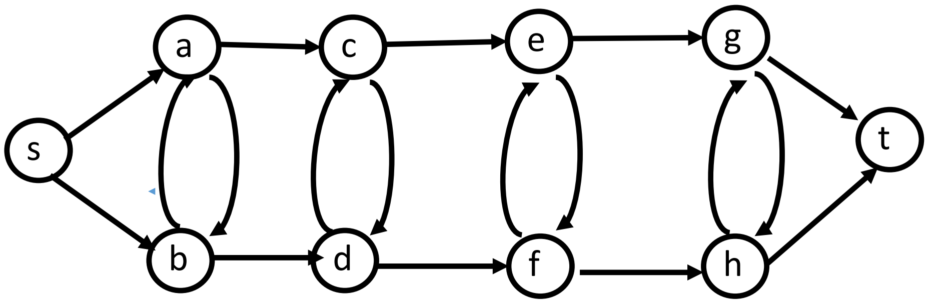
Introduction to Algorithms	**Problem Set 6 (2 problems)**
CS 4820 Fall 2018	**Solutions**

(1) Finding minimum cut closest to t Consider a maximum flow problem with directed graph $G = (V, E)$ with m edges and $n \leq m$ nodes, a source $s \in V$, a sink $t \in V$ and integer capacities $c_e \geq 0$ on each edge. We note that some graphs have many minimum capacity (s,t)-cuts. For example, in the graph below, if all edge capacities are $c_e = 1$, this graph has lots of minimum capacity cuts: any of $\{s\}, \{s, a, b\}, \{s, a, b, c, d\}$, etc. are all s-sides of minimum capacity cuts (cuts of capacity 2).



Assume that you are given a maximum value flow f that was already computed. Give an $O(m)$ time algorithm that finds a minimum capacity (s,t)-cut, (A, B) with $s \in A$ and $t \in B$, with as many nodes in A as possible. In the example above, this would be the cut $A = \{s, a, b, c, d, e, f, g, h\}$ and $B = \{t\}$.

Solutions: The cut we are looking for is cut (A, B) defined as $B = \{v \, \text{such that } G_f \text{ has a path from } v \text{ to } t\}$, and $A = V \setminus B$.

Finding this takes linear time. Starting at t, run a graph search using the edges on the residual graph G_f backwards to see what nodes can reach t. Define that set of nodes as $B = \{v \in V :$ there is a path from v to t in the residual grapg $G_f\}$ and all others as $A = V \setminus B$.

To prove that the algorithm is correct, we first need to show that (A, B) is a min-cut. Recall that for any cut (A, B) the flow value f can also be defined by flow across cut (A, B), and we get

$$v(f) = \sum_{e=(v,w):v \in A, w \in B} f(e) - \sum_{e=(v,w):v \in B, w \in A} f(e) \leq \sum_{e=(v,w):v \in A, w \in B} c_e = c(A, B)$$

Recall that a min-cut has capacity $v(f)$ as f is a maximum flow, so a min-cut is one that has equation in the last inequality: implying a cut is a min cut only if the cut has $f(e) = c_e$ for all edges from A to B and $f(e) = 0$ for all edges B to A. Now we show that the cut just defined has this property. Consider any $(u, v) \in A \times B$. We must have $f(e) = c_e$. If not, there would be an edge (u, v) in the residual graph G_f, producing a path from u to t. This contradicts the fact that u is in A which is the set of vertices without a path to t in G_f. Now consider any $(v, u) \in B \times A$. We must have $f(e) = 0$. If not, there would be an edge (u, v) in the G_f, producing a path from u to t. This contradicts the fact that u is in A just as before. Thus (A, B) is a min-cut.

Second, we need to argue that any node $v \in B$ must be on the B side of any minimum capacity cut, this then shows that A is as large as possible. If there is another min-cut (A', B') with $|B'| < |B|$, there must be some $v \in B$ such that $v \notin B'$ and $v \in A'$. Note that $v \in B$ if there is a v to t path in G_f. If $v \in A'$, then there is now a path from v to t which must pass through some edge e going from A' to B' in the residual graph. This means that $f(e) < c_e$. But this is impossible because (A', B') is a min-cut and a min-cut must have $f(e) = c_e$ for all edges going from A' to B' in G. So, if we select B as above, we will have found the smallest set of nodes possible to have in B for a min cut, which corresponds to the largest A.

Common mistakes (draft):

- Some students tried to perform a BFS in the residual graph G_f starting from t but without reversing the edges of the residual graph or doing something equivalent.
- Some people recognize that the right set B is the set of nodes v with a path from v to t in the residual graph G_f, and then find B by running a path computation from every v separately, this finds the correct cut, but it is not linear time.
- Some students have some part of the proof missing: either not showing that the cut they defined is a min-cut, or not showing that all nodes $v \in B$ have to be on the B side of any min-cut.
- some students forgot to analyze the running time.

(2) Scheduling Interviews

You are scheduling initial phone interviews for n job candidates that have applied to different jobs at the same company (each candidate has applied to just one job). The company has $k \leq n$ recruiters, and each recruiter is qualified to interview candidates only for some of the jobs. Each candidate needs to be assigned to a recruiter to be interviewed.

Candidates will be labeled $0, 1, \ldots, n-1$, and recruiters will be labeled $n, n+1, \ldots, n+k-1$. We will let $\mathsf{neighbors}$ hold the adjacency lists of the bipartite graph representing compatible candidate-recruiter pairs. That is, for each candidate i, $\mathsf{neighbors}[i]$ holds the recruiters that they can be interviewed by. For each recruiter j, $\mathsf{neighbors}[j]$ holds the candidates that recruiter j can interview. You also want to make sure that recruiters are not overloaded: so, we have an array $\mathsf{recruiter\_capacities}$ where $\mathsf{recruiter\_capacities}[j]$ is the maximum number of interviews that recruiter j can do (note that because of our indexing scheme, entries 0 through $n-1$ of this array are empty).

Someone has already tried to find an assignment of candidates to interviewers, but they are having trouble. They have an array $\mathsf{preliminary\_assignment}$ that assigns each candidate i to recruiter $\mathsf{preliminary\_assignment}[i]$ (without overloading any recruiter). Unfortunately, $\mathsf{preliminary\_assignment}[n-1]$ is blank, and they are having trouble filling this last entry.

You must code an efficient algorithm that has the following behavior:

- If there exists a valid assignment that assigns all job candidates to recruiters, output it. Note that **there may be more than one such valid assignment**. We will accept any, as long as it is assigns all job candidates to recruiters in a valid way (i.e., no recruiter is overbooked).
- If no such assignment exists, you plan to ask one of the recruiters j to increase their capacity $\mathsf{recruiter\_capacities}[j]$. Output the list of recruiters j such that if their capacity is increased by 1 (while the other capacities remain the same), then a solution will exist.

Your algorithm must run in $O(m)$ time, where $m = \sum_{i=0}^{n-1} |\mathsf{neighbors}[i]|$ is the number of edges in the graph. Solutions that take longer (e.g., $O(nk)$) will only get partial credit.

We illustrate the problem with the following **example**:

- Suppose we have $n = 3$ candidates, $k = 2$ recruiters, candidate neighbors $\mathsf{neighbors}[0] = (3, 4)$, $\mathsf{neighbors}[1] = (3)$, and $\mathsf{neighbors}[2] = (3)$ (the rest of the array can be inferred from these entries), and recruiter capacities $\mathsf{recruiter\_capacities}[3] = 2, \mathsf{recruiter\_capacities}[4] = 1$. If we are given $\mathsf{preliminary\_assignment} = (3, 3, \cdot)$, then a fully satisfying assignment does exist: it is $\mathsf{valid\_assignment} = (4, 3, 3)$.
- For the (otherwise) same input, if $\mathsf{recruiter\_capacities}[3] = 1, \mathsf{recruiter\_capacities}[4] = 2$, and $\mathsf{preliminary\_assignment} = (4, 3, \cdot)$, then no fully satisfying assignment exists, and the only way to create one is to increase $\mathsf{recruiter\_capacities}[3]$.

Solutions: $\mathsf{neighbors}$ makes up an adjacency list representation of an undirected bipartite graph, while $\mathsf{preliminary\_assignment}$ represents certain edges that are selected in that graph. By definition of $\mathsf{preliminary\_assignment}$, the last candidate vertex is not assigned to any recruiter, while every other candidate is. We wish to (i) check if there exists an assignment of a recruiter to every candidate such that no recruiter is assigned more candidates than their specified limit, and output such an assignment if yes; and (ii) if no, output the recruiters that would enable such an assignment if they were to increase their limit by 1.

To solve this problem, we need to reduce to max flow. In particular, we may draw a directed graph that contains a node for every candidate, a node for every recruiter, and additional "virtual" source and sink nodes. We then draw an edge from the source node to every candidate node, and draw an edge from every recruiter node to the sink. Then, considering each candidate i, we draw an edge from that candidate i to every recruiter in $\mathsf{neighbors}[i]$. For every edge from a recruiter j to the sink, we give it a capacity $\mathsf{recruiter\_capacities}[j]$. All other edges get capacity 1. Now, a flow in this network is equivalent to a candidate-recruiter assignment in our original undirected graph (just take the edges between candidates and recruiters that get flow > 0).

How does this help solve our original problem? We are given a partial assignment covering $n-1$

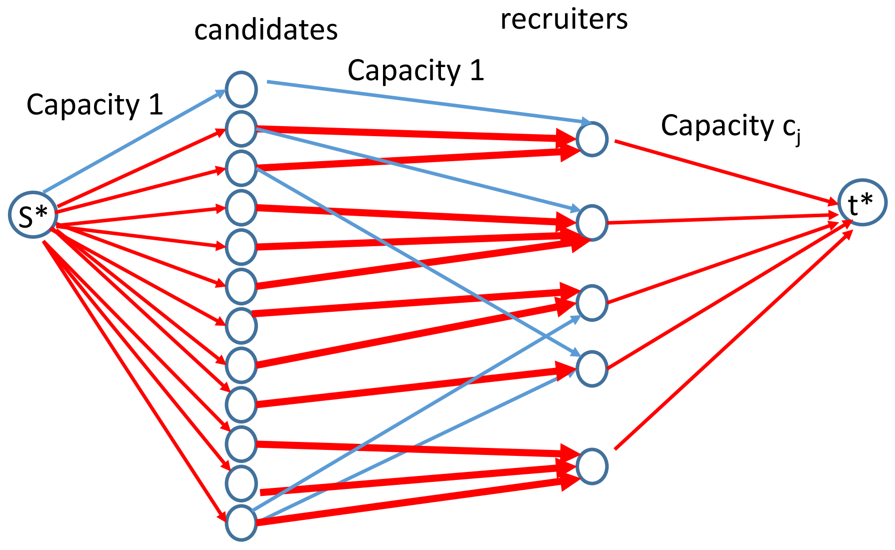


Figure 1: The network flow problem corresponding to candidate-recruiter assignment. Red edges are where the flow of the original assignment is going, blue edges represent the alternate recruiter options for candidates.

candidates - this is equivalent to a flow of value $n-1$ as shown on the figure. Note that in our network, even if we increase the capacity of a recruiter, the max flow will never exceed n, as the capacity of the edges leaving s is only n. Thus, to answer (i) and (ii), we simply need to check if our original network has max-flow value n, and if not, we need to output whether each of the k networks created increasing the capacity of each of the k recruiters has max-flow n.

To answer the first question, we simply use a useful property we have learned: that a flow is maximum if and only if there is no augmenting path if and only if there is no path from the source to sink in the residual network. The difficult part of implementing this is getting it to run in $O(m)$ time. Observe that since we have provided you with adjacency lists, we have provided a graph representation of size $O(m)$.

It is a good idea to convert this network into its residual network under the flow preliminary_assignment. That is, if an edge from a candidate i to a recruiter j is in the preliminary_assignment, delete it from i's adjacency list but keep it in j's (this is equivalent to orienting the edge towards the candidate, as we may push flow "backwards" along it). Otherwise, keep it in i's adjacency list but delete it from j's. Note that we do not even need to consider adding nodes to represent the source and the sink: there will be a source-sink path in this network if and only if the simpler residual network described above has a path from the only unpaired candidate to a recruiter j that is assigned to less than his/her limit in the preliminary assignment.

Thus, we can satisfy our original objectives by simply running DFS on our simplified residual network, starting at the candidate node that is unpaired in the preliminary assignment.

The observation that held with objective (ii) is that the set of recruiters with the property that upon increasing their limit by 1 would produce a path from the source to the sink are exactly the recruiters that are reached by DFS in the simplified residual network. To find these recruiters, while running DFS, we write down all recruiters reached as a 1 in the array bottleneck_recruiters. When DFS is complete, we iterate over bottleneck_recruiters, checking if any recruiter that was reached by DFS was assigned to less candidates than their limit (to check this, it may be helpful to use preliminary_assignment to count the number of candidates assigned to each recruiter, and then compare these results against recruiter_capacities).

If not, then we are done (because bottleneck_recruiters contains exactly the recruiters that, upon increasing their limit by 1, would allow our discovered path from the unassigned candidate to that recruiter to be extended all the way to the sink). If yes, then we need to obtain the path from the unassigned candidate to the recruiter with available capacity. To do this, we may simply run DFS again, using a table that maps each node v to its "parent" (i.e., the unique node that was visited immediately before v). The path can be easily reconstructed from this table in $O(m)$ time. (To speed things up, we can also just construct this table the first time we run DFS.)

Once we have the path, we can assign each candidate i to a recruiter by examining whether i is on the above mentioned path. If so, that candidate will be assigned the recruiter that immediately follows them on this path. If not, that candidate will keep their assignment to the recruiter specified by preliminary_assignment.

Common mistakes (draft)

- some students attempted to do this without using the flows and the residual graph. On very small instances, it may be possible to consider swaps of interviews between pairs on candidates, and this can give you the solution. However, in larger instances this "direct" solution is no longer viable: the feasible assignment of candidates to interviewers can involve a very long chain of changes. This corresponds to a long path in the residual graph, which the flow method will find. But direct swaps are not viable
- some students didn't use the initial assignment. It's not possible to do this in linear time without using the initial assignment. In the terms of the flow, the initial assignment is a flow of value $n-1$, and with n candidates the maximum flow possible is n, so given the initial flow, all we needs is one more augmentation.
- some students decided correctly if there is an assignment, but had trouble finding it: ran a DFS to find an augmenting path, but updated the assignment for all candidates reached by the DFS, even those that the DFS backtracked from, and not only those that are on the path found.
- some students found an assignment in linear time, when there is one, but had trouble getting the list of interviewers whose increased capacity would allow us to accommodate the one extra candidate. A more direct way to do this is to test for each interviewer separately if increasing their capacity by 1 would give us a path in the residual graph. This works also, though it's not linear time. The observation is that this is easy to determine using the original residual graph.