Colorado CSCI 5454: Algorithms
Homework 4

Instructor: Bo Waggoner
Due: September 26, 2019 at 11:59pm
Turn in electronically via Gradescope.

Remember to list people you worked with and any outside sources used.

Problem 1 (6 points)

Consider the initial idea we discussed for a max-flow algorithm: Find a path from s to t where the flow $f(u,v)$ is less than the capacity $c(u,v)$ for every edge (u,v) along the path, and increase the flow on these edges. Repeat until there is no such path.

Part a (4 points) Give an example showing how this algorithm can fail to find the max flow. Describe the algorithm's solution as well as the correct max flow solution. (Try to make the example as small and concise as possible.)

Solution. This actually was covered in class September 19 with this example (suggestion: draw it!). There are two "middle" vertices u, v with edges $s \to u \to t$ and $s \to v \to t$, and also an edge $u \to v$. All edges have capacity 1. The algorithm might initially pick the path $s \to u \to v \to t$ and send flow 1 along it. Now it is stuck because there is no path from s to t with $c(x,y) > f(x,y)$ for each edge (x,y). So the algorithm gives flow 1. But the optimal flow is 2, with one unit flowing along $s \to u \to t$ and the other along $s \to v \to t$.

Part b (2 points) Briefly explain why a correct max-flow algorithm can avoid getting stuck on your example in the same way.

Solution. When the above algorithm is stuck, the residual graph does have a path $s \to v \to u \to t$, because $f(s,v) = 0$, and $c_f(v,u) = f(u,v) = 1$, and $f(u,t) = 0$. So a Ford Fulkerson algorithm would augment this path by 1, resulting in the correct optimal flow.

Problem 2 (6 points)

Part a (4 points) A set of paths from s to t are *edge-disjoint* if they have no edges in common. Given an unweighted, directed graph G and vertices s, t, give an algorithm to count the largest possible set of edge-disjoint paths from s to t. Briefly argue correctness and running time. *Hint: use max flow.*

Solution. Create an instance of max flow. The vertices and edges are the same. The capacities on all edges are 1. Now run Edmonds-Karp on this graph and return the amount of the max flow.

Correctness: We first argue that any set of edge-disjoint paths corresponds to some flow.[1] If we have a set of edge-disjoint paths, then we can send one unit of flow along each of these paths. Each edge carries at most one unit of flow, because of disjointness, so this will be a valid flow with amount equal to the number of disjoint paths.

Next we argue that there is a max flow that equals the number of edge-disjoint paths.[2] We use the Integrality Theorem, which says that Edmonds-Karp will assign integer flow to all edges. Since the capacities are all 1, this means every edge has either flow 1 or 0. So we can find a simple path from s to t using only edges with flow 1. If we delete this path from the graph, the flow goes down by one, and we are still left with a valid flow through the graph (all capacity and flow constraints still satisfied). We can repeat until the flow is zero, after which we will have deleted a number of edge-disjoint paths equal to the max flow.

Running time: Constructing the new instance takes linear time in the size of the graph, so the big-O running time is the same as Edmonds-Karp, $O(|V| \cdot |E|^2)$.

Part b (2 points) Now, with the same input, give an algorithm for finding the minimum number of edges that, if removed from the graph, will make t unreachable from s. Briefly explain correctness and running time.

Solution. If we create a copy of the graph and put a capacity of 1 on each edge, then a min cut is exactly the min number of edges we can remove to disconnect s from t. Constructing the graph takes linear time, just as in the previous part, and we argued in class and the notes that a min cut can be found in time $O(|V| \cdot |E|^2)$ using Edmonds-Karp.

Problem 3 (6 points)

Part a (4 points) Give an algorithm to determine if an unweighted, undirected graph $G = (V, E)$ is bipartite. Argue correctness and running time.
Hint 1: Use the graph search techniques we learned in lectures 2 and 3!
Hint 2: You may use the fact that if a graph has a cycle with an odd number of vertices, then it is not bipartite. For a bonus point, prove this fact.

Solution. Breadth-first search works. Here is an example algorithm that is easy to describe and has optimal big-O complexity. We pick any vertex s and breadth-first-search from s, recording the distance of each vertex from s. (We saw how to do this in previous lectures and homeworks.)

[1] If we didn't prove this, then it might be that there was some set of edge-disjoint paths that couldn't be found by the max flow algorithm.

[2] If we didn't prove this, then it might be that there is a max flow that uses fractions to achieve a number higher than the max set of edge-disjoint paths.

Now we create an array, `component`, and iterate through the vertices. If v has even distance from s, we set `component[v] = 0`, otherwise `component[v] = 1`. Finally, we check if there are any edges inside each component, by iterating over each vertex u and each of its edges (u, v) and checking if `component[u] == component[v]`. If we find any edge connecting vertices in the same component, we return "Not bipartite". Otherwise, return "Bipartite".

(If the graph is not connected, this may not reach all the vertices! We need to repeat with all vertices that are still at distance ∞, by picking one of them and repeating BFS. A graph of disconnected components is bipartite if and only if all its components are bipartite, so we just check each one.)

Running time. This is linear, $O(|V| + |E|)$, because we run BFS which is linear time, then we iterate through the vertices placing them in components in constant time, and then we iterate through all edges with a constant-time check.

Correctness. Suppose the algorithm returns "Bipartite". Then it has found a partition of the vertices into two components, with no edges between two vertices in the same component. So it is correct.

Now suppose the algorithm returns "Not bipartite". We must prove there is *no* partition of the graph into U, V with all edges crossing. The algorithm has found two vertices u, v that are either both even distance from s or both odd distance from s, with an edge between them. Following the path from s to u, to v, and back to s therefore gives an odd-length cycle.[3] Since the graph has an odd-length cycle, it cannot be bipartite. So the algorithm is correct.

Bonus. We prove that if G is bipartite, then any cycle in it is even.[4] Suppose G is bipartite. Then every path has vertices in alternating components, e.g. $U, V, U, V, \ldots$. (This includes simple and non-simple paths.) So in any cycle $v_1, v_2, \ldots, v_k, v_1$, it must be that k is an even number (hence an even number of edges in the cycle).

Part b (2 points) Modify your algorithm to return the partition of the vertices, if the graph does turn out to be bipartite.

Solution. See previous algorithm description.

[3] Note this cycle may not be simple (it may have repeated edges), but this is okay because the question didn't require it to be simple. For example, it may be $u \to w \to s \to w \to v \to u$.

[4] This is a proof by *contrapositive*, i.e. we were asked to prove "an odd cycle $\implies$ not bipartite", and it is equivalent to prove "bipartite $\implies$ no odd cycles".