CMP_SC: Homework 5 answers

1. Recall the Dijkstra algorithm we did in class (and given in Section 24.3 of the book). The Dijkstra algorithm computes the shortest distance from a source vertex to every other vertex in a weighted directed graph as long as the weights are non-negative. (By shortest, we mean that the sum of all weights of the edges should be minimum amongst all paths).

   (a) Consider the weighted directed graph G shown in Figure 1:

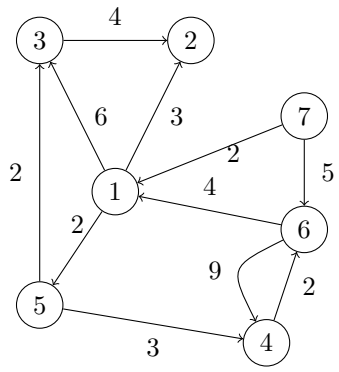


Figure 1: Graph G

   For each vertex u in the graph, show how the values $u.d$ change when Dijkstra's algorithm is executed with the vertex numbered 1 as the source vertex.

Answer: The iteration number, the vertex added to S and $u.d$ values are given below.

Iteration No.	Vertex added to S	1.d	2.d	3.d	4.d	5.d	6.d	7.d
0	NA	0	∞	∞	∞	∞	∞	∞
1	1	0	3	6	∞	2	∞	∞
2	5	0	3	4	5	2	∞	∞
3	2	0	3	4	5	2	∞	∞
4	3	0	3	4	5	2	∞	∞
5	4	0	3	4	5	2	7	∞
6	6	0	3	4	5	2	7	∞
7	7	0	3	4	5	2	7	∞

(b) Dijkstra algorithm is supposed to run only on graphs with non-negative weights. Let us see what happens when we run Dijkstra's algorithm on graphs with negative weights. Consider the weighted directed graph H shown in Figure 2:

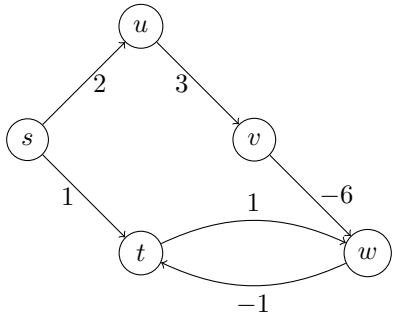


Figure 2: Graph H

- What is the shortest distance from the vertex labeled s to all other vertices in the above graph?
- Suppose we run the Dijkstra's algorithm on the graph H with s as the source vertex. What are the values of the distances computed?

Answer:

- Shortest distance from s to other vertices are as follows:

s	u	v	w	t
0	2	5	-1	-2

- Running Dijkstra's on the graph will give:

s	u	v	w	t
0	2	5	-1	1

2. A weighted undirected graph $G = (V, E)$ is an *almost-tree* if

   - G is connected, and
   - $|E| = |V|$ (that is the number of edges and the number of vertices are exactly the same).

   Give an algorithm that given an almost-tree $G = (V, E)$ computes the minimum spanning tree of G and runs in $O(|V|)$ time. You must justify why your algorithm works and runs in $O(|V|)$ time in order to get full credit.

   Answer: An almost-tree is a spanning tree plus one *extra edge*, which is part of the *unique cycle*. (There can be only one cycle in an almost-tree). If we remove one edge from the unique cycle then we shall have a connected graph with $|V|$-1 edges, which is exactly a tree. Any edge of the cycle will do. On the other hand, if we remove an edge which is not part of the cycle, the graph will be disconnected and we will not get a tree!

   So, our algorithm should first find the unique cycle and then remove one edge of the cycle. The unique cycle can be found by doing a DFS of the graph, which will yield only one back edge. This back edge along with the path that connect the endpoints of the edge in the DFS tree is the unique cycle of the graph.

   Now, the question is which edge of the cycle to remove? Since we want the minimum spanning tree, we must remove the edge of the cycle with the maximum weight.

   The algorithm is $O(|V|)$ since :

   - The DFS and finding the cycle takes $O(|V| + |E|) = O(2|V|)$ time, and
   - finding the maximum-weight edge of the cycle is also $O(|V|)$ time.

3. A sequence of integers $A = a_1, a_2, \ldots, a_n$ is a subsequence of sequence $B = b_1, b_2, \ldots, b_m$ if $a_1, \ldots, a_n$ occur in B in the same order (may or may not occur next to each other). For example,

   - $A = 1, 2, 3, 4$ is a subsequence of $B = 6, 1, 2, 3, 4, 5$.
   - $A = 1, 2, 3, 4$ is **not** a subsequence of $B = 6, 1, 2, 4, 5$.
   - $A = 4, 2, 3, 1$ is a subsequence of $B = 6, 4, 2, 7, 3, 9, 1$.
   - $A = 4, 2, 3, 1$ is a subsequence $B = 3, 4, 2, 7, 3, 9, 4, 3, 1$.
   - $A = 4, 2, 3, 1, 6$ is **not** a subsequence of $B = 4, 2, 7, 3, 9, 6, 4, 3, 1$.

   Give an algorithm that given two sequences of integers A and B checks if A is a subsequence of B or not. Your algorithm must run in $O(n + m)$ time where n and m are the number of elements in the sequences A and B respectively. You must justify why your algorithm works and runs in $O(m + n)$ time in order to get full credit.

Answer: Essentially, we need to first elements of A in B in the same order. Now, since there can be gaps between the elements A in B, observe the following:

- If a_1, the first element of A, is not in B then A cannot be a subsequence of B.
- Assume now that a_1 is in B. a_1 can occur many times in B. Let j_1 be the smallest index such that $b_{j_1} = a_1$. Now, A will be a subsequence of B iff $a_2, a_3, \ldots, a_n$ occur in $b_{j_1}, b_{j_1+1}, \ldots, b_m$ in the same order, i.e., $a_2, a_3, \ldots a_n$ is a subsequence of $b_{j_1}, b_{j_1+1}, \ldots, b_m$. This is because if $a_2, a_3, \ldots, a_n$ occur before b_{j_1} then they do not contribute towards A being a subsequence of B as they do not have a_1 before them.

We can proceed as follows:

(a) We can first check for a_1 in B (start searching from the left). If it is not found in B, then we can declare A to be not a subsequence of B.

(b) If a_1 is found at in B, with the first occurrence at j_1, then we can check for a_2 starting at $j_1 + 1$. If not found then we can declare A to be not a subsequence of B. Otherwise, if found at location j_2, we can search for the a_3 at position $j_2 + 1$ and continue....

This strategy can be easily written as an $O(m + n)$ -algorithm since we only need to traverse the sequence A and B once!

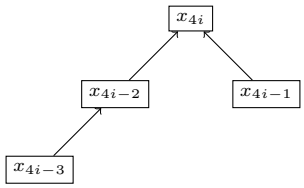
4. Recall the disjoint-set data structure implemented as a forest that we discussed in class and discussed in Section 21.3 of the book. We will assume that the data structure always uses union by rank and path compression heuristics. Assuming that we start a set of objects $x_1, x_2, \ldots x_{16}$, and perform the following sequence of operations:

```
1  for i = 1 to 16
2      MAKE-SET(x_i)
3  for i = 1 to 8
4      UNION(x_{2i-1}, x_{2i})
5  for i = 1 to 4
6      UNION(x_{4i-3}, x_{4i-1})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND-SET(x_2)
11 FIND-SET(x_9)
```

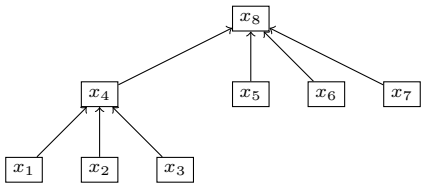
Draw the data structure after the lines 7,8,9,10 and 11 are executed.

Answer: For convenience, I will not draw the self loop at the root of the trees.

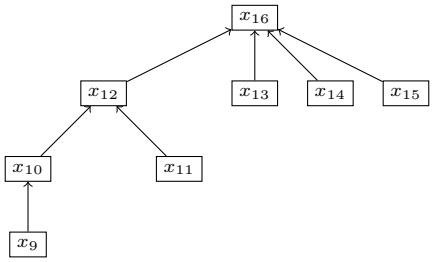
- Before step 7 begins, there are 4 sets S_1, S_2, S_3, S_4 where for each $i = 1, 2, 3, 4$. S_i is of rank 3 and is



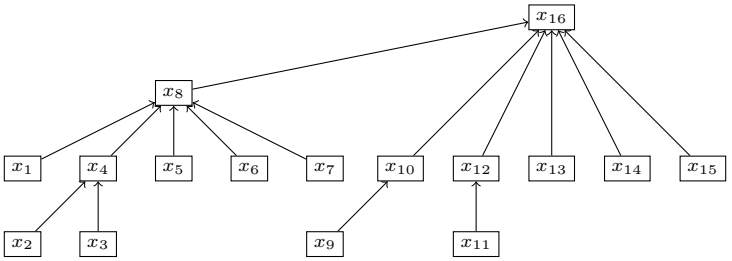
- After step 7 is executed, there are 3 sets A, S_3, S_4 where S_3 and S_4 are as before. A is a set of rank 4 and is



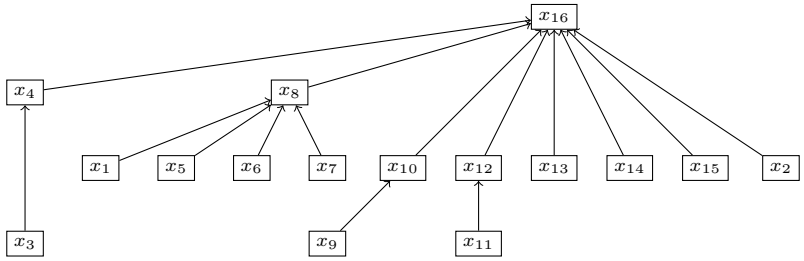
- After step 8 is executed, there are two sets A and B, where A is as before. B is a set of rank 4 and is



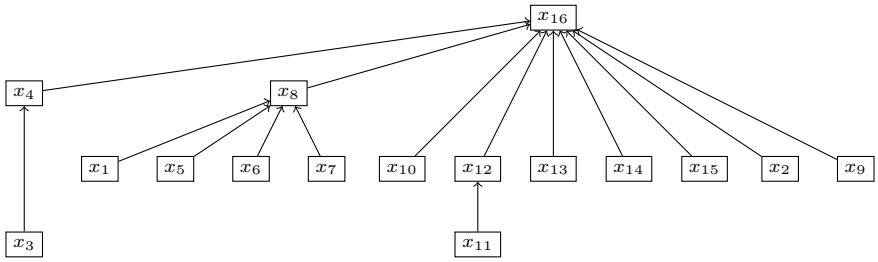
- After step 9 is executed, there is only one set of rank 5 and is



- After Step 10 is executed, the forest consists of one set



- After Step 11 is executed, the forest consists of one set



5. You are given a list L of n natural numbers $d_1, d_2, \ldots, d_n$. Let $m = d_1 + d_2 + \cdots + d_n$. Give an $O(n \log n + m)$ algorithm that takes the list L and returns

   - true if there exists an <u>undirected</u> graph $G = (V, E)$ such that $|V| = n$ and for each number d_i in the list L, there is a vertex of G with degree exactly d_i;
   - and false otherwise.

Answer: Let us first sort the numbers $d_1, d_2, \ldots, d_n$ in decreasing order and call the new list $c_1, c_2, \ldots, c_n$. We can always drop the numbers which are 0 because those vertices do not not have an edge.

Now consider the sequence of $n-1$ numbers

$$c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots c_{n-1}.$$

Basically, the sequence is formed by subtracting 1 from each of the first c_n numbers amongst $c_1, c_2, \ldots, c_n$ and removing the last number c_n.

Now, it is easy to see that if there is an undirected graph with $n-1$ vertices such that the degrees of vertices in decreasing order are $c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots, c_{n-1}$ then there is a graph with n vertices such that the degrees of the vertices in decreasing order are $c_1, c_2, \ldots, c_n$. (Simply add a new vertex and connect it to the first c_n vertices).

Now, also if there is a graph H with n vertices such that the degrees of vertices are $c_1, c_2, \ldots, c_n$ then there must be a graph H' with $n-1$ vertices such that the degree of vertices are $c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots, c_{n-1}$. Why? Consider the vertex u_n of H whose degree is c_n. Now, if its connected to only the vertices of highest degrees then H' is just H with u_n and edges incident on it deleted. Otherwise, there must be vertices v and w such that degree of vertex v is $\geq c_{c_n}$ and w whose degree is $\leq c_{c_{n+1}}$ such that u_n is not incident to v and w is incident to u_n. Now, there must be an edge v' such that v' is adjacent to v but not to w. (Because v has at least as many edges incident to it as w has). Now, we can transform H so that now u_n is incident to v and v' is incident to w.

Therefore, there is an a graph with n vertices such that the degrees of vertices are $c_1, c_2, \ldots, c_n$ if and only if there is an undirected graph with $n-1$ vertices such that the degrees of vertices are $c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots, c_{n-1}$.

This gives a simple recursive algorithm that given $c_1, c_2, \ldots, c_n$ (in decreasing order) checks recursively if there is a graph with $n-1$ vertices such that there degrees of the vertices are $c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots, c_{n-1}$. The function returns true iff the recursive call returns true. The base case is that when we get to just one vertex then the degree must be 0.

Of course, the list $c_1 - 1, c_2 - 1, \ldots c_{c_n} - 1, c_{c_{n+1}}, \ldots, c_{n-1}$ may not be sorted as $c_{c_n} - 1$ may not be $< c_{c_{n+1}}$. This will happen when $c_{c_n} = c_{c_{n+1}}$. So, instead of storing $c_1, c_2, \ldots, c_n$ as list of numbers we store a list of pairs $(b_1, \ell_1), (b_2, \ell_2), \ldots (b_1, \ell_k)$ where $b_1 > b_2 > \ldots b_k$ and for $j = 1, 2, \ldots k$, ℓ_j is the number of times b_j appears in the list $d_1, d_2, \ldots, d_n$.

It is easy to see that this function runs in $O(d_1 + d_2 + \cdots + d_n) = O(m)$. Since we have to first sort $d_1, d_2, \ldots, d_n$, the total running time is $O(n \log n + m)$.