Written Assignment 3

CS 362

April 12, 2022

Damian Franco

Meiling Traeger

1 Problem 1

***Q*uestion:**

Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible base stations. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter r—a client can only be connected to a base station that is within distance r. There is also a load parameter L—no more than L clients can be connected to any single base station.

Our goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

***S*olution(s):**

This question asks us to create and design an algorithm that will solve the mobile computing clients problem. First, lets lay out the landscape of this problem. Clients, referred to as C, in a town need to be connected to a base station which I will refer to as B. There will be a total of n clients and k base stations. Within a graph both of these base stations will be identified as nodes. To account for the various different types of nodes within the graph, the clients will be labeled from $C_1 ... C_n$ and the base stations will be labeled from $B_1 ... B_k$. All nodes will also be associated with coordinates $(x,\ y)$ to indicate the location of the node. Edges will appear between every client C_n within the range r of a base station B_k and will be indicated with $(v_i,\ w_k)$. Each base station has a maximum connection to clients which is referred to as the load capacity, L.

Now that the overview of the problem is expanded on in the first paragraph, lets get into the algorithm details using what we know. This problem could be solved by taking a network flow approach. The overall network/graph will be represented by vertices (we refer to them as nodes) and edges $G = [V,\ E]$. The input within the algorithm will consist of the coordinates and amount $(n\ \&\ k)$ of both the C and B nodes, the load parameter L, and the range value r. It is known that there must be a connection/edge that is between the clients and the base station so let's say for every edge $(v_i,\ w_k)$ where v_i is C_i and w_k is B_k with the capacity of 1. We chose the capacity to be 1 because each individual client must be accounted for in the simplest integer per client.

To make this a true network flow application algorithm let us now create a new "super-source" s to represent the foreground, and a new "super-sink" t to represent the background. The "super-source" s will be connected to every client C_n node with an edge with the capacity of 1. The capacity here will be 1 due to the same capacity values on the edges between the C and $_{\text{B}}$ nodes.

Now edges must be connected between the B_k nodes and the "super-sink" t. These nodes will have an edge capacity of L because it represents the maximum amount of connections between that base station B_k and their clients. A sketch of the paths that could be within the graph will be shown below:

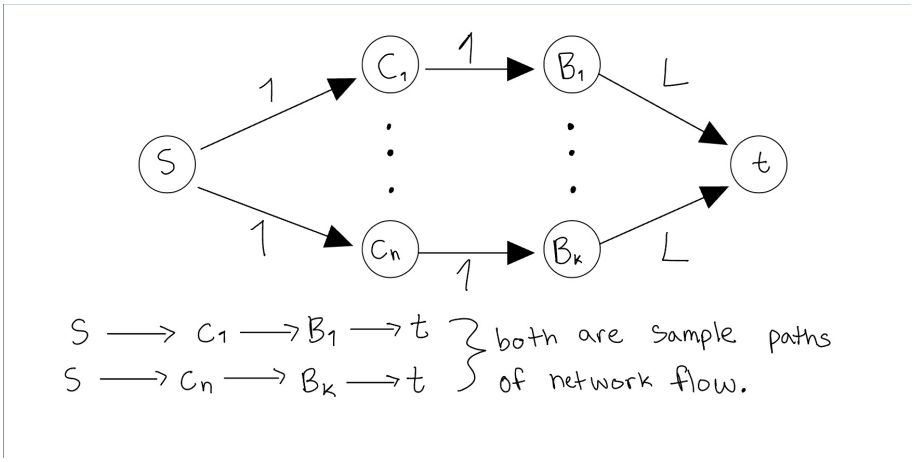

Figure 1: Shows an example of the network. This is merely an example of how the paths are generated and to show the capacities of each edge. The edges between the C and B nodes here are just place holders and are not exact to how the actual graph would be.

The algorithm will begin with the s node and continue to move throughout the graph to each C_i node, sending 1 flow unit through. Next, the one flow unit would be sent from the C_i to a base station B_k within the range r and if the base station has enough flow (maximum L) to add another client. To ensure that there is availability to connect a client to a base station, the algorithm must check the current capacity of the edge between B_k and t. The final step of the traversal through the network would be to send another unit of flow to the t node to finish the path if there is proper capacity left. The overall path that the algorithm will take is the following: s –> C_i –> B_k –> t with a maximum of 1 flow unit sent through the path. The algorithm will then repeat this process of flow sending for all paths within the network ($C_1 ... C_n$ & $B_1 ... B_k$). Since we know that the main point of this algorithm is to find the adequate number of clients that can be simultaneously connected to a base station, this approach would only be correct if the number of clients nodes n is equal to the maximum flow result that is produced from the first part of the algorithm, otherwise there is no correct way to connect the clients properly. Additionally, we have classified this as a maximum flow problem based off the result we are trying to compute. This algorithm should be able to decide whether every client can be connected simultaneously to a base station.

Run time of this algorithm would highly depend on the number of clients n and the number of base stations k. Since we know that this problem is also a maximum flow problem, then the overall run time of this algorithm should be *O(n*k)*.

Algorithm 1 Pseudocode for the network flow algorithm that decide whether every client can be connected simultaneously to a base station.

```
n ← Number of client nodes within the network
k ← Number of base station nodes nodes within the network
C ← Set of all client nodes within the network
B ← Set of all base station nodes within the network
maxFlow = 0 ← Holds the final max flow
for i = 0 till i <= n do
    Travel from t to C_i and send one unit of flow
    Travel from C_i to B_i within range r
    if Capacity from B_i to t > 0 then
        Send one unit of flow and mark path as traveled
        maxFlow++ ← Increment max capacity by 1
        i++ ← Keep incriminating through the network
        Update the residual edge capacity values
    end if
end for
if maxFlow == n then
    return true ← every client CAN be connected
else
    return false ← every client CANNOT be connected
end if
```

2 Problem 2

Question:

Statistically, the arrival of spring typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain antigens present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four types: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.

(a) Let sO, sA, sB, and sAB denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type dO, dA, dB, and dAB for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. The hospital wants to know if the blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type	**supply**	**demand**
O	50	45
A	36	42
B	11	8
AB	8	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. Also, provide an explanation for

this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words flow, cut, or graph in the sense we use them in this book.)

Solution(s):

Part (a):

In this problem we are looking at various blood types. Let us first expand on the overall parameters of the problem at hand. We know that there are four blood types: A, B, O, and AB. Each blood type has strict rules for which blood type it can accept or donate to. These specific rules are based on the antigens that we have in our blood. A person cannot receive blood with a particular antigen if their own blood does not have this antigen present. In general, O is the universal donor. The hospital that we are producing this algorithm for has certain demands that need to be met. Their supply will dictate if the demands are met or not.

To start the algorithm, we are given the supply and demand for each blood type that the hospital has in inventory. We also know that s_O, s_A, s_B, and s_{AB} represent the supply of blood and d_O, d_A, d_B, and d_{AB} represent the demand of each of the blood types. Each supply of the various blood types will be represented with nodes $S_1 ... S_4$ and each demand will be represented with nodes $D_1 ... D_4$. The supply nodes and demand nodes will have an edge between certain nodes that will demonstrate the compatibility of the blood types. For example, a blood type of O can donate to all of the other blood types so if S_1 represents the O blood type supply then there will be an edge between S_1 and all D nodes.

Let us now add a "super source" node s and a "super sink" node t to make this a network flow algorithm. Edges will be generated between s and all the supply nodes S and t. Also all S nodes will also have an edge between itself and the demand nodes D. Capacity on the edge (s, S_i) between the "super source" node and the supply nodes will be assigned the value of each supply S_i. Same goes for the capacity of the edges between the demand nodes (D_i, t) and the "super sink" t. The capacity of the edge that connects each supply and demand node (S_i, D_i) will have the capacity assigned to the value of the connected demand node of D_i. In figure 2, a diagram that illustrates the network of each blood type's supply and demand.

The algorithm will start by first traversing to each supply node from the "super source" node and sending Si units of flow to the supply nodes. No capacity or supply changes will be made for the available supply until a path is completed. After sending S_i units of flow to the supply nodes, the algorithm will then start sending that sufficient amount of supply to the demand nodes starting with blood type O or S_1. We start with the supply node of O (S_1) since it is the universal donor and finish with the AB supply (S_4) because it is more exclusive. We will then send the correct demand amount/flow to each demand node starting with the demand node of the same blood type. Whenever the

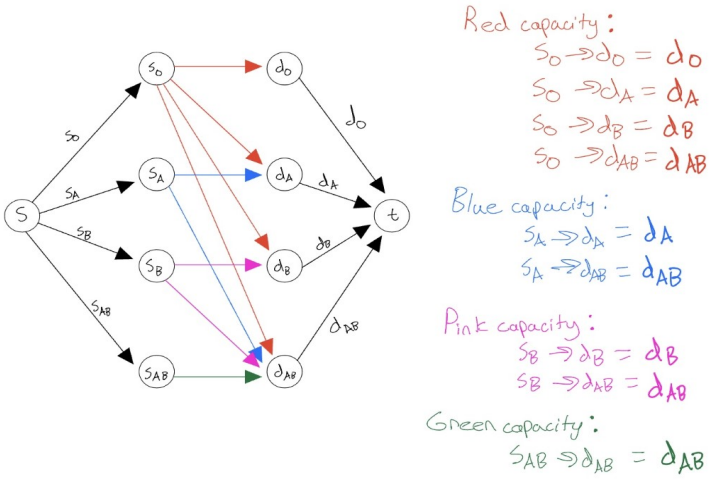

Figure 2: Displays the overall network for the blood supply demand problem. There is also a display of what all the capacity values for all the edges.

supply is sufficient to provide that amount of demand, then there will be D_i units of flow sent to the demand node. If there is not enough supply to meet a demand, then the algorithm will supply the maximum value it can to help ease the demand for later iterations of the algorithm and iterate to the next blood supply. If there is no supply remaining, then the algorithm will either stop or iterate to the next blood type supply ($S_1 \dots S_4$). A good example of how we imagine this algorithm will iterate through blood types is the top-down process ($Red \to Blue \to Pink \to Green$) of the previous figure. This process will repeat for all blood types and stop after S_4. At the end of each iteration, the demand nodes D_i will send the flow amount to the "super sink" which was determined by the value of demand that they were able to generate. If the maximum amount of capacity for each edge between the demand nodes and the "super sink" (D_i, t) was reached, then that indicates that there was a sufficient supply for the demands for that week. If there was still residual demand capacity left over by the end of the algorithm, then there is a failure to meet the demands that week.

Pseudocode for the algorithm is shown on the next page.

Run time for this algorithm will be determined by the amount of supply that we have as input. Adding up all the numerical values of all blood types supplies will give us a run time of $O(log(S_1 + S_2 + S_3 + S_4))$. We think it would most likely be a log function run time due to the fact that this is another max-flow algorithm implementation.

Algorithm 2 Pseudocode for the network flow algorithm that checks if we can meet the blood type demand with the given supply for the hospital that week.

```
S ← Set of supply nodes (S_1...S_4) within the network
D ← Set of demand nodes (D_1...D_4) within the network
totalDemand = D_1 + D_2 + D_3 + D_4 ← Holds the total demand needed
maxFlow = 0 ← Holds the final max capacity
for i = 1 till i > 4 do
    Send S_i units of flow from "super sink" s
    Current node is set to S_i
    for j = 1 till j > 4 do
        if S_i >= D_j then
            Send D_i units of flow to D_j
            Decrease the value of S_i by the amount of flow sent
            maxFlow = maxFlow + S_i ← Update max flow
            Update the residual edge capacity values
        end if
        j++ ← Increment through the demand nodes
    end for
    i++ ← Increment through the supply nodes
end for
if maxFlow == totalDemand then
    return true ← Demand is fulfilled for this week
else
    return false ← Demand is NOT met for this week
end if
```

Part (b):

Using the argument of a minimum-capacity cut, not all of the patients would be able to receive blood. To visualize we would use our graph of the supply and demand nodes. O and A are the only ones that can supply the demand for O and A. The demand for O and A is 87. The supply for O and A is 86. From this cut we are left with 1, that would not fulfill the supply of B and AB. To the clinic administrators, we would say that the demand is greater than the supply for O and A blood, therefore, the demand cannot be satisfied. Figure 3 shows the exact way we were able to determine that there will not be a sufficient amount of supply for the demand in this case.

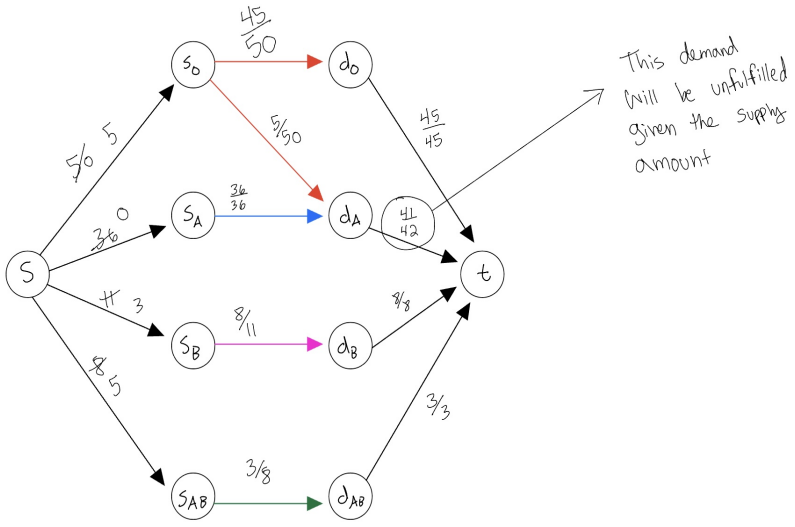

Figure 3: Displays the overall network for the blood supply demand, but with the supply and demand values that were given in the table.

3 Problem 3

***Q*uestion:**

Consider the following problem. You are given a flow network with unit capacity edges: It consists of a directed graph G = (V, E), a source s in V, and a sink t in V; and ce = 1 for every e in E. You are also given a parameter k.

The goal is to delete k edges so as to reduce the maximum s-t flow in G by as much as possible. In other words, you should find a set of edges F is a subset of or equal to E so that |F| = k and the maximum s-t flow in G' = (V, E-F) is as small as possible subject to this.

Give a polynomial-time algorithm to solve this problem.

***S*olution(s):**

This problem asks us to create an algorithm that will be able to delete k edges from the given graph $G = (V, E)$. Deletion of these edges should reduce the maximum s-t flow for the graph by as much as possible. The tip that the problem gives us to the find a set of edges F, which is a subset of E, that the set size is equal to k and the maximum flow is the smallest possible after removing the subset F from the overall graph G

We will approach this problem as a optimization problem and relate this to the Max-Flow Min-cut theorem. In every flow network, there is a flow f and a cut (A, B) so that value of the flow $v(f)$ is equal to the capacity of the cut $c(A, B)$. The Max-Flow Min-cut theorem states that in every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut. Since we know that all edge capacities are equal to 1 then we can view this as the natural special case of the Max-Flow Min-Cut Theorem. We will use both of these theorems to help us solve this problem.

Each edge has the capacity of 1, so the size of F should be equal to the capacity for the s-t cut (A, B) which is generated by the Max-Flow Min-cut Theorem. This indication of the capacity of each edge also allows us to make several different distinctions for this algorithm. In specific, we can make the assumption that the overall flow of the graph f after k cut(s) were made will be equal to f-k due to the fact that all edge weights are 1. With that, this also indicates that the maximum flow f will only be able to be reduced by the amount of edges that are removed. For example, if we remove 5 edges from the graph, then the maximum flow will be only able to be reduced by 5. All of this sets a baseline to the rules that we must follow and how we will be able to right this algorithm.

This algorithm will begin by taking in the graph input $G = (V, E)$, source node s, sink node t, and the parameter k that will indicate the number of edges we will attempt to remove. All edges within the graph will have a capacity of 1. Before traversing through the graph, a single min-cut will be made. Edges that are removed will not be specifically removed, but will be tracked by the number of edges that are connected to the min-cut. If there are k number of edges that were removed for the min-cut, then the algorithm will stop and the maximum flow of the graph will be reduced by the value of k. Since we know

that the maximum reduction of the maximum s-t flow would be k, then we also now that this is a good point to stop.

Run time for this algorithm will be *O(f*k)* according to the steps we took in this algorithm. Since we are finding the min-cut of the graph, we would have to take the maximum flow value f and multiply it by the number of edges that we are cutting k.

Algorithm 3 Pseudocode for reducing the maximum s-t flow within a given graph by utilizing the Max-Flow Min-cut theorem.

```
k ← Number of edges that need to be removed
i = 0 ← Accumulator to keep track of how many edges have been removed
for random edge e in all edges E do
    if i >= k && Max-Flow Min-cut theorem is satisfied then
        Remove current edge e
    else if i <= k && Max-Flow Min-cut theorem is satisfied then
        Break out of the for loop
    else
        Do NOT remove current edge e
        Continue to the next edge and loop
    end if
    i++ ← Increment the accumulator
end for
```

4 Problem 4

***Q*uestion:**

We define the Escape Problem as follows. We are given a directed graph G = (V, E) (picture a network of roads). A certain collection of nodes X is a proper subset of V are designated as populated nodes, and a certain other collection S is a proper subset of V are designated as safe nodes. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

(a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any nodes."

With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

Also, provide an example with the same G, X, and S, in which the answer is yes to the question in (a) but no to the question in (b).

***S*olution(s):**

Part (a):

This question asks us to decide whether a set of evacuation routes exists given the graph $G = (V, E)$, the subset of populated nodes X and the subset of safe nodes V. The goal of this problem is to show that there are evacuations routes. An evacuation route will consist of a node starting in subset X and ending in subset S. There must be no overlapping routes and all routes should be within the overall graph G.

With that, we can decide on where a set of evacuations routes exist within the the graph G by making this a network flow problem. First, a "super source" s will be connected with edges to all nodes within the subset X and a "super sink" will be connect with edges to all nodes within the subset S. All edges leading out of both s will have a capacity of 1. In fact, all edges within the graph, except the ones leading from any node within S and t, will have a capacity of 1. We decided to doe this to satisfy the no overlapping routes rule. Once a route is taken, then there will be no more capacity to send flow units through, so there will also be no overlapping routes. Edges will also be placed between nodes within X either to nodes outside of either subset X and S, directly to a node within subset X, or directly to a node within subset S. All nodes could

be connected, but all that matters is the path finding now. We can use an algorithm like Ford-Fulkerson, which is a polynomial time algorithm, to find all possible paths within the graph. Ford-Fulkerson will successfully increment through each node within the graph and send capacity through a path which could be thought of as our evacuation path. The paths will most likely look like the diagram within Figure 4.

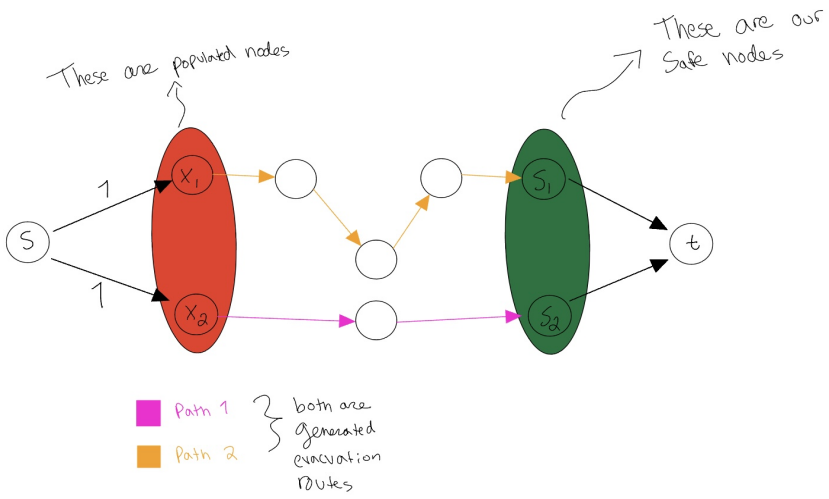

Figure 4: Small example of some paths that could be generated from this approach. All capacities for all edges will be equal to 1, except t which will equal to the value of the number of nodes within the given X subset.

The last point that we would like to make about the algorithm is about the maximum capacity. Capacity of the edges made between the nodes within subset S and the "super sink" will not be 1. This is due to the reason that we are finding multiple paths and computing the maximum flow of the graph. The maximum flow value will help the algorithm decide on whether there is an available escape route(s) within the given graph. Therefore, we assigned the capacity of the "super sink" t equal to the value of the number of nodes within the given X subset. For example, if we get a subset X with 5 edges within it, then we will have a capacity of 5 on each edge between S and t.

This approach should be able to determine the amount of escape routes that the civilians in the population have to the evacuation places.

Part (b):

Part (b) asks us to make the approach we took in part (a) more enforced with no congestion and no node sharing between evacuation routes/paths. The approach to this algorithm will not change much. In fact, everything should remain the same as part (a), besides one rule. No congestion for overlapping edges is already handled in part (a), but not for shared nodes between paths. We first wrote down an example that has a shared node between two paths to help us visualize how we are going to enforce this approach, which can be seen in Figure 5.

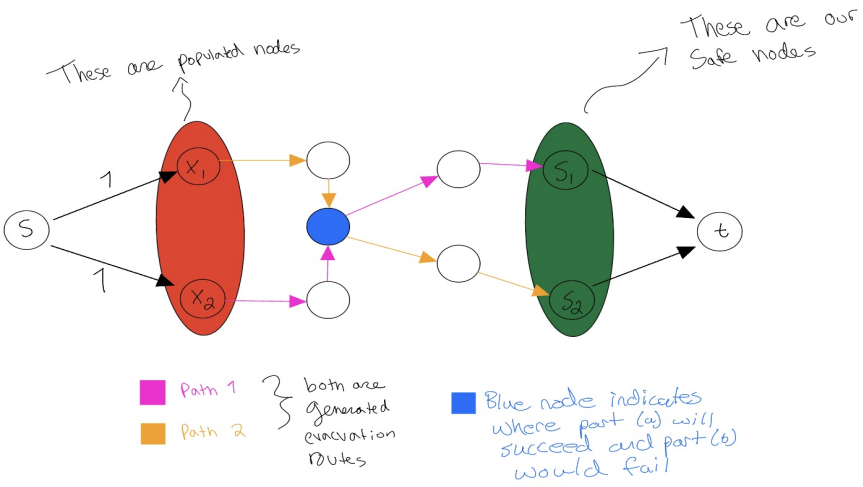

Figure 5: This graph indicates where how the requirements given in part (b) could fail for certain graphs. The blue node would be the main cause to the problem because it is a shared node between the pink and orange path.

We noticed that if we split up the node that has two incoming edges into two separate nodes, then the issue would be resolved and the pink and orange path within the diagram will be able to be completely separated. We then had another problem arise from this when we asked: how will the new two nodes decide on what outgoing edges will be generated? A split node will still have the outgoing edges to all the nodes the singular node had. If the node our current (split) node is pointing towards contains no other incoming edges, then add a outgoing edge to the node from the current node. If the node already has an incoming edge, then do not add a outgoing edge. This seems to fix all the overlapping problems and enforce the rules much stronger. Figure 6 (on the next page) shows a small example of what we mean by splitting up nodes with two incoming edges.

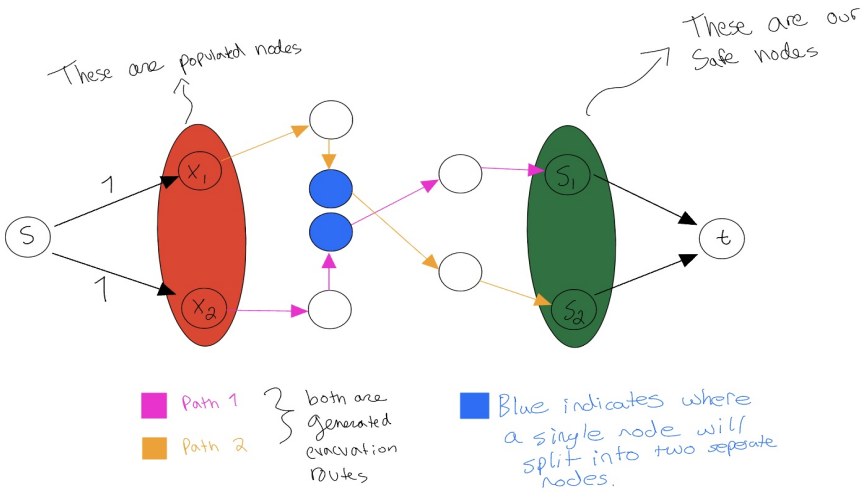

Figure 6: Example of how the use of splitting a node with two incoming edges will result in non-shared nodes between paths. This graph is adapted from the graph in Figure 5 where the requirements in part(b) would not be fulfilled, but here it is satisfied.