CSE 40113: Algorithms
Homework 7

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

Required Problems

1. Describe an algorithm to determine whether a given flow network contains a *unique* maximum (s, t) flow. Then, give an example of a network that has a unique maximum s to t flow, but does *not* contain a unique minimum (s, t)-cut.

   Solution: For the first part: First, we can prove that the maximum flow is unique if and only if the residual graph has no cycles of length ≥ 3. (Note that there are several ways to prove this, but here is one.)

   Forward direction: Suppose the maximum flow f is unique, but that there is a cycle of length ≥ 3 in the residual graph G_f. Consider the bottleneck edge with weight b on this cycle, and augment (as in the augmenting path algorithm) to make a new flow f': for each edge $e = (u \to v)$ in the cycle:

   - if $u \to v$ is an edge in G we set $f'(u \to v) = f(u \to v) + b$. Note that this edge can still carry at least b more flow without exceeding its capacity, as the weight in the residual graph is set to be $c(e) - f(e)$, and we know that value is $\geq b$.
   - if $v \to u$ is an edge in G (so this is a "backwards" edge of the residual graph), we set $f'(v \to u) = f(v \to u) - b$. Note that the flow along this edge stays ≥ 0, as the edge in the residual graph would have weight $f(e)$, and that must be $\geq b$.
   - all other edges have the same flow

   This flow cannot violate the vertex constraint, as the book proves that augmenting along a path does change the fact that flow in equals flow out along every interval vertex; here, since we're using a cycle, we know that flow in and out of every vertex is unchanged. Since this means the flow out of s is unchanged, we have a valid flow f' with the same value as f, contradicting our assumption, so there cannot be any such cycle in G_f if f is a unique maximum flow.

   Backwards direction: Suppose the residual graph has no cycles, but there are two maximum flows f_1 and f_2. Create a flow $f_1 - f_2$, as in section 10.5. Since both flows have the same value, we know the $f_1 - f_2$ satisfies vertex constraints everywhere, since each of f_1 and f_2 do, and that $f_1 - f_2$ must send 0 flow from s to t since the values cancel. In other words, the result in a circulation. Since $f_1 \neq f_2$, they must differ on at least one edge, so that for some e, $f_1(e) - f_2(e) \neq 0$. Following section 10.5 again, since flow in is equal to flow out, some edge e' incident to e must also have non-zero flow. Continuing to build a path, we must eventually find a cycle in the residual graph which has length ≥ 3, contradicting our assumption.

For our algorithm, the key here is to use the residual graph, and modify a traversal such as DFS to check for cycles of length ≥ 3:

```
UniqueFlow(G):
  Compute a maximum flow f in G
  Build G_f (see path 331 of book)
  if no cycle of length ≥ 3
      return true
  else
      return false
```

The only remaining thing is to specify how to find if the graph has no cycles of length ≥ 3. This can be accomplished by modifying any of the WFS variants - in particular, a variant of the directed DFS on page 231 to either ignore immediate back edges or to then calculate the length of the cycle found via the edges explored will work here.

It takes $O(VE)$ time to compute a maximum flow, then $O(V + E)$ time to create the residual graph, and finally $O(V + E)$ time for the modified DFS to find cycles. The total runtime is then $O(VE)$ total.

■

2. The Department of Commuter Silence is implementing a more flexible curriculum with a complex set of graduation requirements. The department offers n different courses, and there are m different requirements. Each requirement specifies a subset of the n courses and the number of courses that must be taken from that subset. The subsets for different requirements may overlap, but each course can be used to satisfy at most one requirement.

   For example, suppose there are $n = 5$ courses A, B, C, D, E and $m = 2$ graduation requirements:

   - You must take at least 2 courses from the subset $\{A, B, C\}$.
   - You must take at least 2 courses from the subset $\{C, D, E\}$.

   Then a student who has only taken courses B, C, D cannot graduate, but a student who has taken either A, B, C, D or B, C, D, E can graduate.

   Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of m requirements (each specifying a subset of the n courses and the number of courses that must be taken from that subset) and the list of courses the student has taken.

   Solution: We create a flow graph quite similar to our bipartite matching problem:

   - G contains a vertex for each class and each requirement, as well as two extra vertices s and t.
   - For each class the student has taken, so create an edge directed from s to the vertex for that class with capacity 1.
   - For each class satisfying a requirement, add an edge from the class' vertex to the requirement's vertex, with capacity 1.

- For each requirement, add an edge for the requirement's vertex to the sink t with capacity equal to the number of classes needed to fulfill the requirement.

Finally, we run max flow, and answer yes (that the student can graduate) if the flow value is equal to the sum of the number of classes needed across all requirement categories.

There are $m + n + 2$ vertices, and $\leq 2 + m + n + mn$ edges total, so the maximum flow algorithm on this graph (via Orlin) takes $O((m + n)mn)$ time total.

Proof of correctness: This is a reduction, so we need to prove that there is a way to assign classes to allow graduation if and only if the maximum flow is equal to the sum of all classes needed across requirement categories. For simplicity, let $C[i]$ be the number of required classes for category i, and let $T = \sum_{i=1}^{m} C[i]$ be the flow we need. First, note that we cannot have flow larger than T, since there is a cut of this value - all of the edges leading into t.

Suppose there is a way to fulfill all requirements. Then we can build flow paths by sending one unit from s to the vertex of the class begin used, then one unit of flow over to the requirement vertex, and finally 1 unit along from the requirement to t. If we fulfill all T requirements, this is T unit size flow paths. We know each class is only used once, so we will not exceed capacity of the edge from s to the class, and we only use it to fulfill one requirement, so again we don't exceed capacity, and since we choose $R[i]$ classes for requirement category i in order to graduate, this means the edges from requirement to t will all be maximized but will not have above their capacity. Since this flow has the appropriate value, does not violate edge constraints, and sets flow in = flow out for every vertex except s and t, we know this results in a valid flow of size T, which is maximum.

Now suppose there is a flow of value T. This decomposes into unit size flow paths from s to t, which must go from s to a class vertex, then to a requirement vertex, then to t. Use this flow path to "assign" the class on this path to fulfill the requirement. Since it has value T, we know each requirement must have $R[i]$ flow paths coming in, which in turn gives an assignment of $R[i]$ classes as desired. Finally, since the flow paths can only send 1 into each class vertex, no class can be used for two requirements, so this gives a valid way to fulfill requirements.

■

3. Ad-hoc networks are made up of low-powered wireless devices. In principle, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network. These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

   We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other backup device within its communication range. We require each device x to have k potential backup devices, all within distance D of x; we call these k

devices the backup set of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n][1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

Solution: Build a graph as follows:

- Create a two vertices for each device, labeled u_i and v_i, as well as two other vertices s and t.
- For all $i \neq j$, add an edge with capacity 1 from vertex u_i to vertex v_j if $d[i][j] \leq D$.
- For all vertices u_i, add an edge from s to u_i with capacity b.
- For all vertices v_j, add an edge from v_j to t with capacity k.

We then compute the maximum flow in this graph. If the flow has value $< nk$, answer that no good collection exists. If the max flow has value nk, we look at the edges u_i to v_j, and for any such edge with flow value 1 along it, we add device i to device j's backup list. (Since there is a cut of size kn, consisting of all edges from v_j to t, we know there cannot be a larger flow.)

Runtime: Flow takes $O(VE)$ via Orlin, and we have $V = 2 + 2n$ and $E \leq 2n + n^2$, for a total of $O(n^3)$ time.

Proof of correctness: If there is a flow of value nk, we can decompose this into flow paths, and for any edge u_i to v_j, if the flow values is 1 then assign device i to be on device j's backup set. For every i, we know that u_i has at most b flow coming in, so that means there are at most b outgoing edges with flow value 1, and hence device i will not be on too many backup lists. For every value, we also know that the edge v_j to t carries k flow, since that's the only way to equal kn total, so we know exactly k flow is coming into v_j and hence there will be k devices assigned as backups.

If we have an assignment of devices as backups so that each is on at most b lists and has exactly k backups, we can translate this to a valid flow in the network of value nk: for each assignment of device i to device j as backup, create a flow path $s \to u_i \to v_j \to t$ of one flow unit. Clearly, given this respects vertex constraints, since any flow in to u_i or v_j also leaves via the next edge. Given the size of the lists and the limits on how many each device is assigned, we know that at most b will be going in to any u_i, and exactly k will be coming out of each v_j, so our flow will also not violate the capacity constraints.

■

4. Sample solved problem: Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity $c(e)$ on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given an integer maximum s-t flow in G, defined by a flow value $f(e)$ on each edge e. Now suppose we pick a specific edge $e \in E$ and increase its capacity by one unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(|V| + |E|)$.

   Solution: The point here is that $O(V + E)$ is not enough time to compute a new maximum flow from scratch, so we need to figure out how to use the flow f that we are given. Intuitively, even after we add 1 to the capacity of edge e, the flow f can't be that far from maximum; after all, we haven't changed the network very much. In fact, it's not hard to show that the maximum flow value can go up by at most 1:

   Claim: Consider the flow network G' obtained by adding 1 to the capacity of any edge e. The value of the maximum flow is either $v(f)$ or $v(f) + 1$, where $v(f)$ is the value of the maximum flow for the original graph G.

   Proof of claim: The value of the maximum flow in G is at least $v(f)$, since f is still a feasible flow in this network - increasing one edge's capacity cannot cause f to violate the edge constraints. It is also integer-valued. So it is enough to show that the maximum-flow value in G is at most $v(f) + 1$. By the Max-Flow Min-Cut Theorem, there is some s-t cut (A, B) in the original flow network G of capacity $v(f)$. Now we ask: What is the capacity of (A, B) in the new flow network G? All the edges crossing (A, B) have the same capacity in G that they did in G, with the possible exception of e (in case e crosses (A, B)). But $c(e)$ only increased by 1, and so the capacity of (A, B) in the new flow network G is at most $v(f) + 1$.

   So we get the following algorithm, which finds if there is an augmenting path in G'_f (our modified graph) and then adds one to ever edge in that path if it exists:

```
IncreaseEdge(G, f, e):G' ← G
  Modify G': increase edge e's capacity by 1
  Build the residual graph G'_f for f in G'
  BFS(s)
  if t is unmarked:
      return f
  else
      x ← t
      while x ≠ s
          f((p(x), x)) ← f((p(x), x)) + 1
          x ← p(x)
```

   For BFS code, I used the following version from page 200 of book which stores parent pointers, but modified for directed graphs:

```
BreathFirstSearch(s):
  put (∅, s) in queue Q
  while Q is not empty
      take (p, v) from queue
      if v is umarked
          mark v
          parent(v) ← p
          for each edge (v, w)
              if (v, w) is unmarked
                  add (v, w) to Q
```

Proof of correctness: Based on the Maxflow-Mincut theorem from the textbook, f will still be the maximum flow precisely if there are no paths in the residual network. If a path does exist in G'_f, we know BFS will find the path, and by our claim above, we know the overall flow will only one until of flow along that path, since otherwise we would exceed e's capacity.

Runtime: We change 1 edge weight, build a residual graph in $O(V + E)$ time, and run BFS in $O(V + E)$ time. The if statement spends at most $O(V)$ time tracing the path and updating the flow values. So, total time is $O(V + E)$.

■