Efficient Algorithms for the All Pairs Shortest Path Problem with Limited Edge Costs

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

Abstract

In this paper we deal with a directed graph $G = (V, E)$ with non-negative integer edge costs where the edge costs are bounded by c and $|V| = n$ and $m = |E|$. We show the all pairs shortest path (APSP) problem can be solved in $O(mn + n^2 \log(c/n)))$ time with the data structure of cascading bucket system. The idea for speed-up is to share a single priority queue among n single source shortest path (SSSP) problems that are solved for APSP. We use the traditional computational model such that comparison-addition operations on distance data and random access with $O(\log n)$ bits can be done in $O(1)$ time. Also the graph is not separated, meaning $m \geq n$. Our complexity is best for a relatively large bound on edge cost, c, such that $c = o(n \log n)$.

1 Introduction

We consider the all pairs shortest path (APSP) problem for a directed graph with non-negative edge costs under the classical computational model of addition-comparison on distances and random accessibility by an $O(\log n)$ bit address. The complexity for this problem under the classical computational model is $O(mn + n^2 \log n)$ with a priority queue such as a Fibonacci heap [6]. We improve the second term of the time complexity of the APSP problem for a directed graph with limited edge costs under this computational model.

To deal with such a graph with edge costs bounded by c, referring to two representatives, Ahuja, Melhorn, Orlin, and Tarjan [1] invented the radix heap and achieved $O(m + n\sqrt{\log c})$ time for SSSP. Thorup [10] improved this complexity to $O(m+n \log \log c)$. If we apply these methods n times for the APSP problem, the time complexity becomes $O(mn + n^2\sqrt{\log c})$ and $O(mn + n^2 \log \log c)$ respectively. We first improve the time complexity for this problem to $O(mn+nc)$. When $c \leq m$, that is, for moderately large c, this complexity is $O(mn)$. We finally reduce this to $O(mn + n^2 \log(c/n))$ [1]. For the priority queue we start from a simple bucket system, whose number of buckets is c and the number of delete-min operations is 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 $c < n$, we can reduce the time for delete-min by looking at the fixed range of values, as observed by Dial [4]. We observe the same idea works for the APSP problem in a better way, as shown in [9]. To the author's knowledge, this simple idea is nowhere else in print.

For the more sophisticated priority queue we use a cascading bucket system of k levels to choose minimum distance vertices one by one and modify the distances of other candidate vertices. This data structure is essentially the same as those in Denardo and Wegman [3] and in [1]. Our contribution is to show how to use the data structure, which was used for the SSSP problem, in the APSP problem without increasing the complexity by a factor of n. We mainly follow the style of Cherkassky, Goldberg and Radzik[2]. Taking an analogy from agriculture, the data structure is similar to a combine-harvester machine with k wheels. The first $k-1$ wheels work inside the machine for threshing crops and refining then to finer grains from wheel to wheel, whereas the last wheel goes the distance of $c(n-1)$ and harvests the crops. For SSSP and APSP, the amount of crops is $O(n)$ and $O(n^2)$ respectively.

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, yet simulated by the conventional comparison-addition model by Pettie [8]. Our contribution of $O(mn + n^2 \log(c/n))$ time complexity is better than existing $O(mn + n^2 \log \log n)$ and $O(mn + n^2 \log \log c)$ when $c = o(n \log n)$.

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 double bucket system. In Section 6 we describe the k-level cascading bucket system, which is a generalization of the double bucket system. In Section 7, we show that the hidden path algorithm [7] can fit into our framework of bucket systems, whereby we can have a further speed-up. Specifically m can be replaced by m^*, where m^* is the number of optimal edges, that is, they are part of shortest paths. Section 8 is the conclusion.

Sections 2 and 5 may be redundant as they are special cases of $k = 1$ and $k = 2$ of the k-level bucket system in Section 6. The sections are included for step by step development of our data structures and algorithms. Also some materials from [9] are repeated

[1] Precisely speaking, this should be $O(mn + n^2 \log(c/n + 1))$.

in this paper for the readers' convenience.

2 Simple data structure

In this section we review a well known data structure of priority queue for developments in subsequent sections. The priority queue allows insert, delete and find-min, and is called the bucket system. This is a special case of $k = 1$ in the general k-level cascading bucket system. Specifically, a bucket system consists of an array of pointers, which point to lists of items, called buckets. 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 indices play the role of key values. To insert x is to append x to $list[i]$ if the key of x is i. To perform decrease-key for item x, we decrease the key value of x, say, from i to j, delete x from $list[i]$ and insert it to $list[j]$. To find the minimum for delete-min, we scan the array from the previous position of the minimum and find the first non-nil list, say, $list[i]$, and 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 items. 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 length of one scan is limited to c, and n delete-min operations are done, the total time for delete-min is $O(cn)$. In comparison-based priority queues such as the Fibonacci heap, delete is an expensive operation, whereas in the bucket system it is $O(1)$. We take advantage of this efficiency of delete in more sophisticated bucket systems in subsequent sections.

If the number of different key values at any stage is bounded by c, and the possible number of key values for the who;e process is larger than c, 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 $V = \{1, \cdots, n\}$ and $E \subseteq V \times V$. The non-negative cost of edge (u, v) is denoted by $cost(u, v)$. We assume that $(v, v) \in E$ with $cost(v, v) = 0$ and that $cost(u, v) = \infty$, 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 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 $OUT(v) = \{w|(v, w) \in E\}$. 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. Combined with line 13, line 11 is decrease-key, and lines 12-13 is insert. In this and next sections, we assume edge costs are integers between 0 and c for a positive integer c.

To have a fixed 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 $O(m + cn)$ [4], and space requirement is $O(c + n)$ if we use a circular structure for $list$. If we maintain c buckets in a Fibonacci heap, we can show the time is $O(m + n \log c)$.

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 review [9], improving this time complexity to $O(mn + nc)$. Precisely speaking this complexity can be $O(mn + min\{nc, n^2 \log \log c\})$, as we can switch between the bucket system and the priority queue in [10], 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 Dijkstra's algorithm into a pair-wise version, which puts pairs of vertices 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 edge $cost(u, v)+cost(v, w) < cost(u, w)$, or (u, w) does not exist, we denote 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.

Algorithm 2

```
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 := ∅;
```

```
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    for w ∈ OUT(v) do update(u, v, w);
9  end;
10 procedure update(u, v, w) begin
11   if < u, w > ∉ S then begin
12     if < u, w > is in F then
         d[u, w] := min{d[u, w], d[u, v] + cost(v, w)}
13     else begin d[u, w] := d[u, v] + cost(v, w);
         F := F ∪ {< u, w >} end;
14     Reorganize F into queue with the new key ;
15   end
16 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 edge appended at the end. The total time taken for all *updates* is obviously $O(mn)$. 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 the shortest one that is an extension of a pseudo edge in S with an edge at the end. The following lemma is similar to Lemma 1, from which subsequent theorems 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 edge (x, y), and 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 1 *The all pairs shortest path problem can be solved in $O(mn + nc)$ time [9].*

For $c \leq m$, the time becomes $O(mn)$. The complexity of deletes in line 7 is $O(n^2)$, which is absorbed into $O(mn)$ if $m \geq n$.

In the paper [7], pseudo edges are extended backward. We can extend pseudo edges into both directions, forward and backward. This version will reduce the time for *update* operations, but it is not known whether we can improve the asymptotic complexity.

5 Double bucket system

Let us describe a double bucket system for SSSP first. We have two levels, level 0 and level 1, of buckets. Level 0 has p buckets and level 1 has $\lceil c/p \rceil$ buckets. Each bucket has its interval that indicates which vertex can be put in the bucket, i.e., if the key of a vertex is in the interval, it can be in the bucket. The range of key values for the frontier, F, is split into $\lceil c/p \rceil$ intervals at level 1, and p values at level 0. We aslo maintain the number of elements in each bucket.

The active pointer, a_i, at level i is pointing to the first non-empty bucket at level i, called the active bucket. For initialization, we express the distance of (s, v) in the SSSP, $d[v]$, by two integers (x, y):

$$d[v] = xp + y \qquad (1)$$
$$(0 \leq x \leq \lceil c/p \rceil - 1, 0 \leq y \leq p - 1)$$

Note that there are up to c possibilities for (x, y). If $d[v]$ is specified by (x, y), and $x = 0$, v is put in the y-th bucket at level 0. If $x \neq 0$, v is put in the x-th bucket at level 1. For general $d[v]$, the range for x in (1) becomes $0 \leq x \leq \lceil cn/p \rceil - 1$, and the possibilities of (x, y) is up to cn. As there are only c possibilities for $d[v]$ for v in F, we can have a circular structure for level 1. The range of x is $a_1 \leq x \leq a_1 + \lceil c/p \rceil - 1$. We use a non-circular structure for level 1 for simplicity.

We describe delete-min. Let $a_0 = p$ when level 0 is empty. Using this information, if level 0 is non-empty, we find the first non-empty bucket. Take all vertices in the bucket and perform update. Then move a_0 to the next non-empty bucket using the information from the numbers of elements of buckets. If level 0 is empty, we go to the first non-empty bucket at level 1 using the active pointer, and we re-distribute all vertices in the bucket to the buckets at level 0 using information from their key values and the active pointer. Suppose this is the x-th bucket at level 1, that is, $a_1 = x$. In this case the bucket contains vertices with key values in the range $[xp, (x+1)p-1]$, meaning that we only know an approximate key value if we pick a vertex from this bucket, but re-distributing vertices to level 0 makes the distances exact with v in the y-th, that is, $(d[v] - xp)$-th bucket at level 0. We say $a_1 p$ is the base for level 0, as the actual key value is the bucket number plus the base. The active pointer a_1 is increased until it hits the next non-empty bucket at level 1. Then we perform the delete-min at level 0 as described above. The vertices move from level 1 to level 0 only once. The scanning of level 0 is done up to n times, each scanning taking $O(p)$ time in the worst case. The movement of the active pointer a_1 is at most $O(cn/p)$, as there are $O(cn)$ values for distances and the active pointer goes over distance p by one movement. Thus the total time for delete-min is $O(pn + cn/p)$. Level 1 goes over $O(cn)$ space. If we organize level 1 into a circular structure we can do it in $O(c/p)$ space apart from $O(n)$ for all buckets.

Next we describe insert and decrease-key. When we insert w with its key $d[w]$, we compute (x, y) by equation (1), and put it into an appropriate bucket, x-th or y-th described above. When we perform decrease-key on w, we delete w from its current location, and insert it with the new key value. This operation is $O(1)$. Thus the total time for insert/decrease-key is $O(m)$.

The total time is given by $O(m + pn + cn/p)$. Let $p = \sqrt{c}$. Then the total time becomes $O(m + n\sqrt{c})$.

Now let us run Algorithm 2 for the APSP problem. The time for insert/decrease-key is $O(mn)$ in total, as update takes place m times for each u at line 8. The time for delete-min is $O(n^2 p + cn/p)$. This is because for each movement of a pair from level 1 to level 0, $O(p)$-time scanning of level 0 can take place, causing $O(pn^2)$ time. The important observation is that the movement of the active pointer at level 1 is still $O(cn/p)$ because we have only $O(cn)$ possibilities for distances in the APSP problem and an increase in the active pointer a_1 effectively skips p values over the distance range, if necessary for a delete-min. Let $p = \sqrt{c/n}$. Then the total time becomes $O(mn + n\sqrt{cn}) = O(mn + n^2\sqrt{c/n})$.

6 k-level cascade bucket system

Now we generalize the previous 2-level bucket system into k-level (cascading) bucket system. We call this cascading because the movement of vertices from level to level looks like water flow from high level to low level. Let a_i be the active pointer, which is defined by the minimum index of the non-empty bucket at level i, initialized to 0. Now the key value $d[v] = cost(s, v)$ is given by

$$d[v] = x_{k-1}p^{k-1} + ... + x_1p + x_0 \qquad (2)$$
$$(0 \le x_0, x_1, ..., x_{k-2} \le p-1, 0 \le x_{k-1} \le \lceil c/p^{k-1} \rceil - 1)$$

The initial insert of v with key $d[v]$ can be done as follows: We compute $(x_{k-1}, ..., x_1, x_0)$ from (2). Let i be such that i is the largest index of non-zero x_i, that is, $x_i \neq a_i$. Then put v in the x_i-th bucket at level i. During putting all v, a_i are set to correct values in time proportional to the number of v inserted.

The range for x_{k-1} becomes $0 \le x_{k-1} \le \lceil cn/p^{k-1} \rceil$ for general $d[v]$. Again we can use a circular structure for the range of $a_{k-1} \le x_{k-1} \le a_{k-1} + \lceil c/p^{k-1} \rceil - 1$.

B_i's are bases of the i-th level, initially all 0. Using the values of a_i's, B_i's are defined by

$$B_i = a_{k-1}p^{k-1} + ... + a_{i+1}p^{i+1}$$
$$B_{i-1} = B_i + a_ip^i, B_{k-1} = 0 \text{ (recurrence formula)}$$

Note that the range of key values for the j-th bucket of i-th level is

$$[B_i + jp^i, B_i + (j+1)p^i - 1] \qquad (3).$$

Furthermore note that B_i are monotone non-decreasing in i and also non-decreasing as computation proceeds. For general $d[v]$, we have the invariant that if vertex v is in level i, the value of i is the largest such that $x_{k-1} = a_{k-1}, ..., x_{i+1} = a_{i+1}$ and $x_i \neq a_i$ in (2). Vertex v is put in the x_i-th bucket in level i correctly at initialization since a_i are all zero, and v is put at level i such that x_i is the first non-zero value when scanned from the largest index. The values of x_i are computed only at initialization in $O(kn)$ time. In the following, the new location of v is determined by (3) with the new value of $d[v]$ being in the interval, and the invariant is kept under the three operations of decrease-key, insert and delete-min. We say interval $[\alpha, \beta]$ at level i is the minimum bucket if the corresponding bucket is non-empty and α is minimum among the intervals at level i.

Let us describe update. Suppose the value of $d[w]$ has been decreased. We delete w from the bucket containing w in $O(1)$ time. Then we compare $d[v]$ with base values B_i of levels i. If w is inserted into the same level, insert can be done in $O(1)$ time. If w goes to a lower level, it takes $O(\ell)$ time where ℓ is the level difference. Since all vertices go from higher to lower levels or stay in the same level, the total time for decrease-key is $O(m + kn)$. Insert of v can be done like decrease-key. The vertex is put at the highest level tentatively and we perform decrease-key with $d[v]$ regarded as the new value. Thus the total time for decrease-key/insert is $O(m + kn)$.

Now we describe the delete-min operation. We maintain the active pointer, a_i, at each level i. If $a_i = p$ except for $i = k-1$, level i is empty. If level 0 is non-empty, we pick up the first non-empty bucket by the active pointer. If level 0 is empty, we go to higher levels for a non-empty level, say the i-th. The key value of vertex v in this bucket given by a_i has B_i as the base. When all vertices v in the bucket are moved to level $i-1$, a_ip^i must be added to the base for level $i-1$. Then the first non-empty bucket at level $i-1$ is re-distributed to level $i-2$, etc. We call this process of repeated re-distribution level by level "cascading". Cascading finally creates at least one non-empty bucket at level 0. Note that for re-distribution we use formula (3) to identify appropriate buckets. Active pointers apart from level i can be set to the minimum bucket at each level in $O(1)$ time through the re-distribution process. Also bases are updated through this process. The pointer a_i moves for the next non-empty bucket, taking $O(p)$ time, when $i < k-1$.

For distribution of vertices to lower levels, all n vertices go from higher level to lower level, taking $O(kn)$ time in total. One find-min takes $O(k+p)$ time if $i < k-1$ and $O(k+c/p^{k-1})$ if $i = k-1$, and there are n find-min operations. Thus the total time for delete-min is $O(kn+pn+cn/p^{k-1})$. The total time for SSSP is $O(m + n(k+p) + cn/p^{k-1}) = O(m + kn + c^{1/k}n)$, where $p = c^{1/k}$. This complexity is $O(m + n\log c)$ when $k = O(\log c)$ and $O(m + n\log c/\log\log c)$ when $k = O(\log c/\log\log c)$.

Now we turn to the APSP problem. Let us use a k-level cascading bucket system as a priority queue in Algorithm 3. Instead of vertices, we maintain vertex pairs in the queue. As $O(mn)$ decrease-key/insert operations are done and $O(n^2)$ pairs are moving to lower levels, the total time for this part is $O(mn + kn^2)$.

To find the minimum, we scan the levels from lower to higher for a non-empty level. Then, after finding the minimum bucket we scan the i-th level for the next non-empty bucket for updating a_i, taking $O(k+p)$ time for each delete-min if $i < k-1$. Thus the total time for delete-min becomes $O(kn^2 + pn^2)$ if $i < k$. The total time for the advancement of the active pointer a_{k-1} is $O(cn/p^{k-1})$, same as that for SSSP. The total time for delete-min becomes $O(kn^2 + pn^2 + cn/p^{k-1})$. Thus the complexity for APSP becomes $O(mn + kn^2 + pn^2 + cn/p^{k-1})$.

Setting $p = (c/n)^{1/k}$ yields the complexity of $O(mn + kn^2 + n^2(c/n)^{1/k})$ for the APSP problem. When $k = 2$, we have the result in the previous section. Further setting $k = \log(c/n)$ yields the complexity of $O(mn + n^2\log(c/n))$. Further optimizing yields $O(mn + n^2\log(c/n)/\log\log(c/n))$ with $k = O(\log(c/n)/\log\log(c/n))$. This comlexity is better than the best for APSP for $c = o(n\log^r n)$ for any $r > 0$.

7 Hidden path algorithm

The hidden path algorithm in [7] is a refinement of Algorithm 2 where update is only by optimal edges. 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.

Algorithm 3 *Hidden path algorithm*

1 **for** $(u, v) \in V \times V$ **do** $d[u, v] := \infty$;
  // array d is the container for the result.
2 **for** $(u, v) \in E$ **do** $d[u, v] := cost(u, v)$;
  $F := \{< u, v > | (u, v) \in E\}$;
3 Organize F in a priority queue with $d[u, v]$ as a key for $e =< u, v >$;
4 $S := \emptyset$;
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 such that < t, u >∈ S 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
```

Note that *update* at line 10 is necessary because when pseudo edge $< t, u >$ was put into S, the edge (u, v) might not have been an optimal edge yet.

The range of distance values in the frontier is limited in the band of c from the following lemma similar to Lemma 2.

LEMMA 4 *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), and 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 $d[w, y] - d[u, v] = d[w, x] - d[u, v] + cost(x, y) \leq c$. Note that pseudo edge $< w, x >$ can be formed before or after edge (x, y) is found to be optimal. In either case, the update of pseudo edge from w to y via x is done in line 10 or 12 respectively with an optimal edge (x, y).

Let m^* be the number of optimal edges. From this lemma, we see the data structures developed in the previous sections are compatible with Algorithm 3, so that all time complexities hold with m replaced with $m^*(\leq m)$ when Algorithm 3 is used in place of Algorithm 2.

8 Concluding remarks

We improved the time bounds for the all pairs shortest path problem when the edge costs are limited by an integer $c > 0$. Although the gain obtained is small, the importance of our contribution is to show that the direct use of Thorup's data structure for the APSP problem is not optimal. Note that our data structure is not optimal either when we set $c = O(n^\alpha)$ for $\alpha > 1$. Therefore we still have some room for approaching the goal complexity of $O(mn)$. It remains to be seen whether the fact that there are only $c(n-1)$ different distances for the solution of APSP is compatible with Thorup's data structure. If so, we could have the complexity of $O(mn + n^2 \log\log(c/n))$. One may be tempted to split the distance range of band-width c into n intervals of size c/n each and use Thorup's data structure for each interval. Delete-min and insert seem to work for our goal, but decrease-key is hard to implement. This is because decrease-key consists two heap operations of delete and insert that may be in different levels. Delete is hard to implement in $O(1)$ time in that data structure.

Ackmowledgment Discussions with Yuji Nakagawa while the author was at Kansai University in 2010 are highly appreciated.

References

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

[2] Cherkassky, B. V., Goldberg, A. V. and Radzik, T., Shortest paths algorithms: Theory and experimental evaluation, Mathematical Programming 73, 129-174 (1996)

[3] Denardo, E. V. and Fox, B. L., Shortest-route methods: I. Reaching, pruning, and buckets, Operations Research 27, 161-186 (1979)

[4] Dial, R. B., Algorithm 360: Shortest path forest with topological ordering, CACM 12, 632-633 (1969)

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

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

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

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

[9] Takaoka, T. and Hashim, M., Sharing Information in All Pairs Shortest Path Algorithms, CATS 11, Perth, CRPIT, 119. Alex Potanin and Taso Viglas Eds., ACS. 131-136 (2011)

[10] Thorup, M., Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem, STOC03, 149-158 (2003)