CSCE-629 Analysis of Algorithms

Spring 2019

Instructor: Dr. Jianer Chen
Office: HRBB 315C
Phone: 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: T,Th 11:00 am–12:30 pm

Teaching Assistant: Qin Huang
Office: HRBB 315A
Phone: 402-6216
Email: `huangqin@email.tamu.edu`
Office Hours: MWF 3:00 pm–4:00 pm

Solutions to Assignment #2

(Prepared with TA Qin Huang)

1. Design algorithms for Min(H), Insert(H, a), and Delete(H, i), where the set H is stored in a heap, a is the element to be inserted into the heap H, and i is the index of the element in the heap H to be deleted. Analyze the complexity of your algorithms.

Solutions.

A heap H is a complete binary tree with possibly some of its rightmost leaves missing. Thus, a heap of n nodes has a height bounded by $\log n$. Moreover, because of its special structure, a heap H of n elements can be represented by an array $A[1... \max]$ and an index n so that $A[1...n]$ holds the n elements of H. If we place the elements in the heap level by level, starting from the root, and following the order from left to right, into the array $A[1...n]$, then it is not difficult to verify that for an element $A[i]$ in the heap, $A[2i]$ and $A[2i+1]$ are the two children of $A[i]$, and $A[\lfloor i/2 \rfloor]$ is the father of $A[i]$.

A min-heap $A[1...n]$ is a heap that satisfies the condition that the value of each element in the heap is not larger than the values of its children, i.e., $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$ for all $1 \leq i \leq \lfloor n/2 \rfloor$. It's easy to verify that for all i, the value $A[i]$ is not larger than that of any of its descendants. In particular, the root $A[1]$ has the minimum value in the heap.

Algorithm 1 Min($A[1 \dots \max], n$)

```
return (A[1]);
```

Algorithm 2 Insert($A[1 \dots \max], n; a$)

```
n = n + 1; A[n] = a; i = n;
while  A[⌊i/2⌋] > A[i] and i ≥ 2  do
    swap A[⌊i/2⌋] with A[i];
    i = ⌊i/2⌋;
end while
```

Min(A, n) takes time $O(1)$, Insert($A, n; a$) and Delete($A, n; i$) takes time $O(\log n)$, i.e., the height of the tree. □

Algorithm 3 Delete($A[1 \dots \max], n; i$)

```
A[i] = A[n];  n = n − 1;
if  i ≥ 2 and A[i] < A[⌊i/2⌋]  then
   while  i ≥ 2 and A[⌊i/2⌋] > A[i]  do
      swap A[i] with A[⌊i/2⌋]; /*pushing up*/
      i = ⌊i/2⌋;
   end while
else
   while (2i ≤ n) and (A[i] > A[2i] or A[i] > A[2i + 1]) do
      if (2i = n) or (A[2i] < A[2i + 1]) then
         swap A[i] and A[2i];  i = 2i;
      else
         swap A[i] and A[2i + 1];  i = 2i + 1;
      end if
   end while
end if
```

Remark. In the following questions, you can assume that your graphs are connected.

2. Write the psuedo-code for the Dijstra's algorithm that solves the SINGLE-SOURCE SHORTEST PATH problem. Analyze the complexity of the algorithm (you can assume that the algorithm uses a heap for fringes and you can use your results in Question 1 directly). Give a formal proof that the algorithm works correctly when the edge weights are all non-negative.

Solutions.

Let $G = (V, E)$ be a weighted, directed graph, with weight function $w : E \to R$. Suppose the source is s.

Time complexity: let $n = |V|, m = |E|$. First to note that Insert(F, v), Min(F) and Delete(F, v) takes time $O(\log n)$, where F can be implemented as a 2-3 tree. Lines 1-3 takes time $O(n)$; lines 4-8 takes time $O(deg(s) \log n = O(m \log n)$, where $deg(s)$ is the out-degree of s. The while loops $n - 1$ times, each time line 10 takes $O(\log n)$ time, and so the total time line 10 takes is $O(n \log n)$. During the whole while loops, lines 11 - 19 takes time $O(m \log n)$ because each directed edge is visited at most once. In total, this algorithm takes time $O((m + n) \log n)$.

Correctness proof: let $\delta(s, u)$ be the distance of the shortest path from s to u. It suffices to show that for each vertex $u \in V$, we have dist$[u] = \delta(s, u)$ at the time when the status of u becomes *in-tree* or when u is removed from F. Initially, dist$[s] = 0$ and thus dist$[s] = \delta(s, s)$.

For the purpose of contradiction, let u be the first vertex for which $dist[u] > \delta(s, u)$. Let P be a shortest path from s to u. Prior to removing u from F, P must consist of at least one vertex other than u whose status is $fringe$ or *unseen*; otherwise if all the vertices on P except u have the status *in-tree*, $dist[u]$ must be updated as the minimum distance, i.e., $dist[u] = \delta(s, u)$, contradiction. Let y be the first vertex along P such that $status[y] \neq$ *in-tree*. Let x be the vertex that appears before y along the path P. Then, $status[x] =$ *in-tree* and $dist[x] = \delta(s, x)$. It follows that $status[y] = fringe$ and y is in F. Thus, $dist[y] \leq dist[x] + w(x, y)$. As x and y are on the shortest path P and all edges are non-negative, we have $dist[x] + w(x, y) \leq \delta(s, u)$. Since it's u that is chosen to be removed from F not y, then $dist[u] \leq dist[y]$. As a result, $dist[u] \leq dist[x] + w(x, y) \leq \delta(s, u)$, contradiction. Therefore, $dist[u] = \delta(s, u)$ for all the vertices. $\square$

Algorithm 4 Dijkstra's algorithm($G = (V, E), s, w$)

```
for each v ∈ V do
   status[v] = unseen;
end for
status[s] = in-tree; dist[s] = 0; F = ∅;
for each edge (s, v) ∈ E do
   status[v] = fringe; dad[v] = s;
   dist[v] = w(s, v); Insert(F, v);
end for
while F ≠ ∅ do
   u = Min(F); status[u] = in-tree; Delete(F, u);
   for each edge (u, v) ∈ E do
      if status[v]= unseen then
         status[v] = fringe; dad[v] = u' dist[v] = dist[u] + w(u, v);
         Insert(F, v);
      else if status[v]==fringe and dist[u] + w(u, v) < dist[v] then
         Delete(F, v); dad[v] = u;
         dist[v] = dist[u] + w(u, v); Insert(F, v);
      end if
   end for
end while
return dad[], dist[];
```

3. Develop a linear-time (i.e., $O(m)$-time) algorithm that solves the Single-Source Shortest Path problem for graphs whose edge weights are positive integers bounded by 10. (**Hint.** You can either modify Dijstra's algorithm or consider using Breath-First-Search.)

Solutions.

For each edge (u, v) in the graph, let $w(u, v)$ be its weight. We construct a new graph as follows: in the given graph, for each edge (u, v), insert $w(u, v) - 1$ vertices into (u, v), say $x_1, x_2, ..., x_{w(u,v)-1}$, such that (u, v) becomes a path $(u, x_1, x_2, ..., x_{w(u,v)-1}, v)$. The number of vertices of the new graph is at most $n + 9m$, and the number of edges is at most $10m$.

Use Breath-First-Search to solve the Single-Source Shortest Path problem. The time required is $O(m + n)$. □

4. Consider an extended version of the Shortest-Path problem. Suppose that you want to traverse from city s to city t. In addition, for some reason, you also need to pass through cities x, y, and z (in any order) during your trip. Your objective is to minimize the cost of the trip. The problem can be formulated as a graph problem as follows: Given a positively weighted directed graph G and five vertices s, t, x, y, z, find a path from s to t that contains the vertices x, y, z such that the path is the shortest over all paths from s to t that contain x, y, z, assuming that these paths are allowed to contain repeated vertices and edges. Develop an $O(m \log n)$-time algorithm for this problem.

Solutions.

The problem requires that the path starts at s, ends at t, and passes through all three vertices x, y, and z. However, we do not know the order in which the path passes through the vertices

x, y, and z. Thus, we consider all possible paths that use a possible order of the vertices x, y, and z. Note that the number of possible orders is bounded by 6. Therefore, the algorithm can go as follows:

1. run Dijkstra's algorithm starting from s, we can obtain the $d(s,x), d(s,y), d(s,z)$, $d(s,t)$;
2. run Dijkstra's algorithm starting from x, we obtain $d(x,y), d(x,z), d(x,t)$;
3. run Dijkstra's algorithm starting from y, we compute $d(y,z), d(y,x), d(y,t)$;
4. run Dijkstra's algorithm starting from z, we compute $d(z,x), d(z,y), d(z,t)$;
5. compute the length of the six possible paths, i.e., (s,x,y,z,t), (s,x,z,y,t), (s,y,x,z,t), (s,y,z,x,t), (s,z,x,y,t), and (s,z,y,x,t), and select the one with the minimum distance. For example, given (s,x,y,z,t), the distance is $d(s,x) + d(x,y) + d(y,z) + d(z,t)$.

Since the above algorithm runs Dijkstra's algorithm four times, and Dijkstra's algorithm runs in time $O(m \log n)$, we conclude that the proposed algorithm solves the given problem and runs in time $O(m \log n)$. $\square$