

Improved Shortest Path Algorithms for Nearly Acyclic Directed Graphs

Lin Tian

Tadao Takaoka

Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand,
Email: lti15@student.canterbury.ac.nz(Lin Tian),
Email: tad@cosc.canterbury.ac.nz(Tadao Takaoka)

Abstract

This paper presents new algorithms for computing single source shortest paths (SSSPs) in a nearly acyclic directed graph G . The first part introduces *higher-order decomposition*. This decomposition is an extension of the technique of strongly connected component (sc-component) decomposition. The second part presents a new method for measuring acyclicity based on modifications to two existing methods. In the new method, we decompose the graph into a 1-dominator set, which is a set of acyclic subgraphs where each subgraph is dominated by one trigger vertex. Meanwhile we compute sc-components of a de-generated graph derived from triggers. Using this preprocessing, a new SSSP algorithm has $O(m + r \log l)$ time complexity, where r is the size of the 1-dominator set, and l is the size of the largest sc-component. In the third part, we modify the concept of a 1-dominator set to that of a 1-2-dominator set. Each of acyclic subgraphs obtained by the 1-2-dominator decomposition are dominated by one or two trigger vertices cooperatively. Such subgraphs are potentially larger than those decomposed by the 1-dominator set. Thus fewer trigger vertices are needed to cover the graph.

Keywords: Algorithm; shortest paths; nearly acyclic graph; strongly-connected component; multidominator set; higher order decomposition

1 Introduction

Let $G = (V, E)$ be a directed graph, where V is the set of vertices, and E is the set of edges. Let each edge have a non-negative cost. The cost of a path is the sum of the costs of edges on the path. The single source shortest path (SSSP) problem is to find a path with minimum cost from a source vertex s to every vertex $v \in V$. Throughout this paper, any unspecified graph indicates a directed graph with positively real-valued edge costs, and all vertices are reachable from the source. Time complexities of algorithms use the worst case time complexity analysis.

For unrestricted graphs, Dijkstra's algorithm (Dijkstra 1959) is still the most efficient algorithm for the SSSP problem. When Dijkstra's algorithm uses an efficient data structure, such as Fibonacci heaps (Fredman & Tarjan 1987) or 2-3 heaps (Takaoka 2003) in its priority queue manipulations, it can achieve $O(m+n \log n)$ time where n is the number of vertices, and m is the number of edges of a given

graph. We assume that the SSSP algorithms that appear in this paper use Fibonacci heaps or 2-3 heaps for their priority queue manipulations. However, for restricted digraphs, we have efficient alternatives, like acyclic graphs with $O(m+n)$ time (Tarjan 1983) and planar graphs with $O(n\sqrt{\log n})$ time (Frederickson 1987).

If a graph is *nearly acyclic*, obviously we should not use the conventional algorithms for general graphs. If we use Dijkstra's algorithm, it does not count the underlying graph structure, and always involves n delete-min operations. In order to efficiently compute the shortest paths for nearly acyclic graphs, several specialized algorithms have been published (Abuaiadh & Kingston 1993, Abuaiadh & Kingston 1994, Takaoka 1998, Saunders & Takaoka 2003, Saunders & Takaoka 2005). These work have shown that we can reduce the number of delete-min operations performed in priority queue manipulations.

However, those specialized algorithms are based on two different measures of what a nearly acyclic graph is. (1) Takaoka gives a definition of acyclicity — the degree of cyclicity of a graph G , $cyc(G)$, is defined by the maximum cardinality of the strongly connected components (sc-components) of G . When the $cyc(G)$ is small, he categorizes the given graph as a nearly acyclic graph (Takaoka 1998). (2) Saunders states that a nearly acyclic graph is a graph that contains relatively few acyclic subgraphs, each subgraph of which is dominated by a vertex, called a trigger (Saunders 2004). Obviously, removal of triggers cuts all cycles in the graph. Saunders' idea is similar to the measure used by Abuaiadh and Kingston (1994), who say a graph is nearly acyclic if there are very few simple cycles in the graph. Note that we need preprocessing to use the above properties of near acyclicity. Here, we measure the near acyclicity of the graph by those parameters such as $k = cyc(G)$ and $r = \text{number of triggers}$. The smaller the values of the parameters are, the more acyclicity the graph has. These two measures (1) and (2) are independent and can not explain one another. We will have a more detailed review of these work in the next section.

The first part of this paper describes an extended technique of sc-components decomposition. We call it the *higher-order decomposition*. It is developed upon Takaoka's technique of strongly connected component decomposition for nearly acyclic graphs (Takaoka 1998), which we have mentioned earlier in this section. In sc-component decomposition, a graph is decomposed into sc-components. Thus, for computing the shortest paths, we run Dijkstra's algorithm only for each sc-component but not the whole graph. When all the sc-components are small, then we can efficiently solve the SSSP problem. Based on the *higher-order decomposition*, we give another definition of acyclicity. That is, the degree of cyclicity

$cyc^h(G)$ is the maximum cardinality of the strongly connected components of the decomposed graph G after the h^{th} order decomposition is made. The original definition introduced by Takaoka (1998) can be represented as: the degree of cyclicity $cyc(G)$ is the maximum cardinality of the strongly connected components of the decomposed graph G after the 1^{th} order decomposition is made.

In the second part of the paper we combine the two measures of near acyclicity into one, and show how to efficiently solve the SSSP problem in nearly acyclic graphs. For preprocessing we use a *hierarchical depth first search* (HDFS) algorithm to decompose a graph. This algorithm does 1-dominator decomposition and decomposition on triggers into sc-components at the same time. The computing time for preprocessing is $O(m)$. We degenerate the graph in such a way that each new vertex is a trigger in the original graph and a new edge exists from a vertex u to a vertex v if there is an edge from the corresponding acyclic subgraph dominated by u to v . Let r be the number of triggers and l be the maximum size of sc-components in the degenerated graph. Using this preprocessing, we show that we can efficiently solve the SSSP problem for nearly acyclic graphs with these parameters in $O(m+r\log l)$ time.

In the third part of this paper, we modify the concept of 1-dominator sets to define 1-2-dominator sets. In a 1-2-dominator set, generally speaking, one or two trigger vertices cooperatively dominate an acyclic structure in a graph. This offers potentially larger acyclic structures than the 1-dominator set does. Thereby, fewer trigger vertices are needed to cover the whole graph, that is, $r' \leq r$, where r' is the number of triggers in the 1-2-dominator decomposition, and r is the number of trigger vertices in the 1-dominator set. Considering efficient shortest path algorithms only do delete-min operations on trigger vertices, fewer trigger vertices can reduce the time for computing shortest paths. When r' is much smaller than r , we can gain efficiency in computing SSSPs for a nearly acyclic graph in $O(m + r'\log r')$ time. We present algorithms to achieve $O(m + r^2)$ time to compute the 1-2-dominator set in a graph.

In the following section, we review previous related work on nearly acyclic graphs. Then in Section 3, we describe the *higher-order decomposition* approach. In Section 4, we present a combined measure of acyclicity, and give the HDFS algorithm to implement this approach. In Section 5, we introduce the 2-dominator set, and present improved algorithms for computing 2-dominator sets. In Section 6, we give evaluations of algorithms presented in this paper. Section 7 gives some concluding remarks of this paper.

2 Review Of Related Work

Abuaiadh and Kingston (1993) suggested that the inherent complexity of the shortest path problem depends on the cycle structures of a graph as well as on its size. They gave an algorithm with $O(m+n\log t)$ time complexity, where t was the number of delete-min operations needed in the priority queue manipulations. For nearly acyclic graphs, t was expected to be small, so their algorithm could efficiently solve the SSSP problem. However, the value of t is defined by an algorithm, and had no direct relation with the static structure of the graph. Later in 1994, they introduced another algorithm with time complexity $O(m+k\log k)$, where k was the number of cycles in a graph. This was improved by Saunders and Takaoka (2005), who defined the concept of 1-dominator set.

Takaoka (1998) gave a definition of acyclicity. The degree of cyclicity of a graph G , $cyc(G)$, was de-

finied as the maximum cardinality of sc-components of G . When $cyc(G)$ was small, he defined the given digraph G to be nearly acyclic. When $cyc(G) = k$, he gave an algorithm with $O(m+n\log k)$ time complexity (Takaoka 1998). Take Figure 1 for example, the degree of cyclicity of the graph is 3, so that $k = 3$.

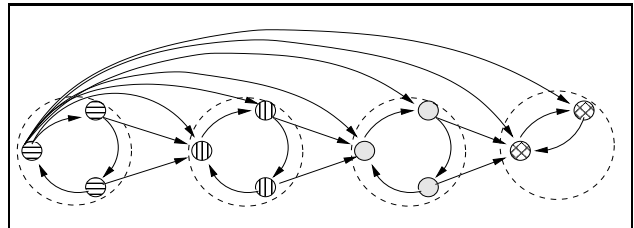


Figure 1: Vertex groups with different patterns form four sc-components in a graph. The largest sc-component has three vertices.

Obviously, Takaoka's approach needs to first compute sc-components. If $F = (V_F, E_F)$ is a depth-first spanning forest of the digraph G and $G_i = (V_i, E_i)$ is a strongly connected component, then $T_i = (V_i, E_i \cap E_F)$ is a tree (Gibbons 1985). The root vertex of the tree T_i is called *the root of the strongly connected component* G_i . If the root of an sc-component is known, then the sc-component is determined when the depth-first search from its root is finished (Gibbons 1985).

For the purpose of encoding the above statement, Algorithm 1 is an algorithm of depth-first search for sc-components by Tarjan (1972). We associate a parameter $lowlink(v)$ with each vertex v . If the vertices are labeled by variable $visitNum(v)$ according to the order in which they are visited in a depth-first search, then $lowlink(v)$ is to be the smallest of $visitNum(v)$ and those of the vertices which are connected by an edge to a descendant of v including v in the tree T_i . That is, if the depth-first search from v can reach a vertex with a lower visit number, v can not be the root of an sc-component, and $lowlink(v)$ keeps a trace of some smaller number of vertices reachable from v (Tarjan 1972).

Algorithm 1. The Depth-First Search for Strongly Connected components algorithm.

```

1. procedure CONNECT( $v$ )
2. {
3.    $visited[v] \leftarrow True; visitNum[v] \leftarrow cnt;$ 
    $cnt \leftarrow cnt + 1; lowlink[v] \leftarrow visitNum[v];$ 
    $T \leftarrow T + v;$ 
4.   for all  $w \in OUT(v)$  do {
5.     if  $visited[w] = False$  then do {
6.       CONNECT( $w$ );
7.        $lowlink[v] \leftarrow \min\{lowlink[v], lowlink[w]\};$ 
8.     };
9.     else if  $visitNum[w] < visitNum[v]$ 
       and  $w \in T$  then
10.       $lowlink[v] \leftarrow \min\{lowlink[v], visitNum[w]\};$ 
11.    if  $lowlink[v] = visitNum[v]$  then do {
12.      repeat
13.         $w \leftarrow \{pop\ vertex\ from\ T\};$ 
14.      until  $w = v;$ 
15.    }
16.  }
17.  $cnt \leftarrow 1; T \leftarrow \emptyset;$ 
18. for all  $v \in V$  do  $visited[v] \leftarrow False;$ 
19. while  $w \in V$  and  $visited[w] = False$  do
   CONNECT( $w$ );

```

In Algorithm 1, the procedure $CONNECT(v)$ perform the depth-first searches for sc-components.

A variable cnt is a count of the global visit order (line 2). At line 4, all the candidates for sc-components are stored in a first-in-last-out stack T . Line 8 updates $lowlink(w)$ if a son of v , w , is found such that $lowlink(w) < lowlink(v)$. Line 10 further updates $lowlink(v)$ if the root of the sc-component containing w is an ancestor of v in the tree T_i . Line 11 identifies roots and those vertices above and including the root on the stack induce an sc-component. They are then popped out from the stack (Gibbons 1985, Tarjan 1972).

Algorithm 2 is an SSSP algorithm based on the sc-component decomposition. First of all, Algorithm 2 calls Algorithm 1 to compute sc-components V_r, V_{r-1}, \dots, V_1 in a topological order where V_1 is the first and V_r is the last sc-component computed. Then, the graph can be degenerated into an acyclic graph $\tilde{G} = (\tilde{V}, \tilde{E})$ where $\tilde{V} = \{V_r, V_{r-1}, \dots, V_1\}$ and $\tilde{E} = \{(v, w) \mid (v, w) \in E, v \in V_i, w \in V_j, 1 \leq i, j \leq r \text{ and } i \neq j\}$ (see Figure 2). In Algorithm 2, (V_i, V_j) indicates a pseudo-edge in degenerated graph \tilde{G} such that one end-point of the edge is in V_i and another end-point is in V_j . Variable $d[v]$ maintains a distance of a path to vertex v .

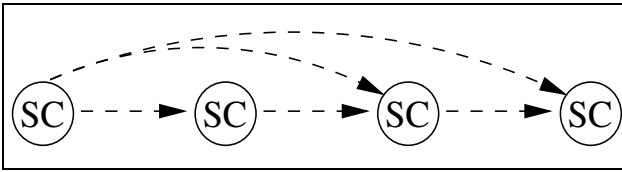


Figure 2: A degenerated acyclic graph \tilde{G} of the graph in Figure 1. \tilde{G} consists of sc-components and pseudo-edges connecting sc-components.

If we see each sc-component as a subgraph, then such a subgraph is defined as $G_i = (V_i, E_i)$ where $V_i \in \tilde{V}$ and $E_i = \{(v, w) \mid v \in V_i \text{ and } w \in V_i\}$. The SSSP problem from a source to all other vertices can be solved along the degenerated graph in the topological order from G_r to G_1 . For each subgraph G_i , we solve the *general shortest paths* (GSS) (Takaoka 1998) at line 5. The GSS is a generalized algorithm from Dijkstra’s SSSP algorithm. In the *general shortest paths* algorithm, there is no source vertex superior to other vertices; all vertices have initial distances from the source s . The vertex with the minimum distance is chosen first, and corresponding updates are made, and so forth. The computation is similar to ordinary Dijkstra’s SSSP algorithm except for the initial distance distribution. Readers are referred to (Takaoka 1998) for more details.

Algorithm 2. Solve the SSSP problem using sc-components decomposition

1. Compute sc-components V_r, V_{r-1}, \dots, V_1
2. **for** $v \in V$ **do** $d[v] \leftarrow \infty$;
3. $d[s] \leftarrow 0$; //For source s let $s \in V_r$ without loss of generality
4. **for** $i \leftarrow r$ **to** 1 **do** {
5. Solve the GSS for G_i ;
6. **for** V_j such that $(V_i, V_j) \in \tilde{E}$ **do**
7. **for** $v \in V_i$
8. **and** $w \in V_j$ such that $(v, w) \in E$ **do**
9. $d[w] \leftarrow \min\{d[w], d[v] + \text{cost}(v, w)\}$;
- }

Obviously, Algorithm 2 runs in $O(m)$ time.

Saunders and Takaoka (2005) offered an acyclic decomposition approach. In this approach, a graph

was decomposed into acyclic structures in $O(m)$ time. Each structure was dominated by a trigger vertex. We denote the 1-dominator set, which is the set of acyclic structures, by D . Note that triggers and acyclic structures correspond one to one. If a nearly acyclic graph had r trigger vertices, they introduced an algorithm with $O(m+r \log r)$ time complexity (Saunders & Takaoka 2005) for solving the SSSP problem. The new parameter r represented the number of acyclic structures in a graph. Intuitively speaking, an acyclic structure in a 1-dominator set is an acyclic subgraph such that any vertex inside can be reached from outside only through the associated trigger vertex. We say that the trigger dominates this acyclic structure. The 1-dominator set is the set of such acyclic structures. As the triggers and acyclic structures correspond one-to-one, we sometimes use the two concepts interchangeably. We also use “acyclic subgraph” and “acyclic structures” interchangeably. Take Figure 3 for example, there are three acyclic structures in the left picture.

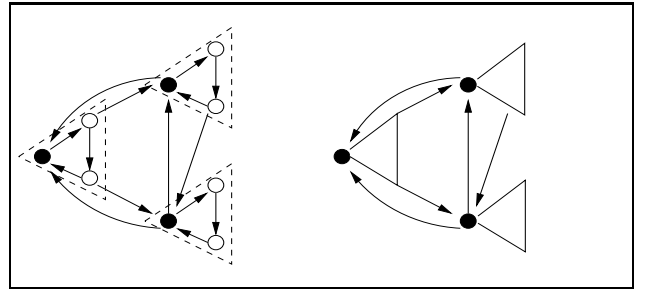


Figure 3: The acyclic structures in the left picture are presented in the right as combinations of a node and a triangle where the node represents a trigger vertex and the triangle represents non-trigger vertices dominated by the trigger vertex. There are three trigger vertices in the graph.

Algorithm 3 is a 1-dominator decomposition algorithm introduced by Saunders and Takaoka (2005) to implement the above statement.

In Algorithm 3, the procedure $rdfs()$ stands for restricted depth-first search. It maintains variable $inCount$ for each vertex v , which initially contains the total number of incoming edges of a vertex, $|IN(v)|$. When it *visits* a vertex v at lines 5-7, it decreases the $inCount$ of the vertex by 1. If it is a new visit, then it will be added into a list L at line 6. If $inCount$ becomes 0, we say it unlocks the vertex and the search goes forward (line 8), that is, the vertex is *traversed*. Unlocked vertices are included into a set A for the acyclic structure dominated by the initial vertex w (line 4). At the end of the search from w , A is the acyclic structure dominated by w , and L is the set of vertices visited (line 6). In this sense, we say, L is the set of visited vertices for possible inclusion into the acyclic structure. At the end $A[v]$ is the acyclic structure dominated by v , and $B[v]$ is the associated boundary set. The algorithm maintains $AC[v]$ which refers to an acyclic structure dominated by trigger vertex v (line 25), and $BS[v]$, which refers to the boundary set of the acyclic structure (line 26). The algorithm also maintains a queue Q containing boundary vertices as trigger vertex candidates (line 30). The algorithm starts from a vertex s , and searches with $rdfs()$ as much as possible. Then it starts searches from boundary vertices again. We assume $n \leq m$ and only treat strongly connected graphs for simplicity. Generalization to a general graph is straightforward. As the graph is strongly connected, the search will eventually come back to s . We note that the searched

part from s is only traversed twice.

Algorithm 3. 1-dominator decomposition

```

1. function AcyclicSet( $w$ ) {
2.   VertexSet  $A, L, B$ ;
3.   procedure rdfs( $u$ ) {
4.      $A \leftarrow A + \{u\}$ ;
5.     for all  $v \in OUT(u)$  do {
6.       if  $v \notin L$  then  $L \leftarrow L + \{v\}$ ;
7.        $inCount[v] \leftarrow inCount[v] - 1$ ;
8.       if  $inCount[v] = 0$  //  $v$  unlocked
9.         then rdfs( $v$ );
10.    }
11.    $A \leftarrow \emptyset$ ;  $L \leftarrow \{w\}$ ;
12.    $inCount[w] \leftarrow inCount[w] + 1$ ;
13.   // prevents re-traversal of  $w$ 
14.   rdfs( $w$ );
15.   for all  $v \in L$  do  $inCount[v] \leftarrow |IN(v)|$ ;
16.   VertexSet  $B \leftarrow L - A$ ; // boundary vertices
17.   return ( $A, B$ );
18. }
19. /***** main program *****/
20. for all  $v \in V$  do  $vertexType[v] \leftarrow unknown$ ;
21. for all  $v \in V$  do  $inCount[v] \leftarrow |IN(v)|$ ;
22.  $Q \leftarrow \{s\}$ ;
23. while  $Q \neq \emptyset$  do {
24.   Remove the next vertex  $u$  from  $Q$ ;
25.   if  $vertexType[u] = unknown$  then {
26.     ( $A, B$ )  $\leftarrow$  AcyclicSet( $u$ );
27.     Let  $AC[u]$  refer to  $A$ ;
28.     Let  $BS[u]$  refer to  $B$ ;
29.      $vertexType[u] \leftarrow trigger$ ;
30.     for all  $v \in A$  do
31.        $vertexType[v] \leftarrow non-trigger$ ;
32.     for all  $v \in B$  do
33.       if  $vertexType[v] = unknown$  and  $v \notin Q$ 
34.         then Add  $v$  to  $Q$ ;
35.   }
36. }

```

The time complexity of Algorithm 3 is $O(m)$, which was proved by Saunders and Takaoka (2005).

Algorithm 4 is an SSSP algorithm based on the results of acyclic decompositions of graphs. It modifies a general SSSP algorithm introduced by Takaoka (1998). Obviously only trigger vertices will be added into the frontier set F (line 8). That means the delete-min operations will not be required by the non-trigger vertices. The distance values $d[v]$ of non-trigger vertices in an acyclic structure will be finalized straightaway with the *decreaseKey* operation in topological order after their associated trigger vertices reach the minimum values.

Algorithm 4. SSSP Algorithm Using Acyclic Decomposition

```

1. procedure decreaseKey( $u$ ) {
2.   for each  $v \in AC[u]$  in topological order do
3.     for each  $w \in OUT[v]$  and  $w \notin S$  do
4.        $d[w] = \text{Min}\{d[w], d[v] + cost(v, w)\}$ ;
5.   }
6. /***** main program *****/
7. for all  $v \in V$  do  $d[v] = \infty$ ;
8. solution set  $S = \emptyset$ ;
9. insert all triggers into frontier set  $F$ ;
10. the source vertex  $s$  and  $d[s] \leftarrow 0$ ;
11. if  $s$  is not a trigger then decreaseKey( $s$ );
12. while  $F \neq \emptyset$  do {
13.    $d[u] = \text{Min}\{d[u] \mid \text{all } u \in F\}$ ;
14.    $F \leftarrow F - u$ ; // delete-min
15.    $S \leftarrow S + u$ ;
16.   decreaseKey( $u$ );
17. }

```

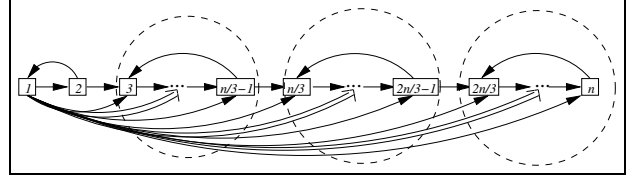


Figure 4: Arrow “ \Rightarrow ” indicates *Vertex 1* has edges to every node unstated. Let *Vertex 1* be the source, $cyc^1(G) = n/3$, $cyc^2(G) = 2$.

In line 8 of Algorithm 4, a trigger means a 1-dominator trigger. In Section 5, this will be a 1-2-dominator trigger.

3 Higher-Order Decomposition

In Takaoka’s approach, the sc-components originally computed in line 1 of Algorithm 2 remain fixed through all GSS’s. When we start one GSS, there is an opportunity that the corresponding sc-component can be further decomposed.

The basic idea behind *higher-order decomposition* is to remove the first vertex selected in GSS (line 5 of Algorithm 2), and then run the sc-component decomposition for the new strongly connected subgraphs. The original sc-component decomposition by Takaoka (1998) is called the *first order* decomposition, and the sc-component decomposition for the subgraphs is called the *second order* decomposition, one for sub-subgraphs is called the *third order* decomposition, and so forth.

After all distance updates are done in lines 6-8 of Algorithm 2, we solve the GSS for G_{i-1} . We call the first vertex whose distance is finalized the *pseudo source* of G_{i-1} .

When all the shortest paths have been computed, we find the largest sc-component in the decomposed graph. Let us indicate the size of the largest sc-component by ρ . An SSSP algorithm based on this approach has the worst case time complexity $O(hm + n \log \rho)$. It can be efficient for some nearly acyclic graphs like a graph in Figure 4.

Thus, we give another definition that the degree of cyclicity $cyc^h(G)$ is the maximum cardinality of the strongly connected components of the decomposed graph G after the h^{th} order decomposition is made.

Let $G = (V, E)$, $V = \{v_1, v_2, \dots, v_n\}$ and $E \subseteq V \times V$. We calculate sc-components V_r, V_{r-1}, \dots, V_1 . According to the new definition, we call the sc-components the *first order sc-components*, and they are defined as $V_r^1, V_{r-1}^1, \dots, V_1^1$. Then, we define the *first order degenerated graph* $\tilde{G}^1 = (\tilde{V}^1, \tilde{E}^1)$ where $\tilde{V}^1 = \{V_r^1, V_{r-1}^1, \dots, V_1^1\}$, $\tilde{E}^1 = \{(v \rightarrow w) \mid edge(v \rightarrow w) \subseteq E, v \in V_i^1 \text{ and } w \in V_j^1, i \neq j\}$ (see Figure 2).

We call edges in \tilde{E}_1 *transient edges* since they connect between sc-components. In contrast, we call an edge $(v \rightarrow w)$ an *inside edge* if both v and w belong to the same sc-component. Thus, graph \tilde{G}^1 is a decomposed acyclic graph of G [1]. Let us define each sc-components in $\tilde{G}^1 = (\tilde{V}^1, \tilde{E}^1)$ to be subgraphs. The subgraphs are defined as $G_i^1 = (V_i^1, E_i^1)$ where $V_i^1 \in \{V_r^1, V_{r-1}^1, \dots, V_1^1\}$, $E_i^1 = \{(v \rightarrow w) \mid v \in V_i^1 \text{ and } w \in V_i^1\}$ (see Figure 2).

We know that G_i^1 is strongly connected. When we erase all incoming edges of the *pseudo source vertex* of G_i^1 , immediately G_i^1 becomes not strongly connected. Based on this property, we use an *SDFS* algorithm (see Algorithm 5) to compute sc-

components $V_r^2, V_{r-1}^2, \dots, V_1^2$ for G_i^1 , and V_r^2 maintains the *pseudo source vertex*. Let us call sc-components $V_r^2, V_{r-1}^2, \dots, V_1^2$ of G_i^1 the *second order sc-components* of G_i^1 .

Algorithm 5 implements the idea of *higher-order decomposition* by modifying Algorithm 1. It is called *Selective Depth-First Search* (SDFS) algorithm. In this algorithm, a set SV maintains source or pseudo source vertices. In the following, SV is a singleton. This can be generalized into a set if we finalize distances for a few vertices in solving a GSS. We do not seek this possibility in Algorithm 6. Variable $Graph[w]$ maintains an index i of a subgraph \tilde{G}_i^h which w belongs to.

In Algorithm 5 at line 21 we initialize the vertices in SV to be *visited* because they are selected not to join the sc-components search. At line 22, we call the recursive procedure $CONNECT(v)$ until every vertex of the graph becomes *visited*. After all the sc-components of the graph or degenerated graph are determined, from line 23 to line 26, we save vertex $v \in SV$ into a separate sc-component.

Now we look at the recursive procedure $CONNECT(v)$ (line 1). At line 8, if w is already in a sc-component, we reserve this edge in \tilde{E}^h , and erase the edge from $OUT(v)$. The reason of erasing the edge from $OUT(v)$ is that the edge connects strongly connected subgraphs, and we do not visit the edge again if we have to further decompose the subgraph which v belongs to. When $CONNECT(v)$ has processed all the outgoing edges of v , $lowlink[v] = visitNum[v]$ if and only if v is the root of the sc-component containing v (line 10). If line 10 is satisfied, we determine an sc-component (line 11 to 15).

Algorithm 5. Selective Depth-First Search algorithm (SDFS). In the first order decomposition, $SV = \{s\}, h = 1$.

```

1. procedure CONNECT(v)
2. {
3.   visited[v] ← True; visitNum[v] ← cnt;
   cnt ← cnt + 1; lowlink[v] ← visitNum[v];
   Graph[v] ← ∞; // ∞ > n
4.   T ← T + v;
5.   for each w ∈ OUT(v) {
6.     if visited[w] = False then {
       CONNECT(w);
       lowlink[v] ← min{lowlink[v], lowlink[w]};
     }
7.     else if visitNum[w] < visitNum[v]
       and w ∈ T then
       lowlink[v] ← min{lowlink[v], visitNum[w]};
8.     if 1 ≤ Graph[w] < r then {
        $\tilde{E}_i^h \leftarrow \tilde{E}_i^h + (v \rightarrow w)$ ; //transient edge
        $OUT(v) \leftarrow OUT(v) - (v \rightarrow w)$ ;
       // OUT(v) is a set of inside edges at the end
     }
9.   }
10.  if lowlink[v] = visitNum[v] then do {
11.    repeat
12.      w ← popvertexfromT;
        $V_r \leftarrow V_r + w$ ;  $Graph[w] \leftarrow r$ ;
13.    until w = v;
14.    r ← r + 1;
15.  }
16. }
/***** main program *****/
17. function SDFS( $G_i^h, SV$ ) //graph  $G_i^h$  to be
decomposed with source in set  $SV$ .
18. {
19.   r ← 1; cnt ← 1; T ← ∅;
20.   for each v ∈ V do {

```

```

   visited[v] ← False;
   Graph[v] ← NULL;
}
21. for v ∈ SV do visited[v] ← True;
22. while w ∈ V and visited[w] = False do
   CONNECT(w);
23. for v ∈ SV do {
24.    $V_r \leftarrow V_r + v$ ;
25.   for w ∈ OUT(v) do
     if 1 ≤ Graph[w] < r
     then  $\tilde{E}_i^h \leftarrow \tilde{E}_i^h + (v \rightarrow w)$ ;
26. }
27. }
```

Algorithm 6 is an SSSP algorithm based on the higher order decomposition. It is generalized from Algorithm 2. The algorithm starts a recursive call a procedure $Dynamic(G, SV)$ at line 16, and $SV = \{s\}$ because it always starts from the first order decomposition, so the source vertex is the only vertex in set SV . Now let us look at the procedure $Dynamic(G, SV)$. At line 2 we call Algorithm 5 to compute the h^{th} order sc-components V_r^h, \dots, V_1^h . Note that the r will be different from one decomposition to another. From line 4 to line 6, we update $d[w]$ according to edges $(v \rightarrow w)$, $v \in V_i^h$ and $w \in V_j^h, (r + 1 \geq i > j)$. At line 7 we find its *pseudo source vertex* for a h^{th} order subgraph G_i^h , and then update the set SV of it (line 8). At line 9, we terminate the recursive call if conditions $(|G_i^h| - 1 > c_1)$ or $(h + 1 \leq c_2)$ are satisfied, c_1, c_2 are constants. The program will decompose the graph until each V_i^h has only c_1 vertices or the graph has been decomposed c_2 times. Generally speaking, the bigger c_2 is, the more sc-components we will have. After terminating the recursive call, line 10 solves the SSSP problem for the subgraph G_i^h .

Algorithm 6. For the second order decomposition, $c_2 = 2$.

```

1. procedure Dynamic(G, SV) {
2.   Call SDFS(G, SV) to compute  $h^{th}$  order
   sc-components  $V_r^h, \dots, V_1^h$ ;
3.   for i ← r to 1 do {
4.     for  $V_j^h$  such that  $(V_i^h, V_j^h) \in \tilde{E}^h$  do
5.       for v ∈  $V_i^h$  do for w ∈  $V_j^h$  do
6.          $d[w] \leftarrow \min\{d[w], d[v] + cost(v, w)\}$ ;
7.          $v_{min} \leftarrow w$  that gives
           min{ $d[w] \mid w \in V_{i-1}^h$ };
8.          $SV \leftarrow \{v_{min}\}$ ;
9.         if  $(|G_i^h| > c_1)$  and  $(h + 1 \leq c_2)$ 
           then do {
             h ← h + 1;
             Dynamic( $G_i^h, SV$ );
           }
10.        else Solve the SSSPs for  $G_i^h$ ;
11.      }
12. }
/***** main program *****/
13. for v ∈ V do  $d[v] \leftarrow \infty$ ;
14. the source vertex s and  $d[s] \leftarrow 0$ ;
15. h ← 1;
16. Dynamic(G, {s});
```

4 A Combined Measure of Acyclicity

In this section we combine the two measures of near acyclicity (see Figure 5). Conceptually we first obtain the 1-dominator set, and degenerate the graph G to

G' where the vertices of G' are the triggers and an edge from u to v in G' exists if an edge exists from some vertex in $AC[u]$ to v . We sometimes refer to this edge in G' as a pseudo edge. Then we search for sc-components in the degenerated graph G' . Obviously the maximum size of the sc-components in the above is good enough for the size of the priority queue in the SSSP algorithm.

In the real programming, Algorithm 7 calls Algorithm 3 for triggers as a subroutine. Specifically it calls *AcyclicSet* at line 3, and only puts the trigger vertex v into a first-in-last-out stack T (line 11). If v can reach another trigger vertex w with a lower visit number, $visitNum[w]$, then v can not be the root of an sc-component in the depth-first search tree. We record this reachability in an array $lowlink[]$. When the depth-first-search finishes, the trigger vertices recorded in T will be popped out until a root vertex u if $visitNum[u] = lowlink[u]$, to form an sc-component (line 16). See Figure 5 for an example, where the degree of cyclicity of the graph G' is 3.

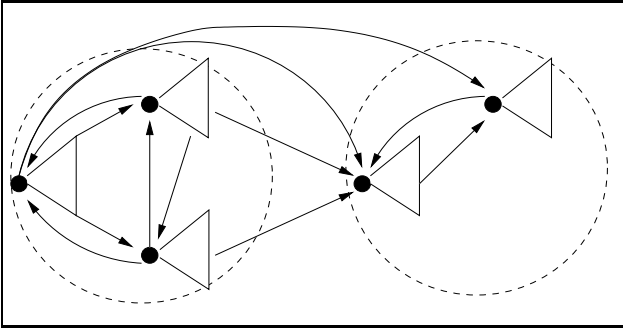


Figure 5: A graph with combined nearly acyclic structures.

In order to implement this approach, the visit number $visitNum$ and the low-link number, $lowlink$, of triggers must be computed,

In Algorithm 7, global variables c and p refer to how many vertices are visited and how many sc-components are identified respectively. Other variables are explained in Table 1.

We call Algorithm 7 the hierarchical depth first search algorithm (HDFS). It first calls function *AcyclicSet*(v) to identify a vertex set $AC[v]$ of an acyclic structure with associated trigger vertex v and boundary vertices of $AC[v]$ (line 3). All the acyclic subgraphs are computed through recursive calls of *hdfs* (line 12), and we can identify the number of triggers, $r = |D|$. In the mean time, the values of $lowlink[v]$, for dominating vertex v of every acyclic part, will be updated from the boundary vertices (lines 13 and 14). If any u of the boundary vertices is *unvisited* ($visitNum[u] = 0$), the depth first search will carry on from that vertex until all boundary vertices update their $lowlink[v]$ values. When a depth first search completes, it checks whether a trigger is a root of an sc-component (line 16). If it is a root, an sc-component of triggers is detected, and we can have the number of triggers in an sc-component, say l_i . Eventually, let $l = \text{Max}\{l_1, l_2, \dots, l_p\}$, then we will have $l = cyc(G')$ where $cyc(G')$ is the degree of cyclicity of the degenerated graph G' . The time for Algorithm 7 is $O(m)$, as each edge is examined only once.

Algorithm 7. Hierarchical Depth First Search

1. **function** *HierarchySets*(v_0) {
2. **procedure** *hdfs*(v) {
3. $(A, B) \leftarrow \text{AcyclicSet}(v)$;
4. Let $AC[v]$ refer to A ;

Table 1: Notations used in algorithms and proofs (sort alphabetically)

A	an acyclic part computed by function <i>AcyclicSet</i> ()
$AC[v]$	reference value pointing to an acyclic structure dominated by vertex v
B	a boundary vertex set
$BS[v]$	reference value pointing to a boundary vertex set of an acyclic structure $AC[v]$
C	an sc-component computed by function <i>HierarchySets</i> ()
c	count of the number of visited vertices
$cost(v,w)$	cost of edge ($v \rightarrow w$)
$d[v]$	distance of a path to vertex v
$IN(v)$	$\{v \mid (w \rightarrow v) \in E\}$
$inCount[v]$	number of untraversed incoming edges of vertex v
L	a vertex set maintains all vertices visited in function <i>AcyclicSet</i>
$lowlink[v]$	pointing to the root of an sc-component which v belongs to.
$OUT(w)$	$\{w \mid (w \rightarrow v) \in E\}$
p	count of the number of sc-components
$SC[p]$	reference value pointing to an sc-component sorted at topological order p
T	first-in-last-out stack
$visitNum[v]$	visited order of vertex v

```

5.   vertexType[v] ← trigger;
6.   for all u ∈ A do
7.     vertexType[u] ← non-trigger;
8.   for all u ∈ B do
9.     if visitNum[u] = 0 do {
10.      c ← c + 1; visitNum[u] ← c;
11.      lowlink[u] ← c;
12.      T ← T + {u};
13.      hdfs(u); // search from unvisited v ∈ B
14.      if u ∈ T then
15.        lowlink[v] ← Min(lowlink[u], lowlink[v]);
16.      else do
17.        lowlink[v] ← Min(lowlink[v], visitNum[u]);
18.      // update lowlink[v] from connected triggers
19.    }
20.   if lowlink[v] = visitNum[v] and v ∈ T do {
21.     VertexSet C ← {pop up vertices from T
22.      until v};
23.     p ← p + 1; // count sc-components
24.     Let SC[p] refer to C;
25.   }
26. }
27. }
28. }
29. }
30. }
31. }
32. }
33. }
34. }

```

Algorithm 8 modifies Algorithm 4. It is an SSSP algorithm using topologically sorted sc-components and topologically sorted vertices in each acyclic structure computed by Algorithm 3. Obviously only trigger vertices in an sc-component will be added into the frontier set F in a topological order (lines 11-14). That means the non-trigger vertices will not do the delete-min operations. The distance values $d[v]$ of the non-trigger vertices in an acyclic subgraph will be decreased straightaway when their associated trigger vertices reach the minimum values.

In Algorithm 8, we assume that sc-components are topologically sorted in the order $p, \dots, 1$. That is, if there is an edge $(v \rightarrow w)$, $v \in SC[i]$, $w \in SC[j]$, then $i > j$. Let us assume that an acyclic component A has k vertices, and these vertices are topologically sorted in the order v_1, \dots, v_k . That is, $\forall 1 \leq i, j \leq k$ if $(v_i, v_j) \in E \Leftrightarrow i < j$. Here, vertex v_1 is the trigger vertex of A .

Algorithm 8. SSSP Algorithm Using Hierarchical Decomposition

```

1. procedure decreaseKey( $u$ ) {
2.   for each  $v \in AC[u]$  in topological order do
3.     for each  $w \in OUT(v)$  and  $w \notin S$  do
4.        $d[w] = \text{Min}\{d[w], d[v] + \text{cost}(v, w)\}$ ;
5. }
/***** main program *****/
6. for all  $v \in V$  do  $d[v] \leftarrow \infty$ ;
7. the source vertex  $s$  and  $d[s] \leftarrow 0$ ;
8. solution vertex set  $S \leftarrow \emptyset$ ;
9.  $(AC, SC, p) \leftarrow \text{HierarchySets}(s)$ ;
10. if  $s$  is not a trigger then decreaseKey( $s$ );
11. for  $i \leftarrow p$  to 1 do {
12.   front vertex set  $F \leftarrow \emptyset$ ;
13.   for  $v \in SC[i]$  do
14.     if  $v$  is a trigger then insert  $v$  into  $F$ ;
15.   while  $F \neq \emptyset$  do {
16.      $d[u] = \text{Min}\{d[u] \mid \text{all } u \in F\}$ ;
17.      $F \leftarrow F - u$ ; // delete-min
18.      $S \leftarrow S + \{u\}$ ;
19.     decreaseKey( $u$ );
20.   }
21. }
```

The computing time of Algorithm 8 is $O(m + \sum_{i=1}^p l_i \log l_i)$ with condition that $l_i \leq l$ and $\sum_{i=1}^p l_i = r$. This time is maximized when we set as many l_i 's to l as possible, and we have the time is bounded by $O(m + r \log l)$.

5 1-2-dominator Set

A 1-2-dominator set is similar to a 1-dominator set. It also decomposes a graph into a collection of acyclic subgraphs, where each subgraph is dominated by one vertex only, or two trigger vertices cooperatively. It offers potentially larger acyclic subgraphs than a 1-dominator set does. Thereby, fewer trigger vertices are needed to cover the whole graph, that is, $r' \leq r$, where r' is the number of triggers in a 1-2-dominator decomposition, and r is the number of triggers in a 1-dominator set. Considering efficient shortest path algorithms only do delete-min operations on trigger vertices, fewer trigger vertices can reduce the time for solving the SSSP problem. When r' is much smaller than r , we can efficiently compute SSSPs in $O(m + r' \log r')$ time.

We can run Algorithm 8 using the results of 1-2-dominator decompositions. In the worst case, two trigger vertices u and v may cooperatively dominate exactly the same acyclic part of the 1-dominator decomposition, that is, $AC[u] = AC[v]$. Then, in the

worst case, a single-source computation using a 1-2-dominator set will scan each acyclic part up to two times. The corresponding worst-case time complexity of the algorithm is $O(2m + r' \log r') = O(m + r' \log r')$, where constant 2 is emphasized in the left-hand side.

Now, we shall show how to compute a 1-2-dominator set. First we compute a 1-dominator set of a given graph. In order to save the time for computing a 1-2-dominator set, we degenerate each acyclic structure into its associated trigger vertex during the 1-dominator decomposition. In Figure 6 the upper picture is degenerated into the lower picture with pseudo edges represented by " \Rightarrow ". We assign a boundary vertex set $BS[v]$ for the trigger vertices. For a boundary vertex w in a set $BS[v]$, we save the number of outgoing edges from $AC[v]$ to w , $inc(v, w)$, together with w in BS .

Algorithm 9 implements this process. It modifies the function *AcyclicSet* of Algorithm 3 with line numbers extended with dots (lines 15-15.4). The main program of Algorithm 3 is the same. In this algorithm, a vertex set A reserves all the non-trigger vertices dominated by the associated trigger vertex, a vertex set L reserves all vertices visited by the current search, a vertex set B contains all the boundary vertices of an acyclic structure.

Algorithm 9. Modified Function for Computing AC and BS

```

1. function AcyclicSet( $w$ ) {
2.   VertexSet  $A, L, B$ ;
   //  $B$  is enhanced with  $inc$ 
3.   procedure rdfs( $u$ ) {
4.      $A \leftarrow A + u$ ;
5.     for all  $v \in OUT(u)$  do {
6.       if  $v \notin L$  then  $L \leftarrow L + \{v\}$ ;
7.        $inCount[v] \leftarrow inCount[v] - 1$ ;
8.       if  $inCount[v] = 0$  then rdfs( $v$ );
9.     }
10.  }
11.   $A \leftarrow \emptyset$ ;  $L \leftarrow \{w\}$ ;
12.   $inCount[w] \leftarrow inCount[w] + 1$ ;
13.  rdfs( $w$ );
14.  VertexSet  $B \leftarrow L - A$ ; // boundary vertices
15.  for each  $v \in B$  do {
15.1     $inc(v_0, v) = |IN(v)| - inCount[v]$ ;
15.2     $BS[v_0] \leftarrow (v, inc(v_0, v))$ ;
15.3     $inCount[v] \leftarrow |IN(v)|$ ;
15.4  }
16.  return ( $A, B$ );
17. }
/***** main program *****/
18. for all  $v \in V$  do vertexType[ $v$ ]  $\leftarrow unknown$ ;
19. for all  $v \in V$  do  $inCount[v] \leftarrow |IN(v)|$ ;
20.  $Q \leftarrow \{s\}$ ;
21. while  $Q \neq \emptyset$  do {
22.   Remove the next vertex  $u$  from  $Q$ ;
23.   if vertexType[ $u$ ] = unknown then {
24.      $(A, B) \leftarrow \text{AcyclicSet}(u)$ ;
25.     Let  $AC[u]$  refer to  $A$ ;
26.     vertexType[ $u$ ]  $\leftarrow trigger$ ;
27.     for all  $v \in A$  do
28.       vertexType[ $v$ ]  $\leftarrow non-trigger$ ;
29.     for all  $v \in B$  do
30.       if vertexType[ $v$ ] = unknown and  $v \notin Q$ 
31.         then Add  $v$  to  $Q$ ;
   }
```

After we finish the computation of 1-dominator decomposition, we look up the tables of boundary sets $BS[v]$, for a 1-dominator trigger v , to do the 1-2-dominator decomposition, rather than traverse the whole graph again.

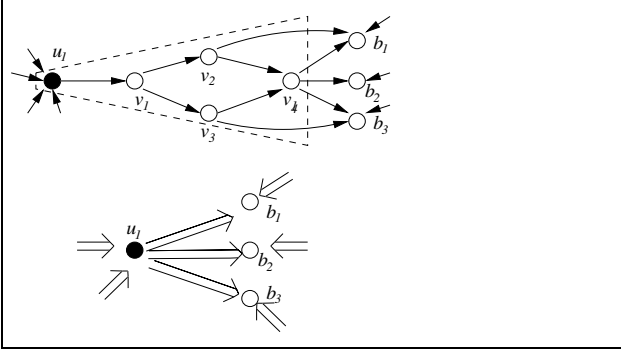


Figure 6: Simplified acyclic structure. Arrows “ \Rightarrow ” represent many edges to or from the same trigger vertex.

From one 1-dominator trigger, say u_1 , we look for a partner, say u_2 , which can co-dominate some part of the graph. The co-dominated part is given to both $AC[u_1]$ and $AC[u_2]$. Unlocked vertices are excluded from the pool of triggers (line 5). The generalized depth-first search $gdfs()$ goes over the degenerated graph.

Algorithm 10 implements this design. In Algorithm 10, a vertex set T_1 reserves all the 1-dominator trigger vertices. Note that $BS[v]$ in function $gdfs()$ plays the role of $OUT(v)$ in Algorithm 3.

Algorithm 10. 1-2-dominator Set Algorithm

```

1. procedure  $gdfs(u_1, u_2, v)$  {
2.   for all  $w \in BS[v]$  {
3.      $inCount[w] \leftarrow inCount[w] - inc(v, w)$ ;
4.     if  $inCount[w] = 0$  do { //  $w$  is unlocked
5.        $T_1 \leftarrow T_1 - \{w\}$ ;
6.        $AC[u_1] \leftarrow AC[u_1] + \{w\}$ ;
7.        $AC[u_2] \leftarrow AC[u_2] + \{w\}$ ;
8.        $gdfs(u_1, u_2, w)$ ;
9.     }
10.  }
11. }
/***** main program *****/
12. Compute 1-dominator set  $T_1$  by Algorithm 9;
13. for  $u_1 \in T_1$  do {
14.    $inCount[u_1] \leftarrow inCount[u_1] + 1$ ;
15.   for all  $v \in BS[u_1]$  do
16.      $inCount[v] \leftarrow inCount[v] - inc(u_1, v)$ ;
17.   for  $u_2 \in T_1 - \{u_1\}$  do {
18.      $inCount[u_2] \leftarrow inCount[u_2] + 1$ ;
19.      $gdfs(u_1, u_2, u_2)$ ;
20.   }
21.    $T_1 \leftarrow T_1 - \{u_1\}$ ;
22.   for all  $v \in BS[u_1]$  do  $inCount[v] \leftarrow |IN(v)|$ ;
23. }
```

At line 12, it takes $O(m)$ to complete a 1-dominator set computation. From line 13 to 23, it takes $O(r^2)$ to check all the possible pairs of 1-dominator triggers. The total time spent by $gdfs$ is $O(m')$ where m' is the number of pseudo edges bounded by m . So, the time complexity of Algorithm 10 is $O(m + r^2)$. Algorithm 9 decomposes set V into 1-dominator structures. Algorithm 10 in fact decomposes V into a mixture of 1-dominator structures and 1-2-dominator ones. This decomposition is set wise unique as the 1-dominator decomposition. More rigorous proof for uniqueness will be given in a future report.

To solve the SSSP problem, we can use Algorithm 8 with the set of remaining triggers computed by Algorithm 10. Algorithm 8 will perform *decreaseKey*

operations on 1-dominator structures once each, and twice on 1-2-dominator structures.

Once the 1-2-dominator set has been computed, we can repeatedly use it for computing SSSPs. When r' is much smaller than r , the decomposition time of the 1-2-dominator set will be balanced off by the time saved from repeatedly computing SSSPs, when only edge costs change in the same graph structure.

6 Evaluation

The new shortest path algorithms presented in this paper are theoretically more efficient than other algorithms for solving the SSSP problem on suitable nearly acyclic graphs. This offers a potential improvement on the running time in practice. In this section, we will look at what exact improvement is possible.

Experimented graphs are line-spanning random graphs. That is, for a graph $G = (V, E)$ with n vertices in V and initially no edges in E , we first add all the edges (v_i, v_{i+1}) for $1 \leq i < n$ into E . Then we repeatedly generate edge $(v \rightarrow w)$ at random such that $v \in V, w \in V, (v \rightarrow w) \notin E$ and $v \neq w$. Here, we use an edge factor f to specify those randomly added edges. Therefore, the total number of edges $m = (1 + f)n$ (Saunders 2004). Each edge has a cost between 1 and 100.

For algorithms presented in sections 4 and 5, we run them in graphs with different nearly acyclic structures. One kind of graph has nearly acyclic structures where the degree of cyclicity is $cyc(G)$ small (see Figure 7 for an example). Another kind of graph has few simple cycles for the graph size (see Figure 8 for an example). The third kind of graph has a combined structures (see Figure 9 for an example). In those experiments, graphs have n vertices and $2.8n$ edges. The number of vertices in the graphs start at 2,000 for Figures 7 and 8 and 2,197 for Figure 9 and doubling for successive values of n up until 128,000 for Figures 7 and 8 and 79,507 for Figure 9. This provides a large enough window to demonstrate the overall trends in algorithm performance.

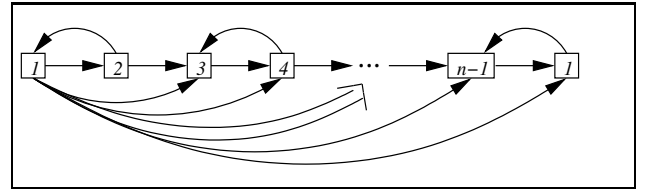


Figure 7: Arrow “ \Rightarrow ” indicates Vertex 1 has edges to each other node $i, 4 < i < n - 1$, in the graph. Let Vertex 1 be the source, $cyc(G) = 2, r = n/2, l = 2$

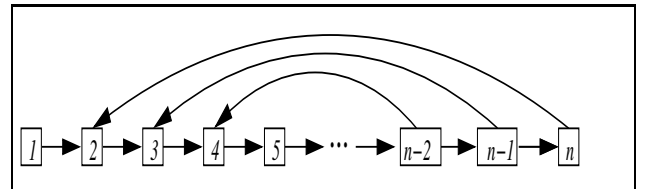


Figure 8: Let Vertex 1 be the source, $cyc(G) = n - 1, r = 3, l = 3$.

Three algorithms have been implemented (see Table 2) for solving the SSSP problem, and the SSSP computation time has been measured for analysis. Figure 10 shows that the new hierarchical approach performs as efficiently as that of the *SC* approach in graphs of Figure 7. We call such graphs *SC* approach favored graphs. Figure 11 shows that the

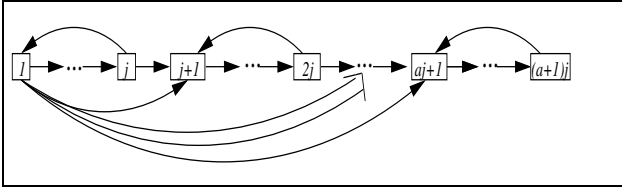


Figure 9: $j = \sqrt{n}$ and $a = \sqrt{n}-1$. Arrow “ \Rightarrow ” indicates *Vertex 1* has edges to every node $ij + 1$, $1 < i < a$, in the graph. Let *Vertex 1* be the source, $cyc(G) = \sqrt{n}$, $r = \sqrt{n}$, $l = 1$.

Table 2: The different algorithms implemented in experiments.

Name	Description
Acyclic	Acyclic approach, Saunders & Takaoka (2005)
Hierarchical	The new hierarchical approach
SC	Strongly connected approach, Takaoka (1998)

new approach performs the SSSP computation as efficiently that of the acyclic approach in acyclic approach favored graphs of Figure 8. Figure 12 shows that the new approach can outperform the other two approaches in graphs with combined nearly acyclic structures (see the graph in Figure 9).

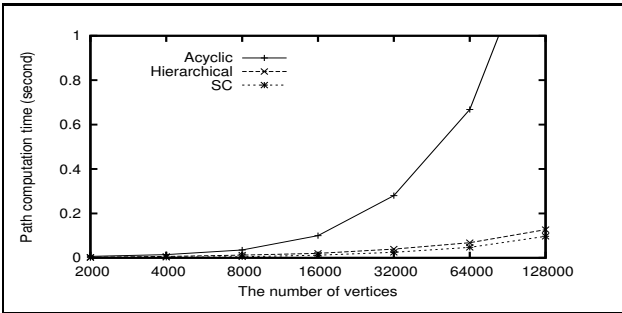


Figure 10: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 7.

For algorithms presented in Section 5, in Figure 13 there are proportions of triggers in 1-dominator sets and 1-2-dominator sets. In the experiments, there are 1,000 vertices in each graph, but the number of edges varies from $1.1 \times 1,000$ to $13.8 \times 1,000$. The abscissa shows the edge factor of each graph. From the Figure 13, we can see that the number of triggers in 1-2-dominator sets is about 20 percent smaller than that in the 1-dominator sets in sparse graphs with edge number between $1.2n$ and $4.2n$.

In Figure 14, there is the time for solving the SSSP problem between using 1-dominator sets and using 1-2-dominator sets. Graphs have 1,000 vertices and edge factors varies from 0.1 to 0.25. The ordinate shows the computation time measured in seconds. We can see from this figure that when the graph is sparse and the edge factor is between 0.005 and 0.25, it reflects the efficiency gained from the reduced number of triggers in 1-2-dominator sets. When the graph becomes denser and the edge factor gets closer to 0.25, the difference of computation time gets smaller and smaller until there is almost no difference at 0.25.

7 Concluding Remarks

In this paper, we introduce three new approaches and algorithms based on them to solve the single-

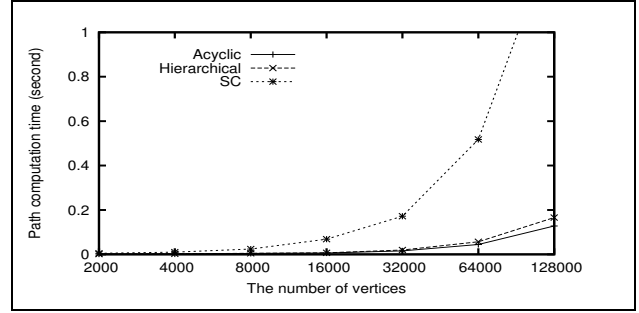


Figure 11: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 8.

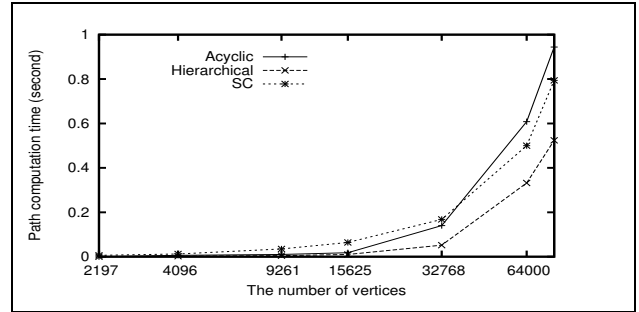


Figure 12: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 9.

source shortest path (SSSP) problem in nearly acyclic graphs. We first present a *higher-order* approach to decompose a graph into strongly connected components (sc-components) as small as possible. Under *higher-order decomposition* framework, we give another definition of acyclicity, which is generalized from the definition of Takaoka’s (1998). That is, the degree of cyclicity is the maximal cardinality of the strongly connected components of the decomposed graph G after the h^{th} order decomposition is made, and the degree of cyclicity is denoted by $cyc^h(G)$. When a graph is decomposed into sc-components, we run Dijkstra’s algorithm only for each sc-component but not the whole graph. When all sc-components are small and the degree of acyclicity $cyc^h(G) = \rho$, we can efficiently compute SSSPs in $O(hm + n \log \rho)$ time. If we repeatedly use the decomposed graph for solving the SSSP problem, we can balance the time used for preprocessing the graph.

Then, we show that we can compute acyclic components and sc-components in a graph at the same time. The benefit of merging these two graph preprocessing approaches is to make a superior measure over the two existing measures of nearly acyclic graphs. Let r be the number of trigger vertices in a

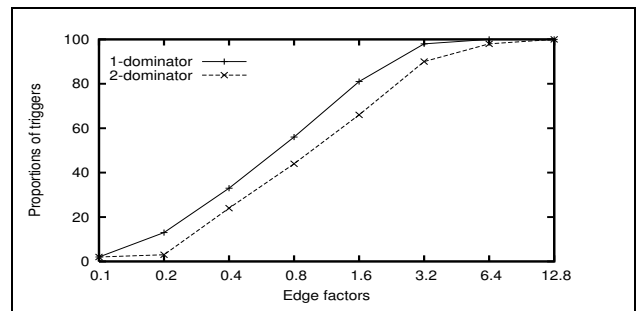


Figure 13: The proportion of triggers in 1-dominator sets and 1-2-dominator sets.

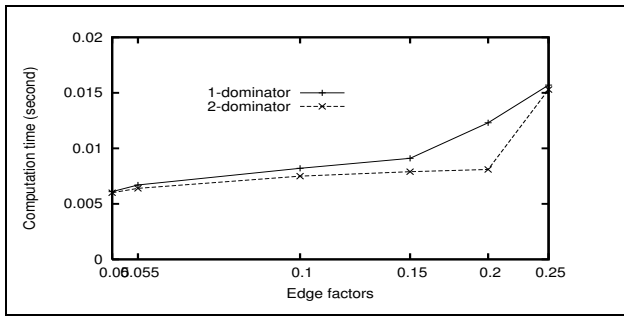


Figure 14: the computing time for SSSP problem between using 1-dominator sets and using 1-2-dominator sets in different density graphs.

1-dominator set, and $l = cyc(G')$ where $cyc(G')$ is the degree of the acyclicity of the degenerated graph G' . When l or r is small, we say that the graph is nearly acyclic, and we can efficiently solve the SSSP problem for the graph in $O(m+r \log l)$ time, which takes advantage of both measures.

We also present a demonstration of a 1-2-dominator set, which can be generalized to a multidominator set, denoted by a k -dominator set. In the 1-2-dominator set, one or two trigger vertices cooperatively dominate an acyclic structure in a graph. This offers potentially larger acyclic structures than the 1-dominator set does. Thereby, fewer trigger vertices are needed to cover the whole graph, that is, $r' \leq r$, where r' is the number of triggers in the 1-2-dominator decomposition, and r is the number of trigger vertices in the 1-dominator set. Considering efficient shortest path algorithms only do delete-min operations on trigger vertices, fewer trigger vertices can reduce the time for computing shortest paths. Once the 1-2-dominator set has been computed, we can repeatedly use it for computing SSSPs. When r' is much smaller than r , the decomposition time of the 1-2-dominator set will be balanced off by the time saved from repeatedly computing SSSPs. It is open whether we can compute the 1-2-dominator set in $O(m)$ time.

References

- Abuaiadh, D. & Kingston, J. H. (1993), 'An Efficient Algorithm for the Shortest Path Problem', *Technical Report 473*.
- Abuaiadh, D. & Kingston, J. H. (1994), 'Efficient Shortest Path Algorithms By Graph Decomposition', *Technical Report 475*.
- Dijkstra, E. W. (1959), 'A note on two problems in connexion with graphs', *Numerische Mathematik* **1**, 269–271.
- Frederickson, G. N. (1987), 'Fast algorithm for shortest path in planar graph with applications', *SIAM Journal on Computing* **16**, 1004–1022.
- Fredman, M. L. & Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *Journal of the ACM* **34**, 569–615.
- Alan Gibbons. (2004), *Algorithm Graph Theory*, Cambridge University Press.
- 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.
- Saunders, S. & Takaoka, T. (2005), 'Efficient Algorithm for Solving Shortest Paths on Nearly Acyclic Directed Graphs', *CATS* **41**.
- Takaoka, T. (1998), 'Shortest Path Algorithm for Nearly Acyclic Directed Graphs', *Theoretical Computer Science* **203**(1), 145–150.
- Takaoka, T. (2003), 'Theory of 2-3 Heaps', *Discrete Applied Mathematics* **126**, 115–128.
- Tarjan, R. (1972), 'Depth-first search and linear graph algorithms', *SIAM Journal on Computing* **1**(2), 146–160.
- Tarjan, R. (1983), 'Data Structures and Network Algorithms', *Society for Industrial and Applied Mathematics* **44**.