Introduction to Algorithms	Problem Set 6 (3 problems)
CS 4820 Fall 2019	Solutions

(1) Scheduling Interviews I (10 points)

We are scheduling initial phone interviews for n job candidates that have applied for jobs at a company. The company has $k \leq n$ recruiters, and each recruiter is qualified to interview only some of the candidates. Candidates will be labeled $0, 1, \ldots, n-1$, and recruiters will be labeled $n, n+1, \ldots, n+k-1$. Each candidate needs to be assigned to a recruiter to be interviewed. We can represent the problem by a bipartite graph where nodes are the job candidates $0, 1, \ldots, n-1$ on one side, and recruiters $n, n+1, \ldots, n+k-1$ on the other side, and a job candidate i is connected to a recruiter j if j can interview i. We also want to make sure that recruiters are not overloaded, which means that for each recruiter j, we have c_j, a maximum number of interviews she could have. A valid schedule is a way to assign candidates to recruiters so that each candidate is assigned to a recruiter, and no recruiter j has more than c_j interviews scheduled (which is called $\mathsf{recruiter\_capacities}[j]$ in the code below).

Someone has already tried to find an assignment of candidates to interviewers, but they are having trouble. They have a valid assignment where each candidate i is assigned 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. In this problem, you are asked to use preliminary_assignment to decide if the there is a perfect schedule. If no such schedule is possible, we ask you to output the list of recruiters j such that if j is willing to interview one extra candidate the assignment will exists.

Specifically, you must code an efficient algorithm in Java 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 assigns all job candidates to recruiters in a valid way (i.e., no recruiter is overbooked and recruiters only interview candidates they are qualified to interview).
- 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.

The time limit of this problem will be 2 second. Your algorithm must run in $O(m)$ time, where m 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 $\mathsf{recruiter\_capacities}[3]$.

Starter code provided. Unlike in previous problems, we have set up a partial code for you that reads the input and writes the output, and sets up useful data-structures, and all you have to do is add the missing part: deciding if an assignment exists or which recruiters can help. Our partial code is available on CMS, called *public_code.java*. You need to add your code between the lines "YOUR CODE STARTS HERE" and "YOUR CODE ENDS HERE".

Note that **you will not parse input nor print output**. We parse the input and provide you with complete and incomplete data structures. You will use the completed data structures to fill the incomplete data structures. Our framework with use the data structures you fill to format and print the output. We describe these data structures below, and also describe the input/output file formats.

Complete data structures we provide:

- The arraylist neighbors, where neighbors$[i]$ is an arraylist all people who can be matched with i. That is, if if $i = 0, 1, \ldots, n-1$ then neighbors$[i]$ is an arraylist of the indices of the recruiters qualified to interview candidate i. If $i = n, n+1, \ldots, n+k-1$ then neighbors$[i]$ is an arraylist of the indices of the candidates that recruiter i is qualified to interview.
- The array recruiter_capacities, where recruiter_capacities$[j]$ is the limit of candidates that recruiter j can interview. Note that because of our indexing scheme, entries 0 through $n-1$ of this array are empty
- The array preliminary_assignment, where preliminary_assignment$[i]$ is the recruiter assigned to interview the i^{th} job candidate in the provided first attempt, for $i = 0, 1, \ldots, n-2$. Recall that preliminary_assignment$[n-1]$ is blank, and they are having trouble filling this last entry.

Data structures we provide that you must fill in:

- The boolean value exists_valid_assignment, which you should set to true if there exists a way to assign every job candidate to a recruiter without increasing their limits (and false otherwise).
- The array valid_assignment, which you should populate *only if* you set the previous variable to true. valid_assignment$[i]$ is the index of the recruiter assigned to candidate i in your full valid assignment. (Note that i ranges over $0, 1, \ldots, n-1$, and the values of the array range over $n, n+1, \ldots, n+k-1$.)
- The array bottleneck_recruiters, which you should populate *only if* you set exists_valid_assignment to false. bottleneck_recruiters$[j]$ should be set to 1 if increasing the limit of recruiter j by 1 (while leaving the other limits the same) would make a full assignment possible, and 0 otherwise. (Note that the array is defined over $j = n, n+1, \ldots, n+k-1$, so **the first n entries must be set to 0.**)

Input file format (just FYI; only work with the data structures above):

- The first line has two numbers, n and k, the number of candidates and recruiters.
- In the i^{th} line of the next n lines (for $i = 0, \ldots, n-1$), there is a number indicating how many recruiters candidate i can interview with, followed by the indices of those recruiters. The indices appearing in the i^{th} line are stored in neighbors$[i]$. (neighbors$[j]$ for $j \geq n$ are inferred from these lines, as well.)
- The next line has k numbers representing the capacities of the recruiters. We store the i^{th} number in recruiter_capacities$[n+i-1]$.
- The last line has $n-1$ numbers representing the preliminary assignment. We store the i^{th} number in preliminary_assignment$[i-1]$.

Output file format (just FYI; only work with the data structures above):

- If there exists a full assignment, the first line will be "Yes", and the second line will be the n entries of valid_assignment.
- If there does not exist an assignment, the first line will have the string "No", and the second line will be the $n+k$ entries of bottleneck_recruiters (where the first n entries are 0).

The only libraries you are allowed to `import` are the ones in `java.util.*`

Warning: be aware that the running time of calling a method of a built-in Java class is usually not constant time, and take this into account when you think about the overall running time of your code. For instance, if you used a LinkedList, and use the `indexOf` method, this will take time linear in the number of elements in the list.

We have set up an **online autograder** at `https://cs4820.cs.cornell.edu/`. You can upload solutions as many times as you like before the homework deadline.[1] **You need to have the main method of your code named Main, must not be "public", must not be part of a package**. When you upload a submission, the autograder will automatically compile and run it on a number of public test-cases[2] that we have prepared for you, checking the result for correctness and outputting any problems that may occur. You should use this facility to verify that you have interpreted the assignment specification correctly. After the deadline elapses, your last submission will be tested against a new set of *private* test cases, which your grade for the assignment will be based on.

Solutions: neighbors makes up an adjacency list representation of an undirected bipartite graph, while preliminary_assignment represents certain edges that are selected in that graph. By definition of 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 neighbors$[i]$. For every edge from a recruiter j to the sink, we give it a capacity 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 $n-1$ candidates - this is equivalent to a flow of value $n-1$. 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 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

[1]The deadline shown on the auto-grader is the latest time at which you will be allowed to submit your code, including the maximum number of slip days. If you submit your code past the actual deadline, you will automatically be charged slip days accordingly, so please be careful.

[2]They will also be available on CMS.

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 abovementioned 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.

Expected mistakes:

- Running DFS once for each recruiter (i.e., iteratively trying to find an augmenting path after increasing each recruiter's limit separately). This will take $O(mk)$ time, or $O(nk^2)$ time if the first mistake was made as well.

(2) Scheduling Interviews II. (10 points) In Problem 1 above, we were assigning candidates to interviewers for a company without scheduling times for the interviews. In this problem, you are asked to set up a more detailed schedule. As in the previous problem, you are scheduling initial phone interviews for n job candidates at a company, with $k \leq n$ potential recruiters. Each candidate needs only one interview for now: concretely, each candidate needs to be matched to exactly one recruiter to be interviewed. The candidates are applying for different kinds of jobs, and recruiters are qualified to interview candidates only for some of the jobs. For each candidate i, you are given a list L_i of the recruiters who are qualified to interview that candidate. You also don't want to overload the recruiters, so each recruiter j has a limit c_j of the maximum number of interviews he or she can do.

Each interview will take one hour, and will be scheduled in one of T possible non-overlapping one-hour time slots $h_1, \dots h_T$, each starting at the top of the hour (e.g. 10 AM, 1 PM, 5 PM …). In addition to the input above, each interviewer and each candidate is asked to list the hours that they would be able to interview or be interviewed. Please note that each interviewer is asked to list strictly more than c_j possible slots to make scheduling easier.

Give an algorithm that finds an interview schedule if one exists. Let $m = \sum_i |L_i|$. Your algorithm should run in time polynomial in n, k, m, and T.

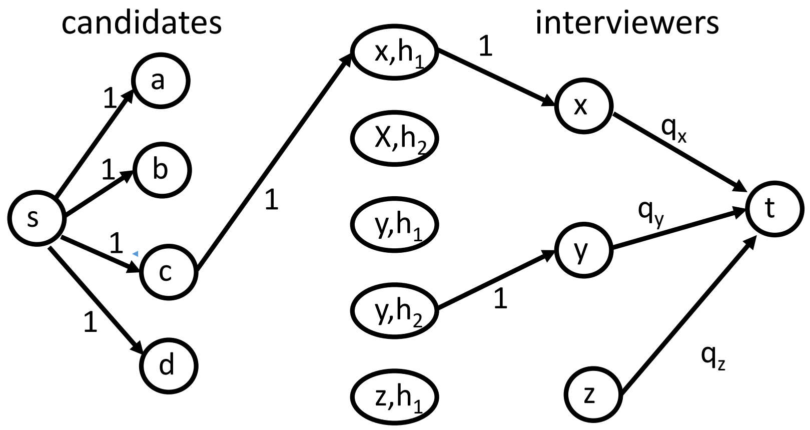
Solution: Solve via maximum flow using Ford-Fulkerson. The solution is illustrated at the figure below. For nodes, we have s and t, a node for each candidate (n total), a node for each interviewer (k total), and a separate node (x, h) for each time h an interviewer x can make ($O(kT)$ total).

For edges we have the following

- edges (s, c) for each candidate c with capacity 1.
- edges (x, t) for each interviewer x with capacity c_x
- edges $((x, h), x)$ for all nodes associated with a time slot h of interviewer x with capacity 1
- edges $(c, (x, h))$ if candidate c can be interviewed by interviewer x and is available at time h.

Finally, the assignment exists if the maximum flow has value n, and then each candidate c is assigned to interviewer x at time h if the edge $(c, (x, h))$ has flow 1.

The resulting graph has at most $N = 2 + n + k + Tk$ nodes, and at most $M \leq N^2$ edges. The



capacities entering t are $C = \sum_j c_j$, where c_j denotes the capacity of interviewer j. Note that we can assume $c_j \le n$ for all j, and so $C \le nk$, and hence a total running time of $O(MC) = O(nk(n + Tk)^2)$.

Proving the solution correct. First note that the Ford-Fulkerson algorithm finds an integer valued flow, so the resulting flow is integer valued. If the total flow has value n, then it has value 1, each edge (s, c) must have flow 1, and exactly one of the edges leaving c will have flow. With the edges $((x, h), x)$ having capacity 1, at most one candidate can be scheduled with x at time h, and with the edges (x, t) having capacity c_x, interviewer x can do at most c_x interviews.

(3) Reduction. (10 points) An *Independent set* in an undirected graph $G = (V, E)$ a set of vertices $I \subset V$ with no two vertices $v, w \in I$ connected by an edge, that is $(v, w) \notin E$. The INDEPENDENT SET problem has input an undirected graph G and an integer k and asks to decide if G has an independent set of size k. This a famous hard problem will be discussed in the lecture on Monday Oct 28 (see also section 8.1 of the book).

Your friend likes getting together small groups of extremely independent people. He claims to have written a super-fast solver for the following graph problem to help him do this: He mapped all people he ever considered inviting as an undirected graph $G = (V, E)$ with the set of nodes representing people, and two nodes $v, w \in V$ connected by an edge if they regularly talk. He views two people $v, w \in V$ as *very independent* if not only they are not connected by an edge, but also they have no shared neighbor u, that is, no node u with edges (uv) and (uw) in E (the reason is that such a u may be influencing both v and w, which does make the two of them less independent). When he is arranging a meeting, he identifies a subset $S \subset V$ of potential candidates, and then looks for k people in S that are very independent. So the VERYINDEPENDENT SET problem is given an undirected graph $G = (V, E)$, a subset of nodes $S \subset V$, and an integer k decide if there is a very independent set $I \subset S$ of size k.

You are impressed by what your friend's code can do, and wonder if this code is useful to solve the famous INDEPENDENT SET Problem also. Show that given an undirected graph G and integer k for the INDEPENDENT SET problem, you can solve this problem by a single call your friend's code. It is okay to modify G to a new graph G' before calling your friend's code as long as G' is polynomial in the size of G. In providing a solution to this problem you need to do the following:

- provide a polynomial time algorithm that converts an input to the INDEPENDENT SET problem, an undirected graph $G = (V, E)$ and an integer k to an input to the VERYINDEPENDENT SET problem, which consists of a graph $G' = (V', E')$, $S' \subseteq V'$ and integer k'
- prove that if $G = (V, E)$ has an independent set of size k then G' has a very independent subset of the set S' of size k'
- and prove that if $G = (V, E)$ has no independent set of size k then G' also doesn't have a very independent subset of the set S' of size k'

Note that the last two proves show that the independent set problem on G is equivalent to the very independent set problem on G'.

(Your solution will show that INDEPENDENT SET $\leq$ VERYINDEPENDENTSET, and in the coming week, we'll show the INDEPENDENT SET problem is NP-complete.)

Solution 1: Given the graph G add an extra node in the middle of every edge, resulting in a new graph G'. So if G has n nodes and m edges, than G' will have $2m$ edges and $n+m$ nodes. Let $S = V$ be the original set of nodes in G. Note that $I \subset V$ is very independent in G' if and only if I is independent in G.

Solution 2: Given the graph G add an extra node in the middle of every edge, and connect all pairs of new nodes with an edge. Let the resulting new graph be $G' = (V', E')$. So if G has n nodes and m edges, than G' will have $2m + m(m-1)/2$ edges and $n+m$ nodes. Let $S = V'$ be at the nodes in G' (including new nodes). Note that $I \subset V$ is very independent in G' if and only if I is independent in G, as above, so if G has a k size independent set than G' has a very independent set of size k.

To show the opposite, notice that a new node x (in the middle of an edge) is at most a distance 2 from all other nodes (distance 1 from all new nodes, and hence distance 2 from the original nodes). So if any new node is includes in a very independent set, this is a set of size 1. So G' has a very independent set of size k with $k > 1$ than is a subset of V and hence an independent set in G. And this is also true for $k = 1$.

Solution 3: Given the graph G add an extra node in the middle of every edge, add two extra nodes s and t. Add edge (s, t), and edges (s, e) for all new nodes in the middle of the edges. Let the resulting new graph be $G' = (V', E')$. So if G has n nodes and m edges, than G' will have $3m + 1$ edges and $n + m + 2$ nodes. Let $S = V'$ be at the nodes in G' (including new nodes). Note that $I \subset V$ is very independent in G' if and only if I is independent in G, as above, and we can add t to the set I and it remains independent. So if G has a k size independent set than G' has a very independent set of size $k + 1$.

To show the opposite, notice that a very independent set I' in G' can contain at most one new node (as no pair of them is very independent). The remaining nodes must be nodes in the original set V, and most be independent. So G' has a very independent set of size $k + 1$ than k nodes of this set is a subset of V and hence an independent set in G of size k.

Mistake of reducing the wrong way: An important mistake to avoid is reducing the wrong way. If we take an instance of the VERY INDEPENDENT SET, and turn it into INDEPENDENT SET that would show that INDEPENDENT SET is at least as hard as the VERY INDEPENDENT SET. This is the wrong direction! Our goal is to show that VERY INDEPENDENT SET is hard.

In fact, the backwards reduction is easier: given a graph for VERY INDEPENDENT SET just add edges for all pairs of nodes that are not very independent.