CS403: Algorithm Design and Analysis

Feb-Jun 2017

Indian Institute of Technology Mandi

Assignment 1

Priyansh Saxena

B14118

Computer Science and Engineering

CS 403 – Algorithm Design and Analysis

Assignment 1

Problem 1: The problem requires a solution for the stable-matching of equal number of men and women who have each given their priorities of the people of opposite gender.

The algorithm used to solve the problem is called the Gale-Shapley algorithm. The outline of the algorithm is as follows:

Initially all $m \in M$ and $w \in W$ are free

While there is a man m who is free and hasn't proposed to every woman

Choose such a man m

Let w be the highest-ranked woman in m 's preference list to whom m has not yet proposed

If w is free then

(m, w) become engaged

Else w is currently engaged to m'

If w prefers m' to m then

m remains free

Else w prefers m to m'

(m, w) become engaged

m' becomes free

Endif

Endif

Endwhile

Return the set S of engaged pairs

The space complexity of the algorithm is $O(n^2)$, as we need to keep track of the priorities of all the men for all the women and vice-versa.

The time-complexity of the algorithm is $O(n^2 \log n)$, which can be reduced to $O(n^2)$ by using arrays instead of priority-queue. This can be shown by the following arguments.

1. Every man needs to propose at least one and at most n women. On an average, every man proposes n/2 women. Therefore, proposals alone take $O(n^2)$ time.

2. For every break-up of a man-woman pair, a man needs to be inserted into the priority-queue. Now, the i-th man in order can face upto n-i rejections before getting a stable match. Therefore, $O(n^2)$ break-ups can happen.

3. Now, insertion into priority-queue is an operation of order O(logn). This makes the total time-complexity $O(n^2 logn)$.

Implementation-based assumption: There are n men and n women, represented by natural numbers from 1 to n separately for men and women.

Problem 2: The problem requires a method to detect a cycle in an undirected graph.

The algoithm used to solve the problem is a modified version of complete-DFS of the graph. Complete-DFS of the graph is used to determine all the connected components of the graph.

The outline of the algorithm is as follows:

```
let S be a stack
while there is a non-visited vertex
    let V be the first non-visited vertex in the graph (by some arbitrary order)
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not visited:
            label v as visited
            for all edges from v to w in G.adjacentEdges(v)
                if w is visited and is not the parent of v
                    set cycleDetected to true
                    printCycle(v,w)
                else if w is not visited
                    S.push(w)
                    set parent[w] to v
```

```
printCycle(start,end)
    BEGIN
    while start is not last
        print start
        start = parent[start]
    print last
    END
```

The space-complexity of the algorithm is O(n+m), where n is the number of the vertices and m is the number of edges in the graph. This is because 2*e values are stored in the adjacency-list represenation of the graph and n values are stored to keep track of which vertices are visited and their parents.

The time-complexity of the algorithm is O(n+m). this can be shwon with the following arguments.

1. The time-complexity of DFS is O(n+m).

2. If we keep track of the vertex from which the last DFS was initiated, the total time taken to determine the next non-visited vertex in the case of termination of a single DFS is of O(n), over all iterations of complete-DFS.

3. The parent of a newly discovered but not visited node is determined in O(1). Similarly, the check for cycle by asserting that a visited vertex is not the parent of the current vertex is done in O(1). These are the only modifications to the original DFS algorithm.

Therefore the running-time of the algorithm is O(n+m).

Problem 3: Determine if a given node ‘w’ can be reached from another given node ‘v’ in the graph, and the minimum number of steps it can be done, if at all.

The algorithm used to solve the problem is a standard BFS over the graph. BFS generates the connected component of the graph that contains the start vertex. If an end vertex is reachable from the start vertex, it will also be present in the connected component and is guaranteed to give the minimum possible distance from the start vertex. The algorithm is given as follows.

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    for each unvisited neighbour w of u

        set distance[w] to distance[u]+1

        put w into Q

The space-complexity of the algorithm is O(n+m), since we need to store the graph (as adjaceny-list) and the distances of all vertices from the start-vertex.

The time-complexity of the standard BFS is O(n+m). This is because every vertex has to be put into the queue at least once and all the edges of the vertex being visited currently are taken into consideration.

V1 + [edges from V1] + V2 + [edges from V2] + ...

Also, the final result is printed in O(1). Therefore the time-complexity of the algorithm is O(n+m).