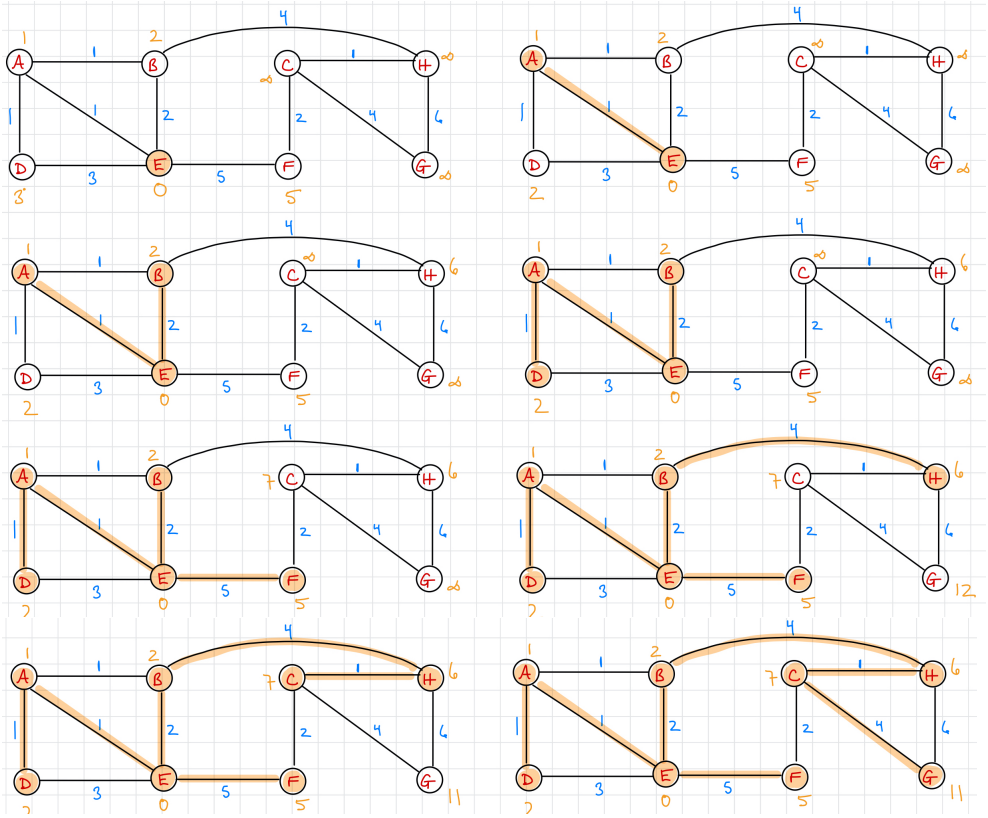
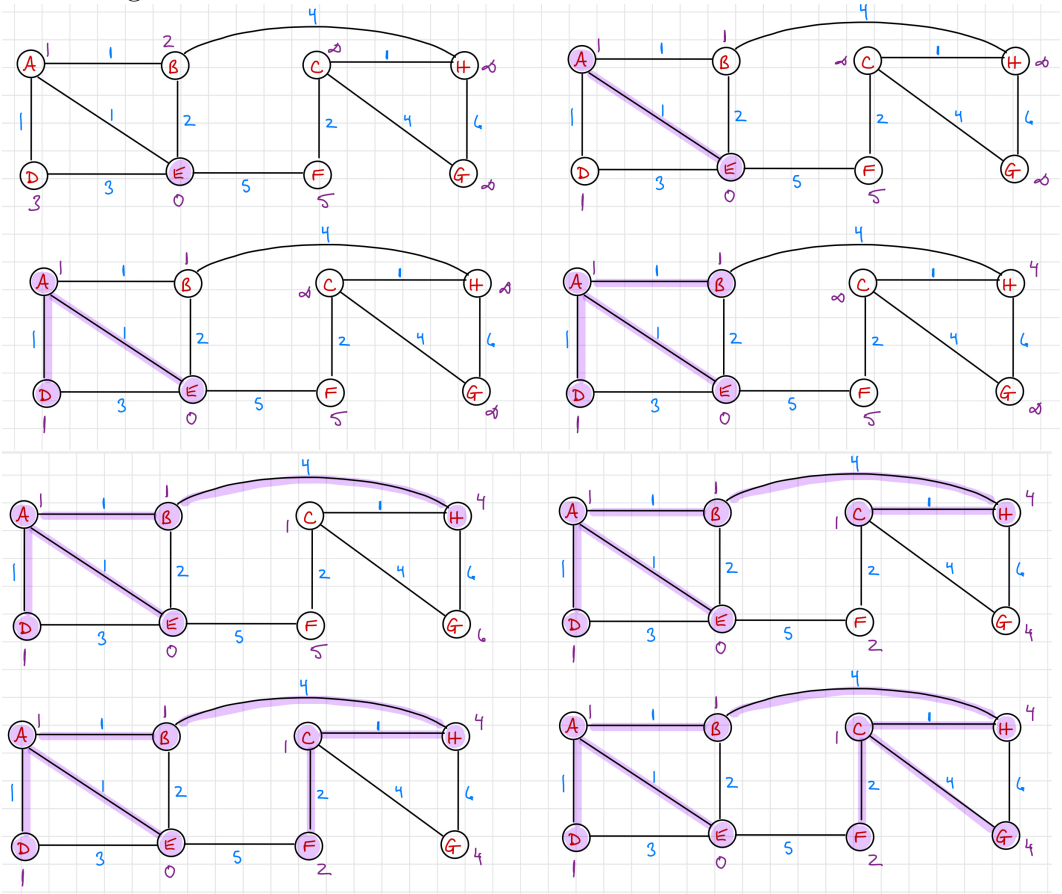
Practice Set 13: solutions

Problem 1

Dijkstra's algorithm:

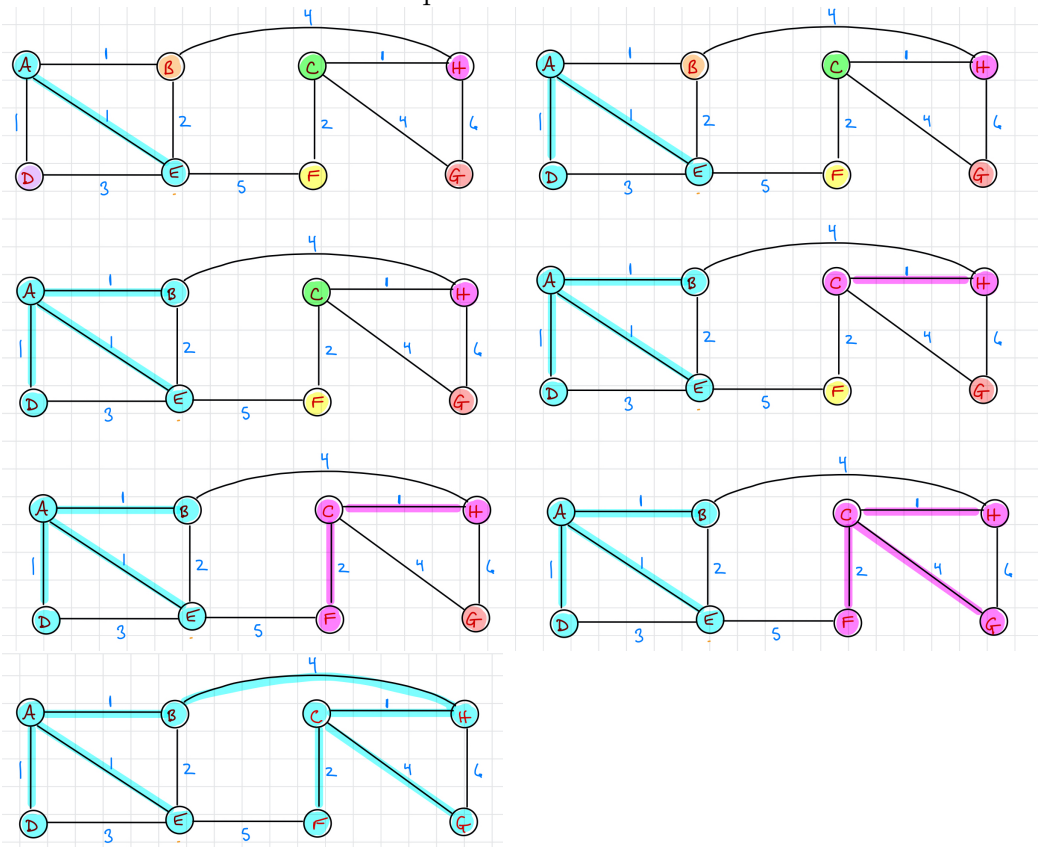


Prim's algorithm:

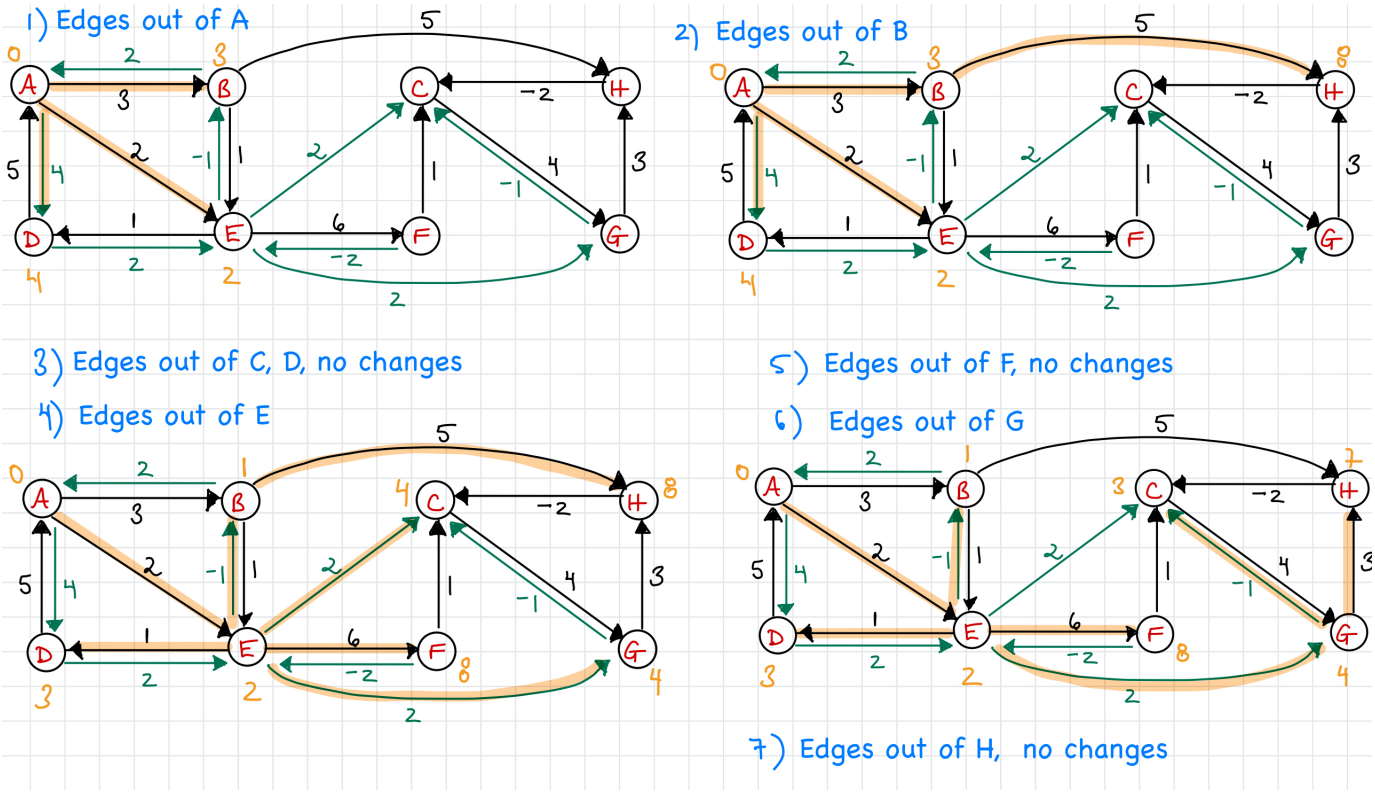


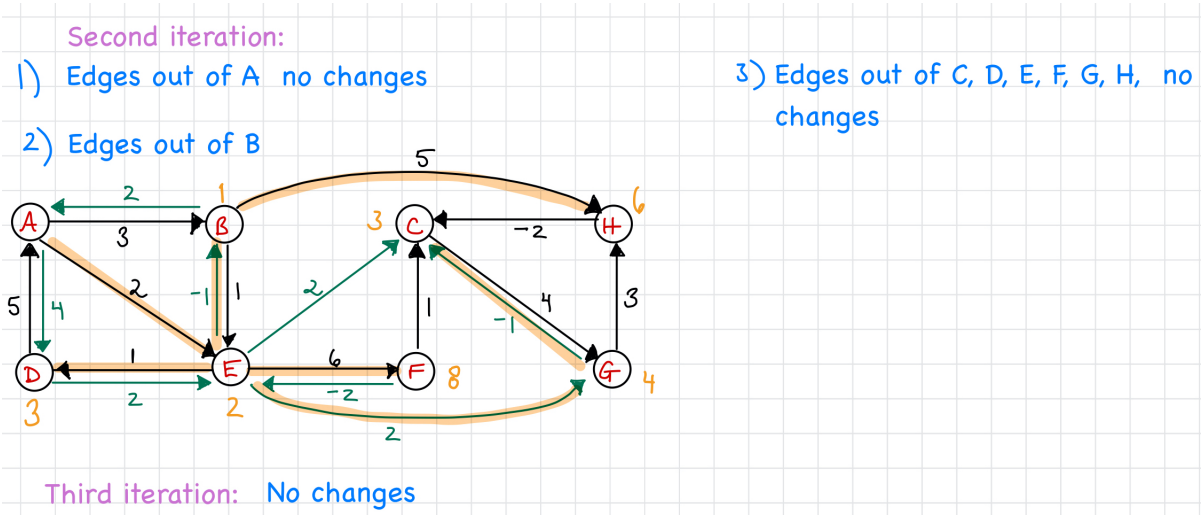
Problem 2

Kruskal is executed left to right, top to bottom. The components are colored differently. The algorithm terminates when there is one component.



Problem 3





After the third iteration, no more changes are made to Bellman-ford. Therefore when all $V-1$ iterations are complete, the above SSSP tree does not change. The final Vth iteration is performed to verify that no more changes are made, and then the algorithm terminates with a correct SSSP tree.

Problem 4

The second step of Bellman ford can be updated so that if an entire iteration is performed without any changes, than we can break out of the for loop and skip step 3. We know that at this point, the resulting tree represents the SSSP.

Step 2:

```
for k = 1 to V − 1
    change = False
    for each edge (u,v) in E:
        if v.d > u.d + w(u,v)
            v.d = u.d + w(u,v)
            change = True
            v.parent = u
     if change = False
        Return True.
```

By returning *True* as soon as we find that no changes have been made, the algorithm will not proceed to Step 3.

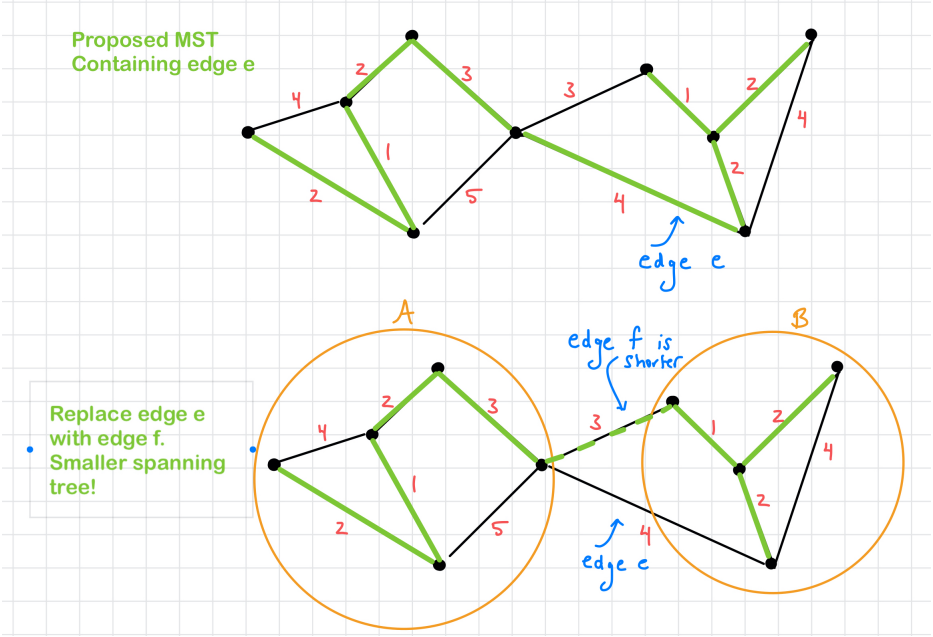
Problem 5

We can use counting sort to sort the edge weights. This is possible because the weights are *integers* and in the range $V \ldots 2V$. Recall that counting sort takes time $O(n+k)$ where n is the number of items to sort and k the the maximum value in the set of numbers. In this case, $n = |E|$, and the range is $|V|$. Therefore the runtime of counting sort is $O(E+V)$. As shown in class, the *Find* operations take time $O(E)$ in total, and the merge operations take time $O(V \log V)$ in total. Therefore the overall runtime of this new version of Kruskal is $O(E+V+V \log V) = O(E+V \log V)$ (as opposed to the original algorithm which takes $O(E \log V)$). This version is asymptotically faster if there are *few* edges.

Problem 6

Suppose it were possible... that there was an MST, T, containing the heaviest edge $e = (u, v)$ on the cycle C. Note that removing any edge in the MST disconnects the graph. Therefore if we remove the edge e, the MST is disconnected into two components. We can call one component A (suppose it contains u) and the other B (suppose it contains vertex v). In the class lectures we called this a *cut* of the vertices of G. Since e is part of a cycle, there is a path in the graph G from u to v that does not involve edge e. One of these edges must cross the cut, suppose it is edge f. Since e is the largest weight on the entire cycle,

then the edge weight of f is smaller than e. If we add edge f to the MST it *reconnects* the MST. This new tree has less total weight that the original tree. Therefore the original T could *not* have been an MST.



Problem 7

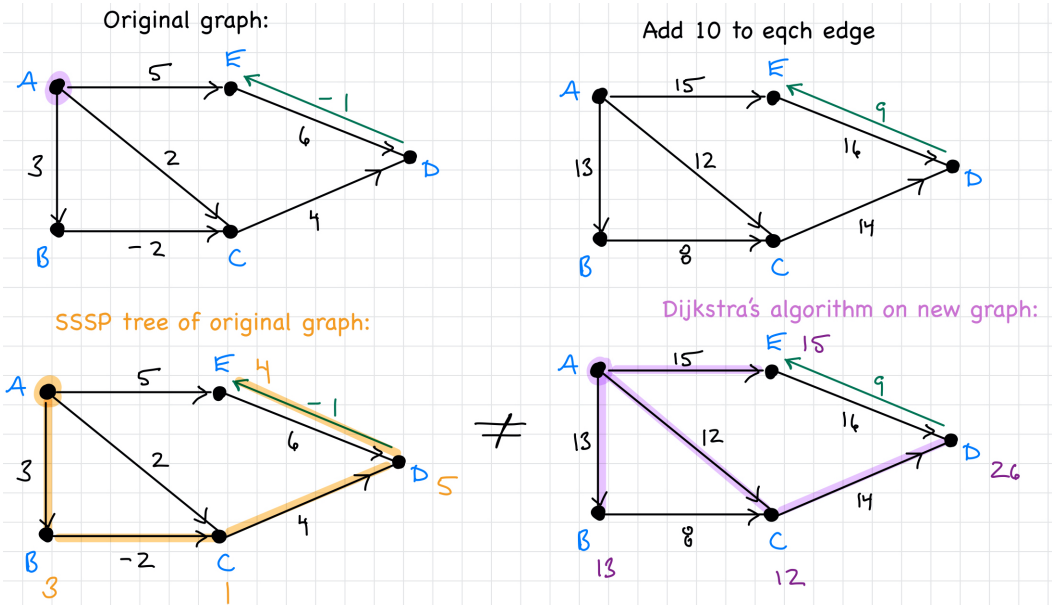
We can carry out Dijkstra's algorithm in the exact same way as the original algorithm, except that we compare *products* instead of sums. The source is initialized with $s.d = 1$. We update $v.d$ whenever $v.d > u.d \cdot w(u, v)$. If so, we set $v.d = u.d \cdot w(u, v)$. The while loop of Dijkstra's is updated as follows:

```
while Q!= NIL
  u = Extract-min(Q)
  for each v in Adj[u]
  if v.d > u.d*w(u,v)
    Decrease-key(Q, v, u.d*w(u,v) )
    v.parent = u
```

Note that this is possible because all weights are greater than 1, therefore the next greedy choice means that there is no other shorter path. This is equivalent to the usual version of Dijkstra that assumes there are no negative weights.

Problem 8

- Dijkstra's algorithm will find the tree that represents the shortest path from s to any vertex in the graph. If all edges have weight 1, then the length of any path is equivalent to the number of edges on the path from s to the vertex. Therefore in this case, Dijkstra finds the tree that is equivalent to the BFS tree. Recall that Dijkstra's algorithm runs in time $O(E \log V)$, whereas BFS runs in time $O(E + V)$. Therefore it would have been faster to use the BFS algorithm to compute this tree.
- By adding -10 to all edges, we have a graph with no negative edge weights. Therefore, we can apply Dijkstra's algorithm. However, the result is not necessarily the SSSP tree. Below is a counter example.



Problem 9

Model the problem as a directed weighted graph G, where each vertex represents a city, and a directed edge A to B represents a train that travels from city A to B with weight $t(A, B)$. Your home is in city S, which we will refer to as the *source* city. Since the problem is to determine the fastest way *home* from each city, we create a new graph G^T, which results from simply flipping the direction of each directed edge. In this new graph, G^T, we will use Dijkstra's algorithm to find the single-source shortest path from the source S. The result is a set of paths representing the fastest route *to* each city from your home town. In the original graph G, these routes correspond to the fastest way *home from* each city.

Runtime: Dijkstra's algorithm runs in time $O(E \log V)$. There are $|V| = n$ cities. There may be as many as $|E| = n(n-1)$ possible train routes, which is the maximum number of directed train routes between n cities. Therefore the overall runtime is $O(n^2 \log n)$.

Problem 10

Prim's algorithm can be updated by simply adding an attribute called $x.children$ for each node x in the graph. The attribute represents a list of children of node x in the MST. The children attributes are all initialised to empty. When a vertex is removed from the queue, it's parent is finalised. Therefore we add a node to the list of children of its parent, only when it is removed from the queue. The main while loop of Prim's algorithm can be updated as :

```
while Q ≠ NIL
    u = Extract-min(Q)
    if u.parent ≠ NIL
        u.parent.children.add(u)
    for each v in Adj[u]
        if v ∈ Q and v.d > weight(u,v)
            Decrease-key(Q,v, w(u,v))
            v.parent = u
```

We can then output the edges of the MST by carrying out a traversal on the edges of the MST. The initial call to the algorithm below is PrintMST(s):

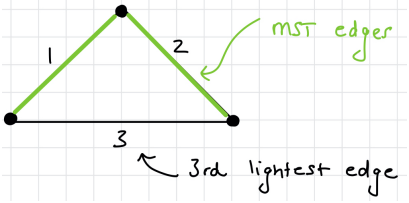
```
PrintMST(u)
    for each v in u.children
        Print edge (u, v)
        PrintMST(v)
```

Problem 11

Suppose the lightest-weight edge e connects vertex u to v. Since this is the lightest edge, then Kruskal's algorithm will select it first to be added to the MST. Therefore it is part of any MST. This is not necessarily true if there is a tie for the lightest edge.

The second-lightest edge must also be part of the MST. Kruskal's algorithm adds the lightest edge, and then the second-lightest edge, as long as it does not connect two vertices in the same component. But after only one edge exists in the MST, it is impossible that the second lightest edge connects two vertices in the same component. Therefore the second-lightest edge is added by Kruskal's algorithm to the MST.

The third-lightest edge is *not* necessarily part of the MST, as shown in the example below:

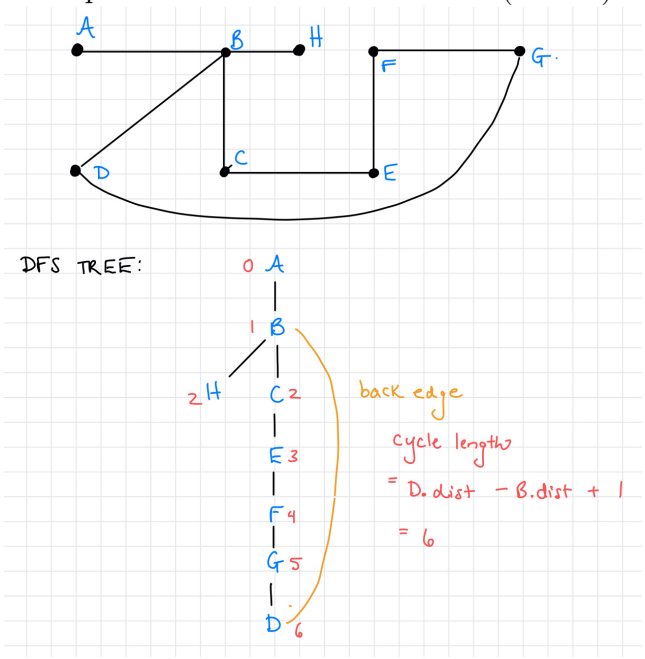


Problem 12

We can model this problem as a weighted directed graph (or undirected depending on the train routes), where each city is a vertex, and each train route between a pair of cities is a directed edge. The weight of an edge between city A and B is exactly the travel time of that train route, $t(A, B)$. Use Dijkstra's algorithm to compute the SSSP using your station as the source. Save this shortest distance in $v.d1$ for each vertex. Next, repeat Dijkstra's algorithm using your friend's station as the source, this time saving the shortest distance in $v.d2$. Each of these calls runs in time $O(E \log V) = O(n^2 \log n)$. Finally, loop through all vertices in the graph and compare $v.d1$ to $v.d2$. The smaller value corresponds to who would arrive first to each station. This loop runs in time $O(V) = O(n)$. The total runtime is $O(n^2 \log n)$.

Problem 13

Recall that in an undirected graph, as DFS executes and find a backedge, we know that a cycle exists. In this problem our job is simply to determine the length of that one cycle. Notice that the lenth of a cycle can be determine by the distance attribute that is set during DFS. The figure on the right illustrates the execution of DFS on an undirected graph with one cycle. The cycle length can be found using the distance attribute. Therefore we can simply updated the pseudo-code from practice set 12 that was used to detect a cycle, in such a way that once the cycle is detected, the length is printed out. Since DFS runs in time $O(V + E)$, this updated version which only includes an extra print statement also runs in $O(V + E)$.



DFS-visit(u):

```
u.visited = true
for each v in Adj[u]
    if v.visited = false
        v.parent = u
        v.distance = u.distance + 1
        DFS-visit(v)
    else if v != u.parent
        Print: Found cycle of length:
               u.distance - v.distance + 1
return false
```

Problem 14

assignment

Problem 15

Model this problem as two directed weighted graphs, G_1 and G_2. Both graphs use the same vertex set, where the vertices represent the n cities. In G_1, an edge (u, v) exists whenever there is **direct road** from city u to city v. The weight of that edge is the toll fee for that road. Similarly, an edge (u, v) exists in G_2 whenever there **direct train** from city u to v, and the weight of that edge is exactly the train ticket fee for that route.

Step 1: Run Dijkstra's algorithm on G_1 using the *London* as the source. Store the shortest distance to each vertex in the attribute $v.d1$. This represents the minimum cost from London to city v using **only roads**.

Step 2: Run Dijkstra's algorithm on G_2, using source *Rome*. Store the shortest distance in the attribute $v.d2$. This represents the minimum cost to get *home* from city v using **only trains**.

Step 3: Now we consider the possibility of **switching from your car to the train** over each possible city. For each vertex in G, set $v.total = v.d1 + v.d2$. This represents the cheapest way to go from Rome to London assuming we switched to the train at city v.

Step 4: Now we simply pick the best city to switch from the car to the train. Loop through all vertices and select the minimum value of $v.total$. This is your cheapest route!

Runtime: Steps 1,2 use Dijkstra's algorithm which runs in time $O(E \log V)$. Since $|E| \leq 2n + 3n$, the runtime is $O(n \log V)$. Steps 3 and 4 run in time $O(V) = O(n)$. The overall runtime is $O(n \log n)$.