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. Let *T* be a tree produced by DFS at the root of a graph *G* and *R* be an isomorphic tree *T* produced by BFS on the root of the graph *G*. We want to prove that , that is *G* cannot contain any edges that are not also in *T*. We can prove this by contradiction. Since the tree produced by DFS and BFS are isomorphic to each other we know that each vertex has the same predecessor/successor relationship in both trees (i.e. edge order is preserved). Let’s assume there exists an edge *(v1,v2)* that is in *G* but not *T*. This could only happen if the edge *(v1,v2)* was missed by DFS, which could happen if *(v2,v1)* was seen first and added to *T* instead (or if *G* is disconnected, which by the problem statement it is not). But by the invariant of isomorphism the ordering of the edges in *T* must be equal for DFS and BFS, so no reverse edge should be in *T*. Thus, having an edge in *G* and not *T* is a contradiction.
   1. We can describe a bipartite graph *G* that maps each person to a night to cook . If the mapping is a perfect matching, then we know each night has a person to cook dinner and no nights are without a cook. This is because we have an equal number of cooks to nights, so a perfect matching would cover all nights with a cook. If it was not perfectly matching than there could be a night that does not have an assigned cook. Figure 1 visualizes the mapping graph *G*.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 1:** An example perfect matching bipartite graph *G* that maps each person to a night to cook . [made with draw.io]

* 1. 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 start and sink vertex and creating an edge from the start 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 to the graph with flow of 0 and capacity of 1.

We can then use BFS to find the augmenting paths and the maximum flow through the graph. 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. Using BFS we can find an augmenting path in O(E) = O(n\*n) = O(n^2) time since at most we search through every edge to find an augmenting path. Figure 2 shows the construction of the network flow diagram used to find the perfect matching.

A diagram of different types of numbers

AI-generated content may be incorrect.

**Figure 2:** Construction of a network flow diagram to find the perfect matching bipartite graph *G.*

* 1. We can construct a network flow diagram to solve this problem. We create a flow graph F with a source, sink and all vertices and edges in G with capacity 1. For each populated city c in X we create an edge (source, c) with a capacity of 1. Similarly, all we create an infinite capacity edge (sink, y) for all safe vertices in S. A set of evacuation routes exists if the maximum flow through F equals the number of vertices in X (populated cities). We can use the Ford-Fulkerson algorithm to compute the maximum flow in the graph F in time . Figure 1 depicts this construction.

A group of ovals with white text

AI-generated content may be incorrect.

**Figure 1:** Flow network graph setup to find evacuation routes from populous cities X to safe cities S. Non X and S cities are not shown.

* 1. We would like to modify F from (a) such that each city has only a capacity of 1, so that no path can use two cities. To do this we can split each vertex into 2 *a* and *b* and if they are a populous cities connect them with an infinite capacity edge (since it will only be used one anyways) or if they are a safe city connect them with a capacity of 1. The latter means that only a single augmenting path from populous city to safe city can go through the original vertex. To maintain the original edges from populous to safe cities we replace edges (x,s) with (xb , sa) with a capacity of 1. Like above if the max flow is equal to |X| then we can find valid escape routes.

A screenshot of a cell phone

AI-generated content may be incorrect.

**Figure 2:** Flow network graph setup to find evacuation routes from populous cities X to safe cities S without overlapping vertices. Non X and S cities are not shown.

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, X3), (X3, S1)], [(X2, X3), (X3, S2)], [X3, S3]. Since no edges are shared in the three paths,

they satisfy the requirements of part (a). But, since the vertex X3 is used multiple times, it does not satisfy the requirements of part (b)

A group of ovals with black text

AI-generated content may be incorrect.

Citations:

[1]https://en.wikipedia.org/wiki/Maximum\_flow\_problem#Maximum\_flow\_with\_vertex\_capacities