Statement of Integrity: I, Derek Zhu, state that I completed this assignment with integrity and by myself.

Homework 5 is here, Due: Aug 5, 2024, to be submitted at here

Problem 1:

My solution:

(a) Zero number of edges we have to add to the MST before adding an edge that includes vertex G. Since Prim's algorithm can pick up any vertex as the start vertex, therefore we can pick up vertex G as the start vertex and add it to MST before adding any other edges to MST.

(b) Zero number of edges we have to add to the MST before adding an edge that includes vertex G. Krustal's algorithm add the edges that have least edge weight to the MST as the first edge, the edge (G,E) ties the least weight with edge (A,B), (C,D), any edge of these three edge could be the first edge to add to the MST, hence we can chose edge (G,E) as the first edge to add to the MST, hence the Zero number of edges we have to add to the MST before adding an edge that includes vertex G.

(c) To use the black box that computes the minimal spanning tree (MST) to instead compute a maximal spanning tree (MST but with maximum weights), we can transform the input graph in such a way. Firstly we negate the edge weights. Secondly, we use the black box device to compute the minimal spanning tree, thirdly The black box will compute the minimal spanning tree of this transformed graph. Finally we interpret the result in this way: the minimal spanning tree of the graph with negated weights corresponds to the maximal spanning tree of the original graph. This is because minimizing the sum of negated weights is equivalent to maximizing the sum of the original weights.

Problem 2:

My Solution: To determine if the proposed four-symbol game (rock, paper, scissors, water) is fair, we need to analyze the game using graph theory principles. In this context, each symbol represents a node in a directed graph. A directed edge from node u to node v indicates that symbol u wins against symbol v.

For the game to be fair, each symbol should have:
1. The same number of outgoing edges (winning cases).
2. The same number of incoming edges (losing cases).
3. No symbol should both win against and lose to the same symbol.

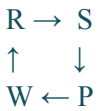
Given the 4 symbols, the maximum number of possible outgoing edges for each node is 4 - 1 = 3, but it would not be a fair game if one symbol wins against the other three symbols. Hence, the number of outgoing edges for each node must be less than 3, at most 2.

If the number of outgoing edges for each node is 2, then we need $2 * 4 = 8$ edges. Since the maximum number of possible one-way edges in a 4-node graph is $3 * 2 = 6$ edges, there must be $8 - 6 = 2$ two-way edges between some pairs of symbols. This conflicts with the rule that no symbol should both win against and lose to the same symbol. Hence, the number of outgoing edges for each node must be less than 2, or at most 1.

If the number of outgoing edges for each node is 1, then we need $1 * 4 = 4$ edges. Since the maximum number of possible one-way edges in a 4-node graph is $3 * 2 = 6$ edges, then there must be $6 - 4 = 2$ pairs of symbols that have no edges between them. This means there are two pairs of symbols that neither win against nor lose to each other; in other words, these pairs of symbols tie.

Therefore, to design a fair game of the proposed four-symbol game, each symbol should win against one, lose to another, and tie with a different one. Let's define the symbols and rules: Rock (R), Paper (P), Scissors (S), Water (W).
- Rock: Wins against Scissors, loses to Water, ties with Paper.
- Scissors: Wins against Paper, loses to Rock, ties with Water.
- Paper: Wins against Water, loses to Scissors, ties with Rock.
- Water: Wins against Rock, loses to Paper, ties with Scissors.

Here's a visual representation of the graph:



By ensuring each symbol wins against one symbol, loses to one symbol, and ties with one symbol, the game maintains balance and fairness. This structure guarantees that no symbol is dominant over others, creating an equal chance for each player to win. This variant of the game with four symbols (rock, paper, scissors, water) is fair and balanced.

Problem 3:

Review concepts:

Complete Graph Kn: A graph with n vertices where every vertex is connected to every other vertex.

Clique: A subset of vertices such that every pair of vertices in the subset is connected by an edge. The size of the largest clique in a graph G is called the clique number and is denoted by **ω(G)**.

Chromatic Number χ(G): The minimum number of colors required to color the vertices of G such that no two adjacent vertices share the same color.

I**nduction Hypothesis**: Assume that for any graph G with n vertices, the chromatic number χ(G) is no less than ω(G) which is the size of the maximal clique of G: $\chi(G) \geq \omega(G)$.

Base Cases:

(1) For a graph G1 with one vertex (n=1):$\chi(G1) = 1$, since only one color is needed $\omega(G1) = 1$. Therefore, $\chi(G1) >= \omega(G1) = 1$, and the base case holds.

(2) For a graph with two vertex (n=2):

Consider Graph $G2 = G1 + v$: Let G2 be a graph with $1 + 1 = 2$ vertices, where one vertex is in G1, and one vertex v added to G1 to form G2. In another word, removing a vertex v from G2 to get a subgraph G1 with one vertex, or adding a vertex v to G1 to get G2. We need to determine if $\chi(G2) \geq \omega(G2)$ still holds after we saw $\chi(G1) >= \omega(G1)$.

If v is not part of any maximal clique in G2 (if there is no edge). The maximal clique in G2 is the same as the maximal clique in G1, which has size $\omega(G1) = 1, or: \omega(G2) = \omega(G1) = 1$. Since v is not part of any new clique, it can be colored with one of the χ(G1) colors used for G1 without any conflict. Thus, $\chi(G2) = \chi(G1)$. Therefore $\chi(G2) \geq \omega(G2)$ still holds since $\chi(G1) >= \omega(G1)$ as discussed above.

If v forms a new maximal clique in G2 (If there is one edge v1−v2): Let C2 be a new maximal clique in G2 that includes v and has size $\omega(G2) = k2 = 2$. Then the vertices of C1 (excluding v) form a clique in G1 of size $\omega(G1) = (k2 - 1) = 2 - 1 = 1$ since C1 is one vertex less than C2. Adding v to G requires at least one more color to form the new clique C2, hence $\chi(G2) \geq \chi(G1) + 1 >= (k2 - 1) + 1 = k2 = 2$, menas:$\chi(G2) >= k2$. Since $\omega(G2) = k2$, therefore $\chi(G2) \geq \omega(G2)$ still holds.

Induction Step: We need to show that the statement holds for a graph G′ with n+1 vertices.

Consider Graph $G' = G + v$: Let G′ be a graph with n+1 vertices, where n vertices are in G, and one vertex v added to G to form G'. In another word, removing a vertex v from G′ to get a subgraph G with n

vertices, or adding a vertex v to G get G'. By the induction hypothesis, $\chi(G) \geq \omega(G)$. We need to determine if $\chi(G') \geq \omega(G')$ still holds.

Case 1: v is not part of any maximal clique in G′. The maximal clique in G′ is the same as the maximal clique in G, which has size $\omega(G) = k$, or: $\omega(G') = \omega(G) = k$. Since v is not part of any new clique, it can be colored with one of the χ(G) colors used for G without any conflict. Thus, $\chi(G') = \chi(G)$. Therefore $\chi(G') \geq \omega(G')$ still holds since $\chi(G') = \chi(G) \geq \omega(G) = \omega(G')$.

Case 2: v forms a new maximal clique in G′: Let C′ be a new maximal clique in G′ that includes v and has size $\omega(G') = k'$. Then the vertices of C (excluding v) form a clique in G of size $\omega(G) = (k' - 1)$ since C is one vertex less than C'. By the induction hypothesis, $\chi(G) \geq \omega(G) = (k' - 1)$, hence $\chi(G) \geq (k' - 1)$. Adding v to G requires at least one more color to form the new clique C′, hence $\chi(G') \geq \chi(G) + 1 >= (k' - 1) + 1 = k'$, menas:$\chi(G') >= k'$. Since $\omega(G') = k'$, therefore $\chi(G') \geq \omega(G')$ still holds.

Both cases (Case 1 and Case 2) demonstrate that for any graph G′ with n+1 vertices , the chromatic number χ(G′) is no less than the size of the maximal clique ω(G′). Thus, by induction, the chromatic number χ(G) of any graph G is at least the size of its maximal clique.

Proved!

Problem 4: My solution:

Solution to (a) :

Idea: we will use a Breadth-First Search (BFS) approach to simulate the spread of the virus through the network of computers.

Given: a list of computers with n computers, a list of triples (Ci,Cj,tk) with m triples sorted order by time tk; the start computer Ca, start time ta, target computer Cb, and target time tb.

Pseudocode:

function **Algorothm_HW5_Question4** (Ca, Cb, ta, tb, triple_list, computer_list):

```
        # Initialize the infection time for all computers to infinity
#1      {infection_time[ci] ← ∞ , for Ci in computer_list with n computers}
        # The infection starts at Ca at time ta
#2      infection_time[Ca] ← ta
        # Process the triples in sorted order
#3      for (Ci, Cj, tk) in triple_list:
            # If Ci is infected and tk >= infection_time[Ci],
#4          if infection_time[Ci] <= tk:
                # then Cj can get infected no later than tk
#5              infection_time[Cj] ← min(infection_time[Cj], tk)
            # Similarly, if Cj is infected and tk >= infection_time[Cj]
#6          if infection_time[Cj] <= tk:
                # then Ci can get infected no later than tk
#7              infection_time[Ci] ← min(infection_time[Ci], tk)
        # Check if Cb is infected by time tb
#8      if infection_time[Cb] <= tb:
#9          Return true
#10     Else return false
```

Solution to (b) :

Runtime:

Line#1: n
Line#2: 1
Line#3: m
Line#4~5: <2m
Line#6~7: <2m
Line#8~10: 2

$Total$: $T(n,m) <= (n + 3 + 5m) = \Omega(n + 5m)$

In our case, n is the number of computers and is a given const, or n can be considered as dominated by m, hence $T(n,m) = T(m) = \Omega(5m) = \Omega(m)$

Proved!

Problem 5(a)

A perfect matching is a matching under which every vertex is matched (refer to textbook 25.1-5 page 715). To describe a bipartite graph G such that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op, we can create a bipartite graph with two sets of vertices. One set represents the people $P = \{p_1, p_2, ..., p_n\}$, and the other set represents the nights $D = \{d_1, d_2, ..., d_n\}$, then we can draw an edge between a person p_i and a night d_j if person p_i is available to cook on night d_j (i.e., $d_j \notin S_i$). Then find out a perfect matching in this bipartite graph corresponds to a feasible dinner schedule. This means every person is assigned exactly one night to cook, and every night has exactly one person assigned to cook, to satisfy their availability constraints. In another word, in graph $G = (V, E)$, $V = P \cup D$, $|P| = |D| = n$, each vertex has one and only one edge connected to it, and there are n edges between vertices $P = \{p_1, p_2, ..., p_n\}$ and vertices $D = \{d_1, d_2, ..., d_n\}$.

Example: If $n = 3$, people $P = \{p_1, p_2, p_3\}$ and nights $D = \{d_1, d_2, d_3\}$. Assume availability sets: $S_1 = \{d_2\}$, $S_2 = \{d_3\}$, $and\ S_3 = \{d_1\}$. The edges in the bipartite graph would be: $(p_1, d_1), (p_1, d_3), (p_2, d_1), (p_2, d_2), (p_3, d_2), (p_3, d_3)$. A perfect matching would indicate a feasible schedule, such as: p1 cooks on d1, p2 cooks on d2, p3 cooks on d3, and the corresponding edges in the bipartite graph would be $(p_1, d_1), (p_2, d_2), (p_3, d3_3)$

Problem 5(b)

Idea: we are tasked with fixing an almost correct schedule where two people are assigned to the same night, and another night is left unassigned. This involves checking for alternative matchings to correct the issue, ensuring that each person is assigned exactly one night and each night is assigned exactly one person. This problem involves finding the maximum matching in a bipartite graph. A matching in a bipartite graph is a set of edges such that no two edges share a common vertex. The goal is to find the maximum number of edges in such a matching. The problem can be modeled as a bipartite graph where one set of vertices represents people and the other set represents nights as we did in above. The task of assigning each person to a unique night (and vice versa) corresponds to finding a matching in this bipartite graph as shown in above example. To correct the schedule done by Alanis, we need to find an augmenting path in the residual graph to adjust the current matching, which is a typical step in algorithms for finding maximum bipartite matchings. **If no augmenting path is found, it means no feasible schedule exists**. Here is the setup: use the almost correct schedule to identify the bipartite graph with vertices P and D as we discussed in above, and the given matching which have two vertices $p_i\ and\ p_j$ both matched to d_k and no vertex matched to d_l. Then we look for an augmenting path in the residual graph to reassign the conflicted nigh, if an augmenting path is found, adjust the matching accordingly by alternating the matched and unmatched edges along the path P.

Pseudocode:

```
function fixSchedule_HW5_Question_5(G, initialMatching, p_i, p_j, d_k, d_l):
#1   residualGraph ← constructResidualGraph(G, initialMatching)
#2   augmentingPath ← findAugmentingPath(residualGraph, p_i, d_l)
#3   if augmentingPath is None:
#4       return "No feasible schedule exists"
#5   else:
#6       initialMatching ← updateMatching(initialMatching, augmentingPath)
#7       return initialMatching

function constructResidualGraph(G, initialMatching):
#8   residualGraph ← {}
#9   for edge in G.edges:
#10      if edge in initialMatching:
#11          residualGraph.addReverseEdge(edge)
#12      else:
#13          residualGraph.addForwardEdge(edge)
#14  return residualGraph

function findAugmentingPath(residualGraph, start, end):
#15  parent = BFS(residualGraph, start, end)
#16  if parent[end] is None:
#17      return None
#18  else:
#19      path ← []
#20      while end is not start:
#21          path.append((parent[end], end))
#22          end ← parent[end]
#23      return path

function updateMatching(matching, augmentingPath):
#24  for (u, v) in augmentingPath:
#25      if (u, v) in matching:
#26          matching.remove((u, v))
#27      else:
#28          matching.add((u, v))
#29  return matching

function BFS(residualGraph, start, end):
#30  parent ← {node: None for node in residualGraph.nodes}
#31  queue ← [start]
#32  while queue:
#33      current ← queue.pop(0)
#34      for neighbor in residualGraph[current]:
#35          if parent[neighbor] is None and residualGraph[current][neighbor] > 0:
#36              parent[neighbor] ← current
#37              queue.append(neighbor)
#38              if neighbor == end: return parent
#40  return parent
```

Time Complexity

1. constructResidualGraph(G, matching) Lines 8-14:
Line#8: Initializing residualGraph takes O(1);
Line#9~13: The for loop iterates over all edges in G. Assuming G has m edges, this loop runs O(m) times. Inside the loop: Checking if an edge is in the matching takes O(1). Adding a reverse or forward edge to residualGraph also takes O(1);
Line#14: returning result takes O(1);
Time Complexity: O(m)

2. BFS(residualGraph, start, end) Lines 30-40:
Initializing the parent dictionary takes O(n), where n is the number of nodes.
Initializing the queue takes O(1).
The while loop processes each node at most once, so it runs O(n) times.
The inner for loop iterates over the neighbors of the current node, which takes O(m) in total over all iterations of the while loop, as each edge is considered once.
Time Complexity: O(n+m)

3. findAugmentingPath(residualGraph, start, end)Lines 15-23:
Line 15: Calls the BFS function.
Lines 16-23: If an augmenting path is found, constructing the path takes O(n) in the worst case (if the path length is n).
Time Complexity of findAugmentingPath: **O(n+m)**

4. updateMatching(matching, augmentingPath)Lines 24-29:
The for loop iterates over the edges in the augmenting path. In the worst case, the path length is O(n).
Checking membership in the matching and adding/removing an edge both take O(1).
Time Complexity: **O(n)**

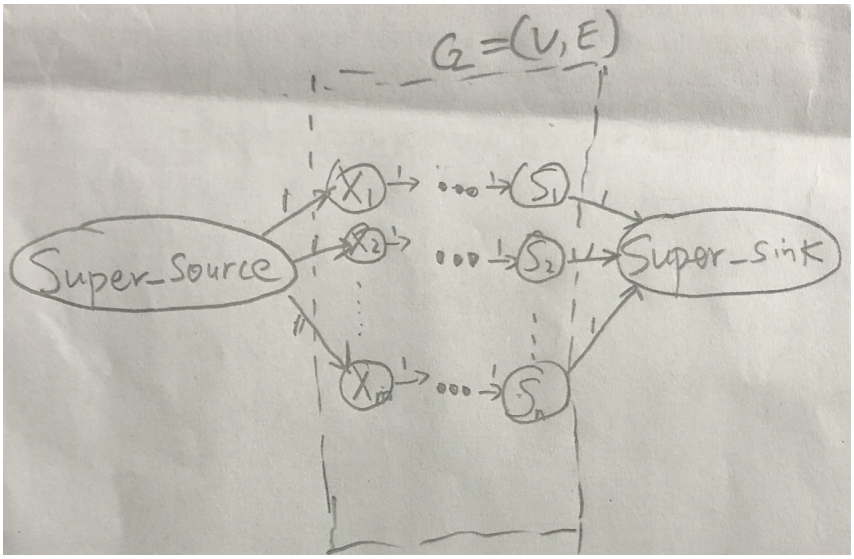
5. fixSchedule(G, initialMatching, p_i, p_j, d_k, d_l)
Line 1: calls constructResidualGraph, which takes O(m).
Line 2: calls findAugmentingPath, which takes O(n+m).
Lines 3-4: Checking the condition and returning a string takes O(1).
Lines 5-7: If an augmenting path is found, calling updateMatching takes)O(n).
Time Complexity: O(m)+O(n+m)+O(n)
Since each person can potentially be connected to each night they are available to cook. In the worst case, if every person is available every night, then $m = n^2$.
hence: $Time\ Complexity = O(m) + O(n + m) + O(n) = O(n^2)$

Problem 6:

My solution to (a):

Idea: to determine if a set of evacuation routes exists, we can model the problem using flow networks so that the maximum flow calculation in the flow network directly determines the feasibility of the evacuation routes: if the maximum flow equals $|X|$ then a set of evacuation routes exists, as each vertex in X can be matched to a unique vertex in S through disjoint paths, otherwise no such set of evacuation routes exists, as not all vertices in X can be safely evacuated to S without sharing edges.

Setup: given a directed graph $G = (V, E)$, a set of populated vertices X, and a set of safe vertices S. X and S are disjoint sets. **we add a new vertex super_source, and a new vertex super_sink, connect the super_source to each vertex in X with capacity 1, connect each vertex in S to the super_sink with capacity 1, then assign a capacity of 1 to each edge in the original graph G, this ensures that paths do not share any edges (as each edge can carry at most one unit of flow).** Then to calculate the maximum flow from super_source to super_sink by using the Ford-Fulkerson algorithm or Edmonds-Karp algorithm which has polynomial runtime proved in our textbook.



Pseudocode:

```
FUNCTION findEvacuationRoutes_Question_6a(G, X, S):
        // Step 1: Construct the flow network
#1:     G_f ← constructFlowNetwork(G, X, S)
        // Step 2: Compute the maximum flow from super source s to super sink t
        // by using the Ford-Fulkerson algorithm or Edmonds-Karp algorithm
#2      maxFlow ← Ford-Fulkerson-algorithm(G_f, s, t)
        // Step 3: Check if the maximum flow equals the number of populated vertices |X|
#3      IF maxFlow = |X| THEN
                RETURN "A set of evacuation routes exists"
#4      ELSE
                RETURN "No feasible set of evacuation routes exists"

FUNCTION constructFlowNetwork(G, X, S):
#5      G_f ← new Graph() // Initialize a new graph for the flow network
        // Add super source s and super sink
#6      s ← new Vertex("super_source")
#7      t ← new Vertex("super_sink")
#8      G_f.addVertex(s)
#9      G_f.addVertex(t)
        // Add vertices and edges from the original graph G
#10     FOR EACH vertex v IN G.vertices:
#11             G_f.addVertex(v)
#12      FOR EACH edge (u, v) IN G.edges:
#13             G_f.addEdge(u, v, capacity=1)
        // Connect super source s to each vertex in X with capacity 1
#14     FOR EACH vertex x IN X:
#15             G_f.addEdge(s, x, capacity=1)
        // Connect each vertex in S to super sink t with capacity 1
#16     FOR EACH vertex s IN S:
#17             G_f.addEdge(s, t, capacity=1)
#18     RETURN G_f
```

Runtime :

Line#5~9: O(1); Line#10~17: O(|V|+|E|); Line#18: O(1)

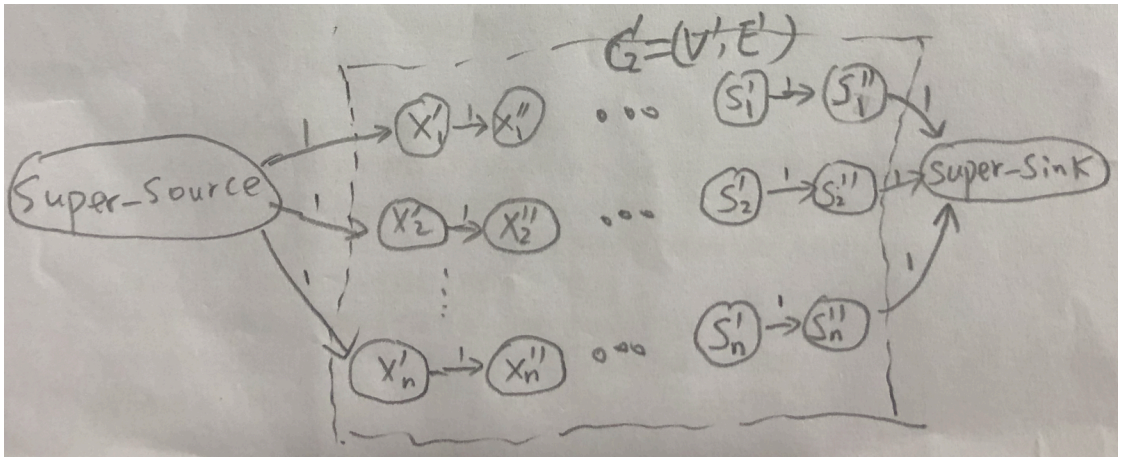
O(1) + O(|V|+|E|) + O(1) = O(|V|+|E|)

Since function **constructFlowNetwork** has runtime O(|V|+|E|) and the Ford-Fulkerson algorithm has polynomial runtime, hence our findEvacuationRoutes_Question_**6a** has polynomial runtime.

My solution to (b):

Idea: in above solution to (a), it is possible that some vertices may be shared even though there is no edge shared. To not share vertex for any path, we need to transform the original graph into a new graph then apply the solution to (a) to the transformed graph to solve the problem (b) by splitting each vertex in X and S into two vertices that connect to each other with unit capacity.

Setup: given a directed graph G = (V, E) , a set of populated vertices X, and a set of safe vertices S. X and S are disjoint sets. we transform G into G' = (V', E') as:
for each $v \in X$, *create two vertices* $v_{in} \in X'$ *and* $v_{out} \in X'$ and an edge $e = (v_{in}, v_{out}) \in E'$.
for each $v \in S$, *create are two vertices* $v_{in} \in S'$ *and* $v_{out} \in S'$ and an edge $e = (v_{in}, v_{out}) \in E'$
With this setup, the problem becomes: given a transformed directed graph G' = (V', E') , a set of populated vertices X', and a set of safe vertices S', X' and S' are disjoint sets. then we can use the solution to (a) above to solve problem (b): we add a new vertex "super_source", and a new vertex "super_sink", connect the super_source to each vertex in X' with capacity 1, connect each vertex in S' to the super_sink with capacity 1, then assign a capacity of 1 to each edge in the original graph G', this ensures that paths do not share any edges (as each edge can carry at most one unit of flow) and do not share any vertex (as each vertex splits into two vertices with an edge added that isn't shared).



Pseudocode

```
findEvacuationRoutes_Question_6b(G, X, S)
#1      G',X', S' = transformGraph(G, X, S)  // runtime: O(|V|+|E|)
#2      Return findEvacuationRoutes_Question_6a(G', X', S')

FUNCTION transformGraph(G):
#3      E' ← empty set  // edges
#4      V' ← empty set  // vertices
#5      X' ← empty set  // populated vertices
$6      S' ← empty set  //safe vertices
#7      FOR EACH vertex v IN G: // split vertices
#8              v_in ← new Vertex(v.name + "_in")
#9              v_out ← new Vertex(v.name + "_out")
#10             add vertex v_in to V'
#11             add vertex v_out to V'
#12             add edge (v_in, v_ou) to E'
#13             If v in X:
#14                     add v_in to X'
#15             else if v in S:
#16                     add v_out to S'
#17     FOR EACH edge (u, v) IN G.edges: // keep original edges
#18             add edge (u_out, v_in) to E'
#19     G'←(E', V') // construct the transformed graph
#20     RETURN G',X', S'
```

Runtime:

line#3~6: initialization: O(1)
line#7~16: Vertex Splitting: O(|v|)
line#17~18: Edge Addition: O(|E|)
line#19: initialization: O(1)
line#20 Return: O(1)
Total: O(|V|+|E|), hence our findEvacuationRoutes_Question_**6b still** has polynomial runtime as findEvacuationRoutes_Question_**6a** does.

Example to show the answer would be "no" for part (b) but "yes" for part (a):
Consider the following graph with populated vertices $X = \{1, 2\}$ and safe vertices $S = \{5, 6\}$:
$1 \ -> \ 3 \ -> \ 5; \quad 2 \ -> \ 4 \ -> \ 6$
In this example, there are clear paths for 1 to 5 and 2 to 6, satisfying both conditions (a) and (b). However, if there is an overlap such as:
$1 \ -> \ 3 \ -> \ 5; \quad 2 \ -> \ 3 \ -> \ 6$
For part (a), this is fine. For part (b), this would violate the vertex-disjoint condition because the vertex 3 is shared, and the answer would be "no" for part (b) but "yes" for part (a).