Sharing Information in All Pairs Shortest Path Algorithms

Tadao Takaoka and Mashitoh Hashim

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz

Abstract

We show two improvements on time complexities of the all pairs shortest path (APSP) problem for directed graphs that satisfy certain properties. The idea for speed-up is information sharing by n single source shortest path (SSSP) problems that are solved for APSP. We consider two parameters, in addition to the numbers of vertices, n, and edges, m. First we improve the time complexity of $O(mn+n^2\sqrt{\log c})$ to $O(mn+nc)$ for the APSP problem with the integer edge costs bounded by c. When $c \leq O(n\sqrt{\log n})$, this complexity is better than the previous one. Next we consider a nearly acyclic graph. We measure the degree of acyclicity by the size, r, of a given set of feedback vertices. If r is small, the given graph can be considered to be nearly acyclic. We improve the existing time complexity of $O(mn+r^3)$ for the all pairs shortest path problem to $O(mn+rn\log n)$ by some kind of information sharing. This complexity is better than the previous one for all values of r under a reasonable assumption of $m \geq n$.

Keywords: All pairs shortest paths, priority queue, sharing information, small edge costs, nearly acyclic graph

1 Introduction

We consider the all pairs shortest path (APSP) problem for a directed graph with non-negative edge costs under the standard computational model of addition-comparison on distances and random accessibility by an $O(\log n)$ bit address. The complexity for this problem under the standard computational model is $O(mn+n^2\log n)$ with a priority queue such as a Fibonacci heap (4) or 2-3 heap (9), where delete-min is $O(\log n)$, and decrease-key and insert are $O(1)$. We improve the second term of the time complexity of the APSP problem for two special types of graphs under this computational model. The problem is well illustrated by the equation $APSP = n \times SSSP +$ information sharing.

One is a directed graph with limited edge costs. When the costs are integers bounded by small c, we improve the existing time complexity of $O(mn+n^2\sqrt{\log c})$ to $O(mn+nc)$. To deal with a graph whose edge costs are non-negative integers bounded by c, Ahuja, Melhorn, Orlin, and Tarjan (1) invented the radix heap that supports the set of n delete-min, m decrease-key and n insert operations in $O(m+n\sqrt{\log c})$ time, meaning the SSSP can be solved in the same amount of time. If we apply this algorithm n times for the APSP problem, the time becomes $O(mn+n^2\sqrt{\log c})$.

For the priority queue we use a simple bucket system such that the number of buckets is c and the number of delete-min operations is at most nc for both SSSP and APSP problems. If edge costs are between 0 and c for SSSP, the tentative distances from which we choose the minimum distance for a vertex to be included into the solution set take values ranging over a band of length c. If c is small, we can reduce the time for delete-min by looking at the small range of values. We observe the same idea works for the APSP problem in a better way.

The other is a directed graph which is nearly acyclic with general edge costs of non-negative real numbers. There can be many definitions for near-acyclicity. Here we define it by the size r of the feedback vertex set, denoted by T. If r is small, we say the given graph is nearly acyclic. For the priority queue in SSSP as part of APSP, we use a Fibonacci heap or 2-3 heap to choose minimum distance vertices one by one and modify the distances to other candidate vertices. The size of queue is n, whereas in the 1-dominator decomposition described below the size can be smaller. We traverse edges forward and backward. When we traverse backward, we can define the single sink problem; we compute shortest paths from all vertices to a specified vertex, called a sink. If we solve r single sink problems for each vertex in T as a sink, and share the result when we solve n single source problems, we can achieve $O(mn+rn\log n)$ time for the APSP problem, which is better than the existing complexity of $O(mn+r^3)$.

A special type of feedback vertices are the roots in the 1-dominator decomposition (7), (8). The 1-dominator decomposition is a collection of maximal acyclic parts A_v dominated by root vertex v. To determine shortest distances from the source to other vertices for the SSSP problem, we can maintain only those root vertices in a priority queue. The distance to non-root vertices can be determined with less time complexity. Then we show the complexity $O(mn+r^2\log r)$ for the APSP problem in (8) can be derived as a special case of our framework. In (8), the APSP problem was solved for the graph derived from the set of roots.

It is still open whether the APSP problem can be solved in $O(mn)$ time. The best bounds for a general directed graph are $O(mn+n^2\log\log n)$ with an extended computational model by Pettie (6) and $o(mn)$ for an unweighted undirected graph by Chan (2). The

rest of the paper is as follows: In Section 2, we introduce a simple data structure of the bucket system. In Section 3, we define shortest path problems; single source and all pairs. In Section 4, we show the bucket system works well for the APSP problem by determining shortest distances directly on vertex pairs. In Section 5, we define the sweeping algorithm on an acyclic graph. In Section 6 we review the theory of 1-dominator decomposition and its application to the APSP problem. In Section 7, we define nearly acyclic graph with a set of size r of feedback vertices, and show the APSP problem can be solved in $O(mn + rn \log n)$ time. Section 8 is the conclusion.

2 Simple data structure

In this section we introduce a well known data structure of priority queue for developments in subsequent sections. The priority queue allows insert, decrease-key, and delete-min, and called the bucket system. Specifically, a bucket system consists of an array of pointers, which point to lists of items. Let $list[i]$ be the i-th list. If the key of item x is i, x appears in $list[i]$. The array element, $list[i]$, is called the i-th bucket. If there is no such x, $list[i]$ is nil. In the bucket system, array positions play the role of key values. To insert x is to append x to $list[i]$ if the key of x is i. To decrease the key of x, we decrease the key value, say, from i to j, and move x from $list[i]$ to $list[j]$. To find the minimum, we scan the array from the previous position of the minimum and find the first non-nil list, say, $list[i]$, then delete the first item in $list[i]$. Since all key values of the items in the list are equal, we can delete all the other items in $O(1)$ time each. Clearly the time for insert and decrease-key are both $O(1)$. The time for delete-min depends on the interval to the next non-nil list. Since we are interested in the total time for all delete-mins, we can say the time for all delete-mins is the time spent to scan the array plus time spent to delete items from the lists. If the size of array is limited to $c > 0$, and n delete-min operations are done, the time is $O(cn)$.

If the length of the range of key values at any stage is bounded by c, and the possible key values are larger than c, to save space, we can maintain a circular structure of size c for the array $list$.

3 Shortest path problems

To prepare for the later development, we describe the single source shortest path algorithm in the following. Let $G = (V, E)$ be a directed graph where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. The non-negative cost of edge (u, v) is denoted by $cost(u, v)$. We assume $cost(v, v) = 0$ and $cost(u, v) = \infty$, if there is no edge from u to v. We specify a vertex, s, as the source. The shortest path from s to vertex v is the path such that the sum of edge costs of the path is minimum among all paths from s to v. The minimum cost is also called the shortest distance. In Dijkstra's algorithm (3) given below, we maintain two sets of vertices, S and F. The set S is the set of vertices to which the shortest distances have been finalized by the algorithm. The set F is the set of vertices which can be reached from S by a single edge. We maintain $d[v]$ for vertex v in S or F. If v is in S, $d[v]$ is the (final) shortest distance to v. If v is in F, $d[v]$ is the distance of the shortest path that lies in S except for the end point v. Let $OUT(v) = \{w|(v, w) \in E\}$, and $IN(v) = \{u|(u, v) \in E\}$. The solution is in array d at the end.

Algorithm 1

```
 1  for v ∈ V do d[v] := ∞;
 2  d[s] := 0; F := {s}; // s is source
 3  Organize F in a priority with d[s] as key;
 4  S := ∅;
 5  while S ≠ V do begin
 6     Find v in F with minimum key and
          delete v from F;
 7     S := S ∪ {v};
 8     for w ∈ OUT(v) do begin
 9        if w is not in S then
10           if w is in F then
11              d[w] := min{d[w], d[v] + cost(v, w)}
12           else begin
                d[w] := d[v] + cost(v, w);
                F := F ∪ {w}
             end
13        Reorganize F into queue with new d[w];
14     end
15  end.
```

Line 6 is delete-min, line 11 is decrease-key, and line 12 is insert. In this and next section, we assume edge costs are integers and $cost(v, w) \leq c$ for all $v, w \in V$ for a positive integer c.

To have a small range for the key values in F, we state and prove the following well known lemma.

LEMMA 1 *For any v and w in F, we have $|d[v] - d[w]| \leq c$*

Proof
Take arbitrary v and w in F such that $d[v] \leq d[w]$. Since w is directly connected with S, we have some u in S such that $d[w] = d[u] + cost(u, w)$. On the other hand, we have $d[u] \leq d[v]$ from the algorithm. Thus we have $d[w] - d[v] = d[u] - d[v] + cost(u, w) \leq c$.

From this we see that the time for Algorithm 1 is $O(m + cn)$, and space requirement is $O(c + n)$ if we use a circular structure for $list$. If we maintain c buckets in a Fibonacci or 2-3 heap, we can show the time is $O(m + n \log c)$.

If we use $IN(v)$ at line 8, and $cost(u, v)$ in lines 11 and 12, we are solving the problem in reverse order from sink s. We call this version, Algorithm 2, the single sink (shortest path) problem, which will be used in Section 7.

Algorithm 2

```
 1  for v ∈ V do d[v] := ∞;
 2  d[s] := 0; F := {s}; // s is the sink.
 3  Organize F in a priority with d[s] as key;
 4  S := ∅;
 5  while S ≠ V do begin
 6     Find v in F with minimum key
          and delete v from F;
 7     S := S ∪ {v};
 8     for u ∈ IN(v) do begin
 9        if u is not in S then
10           if u is in F then
11              d[u] := min{d[u], d[v] + cost(u, v)}
12           else begin d[u] := d[v] + cost(u, v);
                F := F ∪ {u}
             end
13        Reorganize F into queue with new d[u];
14     end
15  end.
```

4 All pairs shortest path problem

If we use Algorithm 1 n times to solve the all pairs shortest path problem with the same kind of priority

queue, the time would be $O(mn + n^2c)$. In this section we improve this time complexity to $O(mn+nc)$. Precisely speaking this complexity can be $O(mn + min\{nc, n^2\sqrt{\log c}\})$, as we can switch between the bucket system and the radix heap depending on the value of c. We call Dijkstra's algorithm vertex oriented, since we expand the solution set S of vertices one by one. We modify the hidden path algorithm (5), which we call pair-wise, since we put pairs into the solution set one by one. Let (u, v) be the shortest edge in the graph. Then obviously $cost(u, v)$ is the shortest distance from u to v. The second shortest edge also gives the shortest distance between the two end points. Suppose the second shortest is (v, w). Then we need to compare $cost(u, v) + cost(v, w)$ and $cost(u, w)$. If $cost(u, v) + cost(v, w) < cost(u, w)$, or (u, w) does not exist, we abbreviate the path (u, v, w) by $< u, w >$ and call it a pseudo edge with cost $cost(u, v) + cost(v, w)$. It is possible to keep track of actual paths, but we focus on the distances of pseudo edges. As the algorithm proceeds, we maintain many pseudo edges with costs which are the costs of the corresponding paths. We maintain pseudo edges with the costs defined in this way as keys in a priority queue. The hidden path algorithm is slightly modifed in the following. An edge $e = (u, v)$ is *optimal* if $cost(u, v)$ is the shortest distance from u to v, that is, if edge (u, v) is returned by the delete-min operation in the following algorithm.

Algorithm 3

```
1  for (u,v) ∈ V × V do d[u,v] := ∞;
   // array d is the container for the result.
2  for (u,v) ∈ E do d[u,v] := cost(u,v);
   F := {< u,v > |(u,v) ∈ E};
3  Organize F in a priority queue with
     d[u,v] as a key for e =< u,v >;
4  S := {(v,v)|v ∈ V};
5  while |S| < n^2 do begin
6    Let e =< u,v > be the minimum
       pseudo edge in F;
7    Delete e from F; S := S ∪ {e};
8    if < u,v > is an edge then begin
9      Mark e = (u,v) optimal;
10     for t ∈ V do update(t,u,v);
11   end;
12   for w ∈ V such that (v,w) is optimal do
       update(u,v,w);
13 end;
14 procedure update(u,v,w) begin
15   if < u,w > ∉ S then begin
16     if < u,w > is in F then
         d[u,w] := min{d[u,w], d[u,v] + cost(v,w)}
17     else begin d[u,w] := d[u,v] + cost(v,w);
       F := F ∪ {< u,w >} end;
18     Reorganize F into queue with the new key ;
19   end
20 end
```

We perform decrease-key or insert in the procedure *update*, which takes $O(1)$ time each. *Update* is to extend a pseudo edge with an optimal edge appended at the end. Note that *update* at line 10 is necessary because when pseudo edge $< t, u >$ was updated, the edge (u, v) might not have been an optimal edge yet. The total time taken for all *updates* is obviously $O(m^*n)$, where m^* is the number of optimal edges. We perform delete-min operations at lines 6-7. If $c < n^2$, several pseudo edges with the same cost may be returned from the same bucket. In this case, those pseudo edges are processed in batch mode without calling the next delete-min. The correctness is seen from the fact that if a pseudo edge is returned at line 6, the corresponding path is an extension of a pseudo edge in S with an extension by an optimal edge at the end. The following lemma is similar to Lemma 1, from which subsequent results can be derived. It guarantees the number of different key values in the priority queue is bounded by c.

LEMMA 2 *For any $< u, v >$ and $< w, y >$ in F, we have $|d[u, v] - d[w, y]| \leq c$*

Proof Let $< u, v >$ and $< w, y >$ in F be such that $d[u, v] \leq d[w, y]$. Let $< w, y >$ be an extension of some pseudo edge $< w, x >$ in S with an optimal edge (x, y), we have $d[w, y] = d[w, x] + cost(x, y)$. On the other hand, we have $d[w, x] \leq d[u, v]$ from the algorithm. Thus we have $d[w, y] - d[u, v] = d[w, x] - d[u, v] + cost(x, y) \leq c$.

We have the following obvious lemma.

LEMMA 3 *The number of different shortest distances for the all pairs shortest path problem for the graph with edge cost bounded by c is at most $c(n - 1)$.*

Theorem The all pairs shortest path problem can be solved in $O(m^*n + nc)$ time, where m^* is the number of optimal edges, satisfying $m^* \geq n$.

For small $c \leq m^*$, the time becomes $O(m^*n)$. If $c \leq m$, since $m^* \leq m$, the time becomes $O(mn)$. If $c > O(n\sqrt{\log n})$, we can switch to the radix heap. The complexity of deletes in line 7 is $O(n^2)$, which is absorbed into $O(m^*n)$ if $m^* \geq n$. We can assume this if the graph is connected when direction of edges is removed. Algorithm 3 can be regarded as simultaneous execution of SSSP's.

In the paper (5), pseudo edges are extended backward. We can extend pseudo edges into both directions, forward and backward. This version will reduce the number of *delete-min* operations, but it is not known whether we can improve the asymptotic complexity.

5 Acyclic graphs and sweeping algorithm

To discuss a nearly acyclic graph, we start with an algorithm for the simpler case of an acyclic graph. Let $G = (V, E)$ is an acyclic graph. Edge costs are non-negative real numbers from this point onwards.

Algorithm 4 Sweeping algorithm

```
1 Topologically sort V and assume
    without loss of generality V = {v_1, ..., v_n}
    where (v_i, v_j) ∈ E ⇔ i < j;
2 d[v_1] := 0;
    //v_1 is the source. This value, 0, will be modified in
    //Algorithm 6
3 for i := 2 to n do d[v_i] := ∞;
4 for i := 1 to n do
5   for v_j such that (v_i, v_j) ∈ E do
6     d[v_j] := min{d[v_j], d[v_i] + cost(v_i, v_j)}.
```

Obviously the time for this algorithm is $O(m + n)$. Similarly to Algorithm 2, we can define a single sink algorithm for an acyclic graph, which sweeps the graph in reverse topological order.

In the next section, we try to find acyclic structures from the given graph, and apply the above algorithm to those structures.

6 1-dominator decomposition

We review the theory of acyclic decomposition developed in (7) for use in the next section. Let $G = (V, E)$ be given. We say A_v is an acyclic structure dominated by v if $A_v - v$ induces an acyclic subgraph of G, every vertex in A_v is reachable from v, and every path from outside of A_v to any vertex in A_v must go through v. A_v can be defined as the fixed point by the following iterative definition.

$A_v \leftarrow \{v\}$;
$A_v \leftarrow A_v \cup \{w | IN(w) \subseteq A_v \wedge IN(w) \neq \emptyset\}$;

The set V can be decomposed into $A_{v_1}, ..., A_{v_r}$ such that they are mutually disjoint and maximal, which is called the 1-dominator decomposition of G. Each v_i is called the root of the maximal acyclic structure A_{v_i}. The following recursive algorithm actually computes acyclic structures A_v's. The above definition looks like a breadth-first approach. The following algorithm, called "forward DFS", computes A_v's one by one in the depth-first fashion. When the algorithm visits a vertex, it decreases the counter, which is initialized to the number of incoming edges, and backtracks. If the counter becomes 0, the algorithm goes through the vertex, or *unlocks* it. We can define backward acyclic structures and "backward DFS" by using IN instead of OUT. By default we mean forward.

Algorithm 5 *Forward-DFS*

```
 1 procedure DFS(v);
 2 begin
 3   for each w in OUT(v) do begin
 4     if w ≠ v0 then begin
 5       count[w] := count[w] − 1;
 6       if count[w] = 0 then begin
 7         is_root[w] := false;
 8         Add w to A_{v0};
 9         DFS(w);
10       end
11     end
12   end
13 end
14 begin /* main program */
15   for each v in V do begin
16     is_root[v] := true;
17     A_v := {v};
18   end
19   for each v in V do begin
20       for each w in V do count[w] := |IN(w)|;
21       v0:=v; count(v0):=count(v0)+1;
         //prevents re-traversal of v0
22       DFS(v);
23   end
24 end // v is a root if is_root[v] = true
```

If w is included in the set A_{v_0} by this algorithm, we say w is dominated by v_0.

LEMMA 4 *(8) If $w \in A_v$, then $A_w \subseteq A_v$. For $u \neq v$, if A_u and A_v are maximal, $A_u \cap A_v = \phi$.*

Some A_w may be included by another A_v later, in which case $is\_root[w]$ becomes false at line 7. After computation is over, the collection of maximal A_v's are set-wise unique in the sense that there may be $A_u = A_v$ for some $u \neq v$. The time for this algorithm is $O(mn)$. In (8), an $O(m)$ time algorithm for this decomposition is described. For our claim of time complexity, this algorithm is sufficient. The following algorithm computes the single source problem from source s with $O(m + r \log r)$ time, given the 1-dominator decomposition.

We maintain only the roots and s in the priority queue. A border vertex w in A_v is one whose out-going edges, if any, go to the roots w of other maximal A_w. The set $EOUT(v)$ (E for extended) is the set of roots to which there are edges from border vertices of A_v. In the sweeping algorithm for A_v in Algorithm 6, all out-going edges are processed in topological order of A_v, including those going from border vertices to the roots of other maximal acyclic sturctures.

This algorithm takes $O(m+r \log r)$ time since each edge is processed in line 8, the total size of $EOUT$'s is $O(m)$ and only at most r vertices are maintained in the queue. If we use this algorithm for n sources, the APSP problem can be solved in $O(mn + nr \log r)$ time, which will be improved in the next section. We can also define a single sink version of this algorithm with backward acyclic structures, which will be used in the next section.

Algorithm 6

```
 1  d[s] := 0;
 2  F := {s}; // A Fibonacci or 2-3 heap is used
      for the priority queue.
 3  Organize F in a priority queue with d[s] as key;
 4  S := ∅;
 5  while S ≠ V do begin
 6    Find v in F with minimum key and
       delete v from F;
 7    S := S ∪ A_v; // In the next, d[v] is used
       for the initial distance from v.
 8    Perform Sweeping Algorithm on A_v from v
       as source;
 9    for w ∈ EOUT(v) do begin
10      if w ∉ S then //w is the root of another A_w.
11        if w ∉ F then F := F ∪ {w};
12      Reorganize F into queue with new d[w];
13    end
14  end.
```

7 Near acyclicity by feedback vertices

A more general definition of a nearly acyclic graph is by the set of feedback vertices, T. Let $r = |T|$. By definition, the induced graph, G', from the complement $T' = V - T$ becomes an acyclic graph. If r is small, the given graph is nearly acyclic. Note that the set of the roots obtained in the 1-dominator decomposition is a set of feedback vertices.

Next we define the reduced graph $G_T = (T, E_T)$, where T is the set of feedback vertices, and E_T is the set of edges defined as follows: an edge (v_i, v_j) for $v_i, v_j \in T$ exists if there is a path from v_i to v_j and its cost is that of the shortest path from v_i to v_j that goes only through the acyclic part T' except for end points v_i and v_j. If there is no path this cost can be defined as infinity.

The costs of $(v_i, v_j) \in E_T$ from each v_i to all other v_j are computed by applying the sweeping algorithm with the source v_i on the original graph that is modified in such a way that the edges from $v \in T$ other than from v_i are cut off. In (8), the APSP problem for G_T is solved, and the solution is shared by n single source problems. If a standard cubic time algorithm is used for APSP, this causes the complexity of $O(mn + r^3)$ since the number of edges $|E_T|$ can be $O(r^2)$. In case that T is the set of roots from the 1-dominator decomposition, $|E_T|$ can be bounded by m, resulting in the total complexity of $O(mn + r^2 \log r)$.

Now the new APSP algorithm is described. We solve single sink problems for all vertices in T, and the result is shared by n single source problems. In

other words, we do not use the graph G_T. We define a generalized single source (GSS) problem (10) by changing the initialization in the SSSP algorithm. Instead of setting $d[s] = 0$ and $d[v] = \infty$ for all other v, we set all $d[v]$ to arbitrary values, say x_v and start the main iteration. This is equivalent to assume a hypothetical source v_0 and set the edge cost of (v_0, v) to x_v. We can generalize other algorithms, single sink, forward and backward sweeping algorithms into the GSS versions. Let graph G_A be the graph obtained from G by removing all incoming edges to vertices in T. From the definition of T, G_A is an acyclic graph.

The APSP algorithm follows. We call the algorithm the 1-dominator version if the roots of the 1-dominator decomposition is used for T, and the feedback vertex version when a set of feedback vertices is used for T. Let $D[u, v]$ be the shortest distance from $u \in V$ to $v \in T$ after the single sink problems are solved in line 1. Array D also serves as the container for the final result.

Algorithm 7 *All pairs shortest paths*
1 **for** *each v in T solve the single sink problem for sink v;*
2 **for** *each u in V* **begin**
3 *Compute shortest distances from u to all $v \in T'$ by performing the forward sweeping algorithm on T';*
4 **for** *each v in T* **do** $d[v] := D[u, v]$;
5 *Compute shortest distances to all $v \in T'$ by performing the GSS forward sweeping algorithm on G_A with the current d;*
6 **for** *each v in T'* **do** $D[u, v] := d[v]$;
7 **end**

Line 1 takes $O(mr + rn \log n)$ time for general T, and $O(mr + r^2 \log r)$ time for T from the 1-dominator decomposition. In those two cases, Algorithm 2 and the single sink version of Algorithm 6 are used respectively. This effort is shared by the following single source problems. Line 3 takes $O(mn)$ time in total. Line 4 takes $O(rn)$ time in total. Lines 5 and 6 take $O(mn)$ time in total. Thus the total time is $O(mn + rn \log n)$ for the general feedback vertex version and $O(mn + r^2 \log r)$ for the 1-dominator version. Other times, such as those for 1-dominator decomposition, topological ordering of T', are absorbed in these complexities. It is possible to change directions of edges all through the algorithm. The definition of r must be modified accordingly.

8 Concluding remarks

We improved the time bounds for the all pairs shortest path problem for special types of graphs by sharing some information among single source problems. In one type, the edge costs are bounded by a small integer c. In this case, a simple bucket system is shared. We assumed the distance values continuously occupy the band of length c. If the values are sparse, there is a room for further speed-up.

The other type is a nearly acyclic graph where we have a small size r of the set of feedback vertices. In this case the result of r single sink problems is shared by n single source problems in a restricted form. One such set of feedback vertices is the set of root vertices in the 1-dominator decomposition, which is treated as a special case of general feedback vertices. It remains to be seen how much we can extend the concept of nearly acyclic graph by identifying an appropriate feedback vertex set. Note that the minimum feedback vertex set is NP-complete.

Added in revision. We came to know the fastest bucket system for delete-min, decrease-key and insert with small integers for SSSP is by Thorup (11). The time is $O(m + n \log \log c)$. Regarding the comment after Lemma 3, we can switch to Thorup's bucket system, when $c > O(n \log \log n)$.

Acknowledgment The authors acknowledge that the constructive comments by the reviewers were very helpful when the paper was revised.

References

[1] Ahuja, K., K. Melhorn, J.B. Orlin, and R.E. Tarjan, Faster algorithms for the shortest path problem, Jour. ACM, 37 (1990) 213-223.

[2] Chan, T., All-Pairs Paths for Unweighted Undirected Graphs in $o(mn)$ Time, Proc. Symp. Discrete Algo. (SODA 06), 514-523 (2006)

[3] Dijkstra, E.W., A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269-271. 1343-1345.

[4] Fredman, M.L. and R,E, Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Jour. ACM 34 (1987) 596-615.

[5] Karger, D.R., D. Koller, and S. J. Phillips, Finding the hidden path: the time bound for all-pairs shortest paths, SIAM Jour. Comput. 22 (1993)1199-1217.

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

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

[8] Saunders, S. and T. Takaoka, Solving shortest paths efficiently on nearly acyclic directed graphs, Theoretical Computer Science, 370(1-3), 94-109 (2007)

[9] Takaoka, T., Theory of 2-3 Heaps, COCOON '99, Lecture Notes in Computer Science (1999)

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

[11] Thorup, M., Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem, Proc. ACM Symp. Theory of Comp. (STOC 2003) 149 – 158, 2003

[12] Vuillemin, J., A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309-314.