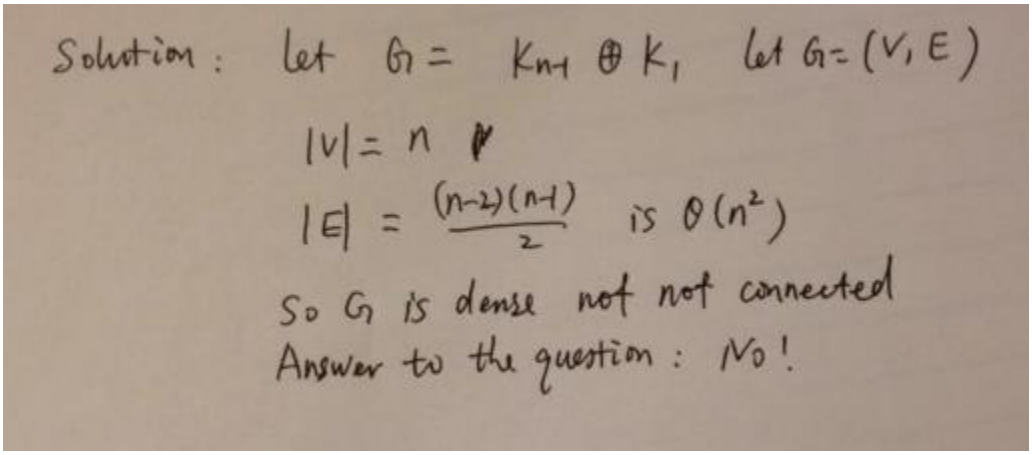
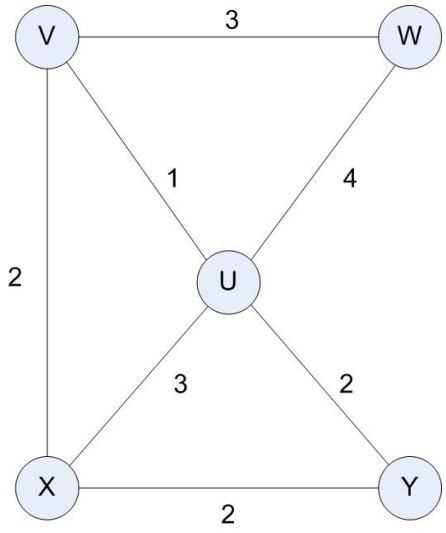
Lab 13 Solutions

1. Must every dense graph be connected? Prove your answer.



2. Carry out the steps of Dijkstra's algorithm to compute the length of the shortest path between vertex V and vertex Y in the graph below. Your final answer should consist of three elements:

    a) The length of the shortest path from V to Y
    b) The list A[] which shows shortest distances between V and every other vertex
    c) The list B[] which shows shortest paths between V and every other vertex



Problem 2 without PQ

Step 0:
$A[v] = 0 \quad B[v] = \phi$

Step 1:
pool $\leftarrow \{(v,u), (v,w), (v,x)\}$
$A[v] + wt(v,u) = 0+1 = 1$ ← minimal
$A[v] + wt(v,w) = 0+3 = 3$
$A[v] + wt(v,x) = 0+2 = 2$
$A[u] = 1 \quad B[u] = B[v] \cup \{(v,u)\} = \{(v,u)\}$
add u to X

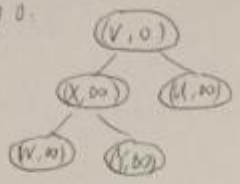
Step 2: pool $\leftarrow \{(v,w), (v,x), (u,w), (u,x), (u,y)\}$
greedy length: 3, 2 ← minimal, 5, 4, 5
$A[x] = 2$
$B[x] = B[v] \cup \{(v,x)\} = \{(v,x)\}$
add x to X

Step 3: pool $\leftarrow \{(v,w), (u,w), (u,y), (x,y)\}$
greedy length: 3 ← minimal, 5, 3, 4
$A[w] = 3$
$B[w] = B[v] \cup \{(v,w)\} = \{(v,w)\}$
add w to X

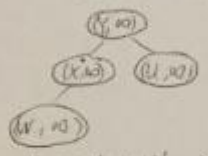
Step 4: pool $\leftarrow \{(u,y), (x,y)\}$
greedy length: 3 ← minimal, 4
$A[y] = 3 \quad B[y] = B[u] \cup \{(u,y)\} = \{(v,u)(u,y)\}$

Problem 2 with PQ

step 0.



step 1: (V, 0) ← remove Min



adj to V and in PQ: W, X, U

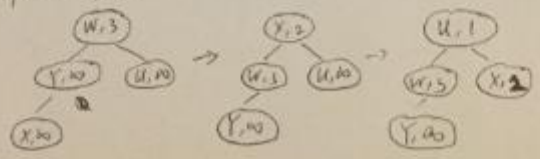
compute greedy lengths for (V,W) (V,X) (V,U)

A[V] + wt(V,W) = 0+3 = 3 < A[W]

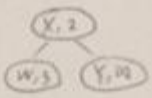
A[V] + wt(V,X) = 0+2 = 2 < A[X]

A[V] + wt(V,U) = 0+1 = 1 < A[U]

update PQ.



Step 2: (u, 1) ← remove Min



adj to u still in PQ: W, X, Y

compute greedy lengths for (u,w) (u,x) (u,y)

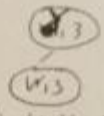
A[u] + wt(u, w) = 1 + 5 = 5 < A[w]

A[u] + wt(u, x) = 1 + 3 = 4 < A[x]

A[u] + wt(u, y) = 1 + 2 = 3 < A[y]

update PQ:

Step 3: (x, 2) ← remove Min



adj to x still in PQ: y

compute greedy length for (x, y)

A[x] + wt(x, y) = 2 + 2 = 4 < A[y]

Step 4: (y, 3) ← remove Min

PQ: (w, 5)

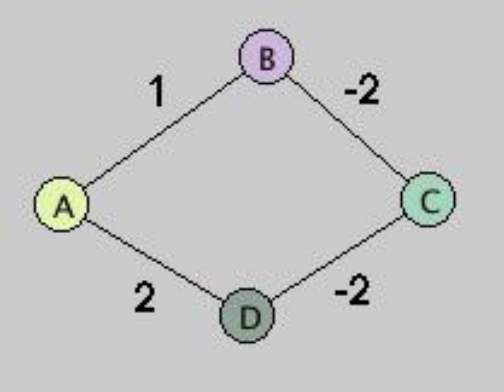
adj to y still in PQ: nothing

Step 5: (w, 5) ← remove Min

adj to y still in PQ: nothing

3. *Points about Dijkstra's Algorithm*

a. What is the shortest path from A to C in the graph below (using any algorithm you like)?

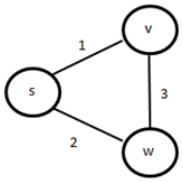


Solution: There is no shortest path because the edge CD can be traversed back and forth as many times as desired to create ever shorter (more negative length) paths.

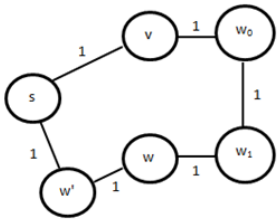
b.

Why is Dijkstra's approach to the shortest path problem better than simply using BFS, as described in the previous lesson?

[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its *level* in the BFS spanning tree. These computed values should be same as values found using Dijkstra]



↓BFS Style



Solution: When edge weights on the original graph are large, the number of new vertices that need to be added causes performance of BFS to slow down. If the sum of the edge weights is as big as n^4, then the running time of the BFS version slows to $\Omega(n^4)$, which is worse than the Dijkstra running time O(mlog n). However, if all edge weights are small, the BFS approach is faster.

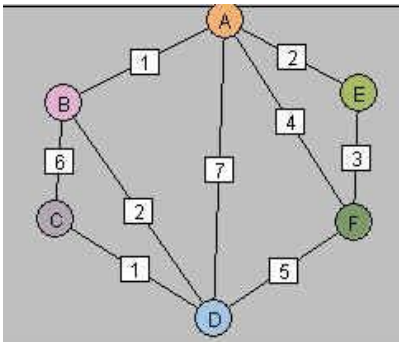
4. Describe an algorithm for deleting a key from a heap-based priority queue that runs in O(log n) time, where n is the number of nodes. (Hint: You may use auxiliary storage as the priority queue is built and maintained. Assume there are no two nodes have the same key.) This technique is needed for the optimized Dijkstra algorithm discussed in the slides.

Solution: Maintain a HashMap where the key of the HashMap is the key used in the priority queue, and the value is the node that contains this key. (This is where we use the assumption that no two nodes have the same key.) This HashMap must be kept synchronized with the priority queue after insert and removeMin operations, but this is easy to do. For instance, when a removeMin operation is done, pulling a node n from the top of the queue, we must also remove n and the key it contains from the HashMap (by looking up this entry in the HashMap using the key).

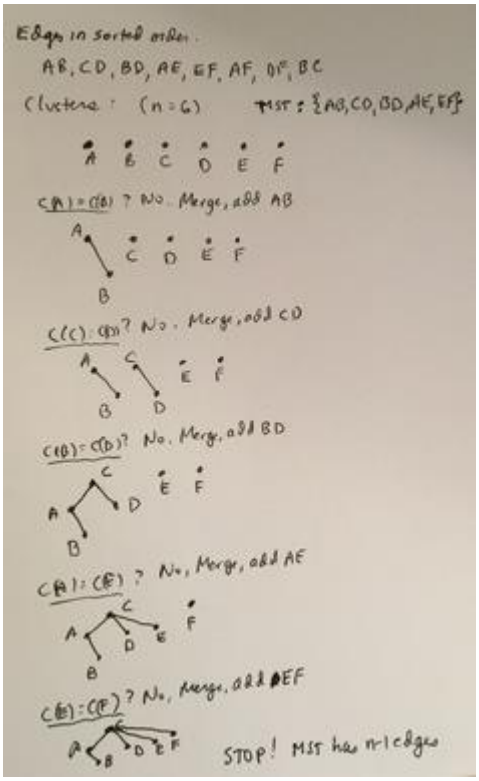
When we need to delete the key, use the key to look up the node in the HashMap, then change the key to a min value (if the keys are integers, use Integer.MIN_VALUE) and do upheap. The modified node will bubble to the root. Then perform removeMin and do downheap in the usual way. The running time of this process is O(log n).

Kruskal's Algorithm and Disjoint Sets

5. Carry out the steps of Kruskal's algorithm for the following weighted graph, using the tree-based DisjointSets data structure to represent clusters. Keep track of edges as they are added to T and show the state of representing trees through each iteration of the main while loop.



Solution:



6. The goal of this exercise is to devise a feasible algorithm that decides whether an input integer is prime. The key fact that you will make use of is the following:

Fact: There is a function *f*, which runs in O(log *n*) (that is, O(length(*n*))), such that for any odd positive integer *n* and any *a* chosen randomly in [1, *n* - 1], if *f*(*a*, *n*) = 1, then *n* is composite, but if *f*(*a*,*n*) = 0, *n* is "probably" prime, but is in fact composite with probability < ½.

A first try at such an algorithm would be:

Algorithm **FirstTry**:

Input: A positive integer n
Ouptut: TRUE if n is prime, FALSE if n is composite

```
if n % 2 = 0 return FALSE
a ← random number in [1, n-1]
if f(a,n) = 1
   return FALSE
return TRUE
```

Notice that **FirstTry** runs in O(log *n*). It also produces a correct result more than half the time.

What could be done to improve the degree of correctness of **FirstTry** but still preserve a reasonably good running time? Explain.

Solution. Our new algorithm will output true correctly with probability $\geq 1 - 1/2^k$, for any specified positive integer k.

The algorithm is the following: Run FirstTry k times. If each run produces TRUE, return TRUE. If one of the runs yields FALSE, return FALSE. We verify that the new algorithm is correct with probability $\geq 1 - 1/2^k$. Let E_i be the event that the algorithm outputs TRUE on the ith run, but the input number is not prime (so FirstTry is incorrect on that run). The E_i are independent and each has probability $\leq$ ½. Therefore

$$\begin{aligned} \Pr(\text{new algorithm is incorrect}) &\leq \Pr(E_1 \cap E_2 \cap E_3 \cap E_k) \\ &= \Pr(E_1) \cdot \Pr(E_2) \cdot \ldots \cdot \Pr(E_k) \\ &= 1/2^k, \end{aligned}$$

as required.

For reasonable choices of k and n, the new algorithm still runs in O(log n). For instance, if k = 50 and n = 100000, this is the case.