CS2040S Final Assessment

MCQ

Q1: Given a BST (which is not balanced) of size **N**, to convert it to a balanced BST (according to AVL property) requires at least:

a. O(1) time
b. O(log N) time
c. O(N) time
d. O(Nlog N) time

Details:

Perform inorder traversal on the BST. This gives us a list of elements in ascending order. The inorder traversal will take O(n) time. Once done, we can construct an AVL tree in O(n) time by picking the middle element as the root of the AVL tree. Doing so will split the list into two sublists: one sublist containing all elements to the left of the root, and one sublist containing all elements to the right of the root. Recursively pick the middle element as the root for each of these subtrees. Doing so will take O(n) time.

Q2: You are given a binary max heap, which uses an array implementation (as covered in lecture). N is known, and is guaranteed to be odd. If we want to find the median value of the heap, we can do so in:

a. O(1) time
b. O(log N) time
c. O(N) time
d. O(N log N) time

Details:

Run quickselect on the underlying array. If preserving the original heap is a concern, we can copy out the contents of the array, and then perform quickselect on the copy instead.

Q3: Given an undirected graph, we want to find out if this graph is a tree. Initially, the only information you have is that this graph is an undirected graph. No additional information is provided.

Which of the following combinations of information will be sufficient to deduce, **for all possible graphs**, if the graph is a tree?

i. Number of vertices in the graph
ii. Number of edges in the graph
iii. In/out degree of each vertex
iv. All edges, given in the form of an adjacency list

a. (i) only
b. (i) and (ii)
c. (i), (ii) and (iii)
d. (i), (ii), (iii), and (iv)

Details:

(i) alone is not enough. For (ii), if the number of edges is not V-1, then it is definitely not a tree. However, if it is V-1, it could either be, or not be a tree, and we need more information. For (iii), if any vertex has degree 0 (except for the special case where V=1), then it is definitely not a tree. If all vertices have degree > 0, then there is still insufficient information to determine if the graph is a tree, as there could be multiple components. Only by getting all edges are we able to perform graph traversal to determine if it is a tree (E=V-1, and there is only one component).

Q4: You are given an initial list of integers, which you should store in a DS of your choice. Afterwards, your DS should be able to support the **removeMin()** and **removeMax()** operations, which removes and returns the smallest/largest element respectively from the DS. It is guaranteed that **no other operations will be performed on the DS**. In order to **minimise the total time complexity** of the removeMin() and removeMax() operations, which DS would be the most ideal?

a. Hash Table
b. Sorted doubly linked list
c. Minimum binary heap
d. AVL tree

Details:

A sorted array will be able to do removeMax() in O(1) time, but removeMin() in O(n) time. A sorted doubly linked list can do both in O(1) time. A minimum binary heap can do removeMin() in O(log n) time, but would need O(n) time to removeMax(). An AVL tree can do both in O(log n) time.

Q5: The following 5 strings are inserted into an initially empty Trie:

bear
bold
cold
cord
dear

How many nodes (including the root) will be present in the Trie after all strings are inserted?

a. 9
b. 10
c. 18
d. 21

Details:
After adding “bear”, there are 1+4 = 5 nodes (**bear**)
After adding “bold”, there are 5+3 = 8 nodes (b**old**)
After adding “cold”, there are 8+4 = 12 nodes (**cold**)
After adding “cord”, there are 12+2 = 14 nodes (cor**d**)
After adding “dear”, there are 14+4 = 18 nodes (**dear**)

Q6: You are asked to draw a directed graph with 5 vertices and 7 directed edges. Additionally, the graph must contain the largest possible number of valid topological sorts. How many valid topological sorts are present in this graph?

a. 1
b. 6
c. 24
d. 120

Details:

The following 7 edges are present in the graph:

(0 -> 1), (0 -> 2), (0 -> 3), (0 -> 4), (1 -> 2), (1 -> 3), (1 -> 4)

This forces vertices 0 and 1 to be the first 2 vertices in the toposort, in that order. 2, 3, and 4 can appear in any order afterwards, for a total of 6 possibilities.

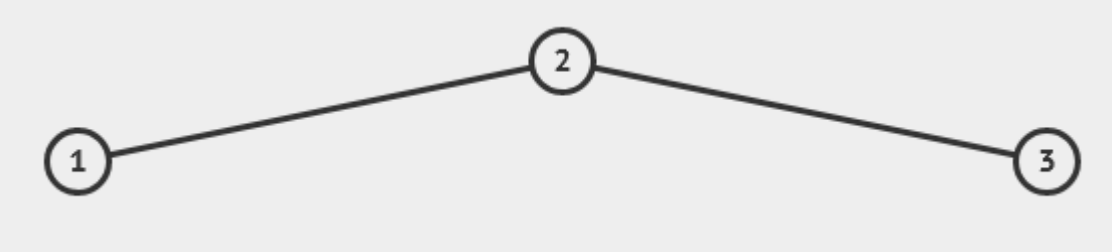
Q7: 8 elements are added one at a time into an initially empty AVL tree. What is the maximum number of rebalancing operations that occurred across all insertions?

a. 3
b. 4
c. 5
d. 6

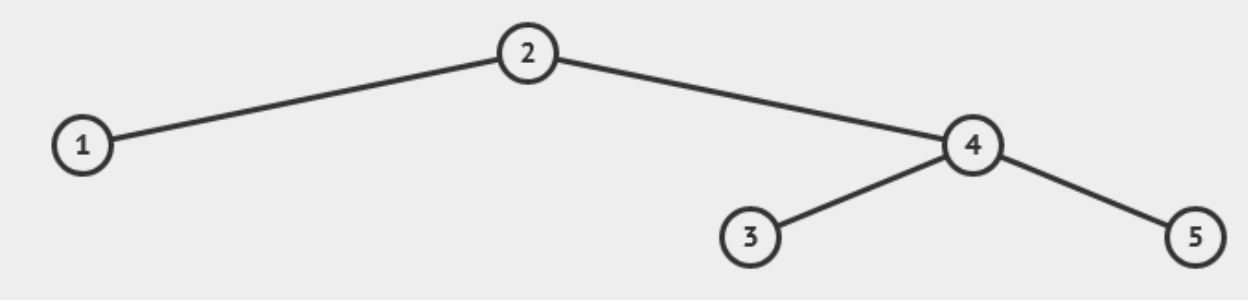
Details:

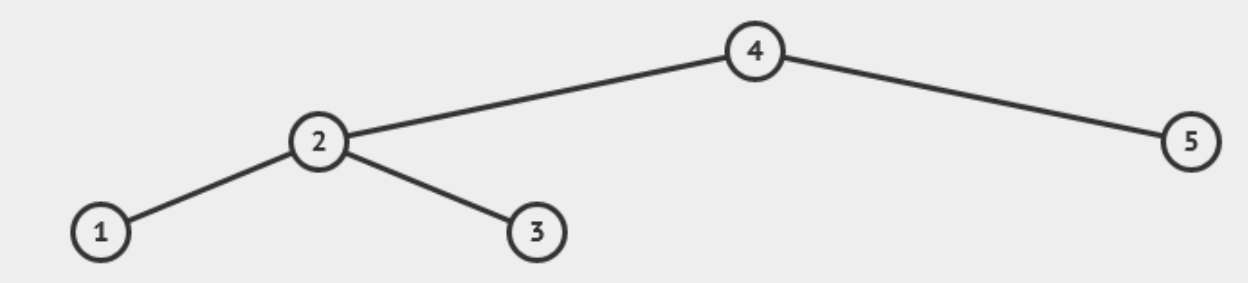
6 is naturally the upper bound for rebalancing operations (a rebalancing operation can only occur when adding to an AVL tree with at least 2 elements, and one insertion will only result in at max 1 rebalancing operation).

There is only one valid AVL tree with 3 elements, so up to 3 elements, the number of rebalancings is still predictable (at most 1 rebalancing after adding the 3rd element, and the resulting tree will be a root with 2 children).

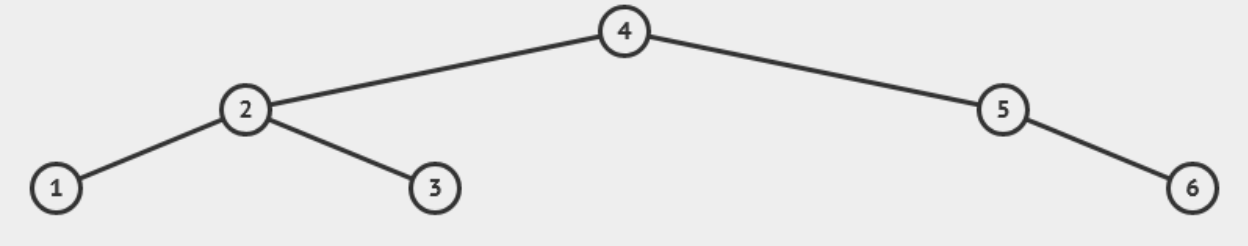


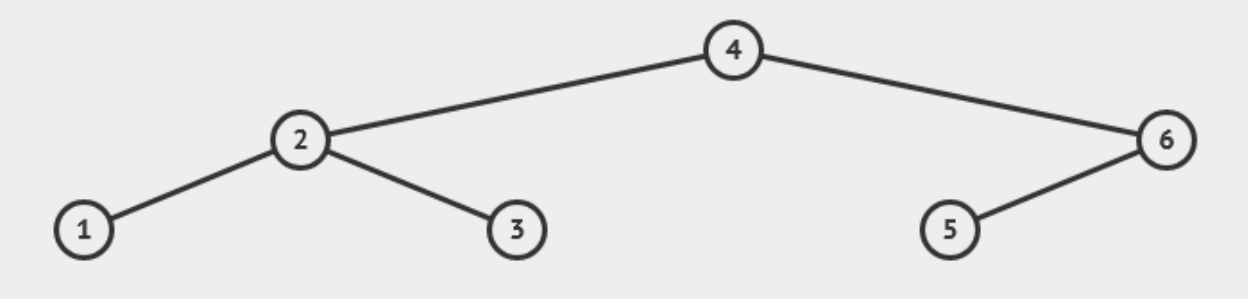
As all nodes are currently perfectly balanced (as all things should be?), we need to add an additional 2 elements to one side of the tree to trigger another rebalancing operation. This results in 5 elements total, and 2 rebalancing operations. The resulting tree will be one of the two below; while other valid AVL trees with 5 elements exist, they cannot be reached by performing two rebalancing operations.



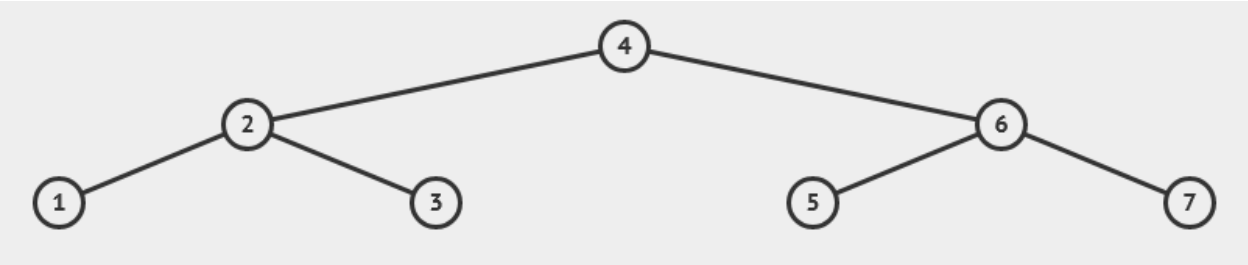


Adding one elements below either of the two leaf nodes at the lowest level would then trigger another rebalancing operation. Taking into account the first picture (we ignore the second, since this is merely a mirror image of the first one, so the same argument applies), 2 possible trees may result.





At this point, simply add an element below the single leaf to trigger another rebalancing operation. Both of the trees above will result in the same tree.



With a perfect binary tree, it is impossible to trigger any rotations via one insertion. The total number of rebalancing operations is 4.

We can ignore the other valid AVL trees of size 5 that can only be formed with less than 2 rebalancing operations. Since one insertion can result in at most one rebalancing operation, it is impossible for them to give an answer > 4 rebalancing operations with 8 elements.

Q8: Given a directed graph with V vertices, what is the maximum number of directed edges that can be present in the graph without forming any cycles?

a. V(V-1)
b. V(V-1)/2
c. V(V+1)
d. V(V+1)/2

Text

Details:

If a directed graph has no cycles, it is a DAG. A DAG has a valid toposort. In a valid toposort, there are no edges from right to left (ie. a vertex (X) to the right of another vertex (Y) in the toposort would mean that an edge from X to Y definitely does not exist). Owing to this, the maximum number of edges is

$(V-1) + (V-2) + (V-3) + \ldots + 1 = V(V-1)/2$

v connects to v -1 edges
v - 1 connects to v - 2 edges etc

Q9: You want to find out if an **undirected, disconnected graph** is reverse-bipartite (not an actual term). For a graph to be reverse-bipartite, vertices can be assigned to one of two sets, and edges that appear in the graph can only be between two vertices in the same set (so these vertices should not be in two different sets). Your program **only needs to return true/false**. This can be done in:

a. **O(1) time**
b. O(V) time
c. O(V + E) time
d. $O(V^2)$ time

Details:

Always true (just put all the vertices into one set, and leave the other set empty)

Q10: You are given an undirected weighted graph G, and the only MST of this graph T. Then, one edge in G (which is also in T) is removed from the graph. You are now required to find an updated MST of G (G is guaranteed to still be connected after the edge is removed). This can be done in :

a. O(V) time
b. O(VlogV) time
c. **O(V + E) time**
d. O(ElogV) time

Details:

run counting CC to label the vertices in the MST (which is now 2 components since one edge is removed) based on which component they belong to. O(V+E) time if using adjacency list.

Now go through all edges in the original graph. find the smallest edge that links 2 vertices that belong to 2 different components and is not the removed edge. O(V+E) time if using adjacency list.

Total time = O(V+E)

Analysis:

1. Given N distinct string keys (N > 1), each of the same length w (w > 2), that is formed from an alphabet set of size R (R > 1), storing all N keys in a Trie cannot take less then O(wNR) space.

False.

If R > N, then it is totally possible for all the N string keys to only differ by their last character. In this case, total space required is O(NR+wR) = O((N+w)R) instead, since the common prefix of length O(w) is stored only once (total space required for this is O(wR)) and only the last character of each of the N strings is stored in their own node (total space required for this is O(NR)). This is better than O(wNR).

2. There is no non-empty binary max heap containing integer keys which also satisfies BST property.

False.

A max heap containing only 1 integer key or 2 distinct integer keys will also satisfy BST property. There are many other counter-examples.

3. In a SCC (strongly connected component), removing any one edge in the SCC will destroy the SCC (meaning not all vertices will be able to visit all other vertices in the SCC after that edge is removed).

False.

For example, a SCC having 4 vertices 0,1,2,3 with directed edges (0,1),(1,3),(3,0),(1,2),(2,3). In this SCC there are 2 directed cycles 0->1->2->3->0 and 0->1->3->0. Removing (1,3) will remove the smaller cycle but it will not destroy the SCC since there is still the larger cycle 0->1->2->3->0.

4. Floyd Warshall is the most efficient algorithm for solving APSP for all kinds of graphs.

False.

For a tree, simply run DFS or BFS from each vertex as source to compute the APSP. This will take O(V^2) time, while using Floyd Warshall will take O(V^3) time.

Structured Questions

1. Given an AVL storing N distinct integer keys, give an algorithm to delete all keys that fall within the range from i to j (i <= j, i and j inclusive) in worst case time <= O(KlogN), where K is the number of keys to be deleted. You cannot use Java library TreeSet or TreeMap in your algorithm. You can only use the AVL operations discussed in lecture. If you need to modify any of the operations, describe your modification.

Answer:

1. Let node = search(i,bBST.root) <- O(logN) time

2. If node exist goto 3 else insert i into the bBST <- O(logN) time

3. Let snode = successor(i,bBST.root) <- O(logN) time

    i. delete(i, bBST.root) <- O(logN) time

    ii. if snode exists goto iii else stop

    iii. if snode.val > j stop

        else i = snode.val, repeat 3.

step 1 and 2 will take O(logN) in total. Step 3 will take O(logN) for each node deleted since there are at most K deleted nodes it will take O(KlogN) time in total. Total time taken = O(logN)+O(KlogN) = O(KlogN).

2. Given an unweighted graph (with at least 1 edge) stored in an adjacency list, give an algorithm that will return 1 if it is an undirected graph, 2 if it is a directed graph and 3 if it is a hybrid graph (a mix of directed and undirected edges). Your algorithm must run in average time <= O(V+E) where V is number of vertices and E is the number of edges in the graph.
*Note: bi-directed edges will not be present in the graph given.

Answer:

```
1. Let H be a hashset
2. For each vertex u in the adjacency list // average O(V+E) time
        For each neighbour v of u
                insert (u,v) into H
3. Let undirected1 = true, undirected2 = false
4. For each vertex u in the adjacency list // average O(V+E) time
        For each neighbour v of u
                if H.contains((u,v)) && H.contains((v,u))
                        undirected2 = true
                else
                        undirected1 = false
5. if (undirected1 == true) return 1
   else if (undirected1 == false && undirected2 == false) return 2
   else return 3
```

Time complexity = time complexity of step 2 + step 4 = average O(V+E) time.

3. There is a game where you have N dots numbered from 1 to N, and for some pairs of dots A,B there is an arrow going from A to B (if there is an arrow going from A to B then there will not be an arrow going from B to A).
The objective of the game is to place the pencil at some starting dot and trace the arrows in such a way as to go through as many dots as possible without lifting the pencil (you may traverse some arrows multiple times to achieve this).
Given there is at least 1 starting dot A' which allows to you to trace the arrows in the above stated way and go through all the other dots, model the game as a graph and give the most efficient algorithm you can think of in terms of worst case time complexity to return one such A'.
Answer:

Model the game as a directed graph where the dots are vertices and there is a directed edge from vertex i to vertex j if there is an arrow going from dot i to dot j.

Now simply run DFS topological sort algo. Without reversing the toposort array, just return the last vertex in the array as an A'. Time complexity is O(V+E) time where V is number of vertices and E is the number of edges in the graph.

This algorithm works because
1.) It is stated there is at least 1 vertex A' which allows you to go visit all the rest of the vertices without lifting the pencil. This also means that all the other vertices are reachable from this vertex.

2.) This means in post-order processing of the vertices using DFS, all vertices reachable from A' will inserted before it in the toposort array. Since all vertices are reachable from it, the last vertex in the toposort array must be such an A'.

3.) You may say the last vertex in the toposort array could be one which can reach all vertices but you must lift the pencil. This cannot be the case. If it can reach all vertices it clearly has a path to some A' (you can trace that path without lifting the pencil) then from that A' it can reach all other vertices without lifting the pencil. Thus it must be an A' itself.

O(V(V+E)) time algo: Run counting component algorithm, once on each vertex as source. If for any particular vertex as source the number of component is 1 then that vertex is an A'.

4. Routers in a network are often susceptible to failure. Dr Ferdaus of Ferulock fame/infamy has developed his own protocol to test if a router in a network is still functioning.

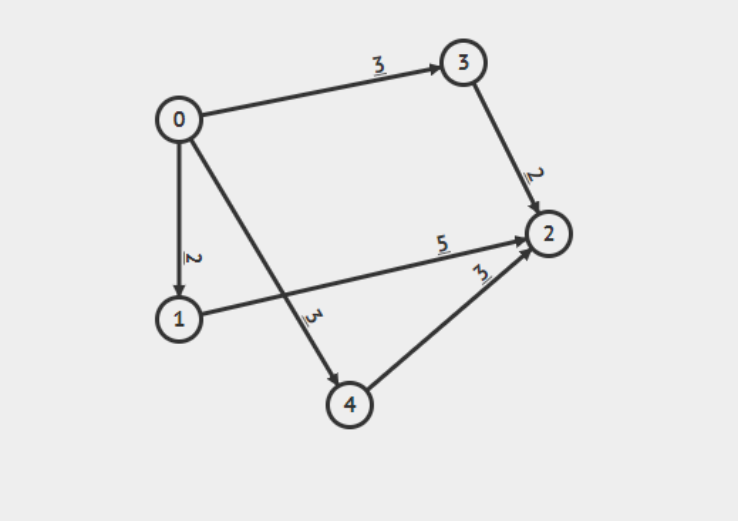
Given a graph representing the network, vertices are the routers (there are V number of them numbered from 0 to V-1) and the routers are connected by E directed edges ($\frac{V(V-1)}{10} \leq E \leq \frac{V(V-1)}{2}$). For any pair of router A and B connected by a directed edge from A to B with integer weight W(A,B) (W(A,B) >= 1), A will send a ping request to B at intervals of W(A,B) seconds (you can assume the time to send the request to B is 0 seconds). If B is alive it will have to send a default message back to A. All the routers in the network is synchronized to a global clock, so they will start counting of their intervals to send pings at the same time.

Now given that the graph as described above is stored in an adjacency list, give the best algorithm in terms of worst case time complexity you can think of to answer the following query (there can be many of such queries):

RequestList(i,K) - Return the 1st K vertices that sends a ping request to vertex i ordered by increasing time of request. Number of vertices sending pings to i <= K <= V. If two different vertices sends a request at the same time, they should be ordered by increasing vertex number.

If required, you can perform pre-processing that takes no more than O(V+E) time. Describe the algorithm for your pre-processing too.

In the example graph given below that represents a network of routers, RequestList(2,4) will return {3,4,3,1} as the first 4 vertices sending pings to vertex 2 in order of increasing time of request. The time of their requests are {2 secs, 3 secs, 4 secs, 5 secs} respectively.



Answer:

The original graph is not good for answering the query since for any vertex i we need to know all vertices who have i as a neighbor and to do so we need to scan the entire adjacency list. To alleviate this problem perform a pre-processing step where all the edges are reversed.

Pre-processing step:
1. Let AL be the given adjacency list. Create another adjaceny list AL' of same size as AL
2. For each vertex u in AL // O(V+E) time
    For each neighbour v of u
        add u to neighbour list of v in AL' with weight W(v,u)
Time taken is O(V+E) to convert AL to AL'

RequestList(i,K):
1. Create a PQ P and array T
2. For each neighbour v of i in AL' // O(K'logK') time where K' is number of neighbors of i
                                    // or can do fast heap create in O(K') time
    insert triplet (W(i,v),j,W(i,v)) into P where they are ordered by the first field then the second field
3. For j = 1 to K // O(KlogK) time
    t = P.dequeue()
    add t.second to back of T
    insert triplet (t.first+t.third,t.second,t.third) into P
4. return T

Total time complexity = O(K'logK') + O(KlogK) = O(KlogK) since K >= K' as given in the problem description.

5. Given a graph G that is a DAG with V vertices and E edges (where E = O(V)) stored in an adjacency list, you want to find the weight of the largest edge along the minimax path from a given source vertex A to a given destination vertex B. Give the best algorithm you can think of in terms of worst case time complexity to do it. You may assume there is at least 1 path from A to B.

Answer:
1. Perform toposort of the DAG using Khan's or modified DFS // O(V+E) time
2. Let D be the distance array init to +inf. D[A] = 0 // here D will store the largest edge weight along the minimax path
3. for each u in topological order //perform 1 pass bellman ford with modified relaxation condition
    for each neighbor v of u
        D[v] = min(D[v], max(D[u],w(u,v)) // basically relaxation condition of Floyd Warshall minimax variant
4. return D[B]

Time taken is time complexity of 1-pass bellman ford = O(V+E) = O(V) since E = O(V)

O(VlogV) time algo: Run Prim's starting from A and stop when it hits B. return the largest edge found so far.

6. The salesman from tutorial 11 has begun another round of travelling around different cities and peddling his wares. This time he has set aside funds to pay the toll fee for every city he passes when getting from some source city A to some destination city B. Of course he still wants the shortest route to get from A to B as time is of the essence. However he has calculated that he has only enough money to pay the toll fee of at most K cities where 1<= K <= 10, thus he cannot pass through more than K cities when getting from A to B (including A and B themselves).
Given the value K and a graph G with V vertices and E edges, where the vertices are cities and bi-directional edges are roads connecting pairs of cities, and edge weight is the travel time (same in both direction), give the best algorithm you can think of in terms of worst case time complexity to find the cost of the shortest path the salesman should use to get from some city A to some other city B that does not involve more than K cities. If no such path exists output "no valid path from A to B".
Answer:
Transform the graph as follows:
For each vertex v in the original graph, create a vertex representing the pair (v,i) for i from 1 to K. Thus v is represented K times in the transformed graph (v,1), (v,2) ... (v,K).
For each vertex (v,i) in the transformed graph there is a directed edge from (v,i) to (u,i+1) if i < K and (v,u) is an edge in the original graph.

Now run modified Dijkstra from vertex (A,1) as source on the transformed graph.
Scan through SP cost of (B,1) to (B,K) and return the minimum among them. If minimum is still +inf then return "No valid path from A to B". Total time complexity is O(ElogE) since blow up in transformed graph is a constant (at most 10x the number of vertices and edges in the original graph).

The above graph transformation is equivalent to modifying the distance array to be a matrix of size D[V][K], where D[v][k'] stores the SP from A to v that uses exactly k' vertices in the path. For the PQ instead of just a pair information make it a triplet (d,v,k) where v is the vertex, d is the current SP estimate, and k is the number of vertices in the current SP from source to v. We can then modify modified Dijkstra as follows to run on the original graph (assume stored in adj list)

```
initSSSP(A) <-- init dist[V][K] to +inf for all except dist[A][1]
                      which is init to 0
PQ.enqueue((0, A, 1))
while PQ is not empty
  (d,u,k) <- PQ.dequeue()
  if d == D[u][k] && k < K // make sure d is up to date and k < K
                           // meaning not yet maxed out # vertices
                                 // along SP
    for each vertex v adjacent to u
      if D[v][k+1] > d + w(u, v) //can relax and now we have new
                                       //SP from A to v using k+1 vertices
        D[v][k+1] = d + w(u, v)
        PQ.enqueue((D[v][k+1], v, k+1))

Now scan dist[B][1] to dist[B][K] to find the smallest among them and return that. If all are
still +inf means there is no valid path involving <= K vertices from A to B.
```

7. The global merchant guild has set her sights on being the most represented guild on the planet. In order to do so, the best way is of course to build an office in every country on the planet. However, the cost is too prohibitive and also some countries are at war with neighbouring countries and it would not be profitable for the guild to do business in such countries. After analyzing all the N countries (the countries are numbered from 0 to N-1) and their relationship to each other in terms of distance, politics etc.., the guild has come up the following possible relationship between a pair of country A,B.

A B 1 - Exactly one office will be built in either A or B because they are close to each other and you do not need one office in both A and B.

A B 0 - No office should be built on both A and B because they are at war and having a presence in either country would not be profitable at this time

A B 2 - An office should be built on both A and B because they are far apart but have very strong economic ties and to maximize profit it would be expedient to build an office in both countries.

Given M such unique pairs of counties and their relationships (there is only 1 relationship per country pair, e.g if there is 0 2 1, there will not be 0 2 0 or 0 2 2), determine the minimum number of offices to build to satisfy the M relationships so as to achieve maximum profitability and also which countries they should be built in. If it is impossible to satisfy the M pairs of relationships, simply return -1. Give the best algorithm in terms of worst case time complexity you can think of to solve the problem.

Answer:

1. Create a 2D array R of size Nx3 initialized to 0.
2. Create an adjacency list AL of size N.
3. Create an array Color of size N initialized to 0. // C can store -1/1 to indicate not build/build
4. Create an arraylist E which store integer pairs (x,y) // x = vertex num, y = color of vertex
5. For each (A,B,x) relationship in the M relationships
    R[A][x] = 1
    R[B][x] = 1
    if x == 1
      add B as neighbour of A in AL and A as neighbour of B in AL // only 1 edges are in AL
6. For i = 0 to N-1
    if (R[i][0] == 1 && R[i][2] == 1) // you cannot both build and not build an office in country i
        return -1
    if (R[i][2] == 1) // you have a 2 relationship for this country so have to build here
        Color[i] = 1
    if (R[i][0] == 1) // you have a 0 relationship for this country so cannot build in this country
        Color[i] = -1
7. For i = 0 to N-1 // perform coloring of the vertices in AL (which may be composed of multiple
        // components) starting from colored vertices so as to check if graph given is
        // bipartite. Color = 1 mean build at vertex, Color = -1 means don't build at vertex
    if (Color[i] == 1 or -1)
        set1 = set2 = 0
        state = DFSBiPart(i,Color[i])
        clear E
        if state == -1 return -1

8. For i = 0 to N-1 // perform a second pass for components which only consists of vertices v where
// Color[v] == 0. i.e, all the edges involved among these vertices are only 1 edges
if (Color[i] == 0)
set1 = set2 = 0
state = DFSBiPart(i,1)
if state == -1 return -1
For each (v,c) in E
if (set1 > set2) // re-color vertices so as to minimize offices built.
Color[v] = c*-1
else
Color[v] = c
clear E

9. Go through Color and return all indices i where Color[i] == 1, add up the number of such indices and return the value.

DFSBiPart(u,c): // modification of DFS to perform bipartite coloring of the graph
Color[u] = c
E.insert((u,c))
if (c < 0) set2++
else set1++
for all neighbor v of u in AL
if Color[v] == c
return -1 // cannot both be c, this means either don't build in both u,v or
// build in both u,v but this cannot be since the relationship
// between u,v is 1 meaning you have to build on only 1 of them.
else if Color[v] == 0 // color not yet colored vertices
return DFSBiPart(v,c*-1)
return 0

Total time is bounded by Step 7 and Step 8.
For step 7 time taken is O(N+M) since it will explore all components that 1 vertices or -1 vertices belong to, and each edge in these components are explored 2 times (once from each end of the edge).
For step 8 time taken is also O(N+M) since it will explore all component that the remaining 0 vertices belong to (re-coloring vertices in a component will not take time more than exploring the component).

So total time complexity is O(N+M).