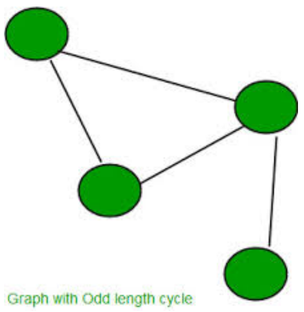
Analysis of Algorithms – HW5

Li-Hen Chen

Department of Computer Science & Engineering, Texas A&M University

1.

First of all, let me introduce what is an odd cycle. An odd cycle means a cycle with an odd number of vertices is called an odd cycle as the below figure shown. There is a lemma that a graph is bipartite if and only if it has no odd cycle. As a result, if a vertex v in an undirected graph G is an odd cycle transversal, then this graph is not bipartite. An odd cycle transversal means that if every cycle of odd length in G contains the vertex v. In order to decide if v is an odd cycle transversal, I decide to use Breath First Search to find if v is an odd cycle transversal.



Algorithm:

Given a graph G and a vertex v

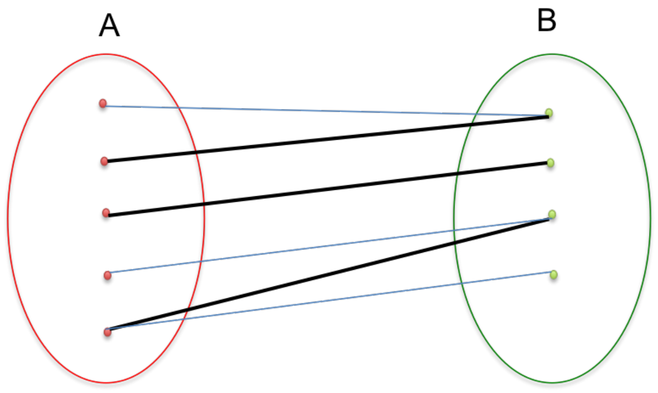
1. Starting from the vertex v using Breath First Search.
2. Record the length (level) and the node when traversing each node.
3. When BFS finishes, look at the table of the lengths of every paths starting from vertex v and ending in vertex v, if lengths of every paths are "odd", output "v is an odd traversal", otherwise output "v is not an odd cycle transversal".

Time Complexity:

This algorithm is only using Breath First Search. As we all know, the time complexity of BFS is linear which can be expressed as $O(|V| + |E|)$, since every vertex and every edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

2.

The problem indicates that each class C_i has an enrollment r_i while each class room R_j has a capacity c_j and a classroom R_j is "feasible" for a class C_i if $\frac{c_j}{2} \leq r_i \leq c_j$. As a result, we can use the "Maximum Bipartite Matching" algorithm to assign the class to proper classroom. As we know the enrollment of each class, we can connect the classes and the classrooms where the capacity of each classroom is feasible for class as below figure. After connecting the classes and the classrooms, we can use the "Maximum Matching Algorithm" to make a feasible assignment of classes to the classrooms such that the as many classes as possible can get held.



Matching

Algorithm

1. Connect the classes and the classrooms where the capacity of each classroom is feasible for the class as Graph G

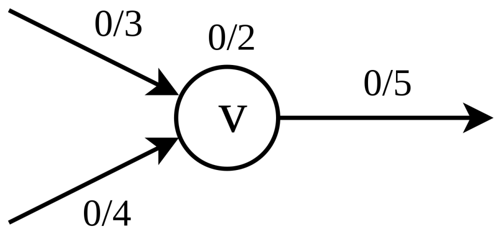
2 . Run the"MaximumBipartiteMatching (G)", so we can get the maximum bipartite for classes and classrooms.

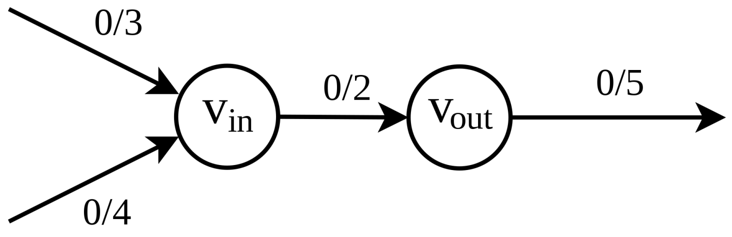
```
MaximumBipartiteMatching(G)
    initialize set $M$ of edges // can be the empty set
    initialize queue $Q$ with all the free vertices in $V$
    while not $Empty(Q)$ do
        $w \leftarrow Front(Q)$
        if $w \in V$ then
            for every vertex $u$ adjacent to $w$ do // $u$ must be in $U$
                if $u$ is free then // augment
                    $M \leftarrow M$ union $(w, u)$
                    $v \leftarrow w$
                    while $v$ is labeled do // follow the augmenting path
                        $u \leftarrow$ label of $v$
                        $M \leftarrow M - (v, u)$  // $(v, u)$ was in previous $M$
                    v ← label of u
                    M ← M union (v, u) // add the edge to the path
                         // start over
                    remove all vertex labels
                    reinitialize Q with all the free vertices in V
                         break // exit the for loop
               else // u is matched
                    if (w, u) not in M and u is unlabeled then
                         label u with w // represents an edge in E-M
                         Enqueue(Q, u)
                         // only way for a U vertex to enter the queue
          else // w ε U and therefore is matched with v
               v ← w's mate // (w, v) is in M
               label v with w // represents in M
               Enqueue(Q, v) // only way for a mated v to enter Q
```

Time Complexity:
The first step only connects the classes and classrooms so the time complexity is O(|E|), which is linear. At the second step, however, each of iteration matches two free vertices, one from V and one from U. So the total number of iterations cannot exceed floor(n/2)+1, where n = |U|+|V|. The time spend on each iteration is O(n+m), where m = |E|, assuming that labeling and determining free or mate are constant. Note that augmenting cost O(m) and is performed only once during an iteration. So the time complexity is $O(n^2 + nm)$.

3.
Now given a graph that not only the edges have capacities its vertices also have capacity. In other words, the amount of flow passing through a vertex cannot exceed its capacity. To find the maximum flow across *N*, we can transform the problem into the maximum flow problem in the original sense by expanding *N*. First, each $v \in V$ is replaced by v_{in} and v_{out}, where v_{in} is connected by edges going into *v* and v_{out} is connected to edges coming out from *v*, then assign capacity $c(v)$ to the edge connecting v_{in} and v_{out} as below figure. In this expanded network, the vertex capacity constraint is removed and therefore the problem can be treated as the original maximum flow problem. Now we

can solve the problem by Ford–Fulkerson algorithm, which can solve the Maximum flow problem.





Algorithm:

1. Modify graph G. Each $v \in V$ is replaced by v_{in} and v_{out}, where v_{in} is connected by edges going into *v* and v_{out} is connected to edges coming out from *v*, then assign capacity $c(v)$ to the edge connecting v_{in} and v_{out}.

2. Run Ford–Fulkerson on graph G.

```
    1. $f(u,v) \leftarrow 0$ for all edges (u, v)
    2. while there is a path p from s to t in $G_f$, such that $c_f(u,v) > 0$ for all edges (u, v)
        1. Find $C_f(p) = \min\{c_f(u,v): (u,v) \in p\}$
        2. For each edge $(u,v) \in p$
            1. $f(u,v) \leftarrow f(u,v) + c_f(p)$
            2. $f(v,u) \leftarrow f(v,u) - c_f(p)$
```

Time Complexity:

For the step 1, the running time depends on the number of edges. So the time complexity of step 1 is O(|V|), which is linear. For the step 2, the Ford-Fulkerson algorithm. By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford–Fulkerson is bounded by O(Ef), , where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O(E) time and increases the flow by an integer amount of at least 1, with the upper bound f. But if in step 2.2, we use BFS, the time complexity can be reduced to $O(VE^2)$.

4.

Suppose we already have a maximum flow f. Consider a new graph G where we set the capacity of edge (u, v) to f(u, v). Run Ford-Fulkerson, with the modification that we remove an edge if its flow reaches its capacity. In other words, if f(u, v) = c(u, v) then there should be no reverse edge appearing in residual network. This will still produce correct output in our case because we never exceed the actual maximum flow through an edge, so it is never advantageous to cancel flow. The augmenting paths chosen in this modified version of Ford-Fulkerson are precisely the ones we want. There are at most |E| because every augmenting path produces at least one edge whose flow is equal to its capacity, which we set to be the actual flow for the edge in a maximum flow, and our modification prevents us from ever destroying this progress.