College of Computer & Information Science
Northeastern University
Fall 2017

CS5800
MS Algorithms
21 November 2017

Problem set 6 (Due Thursday, Dec 7, 11:59 pm)

- The assignment is due at the time and date specified. Late assignments will not be accepted.
- We encourage you to attempt and work out all of the problems on your own. You are permitted to study with friends and discuss the problems; however, you must write up your own solutions, in your own words.
- If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class (or the class sta↵) is strictly prohibited. If you reference any source other than the textbook or class notes, please make sure you cite them in your submission.
- We require that all homework submissions be neat, organized, and typeset. You may use plain text or a word processor like Microsoft Word or LaTeX for your submissions. If you need to draw any diagrams, however, you may draw them with your hand.

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 ith department has m_i members. Also assume that t tables are available and the jth table has a seating capacity of c_j .

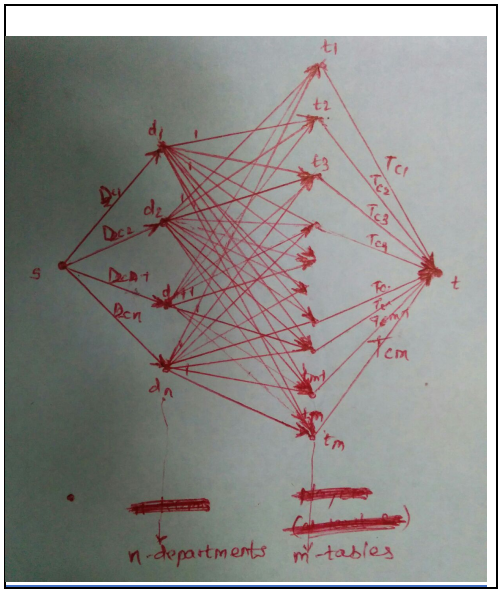
Logic –

Convert given problem into Bipartite Graph.
Apply Maximum Flow and solve the problem

Prerequisites –

1) Convert each department as a node ($d_i \mid i = \{1, 2, \ldots$ total departments$\}$) on left side of bipartite graph
2) Convert each table as a node ($t_j \mid j = \{1, 2, \ldots$ total tables$\}$) on right side of bipartite graph
3) (a) Create a source node *s* on left most side
   (b) Draw an edge between source node and every department (d_i)
   (c) Assign capacity (D_{ci}) to each edge equal to total members of that department
4) (a) Create a sink node *t* on right most side
   (b) Draw an edge between sink node and every table (t_j)
5) (c) Assign capacity (T_{cj}) to each edge equal to total capacity of that table
6) Draw an edge between each department and table pair and give capacity of 1 to that edge

Graph –



Answer –

If value of maximum s-t flow in above graph is less than total employees in the company ($\sum_{i=1}^{n} Dc_i$), then there is no valid seating arrangement possible.

Otherwise for every edge (d_i, t_j) that carries flow in the final flow we assign member d_i to the table t_j.

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.

Logic –

Find a maximum flow value for given graph.
Search whether given edge e is present in any of the min cut.
If it does not belon to any min cut
then return false.
Else increase the capacity of given edge and again calculate the maximum flow for original graph
If given edge e does not belong to any min cut
then return false.
Else return true

Complexity –

Time required to solve maximum flow + O(1)
Thus, running time complexity of given problem is almost equal to complexity of solving maximum flow

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:

$$| U_{i \in I} A_{t_i} | \quad \geq \quad 2| I |$$

for any subset I ⊆ {1, ..., n }

Answer –

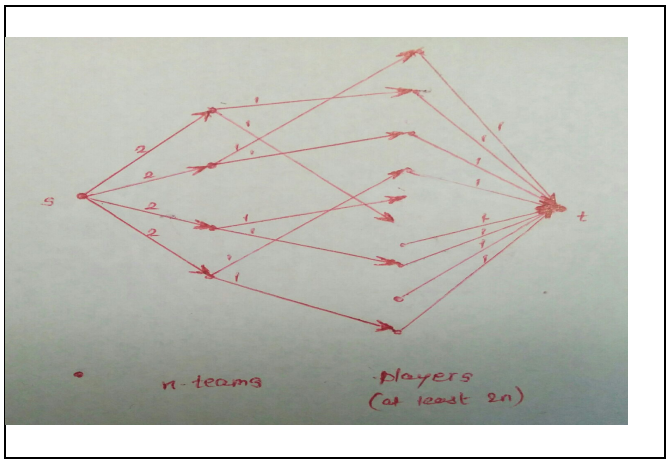
Let`s first convert the problem into a network flow maximization problem.

Bipartite graph can be created for this team player selection problem in such a way that all left side nodes represent the *n* teams and the all right side nodes represent the *total players*. Edges represents the A_{ti} i.e. set of player team willing to get and consider capacity of these edges is 1.

Now by connect all left points to a source node *s* and give them capacity of 2 (as each team can select 2 players).

Connect all players to a sink node *t* and give those edges capacity of 1.

The graph will be constructed something like below –



In above network clearly the for the maximum flow of 2n can be achived only by fully utilizing capacity of each team (implied restriction :- each team should get distinct players). And the restriction can be satisfied iff each team gives preference of distinct players or greater than minimum requirement (> 2), i.e. For *I* teams, the unioun of set of players they want should be at least *2I*.

This can be proved using a cut as well.

Consider a cut *S* that consists of node s, few of the teams ST and few of the players SP, so S = {{s}, ST, SP} and rest all other teams and players are in cut T.

Capacity of the above S-T cut would be summation of 2 for all teams that are not in S cut plus summation of the players that teams ST prefers (1 if player is inside and 1 if he is outside as in both cases cut will be crosssed).

$$S_c = \sum_{t \notin ST} 2 + \sum_{t \in ST \,\&\, p \in SP} 1 + \sum_{t \in ST \,\&\, p \notin SP} 1$$

$$S_c \geq \sum_{t \notin ST} 2 + \bigcup_{t \in ST} A_t$$

(Let us assume that all the teams are in the S cut then the equation would be …)

$$S_c \geq \sum_{t \in T} 2$$

$$S_c \geq 2*|T|$$

$$S_c \geq 2*n$$

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.

Logic –

Convert given problem into Bipartite Graph

Apply Bipartite Matching and find minimum edges required to maximize the flow

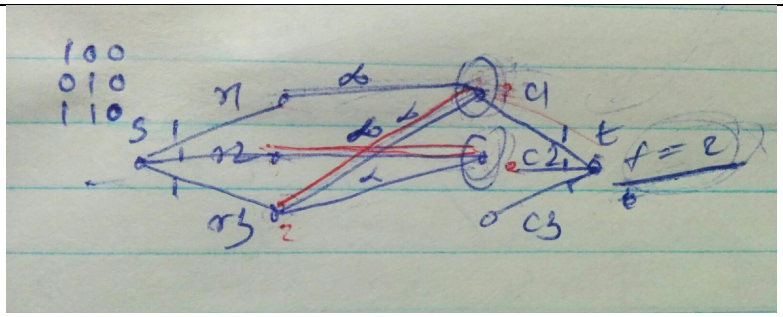
For each vertex in each edge that is in integral flow, select the max out degree vertex add it in a set

Return the set.

Prerequisites – ***output is Graph G***

1) Convert each row to a node ($d_i \mid i = \{1, 2, \ldots n\}$) on left side of bipartite graph
2) Convert each column in matrix to a node ($t_j \mid j = \{1, 2, \ldots n\}$) on right side of bipartite graph
3) (a) Create a source node *s* on left most side
   (b) Draw an edge between source node and every row node (d_i)
   (c) Assign capacity 1 (value does not matter) to the edge
4) (a) Create a sink node *t* on right most side
   (b) Draw an edge between sink node and every column node (t_j)
5) (c) Assign capacity 1 (value should be same as that of step 3. c) to the edge
6) (a) Draw an edge between row d_i and column t_j, if value of cell (d_i, t_j) is 1.
   (b) Set the capacity of the edge to Infinity

Graph –



Answer –

1) For given graph G, find the integral flow path (F) and maximum flow value (V) using bipartite matching principle.
2) For each edge that is present in the integral flow path find the vertex with highest degree and add it in the required node list and if both edges has same degree then save that edge in the equal degree edges list.

```
For each edge e(ri, cj) ∈ F
    if (e.ri.degree == e.cj.degree)
        equalDegreeEdges.add(e)
    else
        requiredNode.add( maxDegreeVertex(e.ri.degree, e.cj.degree) )
        exploredNodes.add(maxDegreeVertex (e.ri.degree, e.cj.degree) )
```

3) For each edge in the equal degree edges list select the vertex that has more unexplored neighbouring vertex and add it in the required nodes list. In case of both vertex has same unexplored neighbours then in that case select any of the vertex.

```
For each edge e(ri, cj) ∈ equalDegreeEdges
    if (e.ri.unexploredNeighbours == e.cj. unexploredNeighbours)
        randomSelect(e.ri, e.cj)
    else
        requiredNode.add(
            maxUnexploreVertex (e.ri. unexploredNeighbours, e.cj. unexploredNeighbours) )
```

4) Return the list of required nodes we found above

```
return requiredNode
```

Complexity –

1) Time required to solve bipartite matching + E
   Thus running time complexity is almost equal to bipartite matching solving (using network flow) problem

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 –

Here red black tree is used. As insert and search operation in reb black tree is done in logn time the time complexity of creating a rbtree and creation of hierarchy will take O(nLogn)

```
class NewEmployee    {

        String bossN;
        ArrayList<String> reportees;

        public NewEmployee(String b, ArrayList<String> r){
                bossN = b;
                reportees = r;
        }
}

class Employee{

        String emp;
        Employee boss;

        public Employee(String e, Employee b)     {
                boss = b;
                emp = e;
        }
}

RedBlackTee<NewEmployee> rbtree;

public RedBlackTree<NewEmployee> createRbtree (String[][] inputEmpList){

        for (i in inputEmpList)     {
                ArrayList<String> rep;
                String employeeName = inputEmpList[i][0];
                String bossName = inputEmpList[i][1];
                if (rbtree.has(bossName))   {
                        NewEmployee ne = rbtree.search(bossName);
                        ne.reportees.add(employeeName);
                }
                else   {
```

```
                        rep.add(employeeName);
                        NewEmployee boss = new NewEmployee(bossName, rep);
                        rbtree.add(boss);
                    }
                    NewEmployee e = new NewEmployee(employeeName, {});
                    rbtree.add(e);

            }
            return rbtree;

    }

    Employee rootCeo = new Employee(ceoName, null);
    public void createHierarchy (RedBlackTree rbtree,  Employee rootCeo)   {

            ArrayList<Employee> reporteeList;
            NewEmployee ceo =  rbtree.search(rootCeo.emp);
            for (String emp : ceo.reportees)    {
                    Employee employee = new Employee(emp, rootCeo);
                    reporteeList.add(employee);
            }
            while(!reporteeList.isEmpty())     {
                    createHierarchy(rbtree, employee);
            }

    }

   return  rootCeo;
```

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

Answer –

Instead of creating a Red Black Tree now we will create a hash map with key value as boss name and values as the names of reportees. As we can insert and retrieve data from hashmap in constant time. Thus the hierarchy buiding operation can be done in O(n)

```
    HashMap<String, ArrayList<String>> rbtree;

    public HashMap<String, ArrayList<String>> createRbtree (String[][]
inputEmpList){

            for (i in inputEmpList)     {
                    ArrayList<String> rep;
                    String employeeName = inputEmpList[i][0];
                    String bossName = inputEmpList[i][1];
                    if (rbtree.contains(bossName))     {
                            NewEmployee ne = rbtree.get(bossName);
                            ne.reportees.add(employeeName);
                    }
                    else    {
                            rep.add(employeeName);
                            NewEmployee boss = new NewEmployee(bossName, rep);
                            rbtree.add(boss);
                    }
                    NewEmployee e = new NewEmployee(employeeName, {});
```

```
                    rbtree.add(e);

                }
                return rbtree;

        }
```

6. (5 points) Prize Collecting Path

You are given a graph G = (V, E) and a subset S ⊆ 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 –

1) Proof given problem is NP :-
   Suppose we are given a path that solves above problem, then we can check whether given solution is correct or not. To do so, we just need to pass through each edge in the solution and check whether it exists or not. Thus, if there is a valid path then we can easily check it in polynomial time. Hence it is NP problem.
2) Proof given problem is NP-hard :-
   Given problem can be shown as NP-hard, by proving that it is equivalent to finding an undirected Hamiltonion path.
   For a given graph G, take each vertex vi ∈ S add it in graph G`, then add all edges (u, v) ∈ G such that either u ∈ S or v ∈ S in graph G`. Then for the sink nodes if there exists path between then include a direct edge between such sink nodes in G`. Finally, fix a vertex (sink if available) vi ∈ S as Source node and other node vj ∈ S as a Sink node.
   Now finding a way between the source and sink vertex in the graph G` is equivalent to a Hamiltonian path. If there is no way to reach, then there is no Hamiltonian path in G.
3) Since we can reduce from Hamiltonian path to our prize node collection problem, we know that the given problem is NP-complete.