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

Dense graph has m dense m

The connected graph m >=

* m dense >=

1. **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:**
   1. **The length of the shortest path from V to Y**
   2. **The list A[] which shows shortest distances between V and every other vertex**
   3. **The list B[] which shows shortest paths between V and every other vertex**



Step 1:

A[V] = 0

B[V] = {}

Put V in S

Step 2:

S = {V}

POOL = {(V, W), (V, U), (V, X)}

Find minimum greedy length

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

A[V] + wt(V, U) = 1 🡨

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

Add U to S and set value of A[U] = 1

B[U] = B[V] U (V, U) = {(V, U)}

Step 3:

S = {V, U}

POOL = {(U, W), (U, X), (U, Y), (V, W), (V, X)}

Find minimum greedy length

A[U] + wt(U, W) = 1 + 4 = 5

A[U] + wt(U, X) = 1 +2 = 3

A[U] + wt(U, Y) = 1 + 2 = 3 🡨

A[U] + wt(V, W) = 1 + 3 = 4

A[U] + wt(V, X) = 1 + 2 = 3

Add Y to S and set value to A[Y] = 3

B[Y] = B[U] U {U, Y} = {(V, U), (U, Y)}

Algorithm complete since Y = V. Computed value

A[V] = 0, A[U] = 1, A[Y] = 3

B[V] = {}, B[U] = {(U, V)}, B[Y] = {(V, U), (U, Y)}

1. **Points about Dijkstra’s Algorithm**
   1. **What is the shortest path from A to C in the graph below?**



Shortest path from A to C is: A-B, B-C



Dijkstra allows assigning distances other than 1 for each step. For example, in routing the distances (or weights) could be assigned by speed, cost, preference, etc. The algorithm then gives you the shortest path from your source to every node in the traversed graph.

Meanwhile BFS basically just expands the search by one “step” (link, edge, whatever you want to call it in your application) on every iteration, which happens to have the effect of finding the smallest number of steps it takes to get to any given node from your source (“root”).

1. **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.**

I don’t know

1. 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.



Sort all the edges by ascending order

AB, CD, AE, BD, EF, AF, DF, BC, AD

|  |  |
| --- | --- |
| C(A) | {A} 🡪 {ABE} 🡪 {ABECD} 🡪 {ABECDEF} |
| C(B) | {B} 🡪 {ABE} 🡪 {ABECD} 🡪 {ABECDEF} |
| C(C) | {C} 🡪 {CD} 🡪 {ABECD} 🡪 {ABECDEF} |
| C(D) | {D} 🡪 {CD} 🡪 {ABECD} 🡪 {ABECDEF} |
| C(E) | {E} 🡪 {ABECDEF} |
| C(F) | {F} 🡪 {ABECDEF} |

T = {}

C(A) & C(B): C(A) ≠ C(B) => Add AB to T = {AB}

C(C) & C(D): C(C) ≠ C(D) => Add CD to T = {AB, CD}

C(A) & C(E): C(A) ≠ C(E) => Add AE to T = {AB, CD, AE}

C(B) & C(D): C(B) ≠ C(D) => Add BD to T = {AB, CD, AE, BD}

C(E) & C(F): C(E) ≠ C(F) => Add EF to T = {AB, CD, AE, BD, EF}

C(A) & C(F): C(E) = C(F) => Discard AF

C(D) & C(F): C(D) = C(F) => Discard DF

C(B) & C(C): C(B) = C(C) => Discard BC

C(A) & C(D): C(A) = C(D) => Discard AD

* The minimal spanning tree is T = {AB, CD, AE, BD, EF}