CS-UY 2413: Design&Analysis of Algorithms—Spring 2017 Prof. Boris Aronov
Department of Computer Science and Engineering Prof. Lisa Hellerstein

Homework 8 Solutions

1. Consider the binary heap implementation in Figure 4.16 of the textbook (near the end of Chapter 4). Assume that the elements that could be placed in the heap are numbered consecutively starting at 1. When we refer to element x, x will be the number of the element.

   The heap is stored in an array h whose first index is 1. The keys are stored in an array called *key*, where $key[x]$ is the key of element x. (For Dijkstra's algorithm, the elements are vertices numbered from 1 to $|V|$, and we use the dist array as the key array.)

   In the pseudocode, $|h|$ denotes the number of elements in the heap, and $h^{-1}(x)$ denotes the position (index) of x in array h, so $i = h^{-1}(x)$ implies $h[i] = x$.

   We would like to implement $|h|$ and $h^{-1}(x)$ so they run in constant time. To calculate $|h|$ in constant time, we can use a variable *heapsize*. To calculate $h^{-1}(x)$ in constant time, we can keep an "auxiliary" array array A where $A[x]$ stores the index of h containing element x. For example, if x is at the top of the heap, $A[x] = 1$.

   Although the pseudocode in the text uses $h^{-1}(x)$ and $|h|$, it does not include the necessary lines of code to update heapsize and/or array A. Below is the pseudocode from Figure 4.16 for insert, decreasekey, and the function bubbleup. (We have changed some round brackets to square brackets when array indices are indicated.) Revise this pseudocode to include any necessary updates to heapsize and array A. Replace $h^{-1}(x)$ with an appropriate expression involving A, and $|h|$ with heapsize. Give the revised versions of insert, decreasekey and bubbleup as your answer to this question.

   procedure INSERT(h,x)
   bubbleup(h, x, $|h| + 1$)

   procedure DECREASEKEY(h,x)
   bubbleup(h, x, $h^{-1}(x)$)

   procedure BUBBLEUP(h,x,i)
   (place element x in position i of h, and let it bubble up)
   $p = \lfloor \frac{i}{2} \rfloor$
   while $i \neq 1$ and key[$h[p]$] > key[x] **do**
   $h[i] = h[p]$; $i = p$; $p = \lfloor \frac{i}{2} \rfloor$
   $h[i] = x$

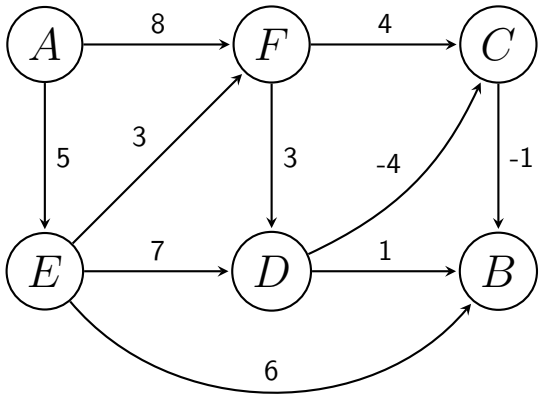
Solution:

procedure INSERT(h,x)
    bubbleup(h, x, $heapsize + 1$)
    $heapsize$ += 1

procedure DECREASEKEY(h,x)
    bubbleup(h, x, $A[x]$)

procedure BUBBLEUP(h,x,i)
    $p = \lfloor \frac{i}{2} \rfloor$
    while $i \neq 1$ and key$[h[p]] >$ key$[x]$ **do**
        $h[i] = h[p]$; $A[h[i]] = i$ (*or* $A[h[p]] = i$); $i = p$; $p = \lfloor \frac{i}{2} \rfloor$
    $h[i] = x$
    $A[x] = i$

2. Run DAG-SHORTEST-PATHS on the directed acyclic graph below to find all shortest distances from A. Give the topological order. Then give the contents of the dist array after processing each vertex.



Solution:

Topological order: A E F D C B

Visiting	A	B	C	D	E	F
At the start	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A	0/nil	∞/nil	∞/nil	∞/nil	5/A	8/A
E	0/nil	11/E	∞/nil	12/E	5/A	8/A
F	0/nil	11/E	12/F	11/F	5/A	8/A
D	0/nil	11/E	7/D	11/F	5/A	8/A
C	0/nil	6/C	7/D	11/F	5/A	8/A
B	0/nil	6/C	7/D	11/F	5/A	8/A

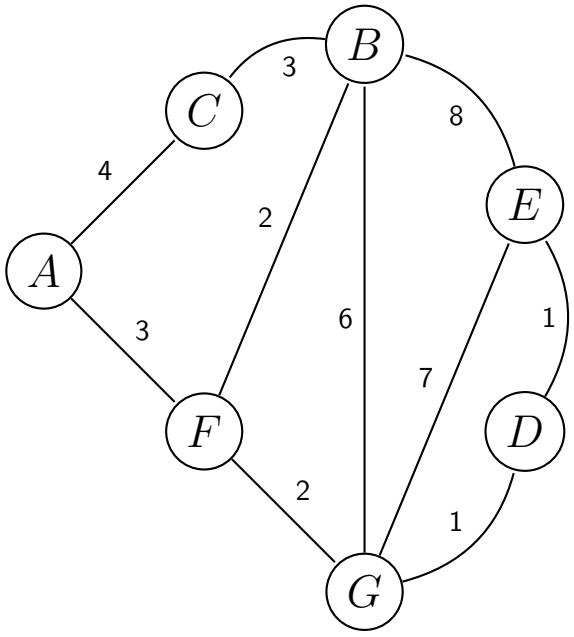
More specifically, if you want to represent the dist array after processing each vertex:
After processing A: $dist = [0, \infty, \infty, \infty, 5, 8]$
After processing E: $dist = [0, 11, \infty, 12, 5, 8]$
After processing F: $dist = [0, 11, 12, 11, 5, 8]$
After processing D: $dist = [0, 11, 7, 11, 5, 8]$
After processing C: $dist = [0, 6, 7, 11, 5, 8]$
After processing B: $dist = [0, 6, 7, 11, 5, 8]$

3. Run the Bellman-Ford algorithm on the graph in the previous problem to find all shortest distances from A, where the edges are processed in lexicographical order (AB, AC, AD, ..., BA, BC, BD, ..., CA, CB, CD, ...; skip the edges that are not actually in the graph). Show the distances for all vertices after each round of UPDATE-ing all edges.

   Solution:

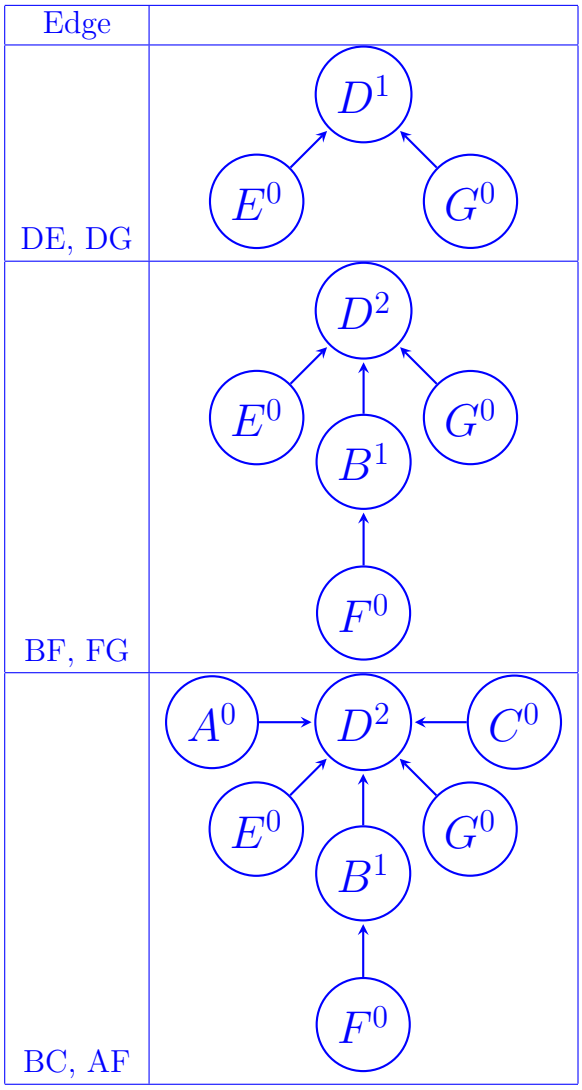
	A	B	C	D	E	F
At the start	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
Round 1	0/nil	11/E	12/F	11/F	5/A	8/A
Round 2	0/nil	11/C	7/D	11/F	5/A	8/A
Round 3	0/nil	6/C	7/D	11/F	5/A	8/A
Round 4	0/nil	6/C	7/D	11/F	5/A	8/A
Round 5	0/nil	6/C	7/D	11/F	5/A	8/A

4. Consider the MST problem on the graph below.



(a) Run Kruskal's algorithm. Show the disjoint sets (Union-Find) data structure after every significant change. (See Figure 5.6 from the textbook.) Do not use path compression.

Solution: Single-node trees are not shown.



(b) What is the total weight of the minimum weight spanning tree on the above graph?

Solution:

The total weight of the minimum spanning tree is 12.

(c) Run Prim's algorithm on the same graph, but just until 3 vertices are in the tree. Whenever there is a choice of nodes, always use alphabetic ordering (start from node A). The first two rows of the table are shown to get you started. **You only need to compute the next two rows of the table!**

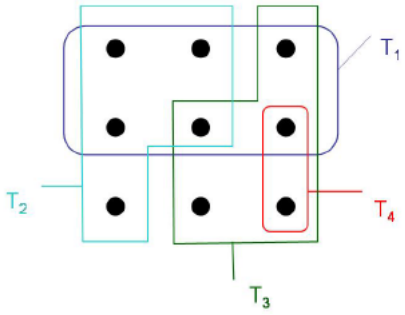
Set	A	B	C	D	E	F	G
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
+A	0/nil	∞/nil	4/A	∞/nil	∞/nil	3/A	∞/nil

Solution:

Set	A	B	C	D	E	F	G
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
+A	0/nil	∞/nil	4/A	∞/nil	∞/nil	3/A	∞/nil
+F	0/nil	2/F	4/A	∞/nil	∞/nil	3/A	2/F
+B	0/nil	2/F	3/B	∞/nil	8/B	3/A	2/F

5. In the Set Cover problem, discussed in Chap. 5 of the textbook, you are given as input a finite set B of elements and a list of subsets of those elements, $S_1, \ldots, S_m \subseteq B$. (This problem was discussed in lecture in Section C.) We call set B the *ground* set. We can think of each set S_i as "covering" the elements it contains. The problem is to choose subsets S_i from this list whose union is equal to B (so that together, the subsets "cover" all the elements of B). The number of chosen subsets should be as small as possible. We call the set of chosen subsets a *minimum cover* of the ground set.

   Consider the following set cover instance, which we are representing graphically.



   Each black dot is an element of set B. The sets S_i are called T_i here.

   (a) There is a cover of all the elements that consists of only two subsets T_i. Which are the two subsets?

   Solution:

   T_2 and T_3

(b) The greedy algorithm for the Set Cover problem chooses the subsets S_i in a greedy way, each time choosing the subset S_i that covers the largest number of elements of B that haven't already been covered. This process is repeated until a cover is formed. This greedy algorithm is NOT guaranteed to find the cover with the fewest number of sets S_i (although as shown in Chap. 5, it does "approximately" solve the problem).

Again, consider the set cover instance represented by the graphic above. Which sets would be chosen to form the cover if we used the greedy algorithm? Does this cover use the smallest possible number of sets T_i?

Solution:

The greedy algorithm chooses T_1 first, then T_3 and finally T_2. This cover uses three sets which is not the smallest number of sets. We already found a set cover that uses only two sets in part a.

(c) Consider the following Menu Problem. You are planning a banquet. Each guest will be allowed to choose one main course from a menu. You have a list L of n main courses that you could put on the menu.

There are m invited guests. Many have dietary restrictions, so you have sent them list L in advance. Each has submitted to you a list of those dishes from L that he or she would be willing to eat. Your job is to choose the smallest possible number of main courses from L to put on the menu so each guest will have at least one main course that he or she is willing to eat.

Show that if you had an algorithm for solving the Menu Problem, you could use it to solve the Set Cover problem. That is, explain how to take an instance of the Set Cover Problem (a ground set B and a set of subsets of B), and turn it into an instance of the Menu Problem (a list L of "main courses", and preference lists for a set of "guests") that you can use as input to your Menu Problem algorithm. Then explain how you would use the solution returned by your algorithm (a set of menu items) to produce a solution to your original instance of the Set Cover Problem.

Solution:

An instance of the Set Cover problem consists of the ground set B and a set of its subsets $S_1, S_2, ...$ We can construct an instance of the Menu Problem as follows: Let the set of m guests be given by B. It follows that each subset S_i is a subset of the m guests. Let there be n dishes, each corresponding to a different subset S_i. For each guest, k, we can specify the list of dishes he/she is willing to eat as the list of dishes for which the corresponding subset S_i contains guest k. This defines an instance of the Menu Problem. Thus, a solution to the Menu Problem (find the fewest dishes to satisfy all guests) corresponds precisely to the Minimum Set Cover (find fewest subsets from $S_1, S_2, ...$ that together cover B). Hence, solving the resulting instance of the Menu Problem will solve the original instance of the Set Cover Problem. Specifically, any cover in the original instance corresponds

to a valid assignment of dishes and vice versa. Moreover, the size of the cover corresponds to the number of main courses. Therefore, the minimum-size cover corresponds to the fewest dishes. This is an example of reduction (which you will learn later in the semester).

6. You have n trucks, $T_1, \ldots, T_n$, sitting in a parking lot waiting to be loaded. For each truck, it costs you \$0.10 per minute to keep it in the parking lot. The trucks have different sizes. For each truck T_i, you know the number of minutes t_i that it will take to load the truck. You can only load one truck at a time. Once you're finished loading a truck, it can immediately leave the parking lot, and you can immediately begin loading another truck.

   Describe an algorithm that determines the optimal order in which to load the trucks, so that you will incur the lowest parking fees possible.

   For example, if there are two trucks, with times $t_1 = 4$ and $t_2 = 3$, then loading T_1 first and T_2 second will incur parking fees of $4 * .10 + 7 * .10 = \$1.10$, since the first truck leaves after 4 minutes, and the second after 7 minutes. But if you loaded them in the opposite order, it would only cost you \$1.00.

   Solution:

   We can use a greedy algorithm that load the trucks that take the least amount of time first. This way, we lower our cost by removing trucks as fast as possible. Observe the following:

   Our goal is to minimize the cost. An important fact is that all trucks cost \$0.10 per minute to keep in the parking lot.
   Let T_i^* denote the i^{th} truck to be loaded and t_i^* be the loading time for truck T_i^*.
   Thus, trucks $T_1^*, T_2^*, ..., T_n^*$ will have loading times $t_1^*, t_2^*, ..., t_n^*$.

   Note: Trucks $T_1^*, T_2^*, ..., T_n^*$ are not in any particular order.
   If we load trucks $T_1^*, T_2^*, ..., T_n^*$ in this order our cost will be:
   Cost $= 0.10(t_1^*) + 0.10(t_1^* + t_2^*) + ... + 0.10(t_1^* + t_2^* + ... + t_n^*)$
   Factoring out 0.10:
   Cost $= 0.10(nt_1^* + (n-1)t_2^* + ... + 2t_{n-1}^* + t_n^*)$
   We know that $n > (n-1) > (n-2) > ... > 2 > 1$
   Thus, in order to minimize this, we need $t_1^* < t_2^* < ... < t_n^*$ such that each multiplication of the i^{th} largest term in t_i^* by the i^{th} smallest term in $(n, n-1, ..., 2, 1)$ would yield the smallest result.
   Intuitively, we can view this as saying that if truck i has a faster loading time than truck j, it must be loaded before truck j because if we were to swap trucks i and j in our loading order, we would only increase the total cost. Thus, the greedy algorithm which picks the fastest truck to load at each step is optimal.