm [4] (when used for solving GSS problems), and introduces improvement to Takaoka's algorithm [5].

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 $|IN(v)| > 1$ or $|IN(v)| = 0$.
- 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.

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

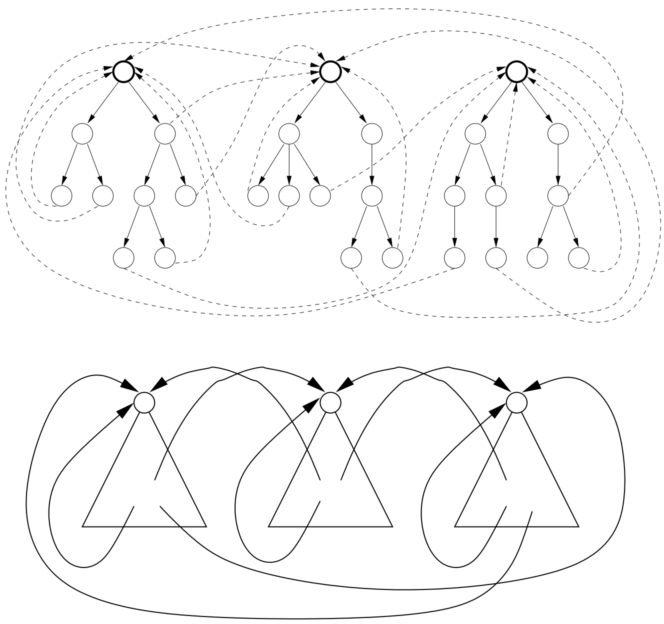

Fig. 1. Example of a graph viewed as linked tree structures.

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. [2] 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. The queue can be maintained in either first-in first-out (breadth first search) or last-in first-out (depth first search). Alternatively, the algorithm can be implemented as a recursive depth-first search, eliminating the need for the algorithm to maintain a queue.

Algorithm 3 First Stage of the New GSS Algorithm

[2] For the special case, where the 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.

```
/* assume trigger vertices are known */
Q = ∅;
for each vertex v do d1[v] = d0[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 {
         d1[w] = min(d1[w], d1[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. This is not strictly necessary for the algorithm to work, but for now it makes the explanation simpler.

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, $v_i \in T$ 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 with nonzero $|IN(v)|$ 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 1

(1) for trigger vertices $u_k \in S$ $(1 \leq k \leq i-1)$, $d[u_k]$ is the shortest distance to vertex u_k.
(2) for all vertices $v \in tree(u_k)$ and all u_k $(1 \leq k \leq i-1)$, $d[v]$ is the shortest distance to vertex v.
(3) for trigger vertices $u \in F$, $d[u]$ is the distance of the shortest path to u

which uses an initial path of zero or more non-triggers, followed by zero or more paths through trees $tree(v)$ for trigger vertices $v \in S$, to reach u.

Proof (by induction)

Basis $i = 1$: Assertions 1 and 2 above are automatically true since S is empty. 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 $v \in tree(u_i)$, 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 $S = \{u_1, u_2, \ldots, u_i\}$. $\square$

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, the first updating pass can be simplified to only include the tree rooted at the source vertex.

The GSS algorithm will perform well when a graph is made up of large tree structures; that is, $r \ll n$. For the same graph, Abuaiadh and Kingston's algorithm could take $O(m + n \log n)$ time to compute GSS since the worst case value for t is n. The worst case value of t is not as bad for conventional single-source,[3] taking at most $O(m + n \log r)$ time since t is at most $r + 1$. Applying tree decomposition with Abuaiadh and Kingston's concept of easy vertices produces a hybrid GSS algorithm with a worst case time complexity of $O(m + r \log t)$, where t is the number of easy trigger vertices resulting from r trigger vertices.

[3] In conventional single-source, a *delete_min* always occurs at the source vertex and all other non-triggers are encountered as easy vertices.

This new GSS algorithm can be applied in Takaoka's single source algorithm for acyclic graphs [5] when solving GSS on each SC component. This gives a time complexity of $O(m + r \log k)$, where k is the maximum number of trigger vertices in any single SC component, and r is the total number of trigger vertices in the graph.

3 Shortest Path Algorithms Using Acyclic Decompositions

In Section 2 the graph was decomposed into tree structures. The root vertex 'dominates' the tree since no vertex outside of the tree structure can update the shortest path of vertices in the tree without first updating the root vertex of the tree. This section generalises from decomposing the graph into trees to decomposing the graph into acyclic parts which are dominated by a single trigger vertex.

In order to describe how a graph can be decomposed into acyclic parts, we describe an algorithm which finds all vertices of an acyclic part. This algorithm uses a restricted depth first search, where the depth first search (DFS) at a vertex v is only allowed to proceed if all the incoming edges of v have been traversed. In this description, we say a vertex has been 'reached' if *any* of its incoming edges were traversed, but if *all* incoming edges of a vertex have been traversed, we say that the vertex has been 'unlocked'. Consider a restricted depth first search from a source vertex, v_0. Vertex v_0 can be considered a trigger vertex, and the vertices that are unlocked during the restricted depth first search are marked as non-triggers. All the non-triggers unlocked by the restricted depth first search belong to the acyclic part dominated by vertex v_0. Note that non-triggers do not have incoming edges originating from outside of the acyclic part. Thus, a vertex outside of the acyclic part cannot update the shortest path to a vertex inside the acyclic part without updating the shortest path to v_0.

In order to decompose the entire graph into such acyclic parts, several restricted depth first searches need to be performed. These depth first searches need to be performed so that the minimum number of trigger vertices for such an acyclic decomposition is obtained. Algorithm 5 presents an $O(mn)$ time algorithm for decomposing the entire graph into such acyclic parts.

Algorithm 5 Locate Dominators

/ Global Variables */*
Vertex v_0;
Vertex_Set L;

```
/* Restricted Depth First Search Function */
rdfs(v) {
  for each w in OUT(v) do {
    if w ≠ v_0 then {
      in_count[w] = in_count[w] − 1;
      if w is not in L then insert w in L;
      if in_count[w] == 0 then {
        is_trigger[w] = false;
        A_w = ∅;
        add w to A_{v_0};
        rdfs(w);
      }
    }
  }
}

/* Main Program */
for each v in V do {
  in_count[v] = |IN(v)|;
  is_trigger[v] = true;
  A_v = [v];
}
for each v in V do {
  if is_trigger[v] then {
    v_0 = v;
    L = ∅;
    rdfs(v);

    /* For visited vertices w, reset in_count[]. */
    for each w in L do in_count[w] = |IN(w)|;
  }
}
```

This algorithm cycles through all n possible source vertices v_0, and attempts the restricted depth first search for each of them. If a vertex v_0 has already been unlocked in a previous depth first search, then it is not used as the source for a new depth first search, since this would just be repeating a process that took place earlier. During each restricted depth first search, the array *in_count*[] keeps track of how many incoming edges have been traversed for each vertex. After each restricted depth first search, it is necessary to reset *in_count*[v] for each vertex v that was reached by the search, before a restricted depth first search from the next source can begin. To accomplish this efficiently, Algorithm 5 uses a set L to keep track of vertices that are reached during the current restricted depth first search. The set, L, is implemented using two one dimensional arrays, giving $O(1)$ time for the set operations *insert* and *find*.

For each vertex v, an ordered list A_v keeps track of the vertices in the acyclic part dominated by v. Initially each list A_v contains only vertex v. During the restricted DFS initiated from a vertex v_0, unlocked vertices w are added to A_{v_0}, and A_w is set to empty since w cannot be trigger vertex. A property of the unlocking DFS is that vertices are unlocked in topological order. Thus, the resulting lists of vertices are topologically sorted. At the end of this algorithm, each trigger vertex v, has a corresponding topologically sorted list A_v containing all vertices of the acyclic part dominated by v, including v itself. Non-trigger vertices have A_v set to empty.

A particular restricted depth first search can encounter a vertex v, which was used as a trigger in a previous depth first search, but has had all of its incoming edges traversed in this depth first search. This causes the re-traversal of vertices reached by the previous depth first search, which results in some inefficiency, but can allow the current depth first search to proceed deeper than the depth first search from the previous source allowed.

To summarise, Algorithm 5 determines a set of trigger vertices, where each trigger vertex u identifies an acyclic structure A consisting of u and all non-trigger vertices unlocked from u. We can then say that u dominates A since all paths from outside of A must pass through vertex u in order to reach any vertex inside A.

Theorem 2 *For a given graph, the decomposition is unique in terms of the set of acyclic parts. However, for a given acyclic part there may be a choice in trigger vertex.*

Proof There are n potential trigger vertices u, and each has associated with it a potential acyclic part B_u consisting of u and all vertices which can be unlocked from u. A valid decomposition is defined as a selection of trigger vertices such that no trigger vertex is contained within the acyclic part of another trigger vertex. That is, there is no overlapping of acyclic parts. Because of the 'unlocking' property of acyclic parts, there are only two possible cases of overlap that can occur among the n potential acyclic parts. Consider a vertex, v, inside a potential acyclic part B_u:

(1) B_v does not contain vertex u, and B_v is completely contained within B_u.
(2) (special case): B_v contains vertex u, so B_v and B_u are essentially the same acyclic part, apart for the choice of trigger vertex.

For any instances of Case 1 in the graph, v is eliminated as a potential trigger vertex, since it is forced to be a non-trigger vertex in the acyclic part B_u. In a valid decomposition consisting of trigger vertices v and corresponding acyclic parts B_v, Case 1 cannot apply since there won't be any such vertex u and acyclic part B_u containing v. A valid decomposition is set-wise unique since there is no choice over these Case 1 situations. Case 2 only applies if Case 1 does not. Therefore, Case 2 situations can only occur for trigger vertices u and corresponding acyclic parts B_u that are part of a valid decomposition. Any instances of Case 2 in the graph, allow a choice for which trigger vertex to use in the acyclic part, but do not alter the set-wise uniqueness of the decomposition. $\square$

For a given graph, there is only one possible set of acyclic parts resulting from this type of decomposition. A *valid* trigger vertex is a vertex that can be used as the trigger of an acyclic part in the decomposition. Algorithm 5 considers all possible trigger vertices for the source in an unlocking DFS, including valid trigger vertices. Eventually, the unlocking DFSs that are started from valid trigger vertices will cause any invalid trigger vertices to be marked as non-triggers. When there is more than one valid trigger vertex for an acyclic part, the one which was first encountered (as the source of an unlocking DFS) is used. Let r be the number of trigger vertices resulting from such a decomposition. The value of r is fixed for any given graph, and Algorithm 5 in effect minimises n potential trigger vertices down to a selection of r valid trigger vertices.

Algorithm 5 takes $O(mn)$ time to compute the set of acyclic parts and trigger vertices. This exceeds the time complexity of single-source shortest path algorithms, including GSS. However, a selection of trigger vertices obtained using Algorithm 5 is useful when solving the all-pairs problem by repeating n single-source problems.

Using the selection of trigger vertices v and acyclic parts A_v found using Algorithm 5, the shortest path algorithm can update shortest path distances through each acyclic part independently using the topologically ordered lists. Algorithm 6 shows an all-pairs shortest path algorithm which uses this idea.

Algorithm 6 All-Pairs Algorithm Using Topologically Ordered Acyclic Parts

/ Global Variables */*
Vertex_Set L;

/ Scan distance updates through the acyclic part of trigger vertex u */*
$update(u)$ **{**
  for each vertex v in order from list A_u do **{**
    for each w in $OUT(v)$ **do {**
      $d[w] = min(d[w], d[v] + c(v, w));$
        / If w is a trigger vertex a decrease_key operation may occur. */*

```
      }
    }
}

/* Main Program */
for each v_0 in V do {
    let d be a reference to row v_0 of array D;
    for each vertex v do d[v] = ∞;
    d[v_0] = 0;
    S = ∅;
    insert all trigger vertices into F;
    if not is_trigger[v_0] then {
        let u_0 be the trigger vertex of the acyclic part containing v_0.
        update(u_0);
    }
    while F is not empty do {
        select u such that d[u] is the minimum among u in F; /* delete_min */
        remove u from F;
        add u to S;
        update(u);
    }
}
```

Distance updates through an acyclic part of trigger vertex u are initiated by calling *update*(u). This function scans the vertices v of A_u in topological order, updating the shortest path distances to vertices in $OUT(v)$. The position of each vertex v in the topological order ensures that all possible updates to $d[v]$ have occurred before distance updates for vertices in $OUT(v)$ occur. Thus, the order of distance updates is correct. As with other shortest path algorithms, F is the frontier set, and S is the solutions set. To simplify the description of Algorithm 6, F initially contains all trigger vertices. However, the algorithm can easily be modified so that trigger vertices v are inserted into F the first time an update occurs to $d[v]$. This will not change the worst case time complexity but may offer a constant factor improvement since the time taken by *delete_min* depends on the number of vertices in the frontier set.

Now consider solving one single-source problem from a source vertex, v_0. In Algorithm 6, the two dimensional array $D[v, u]$ is used for storing the shortest path distance from v to u. For the current source vertex, the one dimensional array notation $d[v]$ refers to $D[v_0, v]$. Initially, $d[v_0] = 0$, and $d[v] = \infty$ for all vertices $v \neq v_0$. If v_0 is a non-trigger vertex, the shortest path algorithm first determines the trigger vertex, u_0, of the acyclic part that contains v_0 and calls *update*(u_0) to start distance updates from v_0. To solve the rest of the single-source problem, only the trigger vertices need to be placed in a frontier set and considered for *delete_min* operations. After a *delete_min* operation selects the minimum trigger vertex, u, the shortest path distances through A_u are updated by calling $update(u)$. Then the next *delete_min* operation occurs, and so on, until the frontier set is empty. For cases where v_0 is a non-trigger vertex, distance updates through A_{v_0} are eventually completed when $update(u)$ occurs with u corresponding to u_0.

The correctness proof of Algorithm 6 is similar to the GSS algorithm presented in Section 2 which selects trigger vertices as the roots of trees in the graph. If a Fibonacci heap or equivalent data structure is used for F, the time complexity to solve one single source problem in Algorithm 6 is $O(m+r\log r)$, where r is the number of trigger vertices (or dominators) in the graph. Overall, the time complexity of this improved all-pairs shortest path algorithm is $O(mn+nr\log r)$, which includes the time taken to determine trigger vertices. It remains to be seen if the decomposition algorithm can be improved to be within the time complexity of the corresponding single source algorithm.

A variation of Algorithm 6 is possible which follows an unlocking DFS instead of scanning topologically ordered lists. This would eliminate the need for maintaining topologically ordered lists, but maintaining the *in_count*[] array would increase the computational overhead of Algorithm 6.

4 Shortest Path Algorithms Using Feedback Vertex Sets

The shortest path algorithms of Sections 2 and 3 select trigger vertices based on trees and other acyclic structures in the graph. This allows shortest path distances through the acyclic parts of the graph to 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.

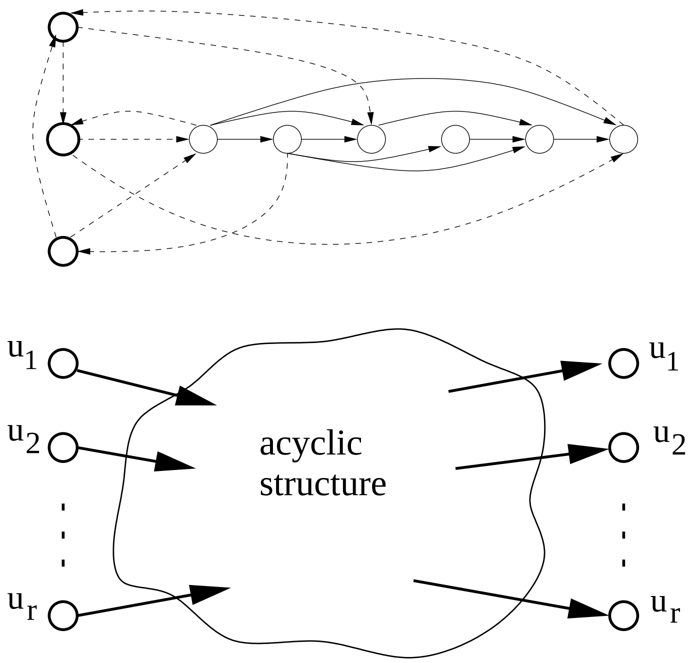

Fig. 2. Example of identifying the graph structure as consisting of trigger vertices and an acyclic part.

The new all-pairs algorithm consists of two stages. Algorithm 7 shows the first stage, and Algorithm 8 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 7. 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.[4] 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 7 First Stage of the New All-Pairs Algorithm

Topologically sort vertices in $\overline{T}$, placing the result into the ordered set L.
for each vertex v_0 in V **do {**
  let d be a reference to row v_0 of array D;
  for each vertex v in V **do** $d[v] = \infty$;
  $d[v_0] = 0$;
  if v_0 is in T **then for** each vertex w in $OUT(v_0)$ **do** $d[w] = c(v_0, w)$;
  for each vertex v in order from L **do {**
    for each vertex w in $OUT(v)$ **do {**
      $d[w] = min(d[w], d[v] + c(v, w))$;
    }
  }
  if v_0 is in T **then {**
    for each vertex u in T with $d[u] \neq \infty$ **do {**
      add edge (v_0, u) with cost $d[u]$ to P;
    }

[4] 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 8). Other first-tentative distances are not important since the same computation can occur during Algorithm 8.

```
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 7, $O(m)$ 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 $u \epsilon T$ and $v \epsilon T$:

- 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 8) completes the all-pairs shortest path computation. Note that distance values from Algorithm 7 are retained in D and used in Algorithm 8. This is important in the correctness of Algorithm 8. The second stage of this all-pairs algorithm can be viewed as repeating the second stage of each single-source problem. For each $v_0 \epsilon V$:

(1) Let $d_1[v_0, u]$ correspond to the value of $D[v_0, u]$ at the end of Algorithm 7. For vertices $u \epsilon T$, 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 $u \epsilon T$, 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 $u_i \epsilon T$ (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 1.2 and Takaoka [5].

(2) The finalised shortest path distances of the form $D[v_0, u]$, where $u \in T$, are then used in calculating shortest path distances of the form $D[v_0, v]$ for vertices $v \in \overline{T}$. A distance $d[v]$, for $v \in \overline{T}$, 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 $u_i \in T$ (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 $v \in \overline{T}$ 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 8 Second Stage of the New All-Pairs Algorithm

```
18. for each vertex v0 in V do {
19.    let d be a reference to row v0 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[v0, 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 8 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 $v \in T$ 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 $v_0 \in V$, 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 [5]. 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.

5 Concluding Remarks

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 [4] and Takaoka [5]. It is possible to combine this new algorithm and the previous algorithms into a hybrid algorithm which incorporates the properties of each.

Extending the concept of trigger vertices, r can be defined as the number of trigger vertices that result from decomposing the graph into acyclic structures. Future work will investigate more general forms of this kind of acyclic decomposition. At present the $O(mn)$ time spent decomposing a graph into acyclic parts will only allow an improved all-pairs algorithm with a time complexity of $O(mn+nr\log r)$. Future work will investigate whether it is possible to achieve this decomposition within the time bounds of a single source algorithm.

In the most general definition of trigger vertices, r can be defined as the number of vertices in a pre-calculated feedback vertex set for the nearly acyclic graph. This allows a new all-pairs algorithm algorithm with $O(mn+nr^2)$ worst case time complexity. In most cases of nearly acyclic graphs, r is less than than $\sqrt{m}$, 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 calculates a, not necessarily minimum, feedback vertex set. Another consideration is whether an efficient single-source algorithm can be devised which uses any supplied feedback vertex set.

Acknowledgements

We would like to thank Jeffrey H. Kingston for providing us with the related work in the thesis of Diab Abuaiadh [6]. Abuaiadh's thesis gave us a more general insight into the concept of using graph decompositions in solving shortest path problems. We also acknowledge the referees whose constructive comments improved the quality of this paper.

References

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

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

[3] A. Gibbons, Algorithmic Graph Theory, Cambridge University Press, 1985.

[4] D. Abuaiadh, J. Kingston, Are Fibonacci heaps optimal?, in: ISAAC '94, Lecture Notes in Computer Science, 1994, pp. 41–50.

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

[6] D. Abuaiadh, On the complexity of the shortest path problem, Ph.D. thesis, Basser Department of Computer Science, University of Sydney, Australia (Jul. 1995).

[7] D. Abuaiadh, J. Kingston, Efficient shortest path algorithms by graph decompositon, Tech. rep., Basser Department of Computer Science, University of Sydney, Australia, Technical Report 93-475 (1994).

[8] T. Takaoka, Theory of 2-3 heaps, in: Proc. COCOON '99, Vol. 1627 of Lecture Notes in Computer Science, 1999, pp. 41–50.

[9] T. Takaoka, Theory of trinomial heaps, in: COCOON '00, Vol. 1858 of Lecture Notes in Computer Science, 2000, pp. 362–372.

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