Homework 6

CS 4104 (Spring 2021)

Assigned on April 12, 2021.
Submit PDF solutions on Canvas by the
11:59pm on April 19, 2021.

My team-mate is: Joseph McAlister
Additional notes.

- For some of the problems in this homework, you will reduce the given problem to one of the flow-related problems we solved in class. Make sure you describe the reduction completely. For example, if you reduce a problem to the maximum network flow problem itself, specify clearly how will convert an input to the original problem into a flow network (nodes, source and sink, edges, directions, and edge capacities). Do not forget to the prove the correctness of your solution in both directions, as we did in class (lest you transmogrify Calvin-sized Hobbes into a dinosaur).
- For some of the problems in this homework, you do not have to reduce the problem "all the way" to computing the maximum flow in a network. A reduction to maximum bipartite matching or to minimum cut may suffice. You can then use the polynomial time algorithms we have developed for these problems to solve the original problem. In other words, you do not have to also describe how to reduce maximum bipartite match matching or minimum cut to the maximum flow problem.

Problem 1 (15 points) Suppose you have a potential solution to a maximum-flow problem, i.e., you have a flow network $G = (V, E)$ (with s and t and edge capacities, defined, of course) and an assignment $f : E \to \mathbb{R}^+$ that satisfies the capacity and conservation conditions. Develop an efficient algorithm to determine if f is indeed a maximum flow.

Solution: We know that a given flow f is a maximum on a network G if there is no residual $s - t$ path on G_f. Using this property, we can modify the Ford-Fulkerson algorithm to quickly evaluate if such a path exists and return true or false accordingly:

Algorithm 1: `is_max_flow`(G, f)

Input: G a network flow, f the assigned flow values for each edge in G
Output: True if the flow is a maximum on the network, False otherwise
`// Construct` G_f `from the input flow assignment`
1 **foreach** $e \in E$ **do**
2 | Set $f(e)$ according to the given flow assignment f
3 **end**
4 **if** *there is an* $s - t$ *path in the residual graph* G_f **then**
5 | **return** False
6 **else**
7 | **return** True
8 **end**

Proof of Correctness: Per the Ford-Fulkerson algorithm, a flow f has maximum value if and only if it has no augmenting path. Therefore, the simple check on line 4 is sufficient to determine if f is a maximum flow on G. If, after assigning the flow values from f there remains an augmenting $s-t$ path, then the flow is not a maximum since the Ford-Fulkerson algorithm could continue to iterate, further augmenting the flow. If no such path is found, the f is a maximum.

Proof of Runtime Complexity: Constructing G_f on lines 1-3 takes $O(m)$ time as it iterates over all edges (and corresponding flow values) in the graph. We can assume that the if statement to check for

an $s-t$ path on the residual graph G_f would take $O(m+n)$ time, using a modified BFS that accounts for edge capacities without increasing runtime complexity.

Therefore, the runtime of the presented algorithm is $O(m+n)$.

Problem 2 (25 points) Given a flow network G, suppose f is flow with positive value, i.e., $\nu(f) > 0$. Consider the subgraph G' of G that only consists of edges that carry positive flow, i.e., an edge e is in G' if and only if $f(e) > 0$. Prove that there must be a simple s-t path in G'. *Note:* G' is not the residual graph.

Solution:

Claim: There exists an $s-t$ path on G'.

It is given that the flow on all the edges in G' is positive. By the conservation condition, we know that in order for flow to enter an internal node, it must also exit the node:

$$\sum_{e \text{ into node } v} f(e) = \sum_{e \text{ out of node } v} f(e)$$

The exceptions to this rule are the source and sink nodes s, t for which

$$\sum_{e \text{ out of } s} f(e) = \sum_{e \text{ into } t} f(e)$$

Therefore, since G' consists of strictly positive flow values, there exists equal amounts of flow out of s and into t.

Inductively, we can use this to show that there must be an $s-t$ path on G'.

Base cases: $|V| = 1$, s connects directly to t. Similarly for $|V| = 2$, s connects to some node v which must connect to t by the conservation condition of an $s-t$ flow.

Inductive Hypothesis: For some graph G' with $n = |V|$ nodes and strictly positive flow values, there exists an $s-t$ path.

Inductive Step: For such a graph G' with $n+1$ nodes, the conservation property still holds. That is, if the flow on G' with n nodes was f, then adding another node v (and corresponding edge(s) with positive flow values) would make the flow of G' with $n+1$ nodes: $f + f(e)_{v,in} = f + f(e)_{v,out}$.

Therefore, an $s-t$ path must exist on G'.

Problem 3 (30 points) You are given a flow network $G = (V, E, c, d)$. Here the function $c : E \to \mathbb{Z}^+$ specifies the positive capacity for each edge. In addition, the function $d : V \to \mathbb{Z}$ is a function that specifies the maximum incoming flow for each node, i.e., for every node v, the total value of the flow coming into v cannot be larger than $d(v)$; note that $d(v)$ may be zero for some nodes. Given a source s and a sink t, develop an algorithm to compute the maximum flow in this network. You may assume that $d(s) = 0$ and $d(t)$ is unbounded.

Solution: We begin by transforming the given flow network $G = (V, E, c, d)$ with both edge *and* node capacities into a flow network $G' = (V, E, c)$ with just edge capacities by replacing the capacity for each node v with a dummy edge e connecting to a dummy node v'.

The dummy edge is assigned the capacity of the initial node, $c(e) \leftarrow d(v)$, which acts as a proxy to enforce the node capacity. Additionally, we transfer all the outbound edges from the initial node v onto v'.

From here, we can apply the standard, node-capacity-agnostic, Ford-Fulkerson algorithm on the modifed flow network G'.

Algorithm 2: `max_flow_with_nodecapacity`(G)

Input: $G = (V, E, c, d)$ a flow network with edge and node capacities

`// Build` G' `from a copy of` V `such that the for loop doesn't consider nodes that have already been "fixed"`

1 $V_c \leftarrow$ `copy`(V)
2 **foreach** $v \in V_c$ **do**
3 $\quad E_v \leftarrow$ all outbound edges of v
4 $\quad$ Remove all outbound edges from v
5 $\quad$ Create a new node v' with outbound edges E_v
6 $\quad e = (v, v')$
7 $\quad c(e) \leftarrow d(v)$
$\quad$ `// Manually set the node capacity of the added dummy node` v' `to be something not producible by the function` d `so we can convert back to` G
8 $\quad d(v') \leftarrow 0.5$
9 **end**

`// Return the max flow computed by the Ford-Fulkerson algorithm`

10 **return** `Ford-Fulkerson`(G')

Proof of Correctness: In order to show that the presented algorithm is correct, we must prove that the stated problem can be reduced to the Max Flow problem which the Ford-Fulkerson algorithm solves.

First, we must show that we can convert the given flow network with node capacities G into a flow network G' for which the Ford-Fulkerson algorithm can correctly compute the max flow.

$G \rightarrow G'$: This portion is straightforward, explained the the introductory strategy, and demonstrated in the algorithm.

Next we must show that the converse relationship is true as well:

$G' \rightarrow G$: In order to convert back to G from G' we simply collapse the added dummy nodes and edges back into their original form. Though the solution presented in the algorithm is a "cheeky" workaround, using a fraction instead of an integer to distinguish dummy nodes from natural nodes in G. This value is arbitrary and only used for the sake of completing the reduction. The reverse process from here, though, is simple. For each node with a non-integer capacity v', we perform the following process:

1. Copy all outbound edges from v' to the node connected by an inbound edge on v'. This node must be v.
2. Delete the dummy edge $e = (v, v')$
3. Delete v'. We do not have to modify $d(v)$ since, though d is omitted from the definition of G' for clarity, our algorithm does not modify this value for v, so it is unchanged.

Having shown that the reduction works both ways, we are guaranteed that the algorithm will compute the max flow on G from the correctness of the Ford-Fulkerson algorithm.

Proof of Runtime Complexity: Building the modified network G' on lines 2 - 8 takes $O(n)$ time since we iterate over each node in the input network G and perform a fixed number of constant time operations on that node. We can assume that directed adjacencies are stored in an array which can be easily copied to/from v, v' and that creating a new edge, a new vertex, and assigning them respective capacities are all constant time operations.

We know that the running time of the Ford-Fulkerson algorithm is $O(mn)$ time, so our algorithm is also $O(n) + O(mn) = O(mn)$.

Problem 4 (30 points) Solve exercise 6 in Chapter 7 (pages 416–417) of your textbook.

Let's call a floor plan, together with n light fixture locations and n switch locations, ergonomic if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the

walls), where the ith wall has endpoints $(x_i, y_i), (x'_i, y'_i)$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is visible from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

Solution: For this problem, we attempt to reduce the input into an instance of the Bipartite Graph Matching algorithm.

This can be achieved by constructing a a bipartite graph $G = (V, E)$ where we ensure bipartness by partitioning the nodes in V into two categories: switches S and fixtures F. We can set $V = \{S \cup F\}$ where $s_i \in S$ is the ith switch, and $f_i \in F$ is the corresponding fixture. We can construct the edges by checking if the line formed by a possible edge $e = (s_i, f_i)$ intersects with any of the m line segments representing walls of the floor plan. Note that we do not attempt to form edges between switch, not between fixtures to maintain the bipartite partition between S, F. If e does not intersect any lines, then it is ergonomic, and we can add it to E.

From here, we simply feed G into the Bipartite Graph Matching algorithm, if a perfect matching is found, then the floor plan is said to be ergonomic, otherwise it is not.

Algorithm 3: `is_ergonomic`(G)

```
   Input: S, F the sets of switch and fixture locations, each of length n, and M the set of walls
   Input: True if the floor plan is ergonomic, false otherwise
1  V ← {S ∪ F}
2  E ← {}
   // Attempt to create edges between switches and fixtures
3  foreach s ∈ S do
4  |  foreach f ∈ F do
5  |  |  e ← (s, f)
6  |  |  foreach m ∈ M do
7  |  |  |  if intersects(e, m) then
   |  |  |  |  // If the edge intersects any walls, then skip and try the next possible
   |  |  |  |     edge
8  |  |  |  |  break
9  |  |  |  end
10 |  |  end
11 |  |  Add e to E with a capacity of 1
12 |  end
13 end
14 n ← |S|
15 f ← bipartite_matching(G = (V ∪ {s, t}, E))
   // If the number of edges (or maximum flow on G from the Bipartite Matching
      Algorithm) in this matching ≠ n, then the matching is is not perfect, and the
      floor plan is not ergonomic
16 return f == n
```

Proof of Correctness: In order to show that the presented algorithm is correct, we must prove that the stated problem can be reduced to the Bipartite Matching problem.

Switches and Fixtures → Bipartite Graph: We've shown in the algorithm that the problem input can be constructed into a bipartite graph: edges are only ever formed between switches and fixtures, never switches to other switches nor fixtures to other fixtures. Therefore the input can be reduced to an instance of the Bipartite Matching Problem.

Bipartite Graph → Switches and Fixtures: To convert back to the input from a bipartite graph G, we must show that a graph has a perfect matching between n pairs of switches and fixtures.

Note that this would only be the case if the input represents and ergonomic floor plan, otherwise the bipartite matching algorithm would not find a perfect matching.

In order to convert back to the input, we simply take all the nodes incident on s which are switches, and similarly all the nodes incident on t are fixtures.

Having shown the the reduction is bidirectional, we are guaranteed to find an ergonomic matching if one exists from the correctness of the Bipartite Graph Matching Algorithm.

Proof of Runtime Complexity: The construction of the graph on lines 3 - 13 takes $O(n^2k)$ time, where k is the number of walls. The double nested for-loop over n switches and n fixtures contributes the n^2 term, and the intersection check for a possible edge connecting a switch to a fixture against all k walls contributes the k term.

Lastly, the Bipartite Matching Algorithm which serves as a wrapper for the Ford-Fulkerson algorithm takes $O(mn)$ time. In the worst case (or most ergonomic floor plan possible), every switch can connect to every fixture without intersecting a wall, so there would be $m = n^2$ edges. In this case, the running time of the Ford-Fulkerson algorithm would be $O(n^3) > O(n^2k)$.

Therefore the overall run time complexity of the presented algorithm is $O(n^3)$.