Homework 7

Solution Key

Problem 1

Given an arbitrary undirected graph G, applying DFS on a given vertex will create a tree. The tree can be used to detect the separating edges and vertices of the graph in linear time.

(a) Show that the root vertex of a DFS tree is a separating vertex of G if and only if the root vertex has multiple children in the DFS tree.

(b) Show that any non-root vertex v of a DFS tree is a separating vertex of G if and only the exists a child of v, w, such that none of w's descendants in the DFS tree have a back-edge to a proper ancestor of v in the DFS tree.

A **descendant** of a vertex is any vertex reachable from v in the DFS tree.

A **proper ancestor** of a vertex v is a vertex v' such that v is a descendant of v' and $v \neq v'$.

A **child** of a vertex v is a direct descendent of v.

Solution:

(a) The root vertex of a DFS tree has no back-edges. If it only has one child then removing the root of the tree keeps the DFS tree connected, so it is not a separating vertex.

If the root vertex of a DFS tree is a separating vertex, then we know that it splits the graph into at least two connected components. For the removal of this vertex to split the DFS tree, it must have degree of at least 2, and since no back-edges of the root vertex can exist, it must have at least 2 children.

(b) Let v be a non-root vertex of a DFS tree that has a child v' where there are no descendants of v' has a back-edge to a proper ancestor of v in the DFS tree. Then we know that the removal of v will separate a proper ancestor of v (which must exist because v is not a root) and v' so it is a separating vertex.

On the other hand, if v is a separating non-root vertex of a DFS tree, we know that there must only be one path to get from a proper ancestor of v to any descendant of v. So there must not be any back-edges from any of the descendants of v (excluding itself) to proper ancestors of v in the DFS tree.

Problem 2

Given a flow network graph G with n nodes and m edges, assume you are given a maximum flow assigment. Propose an algorithm that finds a minimum capacity cut in time $O(m)$.

Algorithm Starting from the source node, run a modified version DFS/BFS marking every node that is seen. Traverse only those edges with positive residual capacity. If traversing an edge in the forward direction then $RC = c(e) - f(e)$. If traversing an edge in the backwards direction then $RC = f(e)$. We stop a DFS/BFS branch upon reaching a node whose forward edges all have residual capacity zero. Let V_s denote the set of all marked vertices. The algorithm returns $(V_s, V \setminus V_s)$.

Correctness To show that the algorithm is correct we note the following fact:

Let $G = (V, E)$ be a network with max flow. Let V_s and V_t be a partition of V such that for every edge e going from V_s to V_t we have that $RC(e) = 0$. Then, (V_s, V_t) is a min-cut of G.

Since (V_s, V_t) is a cut we know that the flow through (V_s, V_t) is equal to the flow through the network, and thus is the max-flow f^* of the network. For every edge e between from V_s to V_t we have that

$$RC(e) = 0 = c(e) - f(e) \implies c(e) = f(e)$$

Let $\mathcal{X}$ denote the set of all edges from V_s to V_t. We have that

$$c(\mathcal{X}) = \sum_{e \in \mathcal{X}} c(e) = \sum_{e \in \mathcal{X}} f(e) = f^*$$

By the min-cut max-flow theorem, since the capacity of $\mathcal{X}$ is equal to the max-flow of the network, $\mathcal{X}$ is a min-cut of the network.

Our proposed algorithm returns two sets V_s and $V \setminus V_s$ such that all forward edges from V_s to $V \setminus V_s$ have residual capacity 0. By the above statement, we have that $(V, V \setminus V_s)$ is a min cut.

Runtime The runtime of our algorithm is given simply by the running time of DFS/BFS since we can create our two output sets by initially starting with two sets, an empty one and the set of a vertices, and moving elements from the latter into the former. This can be done in constant time for a given iteration. The running time of DFS/BFS is $O(n + m)$ where n is the number of nodes and m is the number of edges. Note, however, our since our network is connected that the number of vertices in the graph is $O(m)$. Thus, we can reduce our overall running time to $O(m)$.

Problem 3

The city of Irvine, California, allows for residents to own a maximum of three dogs per household without a breeder's license. Imagine you are running an online pet adoption website for the city for n Irvine residents and m puppies.

Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt puppies that he or she likes and that no resident can adopt more than **three** puppies.

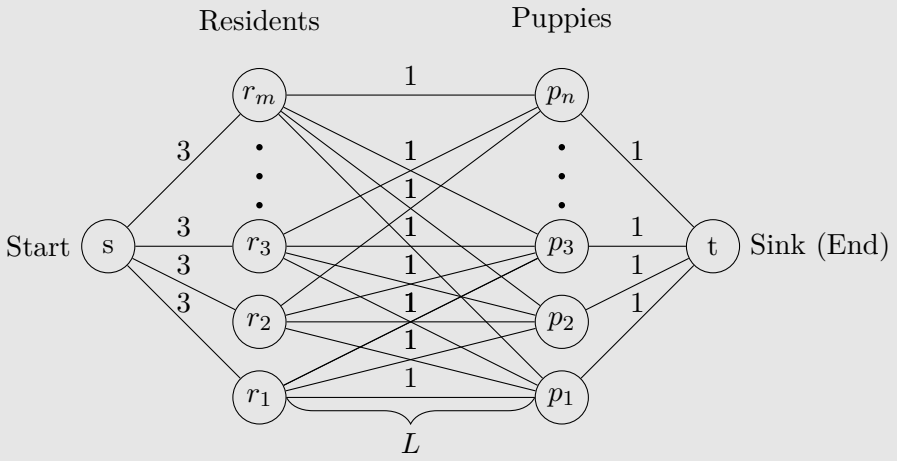
The structure of this algorithm is as follows:

(a) Assume that we are given (along with the residents $r_1, r_2, \ldots, r_n$ and puppies $p_1, p_2, \ldots, p_m$) a list of the puppies that each resident likes, given in the form:

$$L = \{(i, j) : \text{resident } i \text{ likes puppy } j\} \tag{1}$$

So that we have an explicit representation of these relationships.

(b) Construct a graph G with the following structure and edge capacities:



The way this is illustrated, it sort of looks like $m = n$, but the dots are supposed to illustrate that the puppies and residents node columns can (and usually do) have different number of nodes. Also, this appears that L is always complete, but this is not necessarily the case (described more below).

(1) The flow through this graph will represent which puppies get adopted (i.e. if there is flow between r_i and p_j, then that represents r_i adopting puppy p_j.

(2) The edge from the start to each resident r_i has capacity 3 (since each resident can adopt at most 3 puppies).

(3) The edge from each puppy p_j to the sink has capacity 1 (since each puppy can only be adopted once, and a single adoption is one single flow).

(4) We **only** construct an interior edge between r_i and p_j if $(i, j) \in L$ (the graph drawn above represents the worst case, where every resident likes every puppy).

(c) Use the Ford-Fulkerson algorithm on G to find the maximum possible flow network f^*.

(d) From f^* we can construct our solution, where if there is an edge with flow 1 between p_i and r_j, then we say that person p_i has adopted puppy p_j. We can repeat this for all of L to construct a solution list S of adoptions of the form (i, j), and return it.

We prove that this method works by verifying the conditions that we have defined in the problem. In constructing G, we restrict the flow to a maximum of 3 puppies per resident (which aligns with the given condition), and only have an edge with flow (indicating an adoption) between r_i, p_j if r_i likes p_j (from our construction of L) and it is part of the optimal adoption flow.

Further, since have verified that our initial conditions in the construction of G is correct, and we assume (since we proved it in class) that Ford-Fulkerson produces a correct flow (and this flow is what precisely determines the optimal adoptions), we know that this must produce the optimal adoptions for these puppies.

Although the runtime isn't explicitly requested, we know that our initial construction of this graph must have cost $O(mn)$, in the case that L is complete (such that every resident likes every puppy). Further, we recall that Ford-Fulkerson has runtime $O(|f^*|\,(m+n))$, where $|f^*|$ is the flow of the optimal flow map. Therefore, since we know that $|f^*| < 3n$ and $|f^*| < m$, we can say that the overall runtime is $O(\min(m, n)(m+n))$.