Efficient Algorithms for Solving Shortest Paths on Nearly Acyclic Directed Graphs

Shane Saunders **Tadao Takaoka**

Department of Computer Science and Software Engineering,
University of Canterbury, Christchurch, New Zealand,
Email: sas59@student.canterbury.ac.nz (Shane Saunders),
Email: tad@cosc.canterbury.ac.nz (Tadao Takaoka)

Abstract

This paper presents new algorithms for computing shortest paths in a nearly acyclic directed graph $G = (V, E)$. The new algorithms improve on the worst-case running time of previous algorithms. Such algorithms use the concept of a 1-dominator set. A 1-dominator set divides the graph into a unique collection of acyclic subgraphs, where each acyclic subgraph is dominated by a single associated trigger vertex. The previous time for computing a 1-dominator set is improved from $O(mn)$ to $O(m)$, where $m = |E|$ and $n = |V|$. Efficient shortest path algorithms only spend delete-min operations on trigger vertices, thereby making the computation of shortest paths through non-trigger vertices easier. Under this framework, the time complexity for the all-pairs shortest path (APSP) problem is improved from $O(mn + nr \log r)$ to $O(mn + r^2 \log r)$, where r is the number of triggers. Here the second term in the complexity results from delete-min operations in a heap of size r. The time complexity of the APSP problem on the broader class of nearly acyclic graphs, where trigger vertices correspond to any precomputed feedback vertex set, is similarly improved from $O(mn + nr^2)$ to $O(mn + r^3)$. This paper also mentions that the 1-dominator set concept can be generalised to define a bidirectional 1-dominator set and k-dominator sets.

1 Introduction

A directed graph $G = (V, E)$ consists of a set of vertices V, and a set of directed edges E. An edge $e \in E$ is a directed connection $v \rightarrow w$ from one vertex $v \in V$ to another vertex $w \in V$, and has an associated cost $c(e)$. For simplicity, it is assumed that $c(e) \geq 0$. Any path connecting a pair of vertices has an associated distance which corresponds to the sum of the costs of edges that make up the path. The existence of alternative paths between a pair of vertices provides the possibility of some paths being shorter than others in terms of their associated distance. This results in the problem of determining which paths are the shortest. For further description of algorithm and graph theory terms, refer to Gibbons (1985).

Dijkstra's algorithm (Dijkstra 1959) solves the single source shortest path problem on a non-negatively weighted directed graph in $O(m + n \log n)$ worst-case time[1] when applied in conjunction with a Fibonacci heap (Fredman & Tarjan 1987) or 2-3 heap (Takaoka 1999), where m accounts for distance updates from edges, and n accounts for delete-min operations in the heap. The time complexity provided by Dijkstra's algorithm applies to any non-negatively weighted directed graph in general. However, for some classes of directed graphs, such as limited edge cost graphs, planar graphs, and nearly acyclic graphs, it is possible to improve upon the time complexity offered by Dijkstra's algorithm by using a specialised shortest path algorithm. Applying Dijkstra's algorithm from all n possible source vertices solves the all-pairs problem in $O(mn+n^2 \log n)$ time for a graph in general. However, as in the case of single-source, it is possible to achieve a lower time complexity when solving all-pairs on special-case graphs by using specialised algorithms on such graphs.

The specialised shortest path algorithms presented in this paper offer improved efficiency when a directed graph is *nearly acyclic*. Nearly acyclic graphs contain very few cycles relative to the number of vertices they contain. This results in large underlying acyclic subgraphs, within which shortest paths can be computed efficiently. Several shortest path algorithm for nearly acyclic directed graphs have appeared previously (Abuaiadh & Kingston 1994, Abuaiadh 1995, Takaoka 1998, Saunders & Takaoka 2003). The previous publication by Saunders & Takaoka (2003) provides a survey of these algorithms. New algorithms presented in Saunders & Takaoka (2003) introduced the concept of trigger vertices, from which shortest paths can be computed efficiently through underlying acyclic regions in a graph. A graph decomposition, which is now called the 1-dominator set, was devised for specifying such acyclic regions. A 1-dominator set divides a graph into a unique collection of acyclic subgraphs such that any single acyclic subgraph is dominated by a single associated trigger vertex. This provided a single-source algorithm with a time complexity of $O(m + r \log r)$, where r is defined as the number of trigger vertices in the 1-dominator set. If r is small, then the graph is regarded as being nearly acyclic. Saunders and Takaoka also gave a framework which defined trigger vertices more generally as feedback vertices, providing an all-pairs algorithm with a time complexity of $O(mn + nr^2)$, where r is the number of such trigger vertices. Such a definition for trigger vertices is able to encompass a wide range of acyclic structures within a graph.

Extending upon the previous publication by Saunders and Takaoka, this paper introduces new algorithms for providing efficient computation of shortest paths on nearly acyclic graphs. A new all-pairs algorithm is given which uses a 1-dominator set to achieve a time complexity of $O(mn + r^2 \log r)$, improving on the earlier $O(mn + nr \log r)$ algorithm. Here r is the number of trigger vertices in either the monodirectional or bidirectional 1-dominator set of a

[1] The remainder of this paper assumes worst-case time complexities, unless stated otherwise.

graph. The bidirectional 1-dominator set extends the previously presented monodirectional 1-dominator set by allowing acyclic structures to be defined in both directions over edges in the graph. By capturing a more general form of acyclicity, the bidirectional 1-dominator set provides a potentially smaller value for r. Furthermore, the bidirectional 1-dominator set, like its monodirectional counterpart, is set-wise unique for a given graph. The previous algorithm for computing the monodirectional 1-dominator set required $O(mn)$ time in the worst case. A new, improved, algorithm now allows the monodirectional 1-dominator set to be computed in $O(m)$ time. This can be integrated as part of the $O(m + r \log r)$ time required to solve single-source using a 1-dominator set. Under the more flexible definition of trigger vertices, where r is defined as the number of vertices in a precomputed feedback vertex set, the new all-pairs algorithm achieves a time complexity of $O(mn + r^3)$, providing a significant improvement on the earlier $O(mn + nr^2)$ algorithm.

As a starting point, Section 2 provides a mathematical definition for 1-dominator sets. A new algorithm for computing the 1-dominator set of a graph in $O(m)$ time is then presented in Section 3. This is followed by a description of the new shortest path algorithms in Section 4. Final concluding remarks are given in Section 5.

2 A Mathematical Definition for Acyclic Decomposition

This section reviews the acyclic decomposition method previously presented by (Saunders & Takaoka 2003). A mathematical definition for the decomposition is presented, in contrast to the earlier work which defined the decomposition algorithmically. Additionally, the acyclic decomposition is extended to allow acyclic structures defined in both the incoming and outgoing directions from vertices.

Acyclic structures can be defined as either *partial* or *complete*. To distinguish the two forms, partial acyclic structures are denoted using a primed notation, as in A'_u. An acyclic structure can be defined in either the forward or backward direction of edges in the graph. The definition for *forward* acyclic structures will be given first. A forward partial acyclic structure A'_v is defined in the forward direction from some vertex v in the graph, with vertex v 'dominating' all vertices of A'_v. Such an acyclic structure satisfies the following mathematical properties:

- $v \in A'_v$
- $A'_v - \{v\}$ is acyclic; that is, the vertices in $A'_v - \{v\}$ induce an acyclic graph.
- All $w \in A'_v - \{v\}$ satisfy $IN(w) \subseteq A'_v$ and $IN(w) \neq \emptyset$.

Here the notation $IN(w)$ is used to denote the set of vertices that are adjacent to w via incoming edges of w. Intuitively speaking, vertices $w \in A'_v$ can only be reached from outside A'_v by paths through vertex v. A partial acyclic structure A'_v is not necessarily complete. The complete acyclic structure A_v defined in the forward direction from a vertex v must satisfy the additional requirement:

- $w \in A_v$ for all w that satisfy $IN(w) \subset A_v$.

This rule ensures that a complete acyclic structure A_v contains all vertices that are 'dominated' by vertex v. A symmetric definition provides partial and complete *backward* acyclic structures B'_v and B_v respectively; just replace A by B and $IN(w)$ by $OUT(w)$.

In general, let a complete acyclic structure defined on a dominating vertex v be denoted as Φ_v. If only *monodirectional* acyclic structures are being considered then, depending on whether forward or backward acyclic structures are being used, Φ_v can be defined as either $\Phi_v \equiv A_v$ or $\Phi_v \equiv B_v$. In the case of *bidirectional* acyclic structures, Φ_v takes the definition $\Phi_v \equiv A_v \cup B_v$.

With any vertex v in the graph having a corresponding complete acyclic structure, there will be some complete acyclic structures that are contained as subsets of other complete acyclic structures. This is defined in the following theorem:

Theorem 1. *If $v \in \Phi_u$ then $\Phi_v \subseteq \Phi_u$.*

Proof. A detailed proof has been omitted because of space constraints, but can be seen by noting that any vertex which can be included into Φ_v can also be included into Φ_u on the basis that $v \in \Phi_u$. □

As a result of Theorem 1, there will exist some acyclic structures Φ_u that cannot be contained as a subset of any other acyclic structure. Such acyclic structures are referred to as *maximal* acyclic structures, and satisfy the following additional requirement:

- $\Phi_u \subset \Phi_v$ does not hold for any Φ_v.

Thus, among the collection of all complete acyclic structures within a graph, there will exist acyclic structures that are maximal. The set of all such acyclic structures is referred to as the 1-dominator set.

For the purpose of precisely defining the 1-dominator set, let the set Q be defined as:

$$Q = \{\ \Phi_u \mid \Phi_u \text{ is maximal } \}$$

Thus, the set Q contains all maximal acyclic structures. Some maximal acyclic structures may be duplicated in Q, corresponding to cases where $\Phi_v = \Phi_u$ for two different dominating vertices v and u. The 1-dominator set R contains all the acyclic structures in Q, except for any duplicates. For the monodirectional 1-dominator set, R forms a disjoint set of acyclic structures covering the whole graph, and thus is a decomposition. In the case of bidirectional 1-dominator sets, the forward and backward parts of different acyclic structures may overlap.

The vertex u used for denoting a maximal acyclic structure $\Phi_u \in R$ is referred to as a *trigger* vertex. Later, it will be seen that this vertex 'triggers' shortest path distance updates through other vertices in Φ_u. In cases where there exists a duplicate acyclic structure $\Phi_v = \Phi_u$ for some $\Phi_v \in Q$, the placement of Φ_u in R expresses u as the trigger, as opposed to the *alternative trigger* vertex v.

The set Q, which contains all maximal acyclic structures, constitutes a unique set of acyclic structures. The removal of duplicates from Q results in the unique set R since duplicate maximal acyclic structures are indistinguishable in terms of the vertices they contain. Thus, the 1-dominator set R is set-wise unique for a given graph. Only the choice among alternative trigger vertices of an acyclic structure is non-unique.

A generalisation of the 1-dominator set concept defines a more complex form of dominator sets, called k-dominator sets, in which each acyclic structure is dominated by up to k vertices. A complete definition for k-dominator sets appears in the Ph.D. thesis of Shane Saunders (Saunders 2004).

3 Acyclic Decomposition Algorithms

An algorithm for computing monodirectional 1-dominator sets in $O(mn)$ worst-case time was previously presented by Saunders & Takaoka (2003). This section presents an improved algorithm which spends just $O(m)$ worst-case time.

For descriptive purposes, let *acyclicSetA*(u) denote a function which returns a vertex set A containing the vertices of the acyclic structure A_u. This function can be implemented using the restricted depth first search approach previously described in Saunders & Takaoka (2003). A restricted depth first search maintains a value *inCount*$[v]$ for each vertex v, which is the number of untraversed incoming edges of v. If *inCount*$[v]$ becomes zero, then v is said to be unlocked and the search proceeds forward. The 1-dominator set is computed by marking vertices as either *trigger*, *nontrigger*, or *undefined*. Initially, all vertices are marked as *undefined*, identifying themselves as untraversed. The acyclic structures of the 1-dominator set are identified by maintaining reference values $AC[v]$ which refer to the last acyclic set found to contain vertex v. After calling *acyclicSetA*(u) from an untraversed vertex u, the value of $AC[v]$ is assigned to refer to the computed vertex set A for all vertices $v \in A$. Additionally, all vertices contained in A are marked as non-triggers, except for vertex u, which is marked as a trigger. In order for all maximal acyclic structures to be computed, an algorithm must call *acyclicSetA*(u) from any vertex u that remains untraversed, and mark the traversed vertices as triggers or non-triggers accordingly. In this way, all non-maximal acyclic structures are eventually erased, leaving just the maximal acyclic structures which represent the 1-dominator set.

Following the previous approach, one can compute the 1-dominator set simply by considering untraversed vertices u in any arbitrary order for calling *acyclicSetA*(u). However, this results in $O(mn)$ worst-case time since each call *acyclicSetA*(u), taking up to $O(m)$ time, may re-traverse vertices traversed during earlier calls. A more efficient approach is to only initiate calls *acyclicSetA*(u) from vertices bordering previously computed acyclic structures. This takes advantage of the 1-dominator set property that A_w is maximal for any vertex w bordering another maximal acyclic structure A_u. Thus, if a maximal acyclic structure A_u is computed by calling *acyclicSetA*(u), then other maximal acyclic structures A_w can be computed by calling *acyclicSetA*(w) from vertices w bordering A_u, and so forth until all reachable vertices have been traversed.

In order to support this more efficient approach, boundary vertices must be computed along with acyclic structures. This is achieved using a modified version of the function *acyclicSetA*(v), called *bAcyclicSetA*(v), which returns the set of boundary vertices D, in addition to the computed acyclic structure A. The complete function is shown as Algorithm 1. Any vertex visited by *bAcyclicSetA*(v) that is not included into the resulting acyclic structure A, will be a boundary vertex of A. Thus, the set of boundary vertices D is easily computed as $D \leftarrow L - A$, where L is the set of all vertices visited.

Algorithm 1. Function for Computing A_v and Boundary Vertices

```
1.  function bAcyclicSetA(u) {
2.      VertexSet A, L;
3.      procedure rdfs(v) {
4.          A ← A + {v};
5.          for each w ∈ OUT(A) do {
6.              if w ∉ L then L ← L + {w};
                    // w visited
7.              inCount[w] ← inCount[w] − 1;
8.              if inCount[w] = 0 then rdfs(w);
                    // w unlocked
9.          }
10.     }
11.     A ← ∅; L ← {u};
12.     inCount[u] ← inCount[u] + 1;
          // prevents re-traversal of u
13.     rdfs(u);
14.     for each w ∈ L do inCount[w] ← |IN(w)|;
15.     VertexSet D ← L − A;
          // boundary vertices
16.     return (A,D);
17. }
```

Algorithm 2 presents the overall algorithm, which calls *bAcyclicSetA*(v) and explores the resulting boundary vertices. An array entry *vertexType*$[v]$ is used for the purpose of storing the marking of vertex v. The algorithm maintains a queue Q containing boundary vertices to be considered. Initially, the starting vertex s is placed in Q, as no boundary vertices are known. The algorithm proceeds by removing a vertex u from Q. If u has not already been put in an acyclic structure, then the function *bAcyclicSetA*(u) is called to determine the associated acyclic structure A and set of boundary vertices D. The algorithm then assigns the appropriate values to $AC[v]$ and *vertexType*$[v]$ for all vertices v contained in A. Then, each boundary vertex v in D that has not already been put in an acyclic structure is placed in Q, provided that v is not already in Q. This process continues until all located boundary vertices have been exhausted from Q. At this point, all reachable vertices in the graph will have been traversed, and the maximal acyclic structures defined on these vertices determined.

Algorithm 2. Computing the 1-Dominator Set

```
1.  for all v ∈ V do inCount[v] ← |IN(v)|;
2.  for all v ∈ V do vertexType[v] ← unknown;
3.  Q ← {s};
4.  while Q ≠ ∅ do {
5.      Remove the next vertex u from Q;
6.      if vertexType[u] = unknown then {
7.          (A, D) ← bAcyclicSetA(u);
8.          for each v ∈ A do
              let AC[v] refer to A;
9.          for each v ∈ A do
              vertexType[v] ← nontrigger;
10.         vertexType[u] ← trigger;
11.         for each v ∈ D do {
12.             if vertexType[v] = unknown
                  and v ∉ Q then Add v to Q;
13.         }
14.     }
15. }
```

When the algorithm begins, it is unknown whether the starting vertex s is a trigger. In cases where the starting vertex s is a non-trigger, non-maximal acyclic structures will be computed until a trigger vertex is hit. It can be shown that any non-maximal acyclic structure computed will consume only untraversed vertices. Furthermore, all non-maximal acyclic structures computed will belong to the same maximal acyclic structure A_x for some vertex x. In cases where the graph, or subgraph being considered, is strongly connected this trigger vertex x will eventually be hit. At that point, such non-maximal acyclic structures will be erased by the computation of A_x. Since the computation of A_x is the only occurrence of re-traversal, no edge and vertex in the graph is

traversed more than twice. Hence, the algorithm's running time is at worst $O(m)$.

For a strongly connected (SC) graph, all vertices will be reachable from any arbitrary starting vertex s, and all maximal acyclic structures in the graph will be computed in $O(m)$ worst-case time. For other graphs, it is necessary to repeatedly apply this process from each untraversed SC component in the graph in topological order. Let the notation $cover(s)$ denote exactly the same process used by lines 3 to 15 of Algorithm 2. A single call $cover(s)$ will determine the maximal acyclic structures contained within the current SC component and any reachable downstream SC components that still remain untraversed. By calling $cover(s)$ for any SC component remaining untraversed, the whole graph will be considered, and all maximal acyclic structures in the graph will be computed. The combined time of all calls $cover$(s) is at most $O(m)$ since each call only traverses vertices that were not encountered by previous calls. Furthermore, the SC components of the graph, and their topological ordering, can be determined in $O(m)$ worst-case time using Tarjan's algorithm. In summary, the 1-dominator set of any graph can be computed in $O(m)$ worst-case time.

The respective algorithms for computing the backward 1-dominator set are obtained by replacing the function $acyclicSetA(v)$ with a symmetric function $acyclicSetB(v)$. This approach has been extended to compute bidirectional 1-dominator sets in $O(m)$ worst-case time. One such approach is to merge a graph's forward and backward 1-dominator sets. The algorithms for computing bidirectional 1-dominator sets are not given in this paper.

4 Shortest Path Algorithms Using Acyclic Decompositions

This section presents a general framework for using acyclic decompositions to compute shortest paths efficiently. In general, an acyclic decomposition consists of a set of trigger vertices T, and a set of vertices $\overline{T} = V - T$ that forms a subgraph constituting a large acyclic region within G. This different definition for acyclic decomposition by trigger vertices is equivalent to that of the feedback vertex set; that is, the graph is decomposed into a feedback vertex set T and its associated acyclic region $\overline{T}$. The shortest path algorithms presented in this section are able to apply any precomputed feedback vertex set, including the trigger vertices offered by 1-dominator sets, in order to provide efficient computation of shortest paths.

To begin with, the vertices contained in the acyclic region $\overline{T}$ are topologically sorted. A topological ordering of these vertices can easily be computed in $O(m)$ time. This topological ordering is used to achieve efficient computation of shortest paths that involve vertices in $\overline{T}$. In the case of the 1-dominator sets, this topological ordering will have been produced as a by-product of computing acyclic structures.

In order to compute shortest paths efficiently, shortest paths between vertices in T via only vertices in $\overline{T}$ are computed. These shortest paths become edges in a reduced graph P, whose vertices correspond to vertices in T. The cost $c(u, v)$ of an edge $u \rightarrow v$ in P is defined as the cost of the shortest path of the form $u \rightsquigarrow v$ for $u \in T$ and $v \in T$. Here the notation $u \rightsquigarrow v$ is used to denote paths of the form $u, v_1, v_2, \ldots, v_k, v$, where $k \geq 0$ and $v_i \in \overline{T}$ for all $1 \leq i \leq k$. The case where $k = 0$ allows the possibility of $u \rightsquigarrow v$ denoting a single edge directly from u to v in the original graph. An edge $u \rightarrow v$ is created in P if and only if there exists a path of the form $u \rightsquigarrow v$.

In total, $O(mr)$ time is needed when computing P from a general feedback vertex set. This is easily contained within the time complexity required to compute all-pairs. The time required for computing P from the monodirectional or bidirectional 1-dominator is at worst $O(m)$ by scanning each acyclic structure exactly once, and can be integrated as part of the time complexity required for computing single-source.

By computing P and solving shortest paths on P, shortest paths in the whole graph can be computed more efficiently. The previous approach for using P provided an all-pairs algorithm with a time complexity of $O(mn + m'n + nr \log r)$, where r is the number of trigger vertices and m' is the number of edges in P. For a general set of feedback vertices, m' is at worst $O(r^2)$, and the time complexity simplifies to $O(mn + nr^2)$. In contrast, a pseudo-graph P computed from a 1-dominator set has $m' \leq m$, giving a time complexity of $O(mn + nr \log r)$. Interestingly, this result also applies for the new bidirectional 1-dominator set, which allows a potentially smaller value for r. Furthermore, the mono- and bi-directional 1-dominator set both support the solving of single-source in $O(m + r \log r)$ time by computing P in just $O(m)$ time.

Previously, a generalised form of the single-source problem was solved on P. The new all-pairs algorithm presented here takes the approach of solving an all-pairs problem on the reduced graph P. With P containing r vertices and m' edges, the time required to compute all-pairs on P is at worst $O(m'n + r^2 \log r)$ by using Dijkstra's algorithm with a Fibonacci heap or similarly efficient data structure. This determines the shortest path between any pair of trigger vertices. As will be shown, it is then possible to complete the solution to the all-pairs problem on the whole graph in $O(mn)$ time. Overall, all-pairs is solved in $O(mn + m'r + r^2 \log r)$ time; improving on the earlier $O(mn + m'n + nr \log r)$ result.

The new approach is presented as Algorithm 3. Within this algorithm, the result of solving all-pairs on P is represented in an all-pairs matrix M such that $M[x, u]$ denotes the shortest path from x to u for any pair of vertices $x \in T$ and $u \in T$. For descriptive purposes, let $\lambda[v, u]$ be used to denote the hypothetical shortest path distance from vertex v to vertex u. The goal of the algorithm is to compute $\lambda[v, u]$ for every pair of vertices $u \in V$ and $v \in V$. To achieve this, the algorithm first considers each vertex $u \in T$, and determines $\lambda[v, u]$ for all vertices $v \in V$. This is done by backtracking from u, first to other trigger vertices $x \in T$ by consulting array entries $M[x, u]$, and then on to vertices $v \in \overline{T}$ in reverse-topological order. Within this computation, $M[x, u]$ specifies the shortest path distance $\lambda[x, u]$ for vertices $x \in T$. The shortest path distance $\lambda[v, u]$ for vertices $v \in \overline{T}$ is computed by *pulling* shortest path distances from vertices $x \in T$ through the reverse topological order of $\overline{T}$; see $rpull(v)$. This process spends at most $O(m + r) = O(m)$ time for each vertex u. Thus, this process spends $O(mr)$ total time to determine $\lambda[v, u]$ for all $v \in V$ and $u \in T$.

Algorithm 3. Using a Feedback Vertex Set to Solve All-pairs Efficiently

```
1.  procedure pull(v) {
2.      for each w ∈ IN(v) do
            d[v] = min(d[v], d[w] + c(w, v));
3.  }
4.  procedure rpull(v) { // reverse pull
5.      for each w ∈ OUT(v) do
            d[v] = min(d[v], c(v, w) + d[w]);
```

6. **}**
   // Start of Algorithm
7. Assume that T is a set of feedback vertices and $\overline{T}$ is the acyclic part;
8. Compute the reduced graph P;
   // $O(mr)$ time
9. Solve all-pairs on P such that $M[v, u]$ denotes the shortest path from any $v \in T$ to any $u \in T$;
   // $O(m'r + r^2 \log r)$ time
10. Let $D[v, u] = \infty$ for all $v \in V$ and $u \in V$;
    // Compute shortest paths to each trigger
    // vertex u. ($O(mr)$ time)
11. **for** each $u \in T$ **do {**
12. Let $d[v]$ act as a reference to array entries $D[v, u]$;
13. $d[u] = 0$;
14. **for** each $x \in T$ **do** $d[x] = M[x, u]$;
15. **for** each $v \in \overline{T}$ in reverse-topological order **do** *rpull*(v);
16. **}**
    // Finish shortest paths from all
    // source vertices v_0. ($O(mn)$ time)
17. **for** each $v_0 \in V$ **do {**
18. Let $d[v]$ act as a reference to array entries $D[v_0, v]$;
19. $d[v_0] = 0$;
20. **for** each $v \in \overline{T}$ in topological order **do** *pull*(v);
21. **}**

The final stage of Algorithm 3 completes the computation by determining $\lambda[v_0, v]$ for all source vertices $v_0 \in V$ and all vertices $v \in \overline{T}$. This is done by considering each source vertex v_0, and pulling distances $\lambda[v_0, u]$, which were computed for trigger vertices $u \in T$, onto, and through, non-trigger vertices $v \in \overline{T}$ in topological order; see *pull*(v). With this final stage completed, $\lambda[v, u]$ will have been determined for all pairs of vertices $v \in V$ and $u \in V$. Hence, by using Algorithm 3, all-pairs is solved on G in $O(mn + m'r + r^2 \log r)$ total time.

For a general feedback vertex set, the value of m' is bounded by r^2, and the time complexity of the algorithm becomes $O(mn + r^3)$. In contrast, a feedback vertex set arising from the trigger vertices of the 1-dominator set provides value of m' that is bounded by m, giving the algorithm a time complexity of $O(mn + r^2 \log r)$. This time complexity applies for both the mono- and bi-directional 1-dominator set. In the $O(mn + r^3)$ case, it is acceptable to solve the all-pairs problem on P in $O(r^3)$ time, which can be done using Floyd's algorithm instead of using Dijkstra's algorithm, without affecting the overall time complexity.

5 Concluding Remarks

This paper has provided a more efficient algorithm for computing shortest paths on nearly acyclic directed graphs. In general, all-pairs can be solved in $O(mn)$ time when a feedback vertex set of size $r \leq \sqrt[3]{mn}$ can be computed in advance. Recent results presented by Pettie (2004) allow the solving of all-pairs on real-weighted directed graphs in $O(mn + n^2 \log\log n)$ time under the *comparison-addition* model. Applying the reduced-graph framework in conjunction with this state-of-the-art result provides a further improved all-pairs time complexity of $O(mn + r^2 \log\log r)$ where r is the number of trigger vertices in the 1-dominator decomposition of a graph. Finally, some future all-pairs algorithm may improve the current $O(mn + r^3)$ time bound to $O(mn + r^2 \log r)$ for a precomputed set of r feedback vertices.

References

Abuaiadh, D. (1995), On the complexity of the shortest path problem, PhD thesis, Basser Department of Computer Science, University of Sydney, Australia.

Abuaiadh, D. & Kingston, J. (1994), Are Fibonacci heaps optimal?, *in* 'ISAAC '94', Lecture Notes in Computer Science, pp. 41–50.

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

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

Gibbons, A. (1985), *Algorithmic Graph Theory*, Cambridge University Press.

Pettie, S. (2004), 'A new approach to all-pairs shortest paths on real-weighted graphs', *Theoretical Computer Science* **312**(1), 47–74.

Saunders, S. (2004), Improved Shortest Path Algorithms for Nearly Acyclic Graphs, PhD thesis, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand.

Saunders, S. & Takaoka, T. (2003), 'Improved shortest path algorithms for nearly acyclic graphs', *Theoretical Computer Science* **293**(3), 535–556.

Takaoka, T. (1998), 'Shortest path algorithms for nearly acyclic directed graphs', *Theoretical Computer Science* **203**(1), 143–150.

Takaoka, T. (1999), Theory of 2-3 heaps, *in* 'Proc. COCOON '99', Vol. 1627 of *Lecture Notes in Computer Science*, pp. 41–50.