College of Computer & Information Science
Northeastern University
Fall 2017

CS5800
MS Algorithms
10 December 2017

# Sample Solution to Problem set 6

## 1. (5 points) Optimal arrangement for a company retreat

You are organizing a big dinner party for your company retreat. To increase social interaction among the various departments in your company, you would like to set up a seating arrangement so that no two members of the same departments are at the same table. Show how to formulate the problem of finding a seating arrangement that meets this objective as a maximum flow problem.

Assume that the company has $n$ departments and the $i$th department has $m_i$ members. Also assume that $t$ tables are available and the $j$th table has a seating capacity of $c_j$.

**Answer:** The capacitated graph $G(V, E)$ is defined as:

$$\begin{aligned} V &= \{s,t\} \bigcup \{u_i : 1 \le i \le n\} \bigcup \{v_j : 1 \le j \le t\} \\ E &= \{(s,u_i) : 1 \le i \le n\} \bigcup \{(u_i, v_i) : 1 \le i \le n, 1 \le j \le t\} \bigcup \{(v_j, t) : 1 \le j \le t\} \end{aligned}$$

We also have the following equations:

$$\begin{aligned} cap(s,u_i) &= m_i \\ cap(u_i,v_j) &= 1 \\ cap(v_j,t) &= c_j \end{aligned}$$

After we represent the problem as the above, an integral solution to this max flow problem gives an assignment of families to tables. A flow from $u_i$ to $v_j$ is either zero or one, with one meaning that a member of family $i$ sits in table $j$. Thus the maximum flow maximizes the number of people that can be seated under the stated constraint.

## 2. (5 points) Bottleneck edge

Let $G = (V, E)$ be a flow network with source $s$ and sink $t$. We say that an edge $e$ is a *bottleneck* in $G$ if it belongs to *every* minimum capacity cut separating $s$ from $t$. Give a polynomial-time algorithm to determine if a given edge $e$ is a bottleneck in $G$.

**Answer:** Since a bottleneck edge belongs to every min-cut, if we increase the capacity of the edge by 1, every current min-cut will increase by 1. The value of other cuts (which do not include $e$) remain the same, but the capacity of these cuts already exceeded the current valye of the maximum flow. Hence, the minimum-cut of the network (with the capacity of $e$ increased by 1) increases by 1. Hence the max-flow will increase by 1.

So the algorithm is to increase the capacity of $e$ by 1, recompute the maximum flow. If the max-flow increases by 1, then $e$ is a bottleneck edge, otherwise it is not.

For recomputing the maximum flow, one can simply use any efficient maximum flow algorithm. While this would be polynomial-time, one can do much better. In linear time, one can update

the current maximum flow. (Left to the reader; essentially, we need to find one more augmenting path.)

### 3. (5 points) NBA draft

The teams in the NBA chooses new players each year in a process called "the draft". Each year there are $n$ teams $t_1, ..., t_n$ and $2n$ players. The league wants to change the rules of the draft so that each team will give a list of players that it is willing to get and some algorithm will match 2 players for each team, out of the list of players the team is willing to get.

For a team $t_i$ let us denote by $A_{t_i}$ the set of players that the team is willing to get. Show that a necessary and sufficient condition for it to be possible to give each team 2 players out of the list of players it is willing to get, is that:

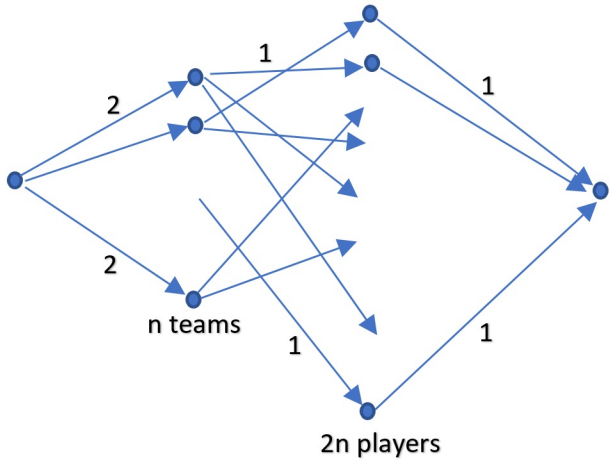
$$\left| \bigcup_{i \in I} A_{t_i} \right| \geq 2|I|$$

for any subset $I \subseteq \{1, ..., n\}$.

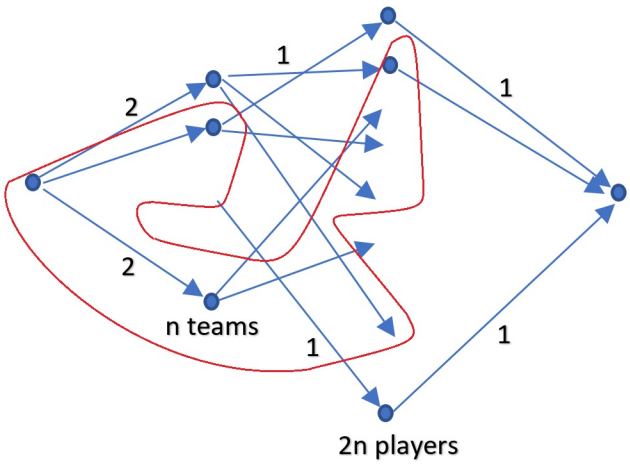
**Answer:** For a team $t_i$ let us denote by $A_{t_i}$ the set of players that the team is willing to get.
The condition is that for any subset of teams $I \subseteq \{1, ..., n\}$, we have:

$$\left| \bigcup_{i \in I} A_{t_i} \right| \geq 2|I|$$

Consider the following flow network:




Clearly there exists a flow of value $2n$ in this network, if and only if there exists a solution to the problem.
It is also clear that if there exists a solution then the condition holds since for any subset of teams $I$, the union of the sets of players they want, contains the $2|I|$ players that they get (in the solution).
To see the other direction, consider a general s-t cut of the graph:




We can write $S = \{s\} \cup T \cup P$, where $T$ is a set of teams and $P$ is a set of players.
The capacity of such a cut would be:

$$\text{cap}(S, \overline{S}) = \sum_{t \notin T} 2 + \sum_{(t,p) \in ((T \times \overline{P}) \cap E)} 1 + \sum_{p \in P} 1$$

Each player that the teams in $T$ want, is either in $P$ or in $\overline{P}$. If it is in $P$, it contributes 1 to the above sums (in the third sum). If it is in $\overline{P}$ it also contributes 1 to the above sums (in the second sum). therefore, the above sum is bigger or equal to:

$$\geq \sum_{t \notin T} 2 + \left| \bigcup_{t \in T} A_t \right| \underbrace{\geq}_{\text{By our assumption}} \sum_{t \notin T} 2 + 2|T| = 2n$$

**4. (5 points) Finding rows and columns in a matrix**

Let $M$ be a matrix with $n$ rows and $n$ columns whose entries are either 1 or 0. Describe an algorithm that finds a minimal set $I$ of rows and columns of $M$, such that any non-zero entry is in one of the rows or columns in $I$.

**Answer:** We construct a graph $G = (V, E)$ in the following way.

$$V = \{r_1, ..., r_n, c_1, ..., c_n\}$$

($r_i$ stands for row $i$ and $c_i$ stands for column $i$)

$$E = \{(r_i, c_j) \mid 1 \leq i, j \leq n,\ M(i, j) = 1\}$$

Clearly the problem is equivalent to finding a minimal set of vertices $C \subseteq V$ such that each edge $e \in E$ has one of its vertices in $C$ (this is a vertex cover).

We make $G$ into a flow network $G' = (V', E')$ by defining:

$$V' = \{s, t\} \cup V$$

$$E' = E \cup \{(s, r_i) \mid 1 \leq i \leq n\} \cup \{(c_i, t) \mid 1 \leq i \leq n\}$$

and $w : E \to \mathbb{R}$ is such that:

$$w(e) = \begin{cases} \infty, & e \in E; \\ 1, & e \notin E. \end{cases}$$

Given an $s-t$ cut $(S, \overline{S})$ of $G'$ we define:

$$U = (\{r_1, ..., r_n\} - S) \cup (S \cap \{c_1, ..., c_n\})$$

We note that if $\text{cap}(S, \overline{S}) < \infty$ then $U$ covers all the edges in $E$. Indeed if there exists an edge $(r_i, c_j)$ that is not covered by $U$ then $r_i \in S$ and $c_j \notin S$ which would imply that $\text{cap}(S, \overline{S}) = \infty$.

It is also clear that $\text{cap}(S, \overline{S}) = |U|$, since the edges that crosses the cut are the edges from $s$ to $(\{r_1, ..., r_n\} - S)$ and the edges from $(S \cap \{c_1, ..., c_n\})$ to $t$.

On the other direction, if $U$ is a cover, we can define an $s-t$ cut $(S, \overline{S})$ by:

$$S = (\{r_1, ..., r_n\} - U) \cup (\{c_1, ..., c_n\} \cap U) \cup \{s\}$$

We have $\text{cap}(S, \overline{S}) < \infty$. Indeed if there exists an edge $(r_i, c_j)$ such that $r_i \in S$ and $c_j \notin S$ then $r_i \notin U$ and $c_j \notin U$ which means that $(r_i, c_j)$ is not covered. Moreover $\text{cap}(S, \overline{S}) = |U|$. Indeed, $r_i \in U$ if and only if $r_i \notin S$ so the edge $(s, r_i)$ crosses the cut iff $r_i \in U$. Similarly $c_j \in U$ if and only if $c_j \in S$ so the edge $(c_j, t)$ crosses the cut iff $c_j \in U$.

We showed that any finite $s-t$ cut $(S, \overline{S})$ defines a cover $U$ and any cover $U$ defines a finite $s-t$ cut such that $\text{cap}(S, \overline{S}) = |U|$. It follows that by finding a minimal cut we can find a minimal cover.

**5. (2 + 3 = 5 points) Reconstructing a tree**

You are asked to reconstruct the reporting hierarchy of a huge company Disorganized, Inc., based on information that is complete but poorly organized. The information is available in an $n$-element array, in which each element of the array is a pair (*emp*, *boss*) where *emp* is the name of an employee and *boss* is the supervisor of the employee, and $n$ is the number of employees in Disorganized. You may assume that the names of all employees are distinct. For the company CEO, the *boss* entry is empty.

Your task is to compute a tree in which each node has the name of an employee *emp* and a parent field pointing to the node corresponding to the supervisor of *emp* (for the CEO, the parent field will point to NIL).

**(a)** Design an $O(n \log n)$ time deterministic algorithm for the problem. Justify the running time of your algorithm.

**Answer:** We create a tree node for each employee with parent pointers initialized to NIL, and create a list of all tree nodes, sorted according to the to the name of the employee. Then, we go through each employee and set its parent pointer to the node corresponding to the supervisor. We can find the supervisor node by doing a binary search on the sorted list.

Sorting takes $O(n \log n)$ time. Finding each parent node takes $O(\log n)$ time, for a total of $O(n \log n)$ time.

**(b)** Using hashing, design an expected $O(n)$ time randomized algorithm for the problem. Justify the running time of your algorithm.

**Answer:** Let $L$ denote the list of nodes given (each node containing the name of an employee and their boss). To distinguish these nodes from the nodes of the tree that we will build, we refer to the nodes in the list as *list-nodes* while we refer to the nodes in the tree as *tree-nodes*. A list-node $x$ has two fields $name[x]$ and $parentname[x]$. For a tree-node $u$, we will maintain

three fields: (i) $name[u]$, the name of $u$, (ii) $parent[u]$, a pointer to the parent of $u$, and (iii) $children[u]$, a doubly linked list containing pointers to the children of $u$.

We use hash tables for efficiently searching a node. In order to ensure expected $O(1)$ time for a search, we will choose our hash function from a class of universal hash functions. In order to resolve collisions, we use chaining. Insertion and deletion take $O(1)$ time as we will maintain each chain as a doubly linked list.

In our hash table, we will store the tree-nodes that we create. There are two ways in which we can hash the tree-nodes. One is using the name of their parent. The other is through their own name. If we hash according to the parent name, there are two kinds of collisions that take place. One is when two nodes have the same parent. And the other is due to the hash function. Due to the latter kind of collisions, it is *not the case that after all the nodes have been hashed, a given slot contains a list of siblings.* We must keep this in mind when we do the construction of the tree. The best way, perhaps, to avoid this problem is to maintain a chain of linked lists in each slot. Each slot maintains a linked list of pointers, each pointer pointing to a linked list of children associated with a parent that maps to this slot. Suppose we are hashing a node $x$ according to its parent's name. So we check slot $A[h[parentname[x]]]$, where $A$ is the hash table and $h$ is the hash function. Instead of just storing the node $x$ into a chain in this slot, we will obtain a pointer to the linked list that maintains the children of $parentname[x]$ and then insert node $x$ (or an associated entity) into this list. If we do not maintain a chain of linked lists, then either we may not be able to reconstruct a correct tree or we may not ensure expected linear running time.

We now give a detailed description of the algorithm in which we hash according to the given name (rather than the parent's name). This avoids some of the problems discussed in the above paragraph. Here is the algorithm:

TREE-CONSTRUCT($L$)
1. for each list-node $x$ in $L$ do
2. create new tree-node $u$;
3. $name[u] \leftarrow name[x]$;
4. $parent[u] \leftarrow nil$;
5. $children[u] \leftarrow nil$;
6. $hi(T, u)$; (hash according to *name*)
7. for each node $x$ in $L$ do
8. $u \leftarrow \quad (T, name[x])$;
9. if $parentname[x] \neq nil$ then $p \leftarrow \quad (T, parentname[x])$;
10. else $root \leftarrow u$;
11. $parent[u] \leftarrow p$;
12. insert $u$ into $children[p]$;
13. return *root*.

In the first for-loop, TREE-CONSTRUCT first creates a tree-node for every list-node in $L$ and inserts the tree-node into the hash table $T$ using universal hashing and chaining. In the second for-loop, the algorithm goes through the list-nodes again and (i) sets the parent pointer of each tree-node (except the root) to point to the appropriate tree-node that we search in the hash table, and (ii) inserts the tree-node into the children list of its parent tree-node. Finally, the algorithm returns the root. Clearly, at the end of the procedure, each tree-node $u$ has a pointer to its parent (since step 9 is executed once for the list-node corresponding to $u$)

and each of its children is inserted into its children linked list exactly once (since step 11 is executed exactly once for each of its children). Note that there is no recursion.

For the running-time, we first consider the first for-loop. In each iteration, steps 2 through 6 take $O(1)$ time. In the second for-loop, step 8 takes expected $O(1)$ time. Steps 9 through 11 take $O(1)$ time. Also, since the children field is a doubly linked list, step 12 takes $O(1)$ time. Finally, step 13 takes $O(1)$ time. Since there are exactly $n$ iterations for each for-loop, the expected running time is $O(n)$.

**6. (5 points) Prize Collecting Path**

You are given a graph $G = (V, E)$ and a subset $S \subseteq V$ of vertices such that each vertex in $S$ has a prize. Consider the problem of determining a simple path in $G$ that visits the maximum number of vertices in $S$.

Formulate a decision version of the above problem and prove that it is NP-complete.

**Answer:** The following applies to both undirected and directed graphs. We define the decision version as follows.

Given a graph $G = (V, E)$, a subset $S \subseteq V$ of vertices, and an nonnegative integer $k$. Is there a simple path in $G$ that visits at least $k$ vertices in $S$?

It is easy to prove membership in NP. A polynomial-size certificate for the problem is a simple path $P$; in linear-time we can check if (a) each edge in $P$ actually is in $G$, (b) $P$ is simple, and (c) $P$ has at least $k$ vertices in $S$.

We give a polynomial-time reduction from the Hamiltonian Path problem. Given an instance $G$ of the Hamiltonian Path problem, we reduce to the following instance of the above decision problem. Set the graph to be $G$, $S$ to be $V$ and $k$ to be $n$. Clearly, $G$ has a Hamiltonian path if and only if the above decision problem has a yes-answer. Also, the reduction is linear-time in the size of $G$.