Theoretical Computer Science 520 (2014) 43–50

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Sharing information for the all pairs shortest path problem ☆

Tadao Takaoka

Department of Computer Science, University of Canterbury, Christchurch, New Zealand

ARTICLE INFO

Article history:
Received 17 July 2012
Received in revised form 2 June 2013
Accepted 3 September 2013
Communicated by G.F. Italiano

Keywords:
Information sharing
All pairs shortest path problem
Single sink shortest path problem
Priority queue
Nearly acyclic graph
Limited edge cost

ABSTRACT

We improve the time complexity of the all pairs shortest path (APSP) problem for special classes of directed graphs. One is a nearly acyclic directed graph and the other is a directed graph with limited edge costs. The common idea for speed-up is information sharing by n single source shortest path (SSSP) problems that are solved for APSP. 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 consider this parameter, r, in addition to the traditional parameters of the number of vertices, n, and that of edges, m. In the first part we improve the existing time complexity of $O(mn+r^3)$ for the all pairs shortest path problem to $O(mn+rn\log n)$. This complexity is equal to or better than the previous one for all values of r under a reasonable assumption of $m \geqslant n$. In the second part, we deal with a directed graph with non-negative integer edge costs bounded by c. 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. 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 $m \geqslant n-1$.

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 $O(mn+n^2\log n)$ by running the single source shortest path (SSSP) algorithm n times, as the SSSP problem can be solved in $O(m+n\log n)$ time by Dijkstra's algorithm with a priority queue such as a Fibonacci heap [7] or 2–3 heap [12].

We improve the second term of the time complexity of the APSP problem for special classes of directed graphs; nearly acyclic directed graph and directed graph with limited edge costs of non-negative integers. The common idea for those two problems is resource sharing, or information sharing. When we solve n SSSP's, we let them share some resource so that we can spend less time than we would need to solve n SSSP's independently.

We start from nearly acyclic graphs. There can be many definitions for near-acyclicity [10,11,13]. Here we define it by the size of the feedback vertex set, denoted by T. Every cycle goes through at least one vertex in T, meaning that removal of T from the graph cuts all cycles. If the size is small, we say the given graph is nearly acyclic. 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 all vertices in T as sinks with $|T|=r$, 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 equal to or better than the existing complexity of $O(mn+r^3)$ [11] for all values of r when $m \geqslant n$. For the priority

☆ A preliminary version of this paper was presented at [15].

E-mail address: tad@cosc.canterbury.ac.nz.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.09.006

queue we use a Fibonacci heap or 2–3 heap to choose minimum distance vertices one by one and modify the distances of other candidate vertices, which are to be included in the solution set in the future.

In the second part of the paper, we consider the all pairs shortest path (APSP) problem for a directed graph with non-negative edge costs bounded by a positive integer under the classical computational model.

Again we improve the second term of the time complexity of $O(mn + n^2 \log n)$. To deal with such a graph with edge costs bounded by c, Ahuja, Melhorn, Orlin, and Tarjan [1] invented the radix heap and achieved $O(m + n\sqrt{\log c})$ time for SSSP. Thorup [16] 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 $O(mn + cn)$. When $c \leqslant m$, 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 with c buckets, which gives nc find-min operations 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 [5]. We observe that the same idea works for the APSP problem in a better way, as shown in [14]. 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 (k will be specified later) to choose minimum distance vertices one by one and modify the distances of other candidate vertices. This data structure is essentially the same as in Denardo and Wegman [4] and in Ahuja et al. [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 [3]. 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 them 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 by Pettie [9] and $o(mn)$ for an unweighted undirected graph by Chan [2]. In [9], extended instructions are simulated by the conventional RAM model. Our complexity of $O(mn + n^2 \log(c/n))$ is better than existing $O(mn + n^2 \log\log n)$ and $O(mn + n^2 \log\log c)$ when $c = n(\log n)^{o(1)}$.

The rest of the paper is as follows: In Section 2, we define shortest path problems: single source, single sink and all pairs. In Section 3, we define the sweeping algorithm on an acyclic graph. In Section 4, we define the generalized single source shortest path problem, where the concept of source is distributed over the entire set of vertices. In Section 5, we define nearly acyclic graph with a set of feedback vertices, and show the APSP problem can be solved in $O(mn + rn \log n)$ time, where r is the size of the feedback vertex set. In Section 6, we introduce a simple data structure of the linear bucket system. In Section 7, we discuss some properties of the shortest path problem with limited edge costs. In Section 8, we introduce the concept of pseudo-edges and show that the bucket system works well for the APSP problem by determining shortest distances directly on pseudo-edges. In Section 9 we describe the k-level cascading bucket system, which is a generalization of the linear bucket system. In Section 10, we show that the hidden path algorithm [8] can fit into our framework of bucket systems, whereby we can have a further speed-up. Specifically m in the complexity bounds can be replaced by m^* in the complexity, where m^* is the number of optimal edges, that is, the edges which are part of shortest paths. Section 11 is the conclusion.

2. 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 $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$. The cost of edge (u, v) is a non-negative real number denoted by $cost(u, v)$. We specify a vertex, s, as the source. A shortest path from s to vertex v is a path such that the sum of edge costs of this path is minimum among all paths from s to v. The minimum cost is also called the shortest distance. In Dijkstra's algorithm [6] 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 vertices v in S and 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 $OUT(v) = \{w \mid (v, w) \in E\}$, and $IN(v) = \{u \mid (u, v) \in E\}$. The solution (the shortest distances from s) is in the array d at the end of the computation. To simplify presentation, only the shortest path distances are calculated, not the shortest paths. We assume all vertices are reachable from the source.

Set F is organized as a heap, such as a Fibonacci heap, that supports delete-min in $O(\log n)$ time, insert/decrease-key in $O(1)$ time. These times are amortized. We could use other heaps that support those operations with the above times in the worst case. Line 6 is delete-min, line 11 is decrease-key, and line 12 is insert.

If we use $IN(v)$ at line 8, u in place of w in lines 8–13 and $cost(u, v)$ in lines 11 and 12, then we name the resulting algorithm Algorithm 2, which solves the single sink shortest path problem. We assume that the sink s is reachable from all vertices.

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

Algorithm 1 Single source.

```
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.
```

The all pairs shortest path (APSP) problem is to compute the shortest distances between all pairs of vertices. An obvious way of solving the APSP is to use Algorithm 1 n times by changing the source n times, or use Algorithm 2 with n sinks. In this approach the time complexity of the APSP becomes n times that of SSSP. We avoid this factor of n by combining Algorithm 1 and Algorithm 2 for solving the APSP for nearly acyclic directed graphs.

3. 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 $G = (V, E)$ be an acyclic graph.

Algorithm 3 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.
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 following sections, we try to find acyclic structures in the given graph, and apply the above algorithm to those structures.

4. Generalized single source problem, GSS

We define a generalized single source (GSS) problem [13] 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 edges (v_0, v) with costs x_v. We can generalize other algorithms, single sink, forward and backward sweeping algorithms into the GSS ones.

5. Near acyclicity by feedback vertices

A 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, we say the given graph is nearly acyclic. In general, a set of T of feedback vertices is part of the input. In some special cases a good feedback set can be computed in an efficient way. One such case is the set of roots in a 1-dominator decomposition, defined in [10] and [11]. Roughly speaking, a 1-dominator decomposition is to decompose the set V into r acyclic parts, members of which can be reached only through the root vertex of each acyclic part. In [11], an $O(m)$ time algorithm for such a decomposition is given.

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 such that there is an edge (v_i, v_j) if there is a path from v_i to v_j in G with all intermediate vertices in T'. The cost of edge (v_i, v_j) for $v_i, v_j \in T$ in the new graph G_T is that of a shortest path from v_i to v_j in G that goes only through vertices in the acyclic part T' except for endpoints v_i and v_j.

The costs of $(v_i, v_j) \in E_T$ from each v_i to all other v_j are computed in $O(m)$ time by applying the sweeping algorithm with the source v_i on the original graph that is modified in such a way that the out-going edges from $v \in T$ other than from v_i are cut off. Let $m' = |E_T|$. In [11], the APSP problem for G_T is solved in $O(m'r + r^2 \log r)$ time, and the solution is shared by n single source problems. In the following, $x_1, \ldots, x_k$ for some k are vertices in T. The shortest path from u to v consists

of the initial portion from u to a vertex x_1 in T, the middle portion through vertices in T or T' and the final portion from a vertex x_k to v, denoted by $(u, \ldots, x_1, \ldots, x_2, \ldots, x_k, \ldots, v)$, where portion $(\ldots)$ is in T'. Once the APSP is solved for G_T, the rest for APSP on G can be done in $O(mn)$ time, resulting in the total time of $O(mn + m'r + r^2 \log r)$ time. For a general feedback vertex set, this causes the complexity of $O(mn + r^3)$ since the number of edges $|E_T|$ can be $O(r^2)$. If the feedback vertex set is that of the roots of a 1-dominator decomposition, $m' \leqslant m$ and the complexity becomes $O(mn + r^2 \log r)$. Note that our result of $O(mn + nr \log r)$ for a general feedback vertex is greater than $O(mn + r^2 \log r)$.

Generally r for the general feedback set is smaller than that for the 1-dominator decomposition. As the cost for the 1-dominator decomposition is not expensive, $O(m)$, we could try this decomposition, and if r for the decomposition is small, we could use the algorithm for the APSP given in [11].

Now the new APSP algorithm is described. We solve single sink problems for all vertices in T as sinks, and the result is shared by n single source problems. In other words, we do not use the graph G_T.

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. Let $D[u, v]$ be the shortest distance from u to v after the single sink problems are solved in line 1. Array D also serves as the container for the final result.

The result for the single source problem for u is accumulated in array d; line 4 is for the initial portion, line 5 is for the middle portion borrowed from line 1 and line 6 is for the final portion.

Algorithm 4 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    d[u] := 0;
4    Compute shortest distances from u to all v ∈ T' by performing the forward sweeping algorithm on the acyclic graph G_A;
5    for each v in T do d[v] := D[u, v];
6    Compute shortest distances from u to all v ∈ T' by performing the GSS forward sweeping algorithm on G_A;
7    for each v in T' do D[u, v] := d[v];
8  end
```

Line 1 takes $O(mr + rn \log n)$ time with Algorithm 2 being used r times. This effort is shared by the following single source problems. Line 4 takes $O(mn)$ time in total. Lines 5 and 7 take $O(rn)$ time and $O(n^2)$ time in total. Line 6 takes $O(mn)$ time in total. Thus the total time is $O(mn + rn \log n)$.

6. Simple priority queue – linear bucket system

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 a linear bucket system, or bucket system for short. 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, that is, to 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 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 is larger than c, we can maintain a circular structure of size c for the array $list$ for space efficiency.

7. Shortest path problems with limited edge costs

From this section onwards, we assume edge costs are integers and $0 \leqslant cost(v, w) \leqslant c$ for all $v, w \in V$ for a positive integer c.

Let us run Algorithm 1. 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 $|d[v] - d[w]| \leqslant c$.*

Proof. Take arbitrary v and w in F such that $d[v] \leqslant d[w]$. 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 $d[u] \leqslant d[v]$ from the algorithm. Thus we have $d[w] - d[v] = d[u] - d[v] + cost(u, w) \leqslant c$.

From this we see that the time for Algorithm 1 is $O(m + cn)$ [5], 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 $O(m + n \log c)$.

8. 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 $O(mn + n^2 c)$. In this section we review [14], 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 choose between the bucket system and the priority queue in [16], 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 $cost(u, v) + cost(v, w) < cost(u, w)$, or (u, w) does not exist, we denote the path (u, v, w) by $\langle u, w\rangle$ 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.

Based on this idea we design the following algorithm, Algorithm 5, which essentially runs n copies of Algorithm 1 concurrently. The n SSSP algorithms share the same priority queue for the set F, which is a set of pseudo-edges. The key for the pseudo-edge, $\langle u, v\rangle$, is denoted by $d[u, v]$, where u indicates we are solving the SSSP with source u and $d[u, v]$ is the distance of a path from u to v. We assume there are pseudo-edges (u, u) with cost 0.

Algorithm 5 Algorithm with pseudo-edges.

```
1  for (u, v) ∈ V × V do d[u, v] := ∞; // array d is the container for the result.
2  for u ∈ V do d[u, u] := 0; F := {⟨u, u⟩ | u ∈ V};
3  Organize F in a priority queue with d[u, u] as a key for e = ⟨u, u⟩;
4  S := ∅;
5  while S ≠ V × V 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 a priority 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 check if the existing pseudo-edge should be replaced by an extended pseudo-edge with an edge appended at the end or insert a new extended pseudo-edge. 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 except for the first can be processed in $O(1)$ time. 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 $\langle u, v\rangle$ and $\langle w, y\rangle$ in F, we have $|d[u, v] - d[w, y]| \leqslant c$.*

Proof. Let $\langle u, v\rangle$ and $\langle w, y\rangle$ in F be such that $d[u, v] \leqslant d[w, y]$. Let $\langle w, y\rangle$ be an extension of some pseudo edge $\langle w, x\rangle$ 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 $d[w, x] \leqslant d[u, v]$ from the algorithm. Thus we have $d[w, y] - d[u, v] = d[w, x] - d[u, v] + cost(x, y) \leqslant c$. □

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 integer edge costs bounded by c is at most $c(n - 1)$.*

Theorem 1. *The all pairs shortest path problem can be solved in $O(mn + cn)$ time* [14].

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

In the paper [8], pseudo edges are extended backward. We can extend pseudo edges into both directions, forward and backward. This version will reduce the time for modified *update* operations, where the priority queue is consulted only when the key value is reduced, but it is not known whether we can improve the asymptotic complexity.

9. *k*-level cascade bucket system

Now we generalize the previous bucket system into the 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 initial key value $d[v] = cost(s, v)$ is given by

$$d[v] = x_{k-1}p^{k-1} + \cdots + x_1 p + x_0 \quad \big(0 \leqslant x_0, x_1, \ldots, x_{k-2} \leqslant p-1, 0 \leqslant x_{k-1} \leqslant \lceil c/p^{k-1} \rceil - 1\big). \tag{2}$$

The initial insert of v with key $d[v]$ can be done as follows: We compute $(x_{k-1}, \ldots, x_1, x_0)$ from (2). Let i be 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 $0 \leqslant x_{k-1} \leqslant \lceil cn/p^{k-1} \rceil$ for general, that is, non-initial, $d[v]$. Again we can use a circular structure for the range of $a_{k-1} \leqslant x_{k-1} \leqslant a_{k-1} + \lceil c/p^{k-1} \rceil - 1$ to save space.

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

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

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

$$\big[B_i + jp^i, B_i + (j+1)p^i - 1\big]. \tag{3}$$

Note also 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 $x_{k-1} = a_{k-1}, \ldots, x_{i+1} = a_{i+1}$ 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 (maximum) bucket if the corresponding bucket is non-empty and α is minimum (maximum) 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 in the maximum bucket 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 $i = k-1$, we regard level i as 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 and pick up the first non-empty bucket. The vertices in this bucket are re-distributed to lower levels. 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 $i-1$, $a_i p^i$ 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 time proportional to the number of movements 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 5. Instead of vertices, we maintain pseudo edges 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 $i < k-1$. The total time for the advancement of the active pointer a_{k-1} is $O(cn/p^{k-1})$, same as that for SSSP, resulting in $O(kn^2 + cn/p^{k-1})$ time for delete-min when $i = k-1$. 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 $k = 1$, 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 complexity is better than the best for APSP for $c = n(\log^r n)^{o(1)}$ for any $r > 0$.

10. Hidden path algorithm

The hidden path algorithm in [8] is a refinement of Algorithm 5 where update is only by out-going 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 6 Hidden path algorithm.

```
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 := {(u, u) | u ∈ V}; for u ∈ V do d[u, u] := 0; Mark all edges non-optimal;
5  while S ≠ V × V 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 ∈ OUT(v) such that (v, w) is optimal do update(u, v, w);
13 end;
14 procedure update(u, v, w)
     The body of update is the same as in Algorithm 5
```

Note that *update* at line 10 is necessary because when pseudo edge $\langle t, u\rangle$ 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, which is another corollary of Lemma 1, similar to Lemma 2.

Lemma 4. *For any $\langle u, v\rangle$ and $\langle w, y\rangle$ in F, we have $|d[u, v] - d[w, y]| \leqslant c$.*

Proof. Let $\langle u, v\rangle$ and $\langle w, y\rangle$ in F be such that $d[u, v] \leqslant d[w, y]$. Let $\langle w, y\rangle$ be an extension of some pseudo edge $\langle w, x\rangle$ 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 $d[w, x] \leqslant d[u, v]$ from the algorithm. Thus we have $d[w, y] - d[u, v] = d[w, x] - d[u, v] + cost(x, y) \leqslant c$. Note that pseudo edge $\langle w, x\rangle$ 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. □

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 6, so that all time complexities hold with m replaced with $m^*(\leqslant m)$ when Algorithm 6 is used in place of Algorithm 5.

11. Concluding remarks

We improved time complexities of the APSP problem for special types of graphs using the idea of information sharing.

First we improved the time complexity of the APSP problem for a nearly acyclic graph where we have a small size r of the set of feedback vertices. The result of r single sink problems is shared by n single source problems.

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. There is a gap between the complexity, $O(mn + rn\log n)$, of the problem with general feedback set and that for the 1-dominator decomposition, $O(mn + r^2 \log r)$. Filling the gap is in the future research agenda.

Next 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 idea is to split the distance range of length cn into n^2 intervals of size c/n each and use Thorup's data structure for each interval. Only the "current" interval is organized in Thorup's data structure and others are just unordered bags. Only when the computation proceeds to a new bag, those elements in the bag are inserted into a new instance of Thorup's data structure. Delete-min and insert seem to work for our goal, but decrease-key is hard to implement. If the result of decrease-key ends up in the same interval, it can be done in $O(1)$ time. If the result goes from the current interval to an unordered bag, decrease-key may be implemented by two heap operations of delete and insert. A delete takes $O(\log\log c)$ time for a general interval of c, and $O(\log\log(c/n))$ time in our situation. It will be our future research to see if we can implement decrease-key in $O(1)$ time in this setting.

Acknowledgements

The author is very thankful to an anonymous referee, whose careful reading and constructive comments greatly improved the quality of the paper.

References

[1] K. Ahuja, K. Melhorn, J.B. Orlin, R.E. Tarjan, Faster algorithms for the shortest path problem, J. ACM 37 (1990) 213–223.
[2] T. Chan, All-pairs paths for unweighted undirected graphs in $o(mn)$ time, in: Proc. Symp. Discrete Algo., SODA 06, 2006, pp. 514–523.
[3] B.V. Cherkassky, A.V. Goldberg, T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, Math. Program. 73 (1996) 129–174.
[4] E.V. Denardo, B.L. Fox, Shortest-route methods: I. Reaching, pruning, and buckets, Oper. Res. 27 (1979) 161–186.
[5] R.B. Dial, Algorithm 360: Shortest path forest with topological ordering, Commun. ACM 12 (1969) 632–633.
[6] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271, 1343–1345.
[7] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (1987) 596–615.
[8] D.R. Karger, D. Koller, S.J. Phillips, Finding the hidden path: the time bound for all-pairs shortest paths, SIAM J. Comput. 22 (1993) 1199–1217.
[9] S. Pettie, A new approach to all-pairs shortest paths on real-weighted graphs, Theor. Comput. Sci. 312 (1) (2004) 47–74.
[10] S. Saunders, T. Takaoka, Improved shortest path algorithms for nearly acyclic graphs, Theor. Comput. Sci. 293 (3) (2003) 535–556.
[11] S. Saunders, T. Takaoka, Solving shortest paths efficiently on nearly acyclic directed graphs, Theor. Comput. Sci. 370 (1–3) (2007) 94–109.
[12] T. Takaoka, Theory of 2–3 heaps, in: COCOON '99, in: Lect. Notes Comput. Sci., 1999.
[13] T. Takaoka, Shortest path algorithms for nearly acyclic directed graphs, Theor. Comput. Sci. 203 (1) (August 1998) 143–150.
[14] T. Takaoka, M. Hashim, Sharing Information in all pairs shortest path algorithms, in: Alex Potanin, Taso Viglas (Eds.), CATS 11, Perth, in: Conf. Res. Pract. Inf. Technol. (CRPIT), vol. 119, ACS, 2011, pp. 131–136.
[15] T. Takaoka, Efficient algorithms for the all pairs shortest path problem with limited edge costs, in: J. Mestre (Ed.), Proc. Computing: The Australasian Theory Symposium, CATS 2012, Melbourne, Australia, in: Conf. Res. Pract. Inf. Technol. (CRPIT), vol. 128, ACS, 2012, pp. 21–26.
[16] M. Thorup, Integer priority queues with decrease key in constant time and the single source shortest paths problem, in: STOC03, 2003, pp. 149–158.