Solving Shortest Paths Efficiently on Nearly Acyclic Directed Graphs

Shane Saunders and Tadao Takaoka

Department of Computer Science and Software Engineering,
University of Canterbury, Christchurch, New Zealand

Abstract

Shortest path problems can be solved very efficiently when a directed graph is nearly acyclic. Earlier results defined a graph decomposition, now called the 1-dominator set, which consists of a unique collection of acyclic structures with each single acyclic structure dominated by a single associated *trigger* vertex. In this framework, a specialised shortest path algorithm only spends delete-min operations on *trigger* vertices, thereby making the computation of shortest paths through non-trigger vertices easier. A previously presented algorithm computed the 1-dominator set in $O(mn)$ worst-case time, which allowed it to be integrated as part of an $O(mn + nr \log r)$ time all-pairs algorithm. Here m and n respectively denote the number of edges and vertices in the graph, while r denotes the number of trigger vertices. A new algorithm presented in this paper computes the 1-dominator set in just $O(m)$ time. This can be integrated as part of the $O(m+r \log r)$ time spent solving single-source, improving on the value of r obtained by the earlier tree-decomposition single-source algorithm. In addition, a new bi-directional form of 1-dominator set is presented, which further improves the value of r by defining acyclic structures in both directions over edges in the graph. The bi-directional 1-dominator set can similarly be computed in $O(m)$ time and included as part of the $O(m + r \log r)$ time spent computing single-source. This paper also presents a new all-pairs algorithm under the more general framework where r is defined as the size of any predetermined feedback vertex set of the graph, improving the previous all-pairs time complexity from $O(mn + nr^2)$ to $O(mn + r^3)$.

Key words: shortest path algorithm, nearly acyclic graph, graph decomposition

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

1 Introduction

A directed graph $G = (V, E)$ consists of a set of vertices V, and a set of directed edges E. Each edge $e \in E$ provides a connection between two vertices in V, and has an associated cost $c(e)$. 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.

Dijkstra's algorithm [3] solves the single source shortest path problem on a non-negatively weighted directed graph in $O(m+n \log n)$ worst-case time when applied in conjunction with a Fibonacci heap [4] or 2-3 heap [10], where m accounts for distance updates from edges, and n accounts for delete-min operations in the heap. Here m denotes the number of edges and n denotes the number of vertices in the graph. 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 $O(mn + n^2 \log n)$ worst-case 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.

This paper deals with shortest path algorithms that are suited to a special class of directed graphs called *nearly acyclic* graphs. 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 previous shortest path algorithms for nearly acyclic directed graphs have been seen. Abuaiadh and Kingston [1] presented a single-source algorithm which uses a heuristic to allow the easy traversal of vertices as acyclic regions of the graph are encountered. In such 'easy' traversal, only distance updates are involved and no delete-min operations. Used in conjunction with a modified Fibonacci heap data structure, they showed that this achieves a worst-case time complexity of $O(m + n \log t)$ where the parameter t is a reduced number of delete-min operations. Takaoka [9], using a different approach provided an algorithm with $O(m+n \log k)$ time complexity. Here k is defined as the size of the largest strongly connected component in the graph, and is expected to be small for nearly acyclic graphs. For a more complete description of strongly connected components and other graph theory terms related to algorithms, refer to Gibbons [5]. The Ph.D. thesis of

Diab Abuaiadh [2] developed a general framework for solving shortest paths by graph decomposition. This framework can be applied to nearly acyclic graphs by decomposing a graph into acyclic structures. However, the time complexity of this method is undefined without specifying the exact acyclic decomposition method to be used. Abuaiadh's thesis describes one form of acyclic decomposition that can be used. More recently, Saunders and Takaoka [7] published other specialised shortest path algorithms that make use of graph decompositions. The new algorithms in [7] 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 an all-pairs algorithm with a time complexity of $O(nm + nr \log r)$, where r is defined as the number of trigger vertices in the 1-dominator set. Here the equivalent single-source time complexity of $O(m + r \log r)$ is only applicable if we exclude the $O(mn)$ time required to compute the 1-dominator set by assuming it is pre-computed. If r is small, then the graph is regarded as being nearly acyclic. Saunders and Takaoka also gave a more general framework for the use of trigger vertices. This defines trigger vertices more generally as feedback vertices, and provides 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. The previous algorithm used by Saunders and Takaoka for computing the 1-dominator set of a graph spent $O(mn)$ worst-case time, which limited its usefulness to the all-pairs problem. This paper presents an improved decomposition algorithm for computing the 1-dominator set in $O(m)$ worst-case time. This is within the time complexity required to compute single-source. Furthermore, a bi-directional variant of the 1-dominator set is presented, which recognises acyclic structures in both directions from trigger vertices, while still maintaining the property of being set-wise unique for a given graph. The forward, backward, and bi-directional 1-dominator set variants are all encompassed by the same general definition. It is shown that the bi-directional 1-dominator set can be computed in $O(m)$ worst-case time by merging the forward and backward 1-dominator sets. All 1-dominator set variations allow single-source to be solved within the previously published $O(m + r \log r)$ worst-case time complexity, where r is the resulting number of trigger vertices. The value of r that results from a bi-directional 1-dominator set is never larger than that for the mono-directional 1-dominator set, and is potentially smaller where a graphs structure favours this. In addition, this paper provides a new algorithm for solving all-pairs in $O(mn+r^2 \log r)$ worst-case

time, where r denotes the number of trigger vertices in a 1-dominator set. This all-pairs time complexity is a marked improvement on the $O(mn + nr \log r)$ time complexity that is required when using the more modest approach of solving single-source from each source vertex in the graph independently. The new all-pairs algorithm is also applicable with r is defined as the number of vertices in a precomputed feedback vertex set. Using this definition for r, all-pairs is solved in a worst-case time complexity of $O(mn+r^3)$. This is a significant improvement on the previously presented feedback vertex set all-pairs algorithm, which had a worst-case time complexity of $O(mn + nr^2)$.

As an introduction to the new material presented in this paper, Section 2 provides an overview of existing algorithms for solving the shortest path problem. Section 3 provides a mathematical definition for 1-dominator sets. A new algorithm for computing the 1-dominator set of a graph in $O(m)$ worst-case time is then presented in Section 4, followed by an algorithm that computes the bi-directional 1-dominator set by merging the forward and backward 1-dominator sets. The presentation of the new shortest path algorithms begins in Section 5. This extends upon the reduced-graph approach previously used by Saunders and Takaoka. It is demonstrated that the reduced graph framework now supports single-source algorithms with time complexities of the form $O(m + r \log r)$ where r is the number of trigger vertices in a 1-dominator decomposition of the graph. A new all-pairs algorithm is then presented which achieves a time complexity of $O(mn + r^2 \log r)$ using the 1-dominator set, and $O(mn + r^3)$ using any general feedback vertex set. Concluding remarks are given in Section 6.

2 An Overview of Existing Shortest Path Algorithms

This section covers some necessary background information, that will ease the introduction of the new shortest path algorithms presented in later sections. As a starting point, Dijkstra's algorithm will be described. Dijkstra's algorithm forms the basis for many specialised forms of shortest path algorithms, such as the algorithms presented in this paper.

Dijkstra's algorithm solves the single-source problem on any directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges in the graph. When computing shortest paths, Dijkstra's algorithm maintains a distance value $d[v]$ corresponding to the distance of the currently shortest known path to each vertex v. A solution set S holds vertices for which $d[v]$ is final, while a frontier set F holds vertices for which $d[v]$ is tentative. Unexplored vertices are neither in S nor F. Dijkstra's algorithm begins by placing the starting vertex s in S with an initial distance of $d[s] = 0$. Vertices v on edges $s \rightarrow v$ are placed in F, with an initial distance value of $d[v] = c(s, v)$, where $c(s, v)$ denotes the

cost of edge $s \to v$. Dijkstra's algorithm progresses by selecting the vertex u in F for which $d[u]$ is minimum. This distance value $d[u]$ cannot be improved further via edges from other vertices in F, and already reflects the shortest distance of paths from vertices in S. Therefore the value of $d[u]$ is known to be final, allowing vertex u to be moved to S. On moving this minimum vertex u from F to S, Dijkstra's algorithm performs a relax operation on u to update the tentative distance of vertices v on outgoing edges $u \to v$ according to the formula $d[v] = \min(d[v], d[u] + c(u, v))$ where $c(u, v)$ is the cost of edge $u \to v$. In cases where such a vertex v is visited for the first time, v is simply added to F with a distance value of $d[v] = d[u] + c(u, v)$. By repeatedly moving the minimum vertex u from F to S and relaxing u, Dijkstra's algorithm eventually computes the shortest path from s to all vertices v that are reachable from s. The trivial shortest path distance of $d[v] = \infty$ holds for any vertex that is unreachable. The actual path tree can be constructed by maintaining values $p[v]$ to specify the preceding vertex on the shortest known path to a vertex v.

The time complexity of Dijkstra's algorithm depends on the data structure that is used for determining the minimum vertex in F. Currently, the most efficient data structure for this purpose is the Fibonacci heap [4], and equivalents such as the 2-3 heap [10] and trinomial heap [11]. These heaps support the insert and decrease-key operations in $O(1)$ amortised or worst-case time, allowing efficient insertion of vertices into F, and updating of distance values. Furthermore, their support for $O(\log n)$ amortised time for delete-min allows the minimum vertex to be determined efficiently, according to n being the maximum number of vertices that will be placed in F. With at most n insertions, n delete-mins, and m decrease-keys, the total time required by Dijkstra's algorithm when using a Fibonacci heap is at worst $O(m + n \log n)$.

The single-source shortest path problem, with a single initial distance of $d[s] = 0$, is a special case of a more general situation. In the more general situation, many vertices can have finite initial distances, which originate from a hypothetical sources via hypothetical edges carrying shortest path distances computed outside of the given graph. This more general situation is referred to as the generalised single-source (GSS) problem. The GSS problem appears in Takaoka's algorithm [9] where separate GSS problems are solved on each strongly connected (SC) component in a graph. The SC components that Takaoka's algorithm works with are first computed in $O(m)$ time by applying Tarjan's algorithm [12] for SC components. The edges of the graph that link together SC components form an outer acyclic structure. Tarjan's algorithm produces a topological ordering of these SC components as a by-product. Considering the SC components in topological order, Takaoka's algorithm initially solves GSS on the first SC component. Shortest path distances from this SC component are extended through outgoing edges and induce initial distances onto vertices contained in downstream SC components of the topological ordering. The computation continues with GSS being solved on the next SC

component in the topological order, which in turn induces further initial distances onto vertices of downstream SC components. This process is repeated until GSS has been solved on all SC components; at which point the single-source problem on the graph will have been computed, with initial distances having been propagated downstream from the SC component that contained the first initial distance of $d[s] = 0$. If the size of the largest SC component is k, then the size of the corresponding frontier set is never greater than k vertices for any single GSS problem. As a result, Takaoka's algorithm provides a worst-case time complexity of $O(m + n \log k)$, where k is the size of the largest SC component in the graph. For nearly acyclic graphs that contain many small SC components, linked together by a large outer acyclic structure, the value of k is expected to be small, allowing Takaoka's algorithm to perform with near $O(m)$ worst-case processing time.

The benefit of using Takaoka's algorithm is seen on the kind of nearly acyclic graphs that do not contain large SC components. Takaoka's algorithm can also be used in conjunction with any GSS-capable single-source algorithm. Thus, if SC components themselves contain acyclic structures, then Takaoka's algorithm can be used in conjunction with a suitable GSS algorithm to achieve the benefits offered by both. Such hybrid algorithms are of potential benefit to a wider range of nearly acyclic graphs than their respective component algorithms alone.

A single-source algorithm presented by Abuaiadh and Kingston [1] provides an alternative method for computing shortest paths on nearly acyclic graphs. Their algorithm works by identifying 'easy' vertices during the process of computing shortest paths. A vertex is identified as 'easy' if all its incoming edges lie in S. For such a vertex v, all possible shortest paths to v have been explored, allowing the vertex to be removed from F without the need for a delete-min operation. By making use of a modified Fibonacci heap, referred to as a relaxed heap, which supports $O(1)$ amortised time for delete operations, easy vertices can be removed from F efficiently. This reduces the number of delete-min operations required to compute single-source. The resulting time complexity for computing single-source thus becomes $O(m + n \log t)$ where t is the number of delete-min operations that were needed. Nearly acyclic graphs typically result in small values for t, thus allowing the algorithm to approach $O(m)$ worst-case running time. Although flexible, this approach has the disadvantage that the resulting number of easy vertices is dependent on the order in which vertices are removed from S. As such, the parameter t which, defines the performance of the algorithm, will depend on factors such as initial distances and edge costs. This makes the performance of this approach less well defined than other approaches which are able to determine their performance parameter directly from the graph structure alone, before shortest paths are computed.

The shortest path algorithms presented by Saunders and Takaoka [7] define their performance parameter in terms of the number of trigger vertices, where trigger vertices are defined in terms of a graph's structure. In these approaches, only trigger vertices need to be considered in delete-min operations, thus reducing the time complexity required to solve a single source problem. All non-trigger vertices in the graph are involved in acyclic regions, through which shortest paths can be computed without the need for delete-min operations. Two definitions for trigger vertices were previously presented. One definition was restricted to acyclic structures that were dominated by a single-trigger vertex. After performing delete-min on a trigger vertex, shortest paths to the associated acyclic structure's non-trigger vertices were finalised and the distance of neighbouring trigger vertices updated. With delete-min operations only ever occurring on trigger vertices, this process resulted in a worst-case time complexity of $O(m + r \log r)$ for solving single-source. However, with $O(mn)$ worst-case time being required to compute the acyclic structures, the algorithm's application was limited mainly to solving either repeated single-source problems or the all-pairs problem. Another algorithm presented by Saunders and Takaoka defined trigger vertices more generally as any feedback vertex set. This allowed the all-pairs problem to be solved in $O(mn + nr^2)$ worst-case time where r is the number of feedback vertices in a precomputed feedback vertex set for the graph. Nearly acyclic graphs typically have a small set of feedback vertices relative to their total number of vertices, with all non-feedback vertices forming one large monolithic acyclic region through which shortest paths can be computed efficiently. The flexibility offered by feedback vertex sets allows any kind of acyclic region within the graph to be identified, provided that an appropriate feedback vertex set can be computed beforehand. As such, this all-pairs algorithm is of potential benefit to a much wider range of nearly acyclic graphs compared to other algorithms which use more restrictive definitions for trigger vertices. Currently, the use of feedback vertices for computing shortest paths only produces an efficient all-pairs algorithm.

The new shortest path algorithms presented in this paper improve upon the earlier algorithms that were presented by Saunders and Takaoka. Firstly, the time required to compute the set-wise unique acyclic decomposition of a graph has been improved from $O(mn)$ to $O(m)$. As such, the acyclic decomposition can be computed as part of the time required for computing single-source efficiently on nearly acyclic graphs. Secondly, a more advanced form of set-wise unique acyclic decomposition is presented, in which acyclic structures are defined over both the incoming and outgoing edges of vertices. This bidirectional acyclic decomposition provides for a reduced number of trigger vertices, and can also be computed within $O(m)$ worst-case time. Finally, improvements are made to the time complexity that is required to solve all-pairs. The previous worst-case time complexity for solving all-pairs by acyclic decomposition is reduced from $O(mn + nr \log r)$ to $O(mn + r^2 \log r)$. Similarly, the previous worst-case time complexity for solving all-pairs by feedback vertices is reduced

from $O(mn+nr^2)$ to $O(mn+r^3)$. This improved algorithm allows the computation of all-pairs shortest paths in $O(mn)$ worst-case time where a feedback vertex set of $O(\sqrt[3]{mn})$ or fewer vertices can be determined. Such a feedback vertex set does not necessarily have to be minimum. [1] Any existing or future algorithm capable of determining a suitably sized feedback vertex set within the $O(mn)$ time needed for computing all-pairs will prove quite useful with this approach.

3 A Mathematical Definition for Acyclic Decomposition

This section reviews the acyclic decomposition method previously presented by Saunders and Takaoka [7]. The earlier work only provided an algorithmic definition for the decomposition. Now, a more precise mathematical definition is presented. 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*. The definition of complete acyclic structures follows from the definition of partial acyclic structures which will be given first. To distinguish the two forms, partial acyclic structures are denoted using a primed notation, as in A'_u. Acyclic structures can be defined over either the outgoing or incoming edges of vertices, resulting in *forward* and *backward* acyclic structures respectively. The definition for forward acyclic structures will be given first. A forward partial acyclic structure A'_v can be defined in the forward direction from any 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, A'_v may contain vertices w such that every path originating from vertices outside of A'_v to w passes through vertex v. In this sense, v dominates all other vertices $w \in A'_v$. Furthermore, if w is in A'_v, then A'_v must also contain all vertices that participate in acyclic paths from v to w. 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:

[1] Finding a minimum feedback vertex set would require too much effort anyway since that is an NP-complete problem.

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

This rule ensures that a complete acyclic structure A_v contains all vertices that are 'dominated' by vertex v.

The definition for backward acyclic structures is symmetric to that for forward acyclic structures. A partial backward acyclic structure B'_v defined on a vertex v satisfies the requirements:

- $B'_v - \{v\}$ is acyclic.
- $v \in B'_v$
- All $w \in B'_v - \{v\}$ satisfy $OUT(w) \subseteq B'_v$ and $OUT(w) \neq \emptyset$.

Similarly, the respective additional rule for a complete backward acyclic structure B_v is:

- $w \in B_v$ for all w that satisfy $OUT(w) \subseteq B_v$.

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. The following theorems describe this property.

Theorem 1 *If $v \in A_u$ then $A_v \subseteq A_u$.*

PROOF. An acyclic structure A_v can be constructed by starting from a partial acyclic structure $A'_v = \{v\}$ and applying the iterative equation $A'_v \leftarrow A'_v + \{w\}$ while there exists some vertex $w \notin A'_v$ that satisfies $IN(w) \subseteq A'_v$. The following proof by induction shows that any such vertex w must belong to A_u on the basis that $v \in A_u$.

Induction Basis: $A'_v = \{v\}$ satisfies $A'_v \subseteq A_u$.

Induction Step: Assume by previous induction that $A'_v \subseteq A_u$. Any vertex w that is added to A'_v must satisfy $IN(w) \subseteq A'_v$. With $A'_v \subseteq A_u$, such a vertex w must also satisfy $IN(w) \subseteq A_u$. By definition of A_u, this means that $w \in A_u$. Thus, the condition $A'_v \subseteq A_u$ is retained after $A'_v \leftarrow A'_v + \{w\}$. Hence, by induction, and the eventual condition $A'_v = A_v$, it holds that $A_v \subseteq A_u$. $\square$

Theorem 2 *If $v \in B_u$, then $A_v \subseteq A_u \cup B_u$.*

PROOF. First, it is shown by induction that, provided $u \notin A'_v$, any vertex w added to A'_v must belong to B_u on the basis that $v \in B_u$.

Induction Basis: $A'_v = \{v\}$ satisfies $A'_v \subseteq B_u$.

Induction Step: Assume by previous induction that $A'_v \subseteq B_u$. Any vertex w that is added to A_v must satisfy $IN(w) \subseteq A'_v$. With $A'_v \subseteq B_u$, such a vertex w must also satisfy $IN(w) \subseteq B_u$. Thus, any vertex $w' \in IN(w)$ satisfies $w' \in B_u$. If $u \notin A'_v$, then $w' \neq u$. By definition of B_u, any such $w' \neq u$ must satisfy $OUT(w') \subseteq B_u$. This is only possible if $w \in B_u$. Thus, provided that $u \notin A_v$, the condition $A'_v \subseteq B_u$ is retained after $A'_v \leftarrow A'_v + \{w\}$. Hence, by induction, and the eventual condition $A'_v = A_v$, it holds that $A_v \subseteq B_u$ if $u \notin A_v$.

Next, the proof is completed by allowing for the situation $u \in A'_v$. Suppose that $u \in A'_v$. Then, following a proof similar to that of Theorem 1, any further vertices w added to A'_v belong to A_u. Therefore, the general condition $A_v \subset A_u \cup B_u$ is satisfied. □

Theorem 3 *If* $v \in \Phi_u$ *then* $\Phi_v \subseteq \Phi_u$.

PROOF. By Theorem 1, the theorem is true for the definition $\Phi_v \equiv A_v$, and symmetrically true for the definition $\Phi_v \equiv B_v$. The theorem is also seen to be true for the definition $\Phi_v \equiv A_v \cup B_v$ by considering Theorems 1 and 2 in combination. □

As a result of Theorem 3, there will exist some acyclic structures Φ_u that can not 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. The 1-dominator set is named as such because of the fact that any single acyclic structure is dominated by a *single* associated trigger vertex.

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 mono-directional 1-dominator set, R forms a disjoint set of acyclic structures covering the whole graph, and thus is a decomposition. In the case of bi-directional 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 determines u as the trigger instead of 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.

4 Acyclic Decomposition Algorithms

4.1 Computing Monodirectional 1-Dominator Sets

An algorithm for computing mono-directional 1-dominator sets in $O(mn)$ worst-case was previously presented by Saunders and Takaoka [7]. 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 by Saunders and Takaoka. 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 $v \in A$. Ad-

ditionally, 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.

Definition 4 *A vertex v is a boundary vertex of an acyclic structure A_u if and only if $v \notin A_u$ and there exists an edge $w \rightarrow v$ such that $w \in A_u$. Such vertices v are said to border A_u.*

Theorem 5 *A_w is maximal for any vertex w bordering another maximal acyclic structure A_u.*

PROOF. Since w is a boundary vertex of A_u, it must hold that $w \in A_v$ for some maximal acyclic structure A_v, where $A_v \cap A_u = \emptyset$. If $w \in A_v$, then either $w = v$ or $IN(w) \subseteq A_v$ by definition of A_v. Since at least one vertex of $IN(w)$ is contained in A_u, the condition $IN(w) \subseteq A_v$ cannot hold. Therefore $w = v$ must hold, in which case A_w is maximal. □

By Theorem 5, 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 *boundedAcyclicSetA*(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 *boundedAcyclicSetA*(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 boundedAcyclicSetA(u) {
2.      VertexSet A, L;
3.      procedure rdfs(v) { /* v traversed */
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 ← ∅;
12.     L ← {u};
13.     inCount[u] ← inCount[u] + 1; /* prevents re-traversal of u */
14.     rdfs(u);
15.     for each w ∈ L do inCount[w] ← |IN(w)|;
16.     VertexSet D ← L − A; /* boundary vertices */
17.     return (A,D);
18. }
```

Theorem 6 *At the start of any call rdfs(v), it holds that:*

(1) each vertex in A has been traversed exactly once.
(2) the vertices in $A-\{u\}$ induce an acyclic subgraph on their outgoing edges.
(3) each vertex $w \in A - \{u\}$ satisfies both $IN(w) \subseteq A_u$ and $IN(w) \neq \emptyset$.
(4) $v \notin A$.

PROOF. *Induction Basis:* The theorem is trivially true for the initial call *rdfs*(u) where A is empty and all vertices are untraversed.

Induction Step: Let all four conditions hold by previous induction. Any vertex that is traversed will have been added to A. Since vertex v is not in A, and therefore untraversed, the condition that all vertices in A have been traversed exactly once will continue to hold after vertex v is placed in A. Hence condition 1 is satisfied by induction.

Since it is known by previous induction that $IN(w) \subseteq A$ for all $w \in A$, there is no edge from vertex v, which lies outside of A, to a vertex w inside A. Therefore adding v to A cannot create a cycle back to a vertex $w \in A - \{u\}$. Hence condition 2 is satisfied by induction.

Now, at the start of any call *rdfs*(v), where $v \neq u$, the precondition $inCount[v] =$

0 must have been satisfied. By previous induction, all vertices in A have been traversed only once, and with outgoing edge traversals initiated only once per vertex, no edge will have been traversed more than once. Therefore, the value of $inCount[v]$, which starts from $|IN(v)|$ for all $v \neq u$, reaches zero only once all $IN(v)$ incoming edges of v have been traversed. Thus, it holds that $IN(v) \subseteq A$. Also, $IN(v) \neq \emptyset$ since v was reached via some incoming edge. Consequently, the condition that all vertices w currently in $A - \{u\}$ satisfy both $IN(w) \subseteq A_u$ and $IN(w) \neq \emptyset$ will be maintained after adding v to A. Hence condition 3 is satisfied by induction.

Before v is placed in A, it holds by previous induction that all vertices $w \in A - \{u\}$ satisfy the condition $IN(w) \subseteq A_u$ and $IN(w) \neq \emptyset$. As a result, all $v' \in OUT(v)$ satisfy $v' \notin A - \{u\}$ since the condition $IN(v') \subseteq A$ is violated by that fact that $v \in IN(v')$ and $v \notin A$. Therefore, it holds that $v' \notin A - \{u\}$ for any recursive call $rdfs(v')$ that occurs. Such a recursive call also has $v' \neq u$ since $rdfs(u)$ never occurs. This is because the initial value of $inCount[u] = |IN(u)| + 1$ prevents the necessary precondition of $inCount[u] = 0$ from ever being reached, even if all $IN(u)$ incoming edges of u are traversed. Therefore, it holds that $v' \notin A$ for any recursive call $rdfs(v')$ that occurs. Thus, condition 4 of the theorem remains true for any recursive call $rdfs(v)$ as $v \leftarrow v'$. Hence, condition 4 is satisfied by induction. □

Theorem 7 *The function boundedAcyclicSetA(u) computes the complete acyclic set A_u.*

PROOF. If the function *boundedAcyclicSetA*(u) computes the complete acyclic set A_u, then, to keep with the definition of A_u, the following conditions need to be satisfied.

(1) $u \in A_u$.
(2) $A_u - \{u\}$ is acyclic.
(3) All $w \in A_u - \{u\}$ satisfy $IN(w) \subseteq A_u$ and $IN(w) \neq \emptyset$.
(4) $w \in A_u$ for all w that satisfy $IN(w) \subseteq A_u$.

The initial call $rdfs(u)$ adds vertex u to A, thereby satisfying condition 1. By Theorem 6, conditions 2 and 3 always remain true as vertices w are added to A by calls $rdfs(w)$. Finally, condition 4 is satisfied because if there is any vertex w with $IN(w) \subseteq A$, then it is known that $inCount[w]$ will have reached zero, causing $rdfs(w)$ to be called and w added to A. □

Algorithm 2 presents the overall algorithm, which calls *boundedAcyclicSetA*(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, an

arbitrary 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 *boundedAcyclicSetA*(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] ← undefined;
3.  Choose an arbitrary starting vertex s.
4.  Q ← {s};
5.  while Q ≠ ∅ do {
6.      Remove the next vertex u from Q;
7.      if vertexType[u] = undefined then {
8.          (A, D) ← boundedAcyclicSetA(u);
9.          for each v ∈ A do Let AC[v] refer to A;
10.         for each v ∈ A do vertexType[v] ← nontrigger;
11.         vertexType[u] ← trigger;
12.         for each v ∈ D do {
13.             if vertexType[v] = undefined and v ∉ Q then Add v to Q;
14.         }
15.     }
16. }
```

When the algorithm begins, it is unknown whether the starting vertex s is in-fact a trigger. If s is a trigger vertex, then only maximal acyclic structures will be computed. If s is a non-trigger vertex, then non-maximal acyclic structures will be computed until a trigger vertex is hit. These non-maximal acyclic structures will eventually be overwritten, meaning that some re-traversal of vertices will occur. Theorems 8 and 9 describe the constraints of such re-traversal.

Theorem 8 *The only way for Algorithm 2 to re-traverse a vertex w is during a call boundedAcyclicSetA(u) that re-traverses the starting vertex s.*

PROOF. Algorithm 2 initially identifies all vertices v as untraversed and sets *vertexType*$[v]$ to *undefined* (see Line 2). The only traversal of vertices by

Algorithm 2 occurs at Line 8 where the call *boundedAcyclicSetA*(u) traverses all vertices belonging to the acyclic structure A_u. Thereafter, any traversed vertices become marked by setting *vertexType*$[u]$ to *trigger* and *vertexType*$[v]$ to *nontrigger* for other vertices v in A_u (Lines 10 and 11). Such a computation of an acyclic structure A_u only occurs if vertex u is untraversed as identified by checking if *vertexType*$[u]$ = *undefined* at Line 7. Now consider if, during this computation of A_u, some vertex $w \in A_u$ has already been traversed. Since u was not traversed before, w must have been previously reached though some path that does not involve vertex u. However, vertex u lies on all paths that originate from vertices outside of A_u and lead to vertices inside of A_u. Therefore, any previously traversed path to w would have to have originated within A_u, and this is only possible where the starting vertex s satisfies $s \in A_u$. □

Theorem 9 *A non-trigger starting vertex s can be re-traversed only by a single call boundedAcyclicSetA(u) that computes the maximal acyclic structure A_x containing the starting vertex s, where $u = x$ is the resulting trigger vertex.*

PROOF. Consider a call *boundedAcyclicSetA*(u) that, while computing A_u, re-traverses a non-trigger starting vertex s. By the definition of A_u, any cycle passing through at least one vertex of A_u must also pass through vertex u. Therefore, with $s \in A_u$, vertex u lies on any path that leads back to vertex s. Now suppose that vertex s has already been re-traversed. If this were the case, then vertex u would have been traversed previously and *vertexType*$[u]$ set to either *trigger* or *nontrigger*. This is in contradiction to the fact that the call *boundedAcyclicSetA*(u) can only have occurred if *vertexType*$[u]$ = *undefined*. Therefore vertex s cannot have already been re-traversed. Hence the starting vertex s is re-traversed only once. Since the maximal acyclic structure A_x containing s must be computed, this single re-traversal of s must result from a call *boundedAcyclicSetA*(u) that computes A_x with $u = x$ as the resulting trigger vertex. □

Corollary 10 *Combining Theorems 8 and 9, any vertex w that is re-traversed is only re-traversed by the single call boundedAcyclicSetA(x) that computes the maximal acyclic structure A_x containing vertex s.*

Under this limited re-traversal, 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. When running the algorithm on a strongly connected graph or subgraph, this trigger vertex x must eventually be encountered. At that point, such non-maximal acyclic structures will be erased by the computation of A_x.

Corollary 11 *Since the computation of A_x is the only occurrence of re-traversal,*

no edge or vertex is traversed more than twice. Hence the worst-case running time of Algorithm 2 is $O(m)$.

For a strongly connected 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. This is summarised as Algorithm 3. The notation $cover(s)$ denotes exactly the same process used by lines 4 to 16 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. As $cover(s)$ will be called 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. Thus, the overall worst-case time complexity to determine the 1-dominator set remains $O(m)$.

Algorithm 3 Computing the 1-Dominator Set of Any Graph

1. **for** all $v \in V$ **do** $inCount[v] \leftarrow |IN(v)|$;
2. **for** all $v \in V$ **do** $vertexType[v] \leftarrow undefined$;
3. **for** each SC component H in topological order **do {**
4. Choose an arbitrary starting vertex s from H;
5. **if** $vertexType[s] = undefined$ **then** $cover(s)$;
6. **}**

The respective algorithms for computing the backward 1-dominator set are obtained by replacing the function $acyclicSetA(v)$ with a symmetric function $acyclicSetB(v)$.

4.2 Computing Bidirectional 1-Dominator Sets

The $O(mn)$ worst-case time approach for determining monodirectional 1-dominator sets can be extended relatively easily to determine the bidirectional 1-dominator set. This is achieved by calling both $acyclicSetA(v)$ and $acyclicSetB(v)$ from each vertex v in the graph that remains to be traversed, and marking the vertex type of vertices appropriately. Provided that both $acyclicSetA(v)$ and $acyclicSetB(v)$ have been called from any vertex that is left marked as a trigger, a set of trigger vertices denoting the bidirectional 1-dominator set will be determined. The time complexity of such an algorithm is easily kept within $O(mn)$ worst-case time. However, an $O(m)$ worst-case time

algorithm for computing the bidirectional 1-dominator sets is not as easily described as its equivalent monodirectional algorithms. A simpler approach is to determine the forward and backward 1-dominator sets separately and merge these to obtain the bidirectional 1-dominator set.

In order to describe the merging process, the set of trigger vertices from the the forward 1-dominator set is denoted as T_A, and the set of trigger vertices resulting from the backward 1-dominator set is denoted as T_B. Thus, a trigger vertex $u \in T_A$ denotes a maximal acyclic structure A_u in the 1-dominator set. Similarly, a trigger vertex $u \in T_B$ denotes a maximal acyclic structure B_u in the backward 1-dominator set. The result of merging is a set of trigger vertices T_C such that each vertex $u \in T_C$ uniquely denotes a maximal bidirectional acyclic structure $A_u \cup B_u$. Bidirectional trigger vertices do have some relationship to monodirectional trigger vertices, as described by the following theorem.

Theorem 12 *The set of trigger vertices T_C, denoting maximal bidirectional acyclic structures, can be assembled such that $T_C \subseteq T_A$.*

PROOF. Consider any vertex v in the graph. This vertex must be contained within some maximal acyclic structure A_u denoted by a trigger vertex $u \in T_A$. With $v \in A_u$, it holds by Theorem 3 that the bidirectional acyclic structure $A_v \cup B_v$ cannot contain any vertex outside of $A_u \cup B_u$. Thus, for any vertex v, it holds that $A_v \cup B_v \subseteq A_u \cup B_u$ for some vertex $u \in T_A$. As such, any maximal bidirectional acyclic structure $A_v \cup B_v$ can be denoted using a trigger vertex $u \in T_A$. Hence, the set of trigger vertices T_C, denoting maximal bidirectional acyclic structures, can be assembled such that $T_C \subseteq T_A$. $\square$

This theorem states that T_C can be assembled using trigger vertices taken from T_A. Alternatively, a symmetric theorem allows T_C to be assembled using trigger vertices taken from T_B. For the purpose of this description, the approach of constructing T_C from trigger vertices contained in T_A will be used.

To support the merge process, each vertex v is assigned a value $source[v]$ as well as value $dest[v]$. The value of $source[v]$ is determined from the forward 1-dominator set, and identifies the trigger vertex u of the maximal acyclic structure A_u containing v. Similarly, the value of $dest[v]$ is determined from the backward 1-dominator set, and identifies the trigger vertex u of the maximal acyclic structure B_u containing v. Using $source[v]$ and $dest[v]$, the following theorem describes how to identify maximal bi-directional acyclic structures:

Theorem 13 *If $u \in T_A$, then $A_u \cup B_u$ is maximal if and only if $source[dest[u]] = u$.*

PROOF. Let $w = dest[u]$. With $u \in B_w$, Theorem 3 implies that $A_u \cup B_u \subseteq A_w \cup B_w$.

First, consider the case of $source[dest[u]] = u$; that is $source[w] = u$. With $w \in A_u$, Theorem 3 implies the additional condition of $A_w \cup B_w \subseteq A_u \cup B_u$. Combining this with the condition $A_u \cup B_u \subseteq A_w \cup B_w$ gives $A_u \cup B_u = A_w \cup B_w$. Now, if $A_u \cup B_u$ were to be non-maximal, then there would need to exist some vertex $v \notin A_u \cup B_u$ such that $A_u \cup B_u \subset A_v \cup B_v$ and $A_w \cup B_w \subset A_v \cup B_v$. Such a situation requires either $u \in A_v$ and $w \in A_v$, or $u \in B_v$ and $w \in B_v$. This implies that either $A_u \subset A_v$ or $B_w \subset B_v$. However, neither condition can hold since both A_u and B_w are maximal. Consequently, $A_u \cup B_u$ cannot be non-maximal, and is therefore only maximal under the condition $source[dest[u]] = u$.

Now consider the case of $source[dest[u]] \neq u$; that is, $source[w] \neq u$. Then $w \notin A_u$. Additionally, $w \notin B_u$ since $u \in B_w$. Thus, it is impossible to have $A_u \cup B_u = A_w \cup B_w$. This reduces the original condition of $A_u \cup B_u \subseteq A_w \cup B_w$ to $A_u \cup B_u \subset A_w \cup B_w$. Thus, $A_u \cup B_u$ is non-maximal under the condition $source[dest[u]] \neq u$. Hence, $A_u \cup B_u$ is maximal if and only if $source[dest[u]] = u$. □

Corollary 14 *Combining Theorems 12 and 13 implies that T_C contains all and only those vertices $u \in T_A$ for which $source[dest[u]] = u$.*

This precisely defines T_C as follows:

$$T_C = \{u \mid u \in T_A \text{ and } source[dest[u]] = u\}$$

Thus, by checking if $source[dest[u]] = u$ for each $u \in T_A$, a set of trigger vertices T_C denoting the bidirectional 1-dominator set acyclic structures can be constructed. A symmetric definition, using vertices from T_B, defines T_C as follows:

$$T_C = \{u \mid u \in T_B \text{ and } dest[source[u]] = u\}$$

Algorithm 4 Computing Trigger Vertices of the Bidirectional 1-Dominator Set

```
1.  T_C ← ∅;
2.  for each u ∈ T_A do {
3.      if source[dest[u]] = u then T_C ← T_C + {u};
4.  }
```

The merge process is summarised as Algorithm 4. This is based on using T_A,

but the algorithm for using T_B is similar.

Theorem 15 *Algorithm 4 computes T_C in $O(r)$ worst-case time where $r = \min(|T_A|, |T_B|)$.*

PROOF. Line 3 of Algorithm 4 can easily be implemented to take just $O(1)$ time. Thus, the total time required for Algorithm 4 to compute T_C is just $O(r)$ where r is the number of vertices in T_A or T_B, depending on which is being used. Using the smaller of the two gives $r = \min(|T_A|, |T_B|)$. □

After determining T_C, the corresponding maximal bidirectional acyclic structures can easily be determined in $O(m)$ time by calls *acyclicSetA*(u) and *acyclicSetB*(u) for each vertex $u \in T_C$. Note that during these calls, each edge is traversed by at most one *acyclicSetA*(u) call since there is no overlapping among forward acyclic structure parts computed as each one of these is maximal. Similarly, each edge is traversed by at most one *acyclicSetB*(u) call. Hence, with each edge traversed at most twice and $|T_C| < n < m$, these calls take $O(m)$ total time. Thus, the overall process, from computing the forward and backward 1-dominator sets, to merging these and computing the acyclic structures of the bidirectional 1-dominator set, takes $O(m)$ worst-case time.

5 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 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.

5.1 Introducing the Reduced Graph Approach

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 $O(m)$ worst-case 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 $u \to v$ 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 $u \to v$ is created in P if and only if there exists a path of the form $u \rightsquigarrow v$. The specific algorithm for computing P will be described later.

An introduction to the use of P is given by the following review of the all-pairs algorithm of Saunders and Takaoka [7]. Using the definition of $\rightsquigarrow$, any path in the graph, from a source vertex $s \in V$ to a target vertex $v \in V$, can be expressed in the form $s \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_l \rightsquigarrow v$, where $l \geq 0$ and $u_i \in T$ for all $1 \leq i \leq l$. The process of computing single-source begins with a distance of $d[s] = 0$ for the source vertex s, and $d[v] = \infty$ for all other vertices. The first stage of the algorithm computes all shortest paths of the form $s \rightsquigarrow u_1$ for all possible u_1. This is achieved by scanning the vertices in $\overline{T}$ in topological order, and updating shortest path distances over their outgoing edges. The process then continues by solving generalised single-source (GSS) on the reduced graph P to compute shortest paths of the form $s \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_l$. In a GSS problem, each vertex v has some arbitrary, possibly infinite, initial distance $d[v]$ that must be reduced to its final shortest path distance. Finally, these shortest path distances are pushed onto vertices in $\overline{T}$, and extended through the topological order of $\overline{T}$ to compute shortest paths of the form $s \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_l \rightsquigarrow v$. As such, all shortest paths from s will have been computed. Hence the single-source problem is solved.

The time required to scan the topological ordering of vertices is at worst $O(m)$. The time required to solve GSS on P is $O(m' + r \log r)$ where $r = |T|$ and m' is the number of edges in P. Thus, the overall time complexity for solving single-source by this approach is $O(m + m' + r \log r)$, excluding the time required to compute the pseudo-graph P. This provides a corresponding worst-case time complexity of $O(mn + m'n + nr \log r)$ for solving all-pairs.

For a general feedback vertex set, the value of m' is bounded by r^2, providing an all-pairs time complexity of $O(mn + nr^2)$. Computing P from a general feedback vertex set, as previously presented by Saunders and Takaoka [7], requires a scan through $\overline{T}$ from each trigger. In total, $O(mr)$ worst-case time is needed when computing P from a general feedback vertex set. However,

this is easily contained within the time complexity required to compute all-pairs. For a mono-directional or bi-directional 1-dominator set, the value of m' will not exceed the value of m, and the time complexity simplifies to $O(m + r \log r)$ for single-source and $O(mn + nr \log r)$ for all-pairs. The time required for computing P from the mono-directional or bi-directional 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.

5.2 Computing the Reduced Graph

This section describes specific algorithms for computing P. Computing the reduced graph from a feedback vertex set in general requires $O(mn)$ worst-case time. This lowers to $O(m)$ worst-case time when using a feedback vertex set computed from the 1-dominator set. Consider computing P from a forward 1-dominator set. The time taken for construction can be limited to $O(m)$ because each edge is scanned only once given the non-overlapping property of the acyclic parts that are scanned from each trigger vertex. In the case of the forward 1-dominator set, a pseudo-edge (u, w) is only created if there exists an edge (v, w) where $v \in A_u$. Such an edge does not relate to the creation of any other pseudo-edge since it cannot participate in any acyclic part other than A_u, and can only have a single destination vertex. Thus, for any pseudo-edge in P there exists a corresponding edge in G, which implies that $m' \leq m$ for a forward 1-dominator set.

The algorithm for computing P from a bi-directional 1-dominator set is also $O(m)$, but is slightly more complicated. This is presented as Algorithm 5.

Algorithm 5 Computing the Reduced Graph from the Bidirectional 1-Dominator Set

```
        /* Initialisation */
 1.     for all v do {
 2.         l[v] = ∞;
 3.         d[v] = ∞;
 4.     }
 5.     R = ∅; /* Holds entries of p that will be reset. */
 6.     for all u' ∈ T do p[u'] = none;
        /* Calculate destination distances. */
 7.     for each u ∈ T do {
 8.         for each v selected in reverse-topological order from B_u − u do {
 9.             for each w ∈ OUT(v) do l[v] = min(l[v], c(v, w) + l[w]);
10.         }
11.         for each v in B_u do dest[v] = u;
```

```
12.     }
        /* Calculate pseudo-graph P. */
13.     for each u ∈ T do {
14.         for each v selected in topological order from A_u do {
15.             for each w ∈ OUT(v) do {
16.                 d[w] = min(d[w], d[v] + c(v, w));
17.                 if w ∉ A_u then {
18.                     u' = dest[w];
19.                     if p[u'] = none then {
20.                         create new pseudo edge e = (u, u') in P;
21.                         p[u'] = pointer to e;
22.                         c(e) = d[w] + l[w];
23.                         R = R + u';
24.                     }
25.                     else {
26.                         let e be the pseudo-edge pointed to by p[u'];
27.                         c(e) = min(c(e), d[w] + l[w]);
28.                     }
29.                 }
30.             }
31.         }
32.         for all u' ∈ R do p[u'] = none;
33.     }
```

The algorithm uses three one-dimensional arrays; $d[v]$ holds distance calculations involving vertex v, $l[v]$ holds the distance to the destination trigger vertex $dest[v]$ of vertex v, and $p[u']$ provides a pointer used to efficiently access a pseudo-edge (u, u').

For all destination trigger vertices u, the algorithm first computes distances $l[v]$ as the distance of the shortest path from $v \in B_u$ to u via only vertices in B_u. Additionally, the algorithm assigns $dest[v] = u$. Then for all source trigger vertices u, the algorithm computes distances $d[w]$ as the shortest path from u to w via only vertices in A_u. When $w \notin A_u$, it is known that $w \in B_{u'}$ where $u' = dest[w]$, in which case the cost $c(e)$ of the pseudo-edge $e = (u, u')$ is possibly updated using $d[w]+l[w]$. Overall, each edge in the graph is traversed at most twice after scanning backward acyclic structures and forward acyclic structures. Furthermore, each update to a pseudo-edge distance is triggered by an individual edge scan from a forward acyclic structure. Hence, the overall time complexity is $O(m)$.

Remark 16 *Some redundant computation occurs where an edge contained in both a forward and backward acyclic structure is scanned twice. This redundancy is avoidable by instead scanning* forward-only *acyclic structures A^*_u, rather than complete forward acyclic structures A_u. A forward-only acyclic*

structure A^*_u contains all the vertices of A_u except any non-trigger vertices that are also contained in backward acyclic structures $B_{u'}$ for trigger vertices $u' \in T$; that is:

$$A^*_u = A_u - (L - \{u\}) \text{ where } L = \bigcup_{u' \in T} B_{u'}$$

Another possibility is to use backward-only acyclic structures B^*_u, which can be defined similarly.

Pointers to pseudo-edges from the current trigger u are available in the one-dimensional array p. It would not be feasible to access pseudo-edges via a two-dimensional array as this would require $O(r^2)$ time. By using array p, each pseudo-edge can be accessed in $O(1)$ time so that the algorithm will not exceed the $O(m)$ time complexity requirement. The algorithm uses the set R to track which entries of p have been changed, so that entries of p can be reset efficiently before moving on to the next source trigger vertex. This avoids producing an $O(r^2)$ term in the time complexity; as would happen had all r entries of p been reset each time a different source trigger vertex was considered. Once again, the condition $m' < m$ holds which allows a single source problem to be computed or recomputed in $O(m + r \log r)$ worst-case time by this approach.

5.3 A New All-Pairs Algorithm

This section presents a new all-pairs algorithm for nearly acyclic graphs. Instead of solving GSS problems on P, the new algorithm solves a single all-pairs problem on P and then spends just $O(mn)$ time to complete the solution to all-pairs on the whole graph, thereby achieving an overall time complexity of $O(mn + m'r + r^2 \log r)$ where r is the number of vertices in P and m' is the number of edges in P. This significantly improves on the $O(mn+m'n+nr \log r)$ worst-case time complexity of the original algorithm.

The new approach is presented as Algorithm 6. First, all-pairs is solved on P, and the result 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 trigger 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 6 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 T̄ 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 ∈ T to any u ∈ T; /* O(m'r + r^2 log r) time */
10. Let D[v, u] = ∞ for all v ∈ V and u ∈ V;
    /* Compute shortest paths to each trigger vertex u. (O(mr) time) */
11. for each u ∈ T do {
12.     Let d[v] act as a reference to array entries D[v, u];
13.     d[u] = 0;
14.     for each x ∈ T do d[x] = M[x, u];
15.     for each v ∈ T̄ in reverse-topological order do rpull(v);
16. }
    /* Finish shortest paths from all source vertices v0. (O(mn) time) */
17. for each v0 ∈ V do {
18.     Let d[v] act as a reference to array entries D[v0, v];
19.     d[v0] = 0;
20.     for each v ∈ T̄ in topological order do pull(v);
21. }
```

The final stage of Algorithm 6 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 $u \in V$.

Theorem 17 *Algorithm 6 computes all-pairs in* $O(mn+m'r+r^2 \log r)$ *worst-case time.*

PROOF. Firstly, solving all-pairs on P takes $O(m'r + r^2 \log r)$ worst-case time where r and m' respectively denote the number of vertices and edges in P. This gives $\lambda[v, u]$ for all $v \in T$ and $u \in T$. Next, the process of computing shortest paths to each trigger vertex gives $\lambda[v, u]$ for all $v \in \overline{T}$ and $u \in T$ and takes $O(m)$ time per trigger vertex, giving a total time of $O(mr)$. Finally, the process of computing shortest paths to all non-trigger vertices gives $\lambda[v, u]$ for all $v \in V$ and $u \in \overline{T}$ in the graph takes $O(m)$time per vertex, giving a total time of $O(mn)$. At this point the computation of all-pairs on the whole graph is complete, having given $\lambda[v, u]$ for all $v \in V$ and $u \in V$. Combining the worst-case time complexities of each stage gives $O(m'r+r^2 \log r)+O(mr)+O(mn) = O(mn + m'r + r^2 \log r)$. □

For a general feedback vertex set, the value of m' is bounded by r^2, and the $O(mn+m'r+r^2 \log r)$ time complexity of the algorithm becomes $O(mn+r^3)$. In this case, it is sufficient to simply spend $O(r^3)$ time using Floyd's algorithm to solve the all-pairs problem on P, without affecting the overall time complexity. For a feedback vertex set arising from the trigger vertices of the 1-dominator set, the value of m' is bounded by m, making the $O(mn + m'r + r^2 \log r)$ time complexity of the algorithm $O(mn + r^2 \log r)$. This time complexity applies for both the mono- and bi-directional 1-dominator set.

6 Concluding Remarks

This paper has provided a more efficient method for using reduced graphs to compute shortest paths efficiently on nearly acyclic directed graphs. In general, all-pairs can be solved in $O(mn)$ worst-case time when a feedback vertex set of size $r \leq \sqrt[3]{mn}$ can be computed in advance. Recent results presented by Pettie [6] allow the solving of all-pairs on real-weighted directed graphs in $O(mn + n^2 \log \log n)$ worst-case 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.

The reduced graph approach may also be useful for solving shortest paths on other types of graphs. Suppose that there is some graph property, say λ, which allows shortest path to be computed efficiently. A shortest path algorithm for nearly-λ graphs would use trigger vertices to separate a graph into a λ part, thereby computing shortest paths more efficiently. For example, a shortest path algorithm for nearly-planar graphs may be possible. Finally, the $O(mn + r^3)$ worst-case time complexity required for solving all-pairs, where r is the size

of any precomputed feedback vertex set, may be lowered to $O(mn + r^2 \log r)$ by some improved approach in the future.

A more complex form of dominator sets, called k-dominator sets, can be defined, in which each acyclic structure is dominated by up to k vertices. The k-dominator set definition is basically a generalisation of the 1 dominator set definition, but is less relevant to computing shortest paths. A complete definition for k-dominator sets appears in the Ph.D. thesis of Shane Saunders [8].

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] Diab Abuaiadh. *On the complexity of the shortest path problem.* PhD thesis, Basser Department of Computer Science, University of Sydney, Australia, July 1995.

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

[4] 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.

[5] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

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

[7] S. Saunders and T. Takaoka. Improved shortest path algorithms for nearly acyclic graphs. *Theoretical Computer Science*, 293(3):535–556, February 2003.

[8] Shane Saunders. *Improved Shortest Path Algorithms for Nearly Acyclic Graphs.* PhD thesis, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand, 2004.

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

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

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

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