CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #4

Name(s): Raashi Maheshwari, Tanvi Yamarthy

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size two or three students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. A **walk** in a directed graph $G = (V, E)$ from a vertex s to a vertex t, is a sequence of vertices $v_1, v_2, \ldots, v_k$ where $v_1 = s$ and $v_k = t$ such that for any $i < k$, (v_i, v_{i+1}) is an edge in G. The length of a walk is defined as the number of vertices inside it minus one, i.e., the number of edges (so the walk $v_1, v_2, \ldots, v_k$ has length $k - 1$).

Note that the only difference of a walk with a **path** we defined in the course is that a walk can contain the same vertex (or edge) more than once, while a path consists of only distinct vertices and edges.

Design and analyze an $O(n + m)$ time algorithm that given a directed graph $G = (V, E)$ and two vertices s and t, outputs Yes if there is a walk from s to t in G whose length is even, and No otherwise. **(25 points)**

Solution. The question asks us to determine if there is a walk from vertex s to vertex t such that their length is even. In class, we learned that in terms of graph theory, a walk can be defined a sequence of edges + vertices that have one starting vertex, and one ending vertex that serves as the endpoint. The logic behind this question is that we can use breadth first traversal (BFS) in order to visit the different vertices in the graph, mark them as visited, and tracks the distance between a current vertex and the origin vertex (or the source), vertex s. It is important to remember that we are given the graph itself, and vertices s and t as the input for this function. Using BFS also has two main advantages, the first is that it will help us determine if a vertex has an an even/odd distance from s, as it traverses, and secondly, in class we learned that Breadth First Search is an algorithm that follows the "shortest path principle", meaning that the vertices that are visited will be done so in increasing distance, helping us identify even length walks from odd length ones. Now, let us understand the psuedo code and design for this algorithm below:

```
function hasEvenWalk(graph G, vertex S, vertex T):

   lengthG=G.size() //this will give us the total number of vertices in the graph
   boolean[][] Visited=new boolaean [n][2]
   //makes a 2D array with vertices
   n columns and 2 rows where the
   first column will have the element
   the second one will way say if its even/odd using 0 or 1.


   queue NewQ =new Linked List()
   //this line is intaitlizing our "BFS" by using a queue.

   NewQ.add(nw int []{s,0};
   //add s, 0 to the queue
   showing the source vertex
   and 0 to indicate its "even distance from s" (0).

   visited[s,0]=true
   //mark visited as true

   while(queue is not empty):
        Poll array[] =NewQ.poll()
        vertex=array[0]
        evenOdd=veretx[1]
        //poll function gives us the elemetn and the front of the queue
        while removing the element
        from the quue, similar to peek.
        We are storing the vertex and
        if the distance is even or odd.

        for (all neighbors of the vertex):
            int newParity=1-evenOdd
            //if the current point has
            an even distance ,
            the neighbor right next
            to it will have an odd distance and so forth.


            if (we didn't visit [enightbor][evenOdd]}:
                   visited [neighbor][evenOdd}] == true
                   //mark neighbor as visited
```

```
                    NewQ.add(visited[neighbor][evenOdd]
                    //add to the queue



         return visiteed[T][0]
         //see if T is visited and if T is an even distance. if it is visited it
         will be in the
         visited array and thus
         should return true. if not
         it means it hasn't been visited yet.
```

Now Let us look at the ***proof of correctness*** and run time analysis: If we are using induction, we can start by stating that our base case is 0, meaning that the length of the walk itself will be 0. This means that in our visited array we se this equal to true and since 0 represents even, our final statement will return true. In our inductive hypothesis, we want to assume that this algorithm works for all works of length k, and that it is true for k+1. The reason the algorothm handles that is because for every vertex it iterates through its neighbor as well filling out the parity that needs to go in the visited array. When it sees that 1 os a neighbor of 0, it will flip from 0 t0 1 and become an odd number that gets marked into the visited array. The runtime of this algorithm is indeed in O(n+m) because the graph is represented in the form of an adjanecy list where we visit every vertex and find all of its corresponding "edges" via neighbors. This gives us an O(n+m) time complexity where n represents vertices and m is representing the edges.

Problem 2. We have an undirected graph $G = (V, E)$ and two vertices s and t. Additionally, we are given a list of new edges F which do not belong to the graph G. For any edge $f \in F$, we define the graph G_f as the graph obtained by adding the edge f to the original graph G.

Design an algorithm that in time $O(n + m + |F|)$ finds the edge $f \in F$ such that the distance from s to t in G_f is minimized among all choices of an edge from F. **(25 points)**

Solution. The question essentially asks us to evaluate all the edges in the graph and design an algorithm in which an edge is returned such that it has the shortest path distance from s to T. The edge that is returned is an edge from F that we are trying to find. We are trying to see which edge, if added will still return a shortest minimal path from S to T (vertices). In the context of our question, F is an input that will be taken in as a list of edges where each edge is a new edge that could be potentially added to G. This graph is simple, undirected, and unweighted. Now, let us go through the algorithmic approach for this question. We start by running BFS from vertex S. This ensures that we can find the shortest distance from S to all the other vertices in the graph. Then, we can run bfs on T, to find the shortest distance from vertex t to all the other vertices. Then, for each new edge in the list of tuples, F, we want to find the shortest path from S to T if that new edge is added. As we iterate through the different edges in F, we want to keep track of the one that produces the minimal path (the best edge), and then return that edge at the end of the algorithm. Let us now walk through the psuedo code for this. It is important to note that BFS is a black box algorithm that we discussed in class, and thus in the psuedo code, we will just write *use bfs* to indicate the algorithm being done.

```
function findNewBestEdge(Graph G, S, T, F):

dOnS=bfs(G, S)
//Run BFS on vertex S, donS[V],
gives the shortest path distance
from vertex S to every other vertex V.

dOnT=bfs(G,T)
//Run BFS on vertex T, donT[V],
gives the shortest path
distance from vertex T to
every other vertex V.

bestPath=infinity
bestEdge= Null;

OgDistance=donS[T]
//get the original minimal path distance from vertex S and T

for each new edge (u,v) in F:
    newDis=min(dOnS[u]+1+donT[v],
    dOnS[v]+1+donT[u] )

    //find the path distance from u to v and v to u

    if newDis<bestPath:
        bestPath=newDis
        bestEdge=u,v
        //set the best path to the minimal path lenth we found with new edge


return bestEdge
```

Now that we have gone over the algorithm itself, let us go over the proof of correctness and the run time. Since we are using a conventionally proven black box algorithm that shortest path distnaces in a graph (BFS), we can be assured that our algorithm is correct. We call BFS twice and then we iterate thorugh all the edges in F (u,v), which also give us new path distances. Then we want to return the edge that mimixs the shortest path distance from vertex s to vertex t.In our base case, the length of F would be 0, in which it would have nothing to traverse over, and no new edge would be added to our group. In our inductive step, we have to show our algorithm holds true for our k+1 edges. If we have our first k edges e1, e2, ek, then when adding our ek+1 edge, we would compare its minimal path distance with ek, and update the best edge and best path accordingly. The runtime of a typical BFS algorith is O(n+m), and we call it both on S, and T. But in our algorithm, we are also iterating through the edges in F, which gets us an O(F) run time. Thus, the total time complexity would be O(n+m+F), which is the desired complexity that we are looking for.

Problem 3. Let $G = (V, E)$ be a connected, undirected graph. We start with coins placed on arbitrarily chosen vertices of G, and each coin can move to an adjacent vertex at every step. The objective is to determine if it is possible to bring all coins to the same vertex, and if so, compute the minimum number of moves required.

(a) Suppose there are two coins placed on two given vertices $u, v \in V$, which may or may not be distinct. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is possible. The input to your algorithm consists of the graph $G = (V, E)$ and the initial vertices u and v.

(b) Now suppose there are three coins placed on three given vertices of G. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where all three coins are on the same vertex, or to report correctly that no such configuration is possible.

(c) Finally, suppose there are forty-two coins placed on forty-two given vertices of G. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. For full credit, your algorithm should run in $O(V + E)$ time.

(40 points)

Solution. Let us begin by solving part A of this question. Part A asks us to devise an algorithm for the "minimum # of steps to reach a configuration where both coins are on the same vertex". It also gives our inputs for the function which are the graph G, and the initial vertices u and v.

When hearing the words "minimum # of steps", that immediately strikes to us that we can use the black-box algorithm that we learned in class BFS, or breadth first search. We can use BFS on vertex u to find the shortest path distance from u to each of the vertices in our given set of vertices. Then, we can do the same thing for the vertex v where we perform BFS on it. Then, after keeping track of the path distances for vertex point u, and v, we want to find a commonality between them to see if both the coins will reach another vertex in the set of vertices at the same time. Once we find that minimal distance and that vertex which gets us this minimal distance, we can output that.

Now, let us look at the basic pseudocode for this algorithm below:

```
def minCoins(Graph G, vertex u, vertex v):
        DofU= bfs(G, u) //finding the shortest path from starting u
        DofV=bfs(G, v)
        min=infinity
        for all vertices, vertex in G:
            if DofU[vertex]=DofV[vertex]:
                min=min(min,DofU[vertex])
        if min is STILL infinity:
            return -1
            //not possible for the coins to collide at a vertex
        else{
            return min
            //poossible for coins to collide
        }
```

Proof of Correctness and Runtime Analysis: This psuedo code describes exactly what we discussed above in words. Now, let us understand the proof of correctness and runtime for this algorithm. This algorithm relies on BFS which maintenance its invariant as the vertices that are being explored are done from order of increasing distance. Then to find the "meeting point" of the coins, we are using a conditional clause to indicate the vertex that the coins meet together, and then as per BFS, we ensure that we hit all the reachable vertices in the graph from both sources u and v. This indicates, that the algorithm takes all the required steps to identify the minimum steps that are needed to bring both coins together. The total runtime is O(V+E), as this is the runtime for a typical BFS traversal, where V represents the vertices in the graph and e represents the edges in the graph.

Now, let us look at the algorithm for part B. The only difference between the question for part A, and part B, is that in part B we are trying to find the minimum steps needed for 3 coins to meet, instead of 2. Our inputs will be the graph G, vertex u, vertex v, and vertex w. We are going to write a unique form of a BFS function that can handle "3" different states or positions. We want to find the point where coins (represented as 'a', 'b', 'c') are at the same vertex and at that point we can return the distance (the min steps). Let us look at the pseudo code below:

```
def min3Steps(Graph g, vetex u, vertex w, vertex w):
    queueNew=dequeue[u,v,w,0]
    visitedSet=set()
    vistedSet.add(u,v,w)
    //This ensures that we are setting
    our queue that we will need to perform
    our BFS traversal on


    while queue is not Empty:
        a,b,c,distance=queue.popLeft()
        //assign varaibles a,b,c with the vertices that we pop from our queue.

        if (a=b=c):
            return distance
            //this tells us that if a, b, c all equal each other, the 3 coins intersect,
            and the distance variable
            is their minimal steps
            that we can return

        for a in n Graph g.neighbors a:
            for b in G.neighbors b
                for c in G.neighbors c
                    next=(a.next, b.next, c.next)

            if next is not in visited set:
                visited.add(next)
                queue.enqueue(next)

    return -1;
```

The above pseudocode translates the algorithm into code using sets and queues, adding each vertex in, checking if it has been visited or not, and identifying where the intersection point between a, b, and c is. Now, how can we determine the **proof of correctness** for this algorithm?

The pseudocode handles three major things:

1. Performing BFS to ensure the vertices are explored in order of increasing distance from the source vertex.
2. Checking if there is a point where $a = b = c$, and returning the minimum number of steps if found.

3. Using "next states" to keep track of how each of the three "coins" (or a, b, c) will move.

The algorithm terminates as soon as $a = b = c$ is found. If this condition is not met, it returns -1. Thus, the algorithm handles all the cases needed to achieve the required end goal.

For this algorithm, the **runtime** is $O(V^3)$, as the triple nested loop dominates the other algorithmic operations, resulting in a cubic runtime complexity.

Now, the final part of this question is part c. Part c states that there are 42 coins placed on 42 unique vertices, and just like part a, and b, the question asks us to find a way to bring the coins together on the same vertex, using the minimal number of required steps. To tackle this question, we can use some of the same logic that we used for part a and b, which is using BFS (breadth first search) in order to find the minimal most distance between the source vertex and the other 41 vertices. Then, after calculating the minimal distances, we want to see how we can bring these coins together to one vertex, and to do that we can use a 2d Array to store the distance from coin x and the source (or where it started off). We want to see if there is a point where all coins can reach the same vertex, and if there is w return that minimal distance, or we will return -1, to indicate that there is no solution found. Let us look at the code below:

```
function coins42(Graph G, array of 42 vertices: array):
   totalVertices=length(Graph G)
   distancesArray=[42][totalVertices] and set these all to infinity

   for i in range 0 to 41:
       distancesArray[i]=bfs(G, array[i],totalVertices)

   minMoves=set to infinity in order to update minimum

   for i in rage totalVertices-1:
       maximumPath=0
       for j=0 to j=41:
           if distancesArray[j][i]==-1:
               maximumPath=infinity
               return
           else:
               maximumPath=max(maximumPath, distancesArray[j][i]

       if maximumPath< infinity:
           minMoves=min(minMoves,maximumPath)
           //update minimum by comparasions

       return minMoves if minMoves is not infinity
       else{
           return -1
       }
```

Now, let us analyze the proof of correctness, and runtime for this algorithm. We can start with our base case, where we ar assuming that there is only one vertex. If there is only one vertex, then by common sense, we can derive that all 42 coins must fall on that same vertex, and therefore, the minimum number of steps that the coins must move are 0 steps. For our inductive step, we want to prove that this algorithm applies for all k+1 coins. The good thing about using a black box algorithm like BFS, is that its' logic and reasoning is most usually proven to be correct. Our BFS algorithm that is used in the psuedo code will return the shortest path distance that each coin must take in order to get to every other coin, and thus it will always return the minimum number of steps, because we are relying on a heavily functionable algorithm like BFS. And lastly, let us analyze the run time. The runtime for this algorithm will be $O(V+E)$, as stated by the question, and this is derived from our BFS which traverses through every single vertex, V, and edge, E, in the graph.

Problem 4. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a French flag walk if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

Describe an algorithm to find all vertices in G that can be reached from a given vertex v through a French flag walk. **(25 points)**

Solution. To find all vertices in the graph that can be reached from a given vertex through a French flag walk, we can choose to use BFS. BFS works slightly better with queues, so we implement our algorithm by using queues.

The pseudocode would be as follows:

```
def french_flag_walk(graph, start_vertex) {
    queue = new empty queue //The queue should store (vertex, color_path)

    queue.append([start_vertex,0])

    visited = empty set()

    french_flag_vertices = set([start_vertex])

    while queue is not empty {
        curr_vertex, curr_path = queue.pop()


        for every neighbor of the curr_vertex in the graph {
            next_color = (curr_path + 1) % 3

            if graph[curr_vertex][neighbor] == next_color {
                if(neighbor, next_color) is NOT in visited {
                    queue.append((neighbor, next_color))
                    visited.add((neighbor, next_color))
                    french_flag_vertices.add(neighbor)
                }
            }
        }
    }
    return french_flag_vertices
}
```

Time Complexity: The time complexity of the algorithm would be: $O(V + E)$ where V is the number of vertices present in the graph and E is the number of edges.

To analyze why this is, we can check the amount of times we visit the vertexes and the edges. Each vertex can be visited at most 3 times to check for the 3 different colors of red, white, and blue. This would mean that the total vertex visits would be 3V.

For each vertex, we check all of its adjacent edges and each edge will be checked at most 3 times for the 3 different colors. This would mean that the total number of traversals for the edges would be 3E.

Putting that together, we would get that: $O(3v + 3E)$ which would simplify to a running time of: $O(V + E)$

Proof of Correctness: To prove that the algorithm correctly finds all the vertices reachable via a French flag walk from the start, we need to prove 3 different things: the vertices found are correct, the algorithm is accurate in finding all reachable vertices, and that the algorithm terminates as needed.

To prove that all vertices are correct, we nee to prove that there exists a valid French flag walk from the start vertex to v. To do this, we consider that every path in the algorithm would satisfy the French flag walk condition. The algorithm will only add a neighbor if its edge color matches the current color state (red $\rightarrow$ white $\rightarrow$ blue $\rightarrow$ red). This thoroughly checks that the edges follow the correct path, and therefore, any path found maintains the French flag walk.

If we assume that w is a vertex that is reachable by a French flag walk from the start vertex but for a proof of contradiction, we assume it is not found by the algorithm. Our algorithm works in that sense that given that BFS explores the paths systematically, the algorithm tracks the visited vertexes, and that color stage tracking prevents unnecessary path exploration. If w is NOT found, then either BFS did not explore a valid path or the path to vertex w violates the French flag walk condition. This concept though, contradicts the algorithm which is accurate in systematically checking all possible paths, tracking the color stages, and not revisiting paths/vertexes. Therefore, we know that vertex w must be found through the algorithm.

The last thing to prove that our algorithm works is to show that it terminates when needed. We know that the algorithm will eventually end because we have a finite number of paths (V * 3 different colors), each path (containing a vertex and color) is visited at most once and that the queue will become empty.

Problem 5. A vertex v in a connected undirected graph G is called a cut vertex if the subgraph $G - v$ (obtained by removing v from G) is disconnected.

Describe an algorithm that identifies every cut vertex in a given undirected graph in $O(V + E)$ time.

(25 points)

Solution. To solve this problem, we need to first understand what a cut vertex is. By definition, a cut vertex can be identified by one primary thing: If removing a vertex v would split the graph into two or more DISCONNECTED components, then we know there exists some vertices that can only reach each other through the vertex v. By this definition, vertex v can be considered a cut vertex.

This problem can be solved by using DFS traversal with Tarjan's algorithm. To start off, we would use DFS (Depth First Search), which would help us effectively find the cut vertices. While running DFS on the graph, we have to keep track of three main things for every vertex v:

1. *Discovery Time*: The discovery time of a vertex is when we first visit it.
2. *Low Values*: The low values of a vertex is the lowest discovery time that can be reached from each vertex using the back edges.
3. *Parent of v*: This is keeping track of the parent of v in the DFS tree

We would maintain a list that would store all the identified cut vertices and start dfs from any vertex. For simplicity purposes, we start dfs from 0 in our pseudocode algorithm.

The pseudocode for the algorithm is as follows:

```
def findCutVertices(graph):
    time = 0
    visited = new array of size = number of vertices initialized with [False] for
        all vertices
    disc = new array of size = number of vertices initialized with [-1] for all
        vertices

    //set to infinity so that the minimum operation works correctly to keep track
    //of the valid discovery times (which are guaranteed to be less than infinity)
    low = new array of size = number of vertices initialized with [infinity] for
        all vertices
    parent = new array of size = number of vertices initialized with [-1] for all
        verticles
    cutVertices = new empty set

    function dfs(vertex v) {
        mark v as visited: visited[v] = True
        set disc[v] = low [v] = time
        increment time: time++
        children = 0

        for each neighbor n of vertex v {
            if n is NOT visited  {
                parent[n] = v
                children++
                dfs(n)

                //this updates the low value of v after returning from n
                low[v] = min(low[n], low[v])

                //root case
                if (parent[v] == -1 and children > 1) {
                    add v to cutVertices
                }

                //non-root case
                if (parent[v] != -1 and low[n] >= disc[v]) {
                    add v to cutVertices
                }
            }
            else if(n != parent[v]) { //back edge case
                low[v] = min(low[v], disc[n])
            }
        }

    //this is for disconnected graphs: it performs dfs for all unvisited vertices
    for each vertex v in graph {
        if (v is not visited) {
            dfs(v)
        }
    }

    return cutVertices
}
```

The way the solution works is as follows:

1. **Initialization**
   (a) disc array stores the discovery times of the vertices
   (b) low array stores the lowest discovery time that is reachable from a vertex, including the back edges
   (c) parent array keeps track of the parent of each vertex in the DFS tree
   (d) cutVertices is the SET that stores the set of identified cut vertices
2. **DFS Traversal**
   (a) dfs starts by running on one vertex and for each unvisited vertex v, it assigns the vertex a discovery time and low value
   (b) It considers all the unvisited neighbors n of vertex v. Since the vertex is unvisited, it recursively performs DFS and updates the low value of v based on the subtree of its neighbor n. If the neighbor vertex n is already visited and vertex v is NOT its parent, then the low value of v is updated by using the discovery time of u.
3. **Check**
   (a) If vertex v is the root of the DFS tree and has more than one child, we consider v a cut vertex
   (b) If v is NOT the root of the DFS tree and no subtree rooted at its neighbor n cannot reach an ancestor of vertex v, then v is also a cut vertex.
4. **Return**
   (a) Once dfs is finished running on ALL vertices, then the algorithm returns the cutVertices set.

Running time: The algorithm does a single DFS traversal. DFS traverses each vertex and edge only once, which leads to a run time of $O(V + E)$ where E is the number of edges in the graph and V is the number of vertexes in the graph.

Proof of Correctness: To prove that the algorithm only identifies the cut vertices in the graph, we have to check two things: that every cut vertex is identified AND that every vertex that is identified, as a cut vertex is actually a cut vertex.

If we assume that vertex v is a cut vertex, then:

1. If v is NOT the root, then we must have a child n such that

   $$low[n] \geq disc[v]$$

   The algorithm successfully fulfills this condition during DFS and marks v as a cut vertex.

2. If v is the root with more than one child, then the algorithm successfully identifies it using the root condition.

If we assume that every vertex v that is identified as a cut vertex is correct, then: Since v is identified as a cut vertex:

1. If v is NOT the root and we have a child n such that

   $$low[n] \geq disc[v]$$

   then we know that n cannot reach the ancestor of vertex v without the vertex v itself. Since removing v disconnects n and its subtree from the rest of the graph, v can be considered a cut vertex.

2. If v is the root, we know that it has more than one child in the DFS tree. Removing v would mean that the subtrees rooted at v's children would be disconnected from v, thus proving that v is a valid cut vertex.

Problem 6. Suppose we are given a directed acyclic graph (DAG) $G = (V, E)$ whose nodes represent jobs, and whose edges represent precedence constraints. Specifically, each edge $u \to v$ indicates that job u must be completed before job v can begin. Each node v also has a weight $T(v)$ indicating the time required to execute job v.

(a) Describe an algorithm to determine the shortest interval of time in which all jobs in G can be completed.

(b) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time at which job v can begin.

(c) Now describe an algorithm to determine, for each vertex v, the latest time at which job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (a)), assuming that every job except v starts at its earliest start time (computed in part (b)). **(40 points)**

Solution. Part A: Topological sorting is a way to arrange items in a directed graph. We want to code out an algorithm to find the shortest interval of time in which all jobs can be completed and topological sorting is one of the smartest ways we can solve this problem. Vertices are connected with one-way arrows in a directed graphs, which means that topological sorting would ensure that the dependent items appear after the item that it depends on. In other words, topological sort would ensure that the algorithm is processing jobs in such a way that is consistent with precedence constraints.

The pseudocode for it would be as follows:

```
def top_shortest_time {
    sorted = TOPOLOGICAL_SORT(G)

    longest_time = new array[number of vertices]
    for each vertex v in V {
       longest_time[v] = 0
    }

    for each vertex v in order {
        max_time = 0

        for each predecessor p of v {
            max_time = max(max_time, longest_time[u])
        }

        longest_time[v] = max_time + T(v)  //
    }

    return max(longest_time)
}
```

The time complexity of this would be $O(V+E)$ because each vertex is visited once and each edge is processed once (DFS implementation) which leads to a running time of $O(V+E)$ and reeversing the list of the vertices takes $O(V)$ time. The time complexity simplifies to $O(V+E)$

The proof of correctness for this algorithm is as follows: Topological order works in the sense where vertices are ordered so that if there is an edge rom u to v, u appears before v in the order and if you invert toplogical order, it reverses the order. If we take the topological order of a directed acyrlic graph and reverse the list, then in the reversed list, u would appear before v. This reversal of a valid topological order would fulfil the reverse precedence constraints because reversing the order ensures that v is before u and the original topological order still respects the edge from u to v.

Part B: To solve this part, we would use the same topological order we computed in part a. After doing this, we would store an earliest start time for all vertices. For each outgoing edge from each vertex in our topological order, we would update the earliest state time using the max function on the vertex and weight, or the time it takes for the job to be completed (given from the problem).

The pseudocode would be as follows:

```
def earliest_start_time {
    order = TOPOLOGICAL_ORDER(G)

    earliest_time[v] = 0 for all vertices

    for each vertex u in topologicla order {
        for each outgoing edge (u --> v) from all edges {
            earliest_time[v] = max(earliest_start[v], earliest_start[u] + T[u])
        }
    }

    Return earliest_start[v] for all vertices
}
```

The runtime for this algorithm would be: $O(V+E)$. The dfs bsaed topological sort takes $O(V+E)$ for all vertices and edges. Each edge (from u $\rightarrow$ v) is processed once in constant time, leading to a runtime of $O(E)$. Thus, the overall runtime simplifies to $O(V+E)$

The proof of correctness for this algorithm is as follows:

1. Base Case: For source vertices (vertices with no incoming edges), there are no dependencies so we can correctly initialize ES[v] = 0 because such jobs can start immediately at time 0.

2. In a directed acyclic graph (DAG), a topological order guarantees that if there is an edge $u \rightarrow v$, then u will always come before v in the order. This property ensures that by the time we process v, the earliest start times $ES[u]$ for all its predecessors u have already been calculated.

   To calculate the earliest start time for a job v, we use the formula:

   $$ES[v] = \max(ES[v], ES[u] + T[u]),$$

   where $T[u]$ is the time it takes to complete job u. The term $ES[u] + T[u]$ represents the earliest time job u finishes. By taking the maximum over all predecessors, we ensure that $ES[v]$ reflects the earliest time v can start, accounting for all its dependencies.

   Since the algorithm processes jobs in topological order, all predecessors of v are guaranteed to have their $ES[u]$ values correctly computed before v is processed. This ensures that the computed $ES[v]$ is accurate.

3. The algorithm also ensures that for every edge from u $\rightarrow$ v, ES[v] $\geq$ ES[u] + T[u]. This guarantees that v does not start until all its predecessors u are completed.

4. Since the algorithm processes the vertices in topological order, every vertex is processed after its predecessors. Therefore, we know that for every vertex v, all ES[u] values for u $\rightarrow$ v are finalized before ES[v] is updated and the algorithm terminates after all the vertices are processed and ES[v] is finalized for every vertex v.

Part C: To find the latest start time for each job without increasing the overall completion time, you can start by doing a reverse topological sort. This would mean that you would process the jobs in reverse of the order we found in part A. After doing that, we would compute the latest completion time for all vertices by using the total time to complete all jobs. For each vertex u in reverse topological order and for each incoming edge v → u, we would update the latest complete time for v by setting it to the minimum between the latest completion time of v, and latest completion time of u and the weight given of vertex u. To end the algorithm, we would compute the latest start time for each job by doing the latest completion time for that job minus the weight/time needed to execute the job (v). This would leave us with the result that for each job, the latest strat time without increasing the overall completion time is LS[v].

The pseudocode for the algorithm would be as follows:

```
1. Reverse Topological sort on directed acyclic graph

2. latest_completion[v] for all vertices is initialized to infinity

3. For all sink vertices (vertices with no outgoing edges), the latest completion for that vertex i

4. Traverse the vertices in reverse topological order

for each vertex u {
    for each incoming edge (v --> u) {
        latest_completion[v] = min(latest_completion[v], latest_completion[u] - T[u])
    }
}

5. The latest start time for each job v can be computed as: LS[v] = LC[v] - T[v]

6. Return LS[v] for all vertices
```

Runtime Analysis

1. Topological Sort takes $O(V + E)$
2. Latest Completion Times: each edge is processed once ($O(E)$) and each vertex is processed once ($O(V)$).
3. Latest Start Times is calculated for all vertices, taking $O(V)$.

Thus, the total runtime of the algorithm is $O(V + E)$

Proof of Correctness

The algorithm computes the latest start times by propagating completion constraints in reverse topological order.

1. Initialization: The latest completion times ($LC[v]$) are initialized to ∞ for all vertices to ensure proper updates. Sink vertices (with no outgoing edges) are assigned $LC[v] = \text{MaxCompletionTime}$, as these jobs must finish by the total completion time.

2. Reverse Topological Order: Processing vertices in reverse topological order ensures that for every edge $v \to u$, the value $LC[u]$ is computed before $LC[v]$. This guarantees that each job's dependencies are resolved before it is updated.

3. Update Rule: For each edge $v \to u$, the formula

$$LC[v] = \min(LC[v], LC[u] - T[u])$$

ensures that job v finishes just in time for job u to start without increasing the overall completion time.

4. Latest Start Times: The latest start time for job v is calculated as:

$$LS[v] = LC[v] - T[v].$$

This ensures v starts at the latest possible time without delaying its own completion or affecting its successors.

5. Precedence Constraints: The algorithm ensures that for every edge $v \to u$, the condition

$$LS[u] \geq LS[v] + T[v]$$

is satisfied, maintaining the precedence constraints.

Conclusion: By processing all vertices and edges, the algorithm assigns valid latest start times without violating constraints or increasing the total completion time.

Problem 7. Tarjan's algorithm is an efficient method for finding all strongly connected components (SCCs) of a directed graph $G = (V, E)$ in linear time. A strongly connected component of a directed graph is a maximal subgraph in which any two vertices are reachable from each other. Tarjan's algorithm leverages Depth-First Search (DFS) to identify these components, ensuring each node in the graph is part of exactly one SCC.

The main idea of Tarjan's algorithm is to perform a DFS on G, keeping track of each vertex's discovery time v.index (a unique integer assigned in the order of discovery) and a secondary value v.lowlink, which is the smallest discovery index reachable from v in the DFS subtree, including v itself. The key steps of the algorithm are as follows:

1. For each unvisited vertex v in G, initiate a DFS that assigns v.index and v.lowlink.
2. Push each visited vertex onto a stack, and keep it on the stack until its strongly connected component has been fully explored.
3. For each successor w of v:
   - If w has not yet been visited, recursively call the DFS on w. Update v.lowlink as $\min(v.\text{lowlink}, w.\text{lowlink})$.
   - If w is on the stack, update v.lowlink as $\min(v.\text{lowlink}, w.\text{index})$.
4. After visiting all descendants of v, if v.lowlink = v.index, then v is the root of a strongly connected component. Pop vertices from the stack until v is removed, forming one complete SCC.

Using the above description, prove the correctness of Tarjan's algorithm. In particular, demonstrate that:

- The algorithm correctly identifies each strongly connected component.
- Each vertex belongs to exactly one SCC.
- The invariant properties of v.index and v.lowlink are maintained throughout the execution of the algorithm.

(25 points)

Solution. This question asks us to use the given description to prove the correctness of Tarjan's algorithm. We need to prove 3 distinct points which are that the algorithm correctly identifies "each strongly component", that each vertex belongs to exactly one strongly connected componenet, and that the invariant properties of v.index and v.lowlink are maintained throughout the execution of the algorithm. ***Let us start by proving the very first point which is that the algorithm is able to identify each SCC.*** We know that the algorithm performs DFS as it is given to us in the description itself. When the DFS is performed, the graphs' vertices will each be assigned a particular discovery index from v.index. When we reach the point in the algorithm were v.lowlink=v.index, this is how we know that the root of the strongly connected component is v itself. It is important to not think that the v.lowlink will have the smallest discovery index among the other vertices. When this algorithm is being performed, the way that the stack works in space allocation is so that the Strongly Connected Component will have all the nodes within that particular cycle. Once the root (v) is identified for the SCC, the stack pops off all the other nodes that are apart of that cycle. Because DFS is used in this algorithm, that is how we can ensure that every single vertex in the graph is visited at one point, as well as the root itself is found as v.lowlink traverses through the cycle. The critical part of this algorithm is the line : v.lowlink=v.index, because that is how we know that the root is found and that this node can be popped off from the stac to avoid overlapping connected components. Using all of this information, we can draw the conclusion that the algorithm does indeed identify every SCC in the graph.

Now our second property that we must prove is that each vertex belongs to exactly one SCC. How can we ensure that this is indeed the case? Once again we need to go back to our DFS call stack in th algorithm. We know that a vertex gets pushed onto the stack if is being handled by DFS and is being ”visited”. A vertex will only undergo DFS and get pushed into the stack once. Till we get to the point where can clearly identify a strongly connected component using a root (where v.lowlink=v.index), the vertex that is visited and being ”looked at” stays on the stack till it gets popped. Once we find the root, we pop the remaining vertices off our stack to signify our newly formed SCC. The way we can verify that each vertex belongs to exactly one SCC, is because since these vertices get popped off the stack once a root is found, it allows for absolutely no overlap between the vertices in the SCC, ensuring that each vertex belongs to exactly one SCC.

And finally, ***the last property that we need to prove is that the invariant properties of v.index and v.lowlink are maintained throughout the algorithm.*** This appears pretty intuitive at first because based on the first two properties and their proofs, we can see that v.lowlink and v.index are consistently present in order to find the root of the SCC, but let us break it down a little more. When our DFS first begins, we intialize both v.index and v.lowlink using the current depth of the DFS which is of course the smallest index. As DFS has a recursive step, we must note that if a node p, that successes the current node has not been visited yet, then we must call DFS on p, as it is unvisited. Again, we want to follow this idea of constantly updating v.lowlink to min(v.current, p.lowlink), to find the smallest distant node in the SCC. We keep updating the minimum using this logic. The base case for this algorithm appears when v.lowlink=v.index, and at this point we know we have reached the root of the SCC and the vertex is remvoved from the stack itself once and for all so there are no overlaps and so that it cannot be ”revisited” in the past. This goes back to proving our previous point 2. Using this information, we can se that the invariant properties of v.index ad v.lowlink are indeed preserved as the algorithm executes in completion.

Problem 8. Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

(a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

(b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

(25 points)

Solution. (A) Part A of this question asks us to describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased. For this question, our input would be a graph G that has edges which are weighted, we also must be given some kind of already existing minimum spanning tree of T on graph G. And we want to be able to decrease the weight of the edge from w to w', where w' is less than w. The steps in this algorithm included ensuring that the existing minimum spinning tree is valid, then we want to add our edge to the MST which may create a cycle in the minimum spanning weighted graph. This is how I intuitively think of going about this algorithm in my mind: if the input edge e is in the given minimum spanning tree T, then we want to get rid of that specific weight from the tree, which actually makes the tree's runtime more efficient. However, if the edge itself is not in the tree, then we want to add this particular edge to the tree and see if adding this edge creates some sort of cycle in the tree. If a cycle forms in the tree, we must traverse the tree to see which edge contributes the most weight to the spanning tree, and accordingly remove it from the tree, T.

Now, let us look at the psuedo code for this question:

```
function minSpanTreeDec(graph G, tree T, weight w, edge e):

//start by applying the Kruskal's
algorithm to find
whether edge e is in the spanning tree T


if algorithm returns true, if e is in T:
     w=e.weight
     return T

else:
    T.add(e,w) //if it is not in the graph already, add the new edge

    newCycle=DFS(T) /see whether this new edge forms a cycle
    largestEdge=Kruskals(newCycle)

    if largestEdge is not edge e:
        T.remove(largestEdge)

return T
```

As is evident, the algorithm above does exactly what is described above. Now, **let us analyze its run time,** The runtime for this algorithm is O(V+E), and this is because DFS itself takes O(V+E), and an addition O(V) time to find the most weighted edge using Kruskal's algorithm. So combined, this gets us O(2v+E), but simplified it gets us O(V+E), which is our final run time.

Now, for the **proof of correctness**, we can go about this by understanding that if we have an edge e, and we are trying to minimize the weight of it, that means that the weight of the spanning tree T also becomes reduced in return. If no edge e exists itself, then the appropriate action is to add that edge e, and ensure that the weight of the spanning tree is still minimal, by removing it if it indeed is the heaviest edge. This explanation validates the algorithm above.

Part B

In this approach, we follow a similar methodology to part (a). The first step involves checking whether the edge e is already present in the minimum spanning tree T. If the edge is not part of T, no modifications are necessary, and we can simply return the tree as is. However, if the edge e is already in T, we must proceed with caution when updating its weight.

Several key considerations come into play during this process. We need to be mindful that another edge might potentially result in a tree with a lower total weight. When we determine that the edge is valid for weight increase, we update e's weight to the new, higher value. An important step is to handle disjoint components if an edge is removed.

```
def increasingTree(graph G, tree T, edge E, weight W) {

    Black Box: Kruskal's algorithm to find if e is in T {
        if e is NOT in T {
            return T
        }
    }

    e.weight = w
    T.delete(e)

    disjointComponents = DFS(T)
    disjoint1, disjoint2 = disjointComponents[0], disjointComponents[1]

    minimumE = inf

    for each edge E in graph G {
        if E connects disjoint1 to disjoint2 {
            if E.weight < minimumE.weight or minimumE == 0 {
                minimumE = edge
            }
        }
    }

    T.add(minimumE)
    return T

}
```

Proof of Correctness The algorithm ensures the integrity of the minimum spanning tree when increasing an edge's weight. We check whether an edge E is in the tree T or not when increasing the weight of an edge. If E is not part of T, then we know that the tree is already correct and the algorithm returns the same. We need to make sure we are properly updating T so that we can increase T without violating the rules of a minimum spanning tree when the edge E is part of the tree T. This is done when our algorithm splits the tree into two separate connected components, and then finding the edge that can connect them. Once this is done, our algorithm rearranges the tree and returns it. This proves our algorithm is accurate and correct.

Runtime Analysis

- Part (a): $O(V+E)$ for DFS + $O(V)$ for finding the heaviest edge (Kruskal's algorithm). Total runtime is $O(2V+E)$ which simplifies to $O(V+E)$
- Part (b): $O(V+E)$ for DFS + $O(E)$ for finding the lightest edge. Total runtime is $O(V+2E)$ which simplifies to $O(V+E)$

Challenge Yourself. Suppose you are given an arbitrary directed graph G in which each edge is colored either red or blue, along with two special vertices s and t.

(a) Describe an algorithm that either computes a walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

(b) Describe an algorithm that either computes the shortest walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

Solution. For part a, we can use BFS and modify it so that along with tracking the vertices visited, it also tracks the sequence of the edge colors along the walk.

The pseudocode for the algorithm would be as follows:

```
def isPalindrome(sequence){
    return sequence == Reverse(sequence)
}

def MaybePalindrome{
    return length(sequence) <= 2 * length(unique_elements(sequence)) - 1
}

def FindPalindromePath(Graph G, vertex s, vertex t) {
    Q = empty queue that stores curr_vertex, color_path, walk)

    Enqueue to Q the following: (s, [], [s]))

    //keeps track of the visited states
    visited = empty set

    While Q is not empty{
        Dequence from Q: (curr_vertex, color_path, current_path)

        //assign a unique state key
        state key = (curr_vertex, color_sequence)

        //do not add to visited if the state_key is already present in that array
        if state_key is in visited {
            continue
        }

        //add to visited if not already present
        add state_key to visited

        //check if current vertex is at t and forms a palindromic walk
        if curr_vertex = t and isPalindrome(color_path) {
            return current_path
        }

        for each neighbor and edge_color of the neighbors(curr_vertex) {
            color_path2 = color_path + edge_color

            if CanBePalindrome(color_path2) == false {
                continue
            }

            walk2 = (current_path, neighbor)
            Enqueue to Q: (neighbor, color_path2, walk2)
        }
    }

    Return NULL
}
```

Runtime Analysis; To explore every possible possibility, we take the number of vertices V and have to account for the fact that we traverse an edge forward and backward. This would mean that the color sequence can end up with $2V$ elements in the worst-case scenario.

Since BFS explores every unique state, and if we assume that edges can have only two colors, then the number of possible sequences p of length $2V$ is 2^{2V}.

When BFS runs, for each state, you check every outgoing edge from the current vertex, and the maximum number of neighbors would be V. Therefore, for every state, the amount of work being done is $O(V)$.

If we multiply them together:

$$O(V \cdot 2^{2V}) \cdot O(V) = O(V^2 \cdot 2^{2V})$$

However, we only want the dominant term for Big O notation, so the runtime for this algorithm simplifies to:

$$O(V \cdot 2^{2V})$$

Proof of Correctness: To prove that the algorithm is correct, we need to prove three different things: that the algorithm will terminate at some point, that the algorithm is correct in that sense that if it finds a walk, the walk will be a palindromic walk and that the algorithm is complete and will find a palindromic walk from s to t if it exists.

1. The algorithm uses BFS to check all possible paths/walks in the graph and this guarantees that we are checking every possibility in level-order. We know that the algorithm will terminate if a palindromic walk is found. The algorithm will also terminate if all possible walks are checked and no palindromic walk has been found.
2. If the algorithm returns a walk, we are guaranteed that it is a palindromic walk. This is due to two reasons. The color sequence of the walk will be a palindrome because we check for that right with the current color sequence after the algorithm reaches vertex t. This means that the walk did reach the end point of vertex t since that is the only point where we check to see if we have a palindromic walk.
3. The algorithm is guaranteed to check all possible walks from vertex s to vertex t. BFS starts by exploring walks with the lowest length (starting at 1) and making its way up with the size of the length (going to length 2, then 3, etc). This ensures that we explore all possible walks in the order of increasing path length. Since the search includes every possibility, if s to t has a palindromic walk between them, the algorithm will be successful in finding it, making it a complete algorithm.

For part b, you can utilize a priority queue and Dijkstra's algorithm to find the shortest walk from vertex s to vertex t such that the pattern of red and blue edges along the walk is a palindrome.

The pseudocode for the algorithm would be as follows:

```
  def isPalindrome(sequence){
            return sequence == Reverse(sequence)
        }

        def MaybePalindrome{
            return length(sequence) <= 2 * length(unique_elements(sequence)) - 1
        }

def shortestPath {
    PriorityQueue pq = empty priority queue
    visited = empty set

    //parameters for pq are: the length of shortest path, the current vertex,
    //and the path/sequence of colors)
    pq.insert((0, s, []))

    while pq is NOT empty {
        pq.pop(cost, curr_vertex, color_path)

        possibility = (curr_vertex, tuple(color_path))

        if possibility exists in visited {
            continue //if its already visited, we can skip it
        }

        visited.add(possibility)

        if curr_vertex = t and isPalindrome(color_path) {
            return color_path
        }

        for each(neighbor, edge_color) in G[curr_vertex] {
            color_path2 = color_path + edge_color

            if MaybePalindrome(color_path2) == False {
                continue
            }

            pq.insert((length + 1, neighbor, color_path2))
        }
    }

    Return NULL
}
```

Runtime Analysis; To explore every possible possibility, we take the number of vertices V and have to account for the fact that we traverse an edge forward and backward. This would mean that the color sequence can end up with $2V$ elements in the worst-case scenario.

Since BFS explores every unique state, and if we assume that edges can have only two colors, then the number of possible sequences p of length $2V$ is 2^{2V}.

For the priority queue, each insertion or removal takes $O(\log P)$, where P is the number of paths (a vertex and a color sequence).

Thus, we have that the total states are:

$$O(V \cdot 2^{2V}).$$

The operations per state would come out to be:

$$O(\log(V \cdot 2^{2V})) = O(V + 2V) = O(V).$$

Therefore, the total time complexity of this algorithm would be:

$$O(V \cdot 2^{2V} \cdot V) = O(V^2 \cdot 2^{2V}).$$

Proof of Correctness: To prove that the algorithm is correct, we need to prove three different things: that all valid walks are checked, the algorithm terminates when all possible sequences are processed and that the shortest path is guaranteed.

1. The algorithm checks all valid walks because it only skips the states/vertexes that are already visited or are guaranteed to NOT form a palindrome.
2. The algorithm is set to terminate when the priority queue is empty, which makes sure that all possible states/sequences are checked.
3. We are utilizing Dijkstra's algorithm. This ensures that each state is processed in increasing order of the path length. Due to our priority queue ordering, we are guaranteed to get the shortest path the first time we even process vertex t with a palindromic sequence.