Evan Edelstein
EN.605.621.84.FA25
HW 5
The work in this exercise is mine alone without un-cited help. No AI was used to answer these questions.

1) A tree is a graph with the property that there is only one edge to every vertex. A BFS tree is the tree that results from performing BFS on a graph and has the property that the distance from the root to any vertex is equal to the depth of the vertex. Similarly, a DFS tree is the tree that results from performing DFS on a graph and has the property that it traverses down a path until it hits a leaf and then backtracks up. We define T to be the DFS tree on G rooted at u, and R to be a BFS tree on G rooted at u. Let T be isomorphic to R.

   We would like to prove that G=T, or equivalently that G is a tree. We can do this by showing that there are no edges in G that are missing from T and R. We can prove this by contradiction. Assume there exists some edge *e* in G that was not in T and R. We can break this up into two possible cases, where e is between two vertices x and y at the same depth and at adjacent depths.

   In the first case, when e is an edge between two nodes of the same level, the DFS tree will traverse this edge since it will follow the path into x and then from x to y, before it backtracks up and traverses down the path that originally led to y. The BFS tree would not have this edge, since it would violate the depth = distance invariant for y. In case two when the depth of x and y differ by one, BFS will traverse e while DFS would not. BFS would traverse down (x,y) since that path represents the depth of y, which is shorter from x than before e existed. DFS would not include this path since it would traverse the original path to y before backtracking up to x and then skipping the path directly to y. In both cases we contradict the isomorphic property between T and R, and this show that there are no edges in G that are not in T and R, thus G=T.

2)
   a. We can describe a bipartite graph *G* that maps each person p_i to a dinner to cook d_j. A bipartite graph is a set of nodes split into two groups, with edges only between the groups. We can construct a bipartite graph from the set of non-available nights S_j for each cook d_j, by connecting an edge from all cooks to all nights and then removing the ones in S_j. A mapping is a set of edges that connect nodes between the two groups in the bipartite graph. A

perfect matching bipartite graph is a stricter mapping that matches the two groups in a 1:1 manner, with no unmatched vertices. That is, each vertex in group 1 is uniquely mapped to a single vertex in group 2, and all vertices are mapped. If and only if the bipartite graph of cooks to days is perfect matching, do we have a valid cooking schedule, since a valid cooking schedule requires a complete and 1:1 mapping of cooks to dinners. That is all dinner are covered by a cook, and each cook only cooks one dinner. If it was not perfectly matching than there is no guarantee that each cook is assigned to one and only one dinner, instead cooks can be assigned to no dinner or more than one dinner, and dinner can be left unassigned or overbooked.

b. We will use the mechanics of flow graphs to help construct a perfect matching bipartite graph *G* that can be a valid cooking schedule. We note that by finding the maximum flow through the graph described below we can construct the perfect matching graph and valid cooking schedule. We start by creating a super-source and super-sink vertex and creating an edge from the source to each person with a flow of 0 and capacity of 1, and from each day to the sink with a flow of 0 and capacity of 1. We can then add the properly scheduled connections to the graph by creating an edge from the person to their day to cook with a flow of 1 and capacity of 1, to represent an already matched day. We then add the two mismatched edges, $p_i$and p_j to the graph and night d_{n-1} and d_n. This setup is depicted in figure 1. We can then use Ford-Fulkerson method (with BFS) to find the augmenting paths and the maximum flow through the graph. An example of the augmenting path is shown if figure2. If no augmenting paths can be found, then there is no feasible dinner schedule. If it can be found, we can use the augmenting path to match the remaining cooks by flipping the flow along the augmenting path. This results in a matching with one more edge. To find a perfect matching we can run Ford-Fulkerson with $(p_{i,}d_{n-1})$ at a capacity of 1 and p_j with zero capacity edges to d_n and d_{n-1} and then again with swapping p_j for p_i. Using

BFS we can find an augmenting path in O(E+V) = O(n^2 + n) = O(n^2). We know $E = n^2$ since there are n cooks and n nights.

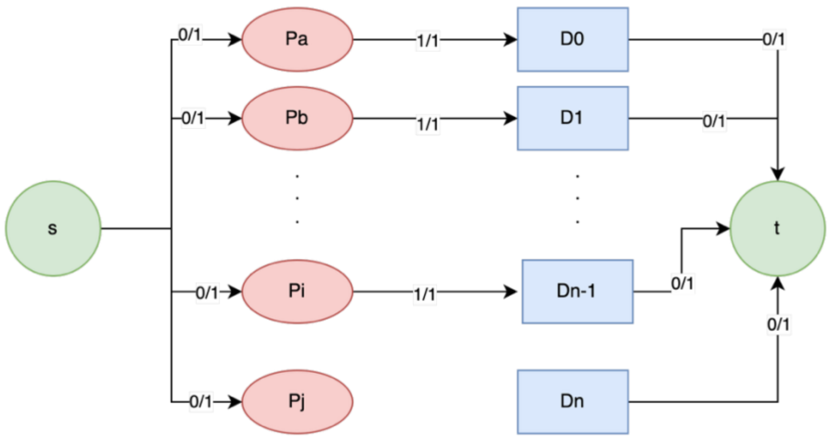

Figure 1: Construction of a network flow diagram to find the perfect matching bipartite graph G.

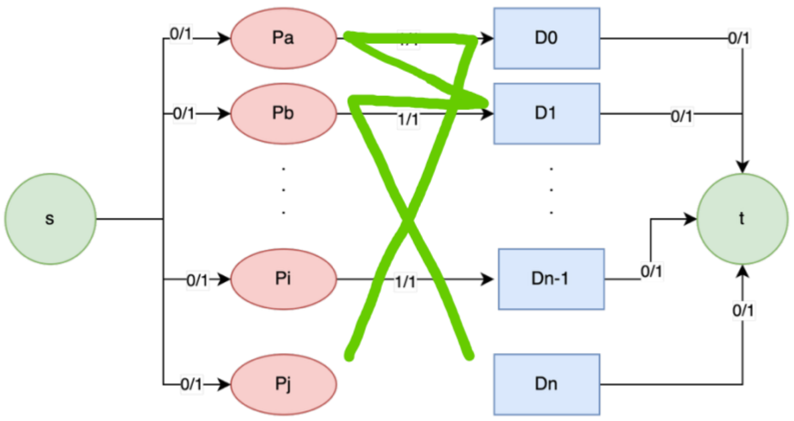

Figure 2: Example Augmenting path shown in green

3)

a. We can construct a network flow diagram to solve this problem. We create a flow graph *F* with a super-source, super-sink and all vertices and edges in *G*. We assign each edge in *E* with a capacity of 1. For each populated city *c* in *X*, we create an edge (source, *c*) with a capacity of 1. We also create an infinite capacity edge (sink, *y*) for all safe vertices *y* in S. A set of evacuation routes exists if the maximum flow through *F* equals the number of vertices in *X* (populated cities). Since this ensures there is a path from each populous city to a safe city. We can use the Edmonds–Karp algorithm to compute the maximum flow in the graph F in time $O(V * E^2)$. Figure 1 depicts this construction. The source and sink placement and capacities ensure conditions i and ii. Because each internal edge has a capacity of 1 it cannot be used by multiple paths and since augmenting paths cannot exceed the capacity of an edge, we satisfy condition iii.

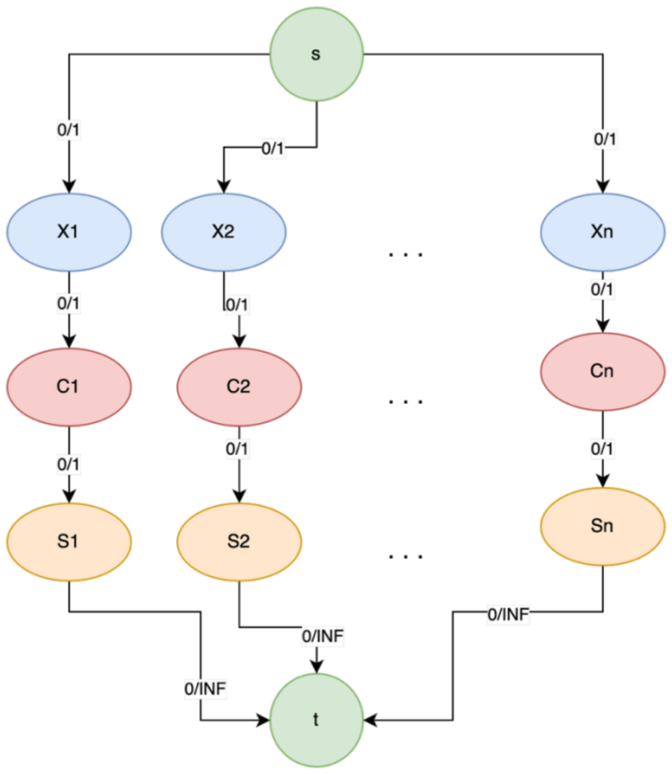

Figure 1: Flow network graph setup to find evacuation routes from populous cities X to safe cities S, through regular cities C.

b. We would like to modify F from (a) such that each city has only a capacity of 1, so that no path can be uses by two cities. To do this we can split each internal vertex *c* into two, c_a and c_b. We connect any incoming edges to c to c_a and any outgoing edges from c to start from c_b. This ensures the correct connection of all the cities. We assign an edge between them with capacity 1 to ensure that *c* is only used once. We create a super-source and attach it to each populous city in X, with a capacity of 1. Similarly, we create a super-sink and attach each safe city in S to it with a capacity of 1. These two setups ensure that each city in X and S is used only once. Figure 2 depicts the construction of F. Like above, if the max flow is equal to the number of cities in X, then we can find valid escape routes that satisfies the three conditions.

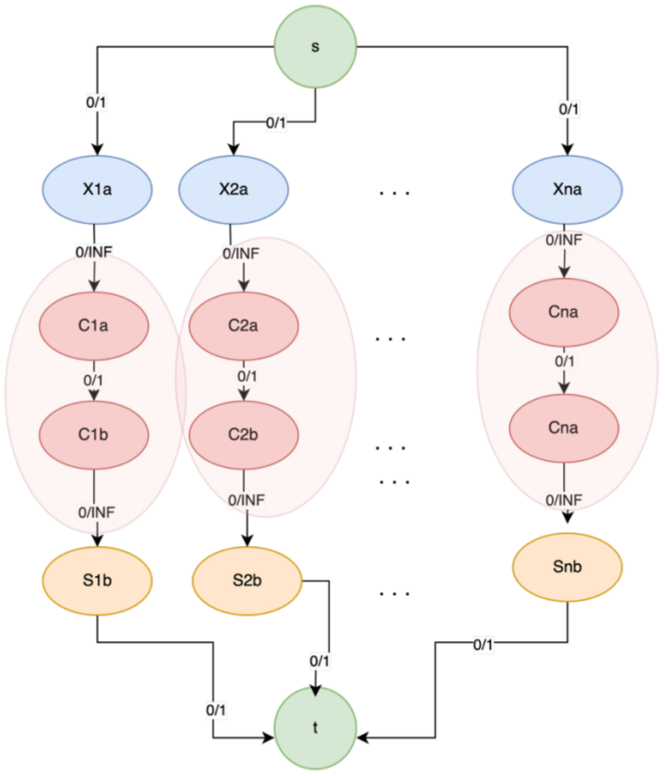

Figure 2: Flow network graph setup to find evacuation routes from populous cities X to safe cities S through regular cities Cm without overlapping vertices.

An example where a graph satisfies (a) but fails (b) is shown below. The three augmenting paths (evacuation routes) through this graph would be: [(X1, C1, S1), (X2, C1, S2)]. Since no edges are shared in the three paths, they satisfy the requirements of part (a). But, since the vertex C1 is used multiple times, it does not satisfy the requirements of part (b).

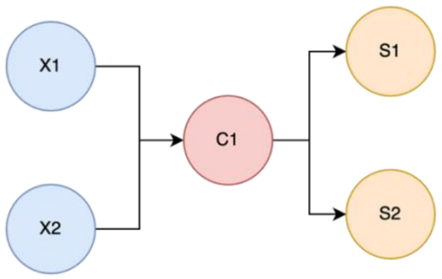

Figure 3: Example graph that satisfies edge constraints but not vertex constraints.

Citations:

[1] Cormen, Thomas H., et al. Introduction to Algorithms, Fourth Edition, MIT Press, 2022. ProQuest Ebook Central, https://ebookcentral-proquest-com.proxy1.library.jhu.edu/lib/jhu/detail.action?docID=6925615.

[2] https://en.wikipedia.org/wiki/Maximum_flow_problem#Maximum_flow_with_vertex_capacities

[3] https://www.drawio.com/