CSCE 629: Analysis of Algorithms
Home Work - 5

Shabarish Kumar Rajendra Prasad
UIN - 228000166

November 2019

1 Question 1

The algorithm is shown below:

Algorithm: oddCycleTraversal $(G = \{V, E\}), v$

```
if is_bipartite(G):
    return False

else:
    G1 = G-v
    if is_bipartite(G1):
        return True
    else:
        return False
```

We know that a graph is bipartite if and only if it does not contain any odd cycles. We also know that we can determine whether a graph is bipartite in linear time using DFS. Thus we first determine whether the graph is bipartite. If it is bipartite, then we know that there are no odd cycles and thus we return False. If not, then we remove the vertex v from the graph G and form a new graph $G1$ which can also be done in linear time. Now we determine whether $G1$ is bipartite. If $G1$ is bipartite, then v is an odd cycle traversal in G. If the graph $G1$ is also not bipartite then v is not an odd cycle traversal in G.

Time complexity

Since both finding whether the graph is bipartite and removing a vertex from the graph can be done in linear time, the time complexity of the entire program is linear.

2 Question 2

First we shall construct a bipartite graph, $G = \{C, R, E\}$ where all the classrooms from $C = \{C_1, C_2, ..., C_m\}$ and all the rooms from $R = \{R_1, R_2, ..., R_n\}$ constitute the vertices of the graph. Here each class C_i in C is associated with an enrollement number r_i and each room R_i in R is associated with a maximum class capacity c_j. The edges of the graph are drawn between the classes and rooms which satisfy the condition $c_j/2 \leq r_i \leq c_j$. Then we shall perform graph matching on this bipartite graph to assign the rooms for classes.

Time complexity

The construction of the bipartite graph takes time of the order $O(mn)$. Then the maximum matching algorithm runs in time $O(|V| * |E|)$. Here the number of vertices in this graph is equal to

$|V| = m + n$ and the number of edges is $|E| = m * n$. Thus the complexity of maximum matching algorithm is $O(n^2m + nm^2)$, which describes the overall time complexity of the program.

3 Question 3

Let $G = \{V, E\}$ be the flow network with both vertex and edge capacities. From this graph, we can construct a new graph $G' = \{V', E'\}$ without vertex capacities. We can do this in the following way. For every vertex v in the graph G, that has a edge capacity, we split that vertex into two vertices v_1 and v_2 in G' and add an edge between v_1 and v_2, with an edge capacity equal to that of the vertex capacity in graph G. For every vertex v that is split in G', if there exists an edge (v, u) in G, then in G' we insert the edge (v_2, u) with the same edge capacity of the edge between (v, u) in G. This makes sure that the value of maximum flow the new graph G' is the same as the value of maximum flow in the graph G. The source in G' will be vertex s_1 while the sink will be t_2.

If f is the flow from s to t in G, then the corresponding flow f' from s_1 to t_2 in G' is created as follows. Every flow in G on the edge (u, v) will be $f(u, v)$, we insert the flow $f'(u_2, v_1)$ such that $f'(u_2, v_1) = f(u, v)$. For each vertex v in G such that $v \neq s$, $f'(v_1, v_2) = \sum_{(u,v)\in E} f(u, v)$ and for the start vertex s, let $f'(s_1, s_2) = \sum_{(s,u)\in E} f(s, u)$. Now we can see that f' is a valid flow in G' and this way we know the flow f' satisfies both the vertex and edge capacity constraints.

Thus the flow f' is the flow in G' corresponding to f in G.

4 Question 4

The algorithm is given below:

Algorithm: isBipartite $(G = \{V, E\}), v$

```
    for each vertex v in G:
        color[v] = white
    for each vertex v in G:
        if color[v] == white:
            group[v] = 0
            if ( not DFS(v)):
                return False
    return True

def DFS(vertex):
    color[v] = grey
    for each edge (v,u) in G:
        if color[u] == white:
            if group[v] == 0:
                group[u] = 1
            else:
                group[u] = 0
            if (not DFS(u)):
                return False
        else:
            if group[v] == group[u]:
                return False
    color[v] = black
    return True
```

The working of the algorithm is simple and is very similar to finding whether the graph is bipartite when it is directed. We run a DFS on the undirected graph and try assigning them to two different groups. Whenever the neighboring vertices of a graph belong to the same group, we return False. Else, if all the neighboring vertices belong to different groups, we return True.

Time complexity

Since the algorithm simply runs a depth first search on the graph, the time complexity of the algorithm can be defined as $O(|V| + |E|)$, which is linear.