Data Structures and Algorithms
Assignment 3 Solutions

Ayush Bansal
Roll No. 160177

April 1, 2018

I Problem 1 Solution

We are given a binary min heap H of size n, i.e. there are n nodes in the tree, a number x and a positive integer $k < n$.
I have to design a $O(k)$ algorithm to determine if x is smaller than k^{th} smallest element in H.
Since, I am given a binary min heap, the node value of the child will always be greater than or equal to that of the parent, thus if $x < ancestor[i].val$ then $x < i.val$ as $ancestor[i] \leq i$.
Also, we can say that if I can find k nodes such that $val \leq x$ then, it is not possible for the k^{th} smallest element to be bigger than x.
So, I will traverse the tree from the root recursively and I will **return true** when a $tree.val > x$ otherwise I will increment the counter as $tree.val \leq x$ and traverse that tree further.
The algorithm will be as follows

Algorithm 1 x is smaller than k^{th} element or not

```
procedure SMALLORNOT(Node* n)          ▷ takes pointer to node & returns Boolean
    if $n = NULL$ then                  ▷ Imitates $\infty$ in case of binary min heap
        return true
    end if
    if $n.val > x$ then                 ▷ $x$ is global variable
        return true
    end if
    $count++$                           ▷ $count$ is global variable, initially 0
    if $count \geq k$ then              ▷ $k$ is global variable
        return false
    end if
    return (SMALLORNOT(n.left)) and (SMALLORNOT(n.right))
end procedure
```

The procedure will be called with the **pointer to root** of the tree.

Proof Of Correctness

Proof. The algorithm is traversing the tree from root to the bottom and as soon as it finds a node which is greater than x it returns **true** for that subtree as the all the values in that subtree will be greater than x and don't need to be tested.
Also the **count** is incremented only when an element is found smaller than or equal to x which is exactly what needs to be counted.
The above procedure returns **false** if we can find atleast k elements smaller than or equal to x as in this case it is not possible that x is smaller than k^{th} smallest element.
The above procedure returns **true** if less than k elements of the tree are smaller than or equal to x which is as it should be. □

Proof Of Time Complexity

Proof. The worst case complexity will occur when at each level it has to reject a subtree (let it be left), so assume at every height of heap, $left.val > x$ and thus we don't need to traverse that tree and the $count$ will not be incremented.
In the above case the time will come to be

$$\begin{aligned}
T(k) &= 1 + (2 + 2 + 2 + \dots k - 1 \text{ times}) \\
T(k) &= 1 + 2(k-1) \\
T(k) &= 2k - 1
\end{aligned}$$

In any other case, the $count$ will be incremented more or equally frequently, thus the worst case time will be **c(2k-1)** which gives us $O(k)$ complexity. □

II Problem 2 Solution

2.1 Part 1

We are given an undirected graph $G = (V, E)$ and I have to find whether it is acyclic of not.
Firstly, I will check if the graph has less than $|V|$ edges i.e. $|E| \leq |V| - 1$ as if its not the case, then graph cannot be acyclic according to the definition of **undirected acyclic graphs**.
If the graph has less than $|V|$ edges, then I will use the following algorithm to find if it is acyclic or not.
Since, the graph is undirected, I am assuming there will be no self loops.
We will define a **parent** array which will be of $|V|$ length and initialize it to -1 or any other value which is not a node.
Now, by traversing each edge of the graph one by one, we will find the parent of each and every node of the graph and if there is a clash at any point, that means we have encountered a cycle.
The parent-finding algorithm will be as follows

Algorithm 2 Find Parent

```
1: procedure FINDPARENT($parent[], node$)                ▷ This will recursively find parents
2:     if $parent[node] = -1$ then
3:         return node
4:     end if
5:     return FINDPARENT(parent, parent[node])
6: end procedure
```

The cycle finding algorithm will be as follows

Algorithm 3 Acyclic or not

```
1: procedure ISACYCLIC(Graph g)
2:     if ( then$|E| \geq |V|$)
3:         return false
4:     end if
5:     for all edges in g do
6:         $x \leftarrow$FINDPARENT(parent, edge.src)
7:         $y \leftarrow$FINDPARENT(parent, edge.dest)
8:         if $x = y$ then
9:             return false
10:        end if
11:        $x \leftarrow$FINDPARENT(parent, x)
12:        $y \leftarrow$FINDPARENT(parent, y)
13:        $parent[x] \leftarrow y$
14:    end for
15:    return true
16: end procedure
```

Time Complexity
The time complexity of the above algorithm will be $O(|E|)$ but, since the edges are less than $|V|$ i.e. $|E| \leq |V|$.
The algorithm becomes $O(|V|)$ which is what we wanted

2.2 Part 2

We will use the algorithm defined in previous part i.e. 3 to first determine that the given graph is **acyclic** or not, as by the definition of a tree, it is a **connected acyclic graph**.
If graph is **cyclic** then we simply **return false**, as it can't be a tree.
If graph is **acyclic** then we check if it has exactly $|V|-1$ edges or not as a tree with $|V|$ nodes contains exactly $|V|-1$ edges (no more no less).
The algorithm for the procedure **treeOrNot**

Algorithm 4 Graph is Tree or not

```
procedure TREEORNOT(Graph g)                              ▷ returns a Boolean
    if ISACYCLIC(g) then
        $count \leftarrow 0$
        for all edges $\in g$ do                          ▷ Getting the total num of edges
            $count \leftarrow count + 1$
            if $count > (|V| - 1)$ then
                break
            end if
        end for
        if $count = (|V| - 1)$ then
            return true                                   ▷ Total num of edges are $|V| - 1$
        end if
    end if
    return false
end procedure
```

The proof of correctness is clear from the algorithm, as the algorithm clearly finds if the graph is **connected acyclic graph** or not which is the definition of the **Tree**.

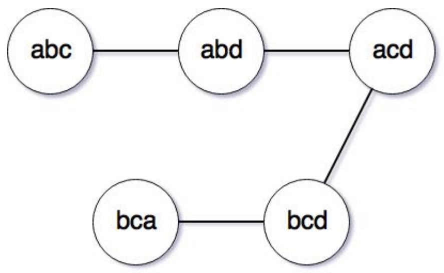
Time Complexity
The time complexity of **isAcyclic** Procedure is $O(|V|)$ and the iterations in finding if the number of edges are $|V|-1$ or not are also $O(|V|)$ which gives us the time complexity $O(|V|)$.

III Problem 3 Solution

We are given a **dictionary** D of words, where each word is string of English alphabets.
Two words X and Y in the dictionary are related or connected if Y can reached from X by single insertion, deletion or replacement of an alphabet or vice versa (as per Allowed edits).
Given 2 words A and B in the dictionary, I have to find the minimum number of allowed edits to reach from A to B.

This can be thought of a graph problem such all the **words** in the dictionary are **nodes** and there is an **edge** between 2 nodes if they are **connected** according to **allowed edits**
While building up the graph we will compare each word with other words and add edges between nodes which are seen connected.
Also, notice according to allowed edits, if you can move from X to Y, then you can also move from Y to X, which means the graph will be **undirected**.
For the dictionary (array of words) given as an example

$$D = \{abc, bcd, acd, abd, bcd, bca, bac\}$$



Above graph will be formed from the given dictionary.
Below is the algorithm for constructing the graph.

Algorithm 5 Making graph from dictionary

```
1: procedure BUILDGRAPH(Dict D)                                ▷ Takes dict. as input
2:     for all words in D do                                   ▷ Traversing dictionary
3:         for all n_words after current word do   ▷ Comparing all words occuring after current
4:             $flag$ ←COMPARE(word, n_word)
5:             if $flag = true$ then                           ▷ Adding edge to undirected graph
6:                 ADDEDGE(word,n_word)
7:                 ADDEDGE(n_word,word)
8:             end if
9:         end for
10:    end for
11:    return Graph
12: end procedure
```

Now, since the graph is constructed and we will traverse the graph to find the path with minimum number of allowed edits, in short the shortest path between the nodes A and B in this graph.
Since we are going from A to B, we will pick A to be our initial vertex and apply BFS algorithm with source vertex as A to find the shortest path to B as follows

- All vertices adjacent to A will be at a distance of 1 from A.
- All vertices adjacent to the neighbours of A (but not A) will be at distance 2 from A.
- Following and expanding the above path keeping track of parents of nodes will give us the shortest path.

Now, there might be 2 possibilities here, either there exists a path between A and B or there doesn't.

- If there does exist a path between A and B, then it will be found by the algorithm as we are moving to adjacent levels (neighbours) one by one.
- If there doesn't exist a path between A and B, then the graph must have been disconnected and during traversal of component containing A, B will never be encountered and thus we can output "Path does not exist" after component containing A is traversed completely.

If we apply above algorithm to the mentioned example, we will get the path

$$abc, abd, acd, bcd$$

Proof of Correctness

Proof. Firstly, during constructing of undirected graph, we are comparing each node with an other node once, so it is not possible for a connection to be left out between 2 words.
Out of the 2 possibilities of existence of paths, I am assuming the first one because second is trivially proved as B will never be encountered during traversal and thus no path exists.
In case of existence of path, we are expanding from the source vertex A level by level as in the case of BFS, so we will reach the end of the component containing A and since B is part of that component, we will reach B eventually.
Also, at every point we are calculating the shortest path for every node from A so we will always lead to a path with shortest length when we reach B. □

Time Complexity

Solution. First finding out the time complexity of the **buildGraph** procedure, we are traversing the list word by word and then multiplying that with the number of words after that.
Also, assuming right now that complexity of **compare** procedure is k (depends on string size).

$$T(n,k) = k * ((n-1) + (n-2) + (n-3) + (n-4) + \cdots + 1)$$
$$T(n,k) = k * \frac{n(n-1)}{2}$$

Now, finding out the time complexity of the shortest path finding algorithm, the time complexity of BFS is $O(|V| + |E|)$, so the time complexity of this will also be $O(|V| + |E|)$.

Worst Case Time Complexity

The worst case time complexity for the **buildGraph** algorithm will be $O(cd^2)$ where d is number of vertices and c is the time complexity of compare statement.
The worst case time complexity for the **BFS** algorithm will be $O(cd^2)$ only as the max number of edges can be $\frac{d(d-1)}{2}$
So the worst case time complexity is

$$T(c,d) = O(cd^2)$$

□

IV Problem 4 Solution

In this question we are given a **hashtable** of size 10 and the hash function $h(g) = gmod10$. The collision resolution technique is Linear Probing.
There is insertion of 6 values in the table -> 42,23,34,52,46,33 not particularly in this order. After these insertions, table looks like this.

0	nothing
1	nothing
2	42
3	23
4	34
5	52
6	46
7	33
8	nothing
9	nothing

In case of Linear Probing if there is a clash, then the element is inserted in the next free slot.
So, depending on the table generated after 6 insertions, the following must be true:

$$\text{52 must come after } 42, 23, 34 \text{ as 52 clashes with } 2, 3, 4 \text{ slots} \tag{4.1}$$

$$\text{33 must come after } 23, 34, 52, 46 \text{ as 33 clashed with } 3, 4, 5, 6 \text{ slots} \tag{4.2}$$

Using eq (4.1) and (4.2), we get that

$$\text{33 will be inserted at last and 52 will be inserted at } 4^{th} \text{ or } 5^{th}\text{time.}$$

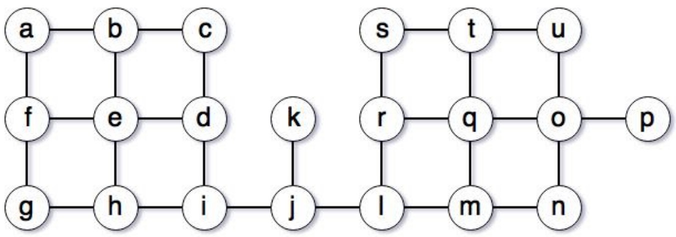
So, there can be 2 possible ways to insert 52 (46 may be inserted before or after 52) and there is only 1 way to insert 33.
If we insert 52 at the 4^{th} time, then the 3 numbers before this must be $\{42, 23, 34\}$ in any order which gives us 3! ways.
If we insert 52 at the 5^{th} time, then the 4 numbers before this must be $\{42, 23, 34, 46\}$ in any order which gives us 4! ways.
Thus, total number of ways these 6 integers can be inserted one after the other into the hash table are

$$3! + 4! = 30$$

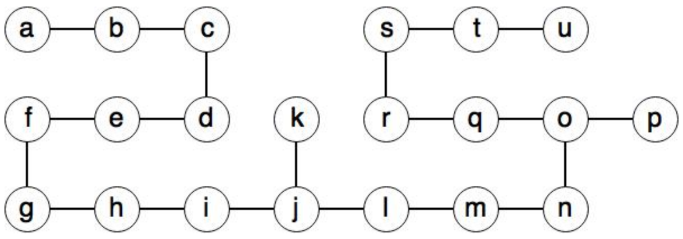
V Problem 5 Solution

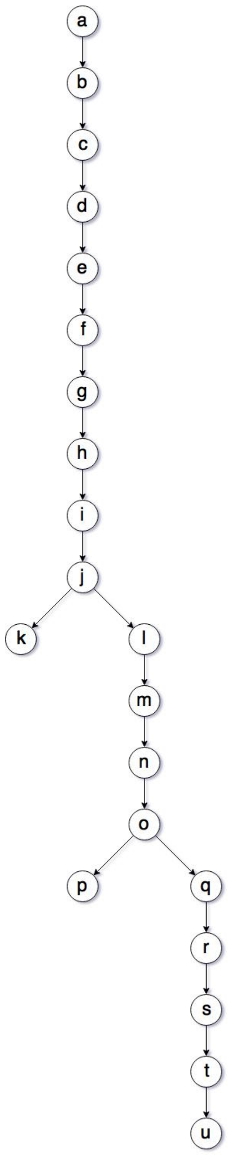
We are given the following graph and I have to find the a **DFS spanning tree** with **maximum possible height**.
Choosing **a** as the root vertex, applying DFS, we get the following **DFS tree** and **spanning tree by DFS**.

Graph



Spanning Tree by DFS





DFS tree

In DFS, when moving from one node to another, there is possibility of moving on any of the unvisited node.
If all of the node's neighbours have been visited then you move back to the node you came from, till you find a new node to visit. You keep on doing this procedure till all the nodes have been visited once and note the order of the traversal.
The different traversal paths while applying DFS algorithm, will get you many spanning trees out of which I have to report the spanning tree with the maximum height.
Since the number of nodes in this graph is 21, the maximum height $H \leq 21$.
While traversing we can notice that the nodes k and p are 1 degree nodes and thus they will form the leaves in the maximum height DFS tree, also they are connected to a node which is part of a cycle so they will branch out.
Thus, the maximum height $H \leq (21 - 2)$ and from the DFS tree figure on page 9 we can see that the maximum height of the tree is 19.

VI Problem 6 Solution

The problem consists of a set of islands and each island is connected to one or more islands via bridges.
Each bridge has a token associated with it which is used for crossing that bridge and each token is unique. Also there are a total of n bridges.
John (our person of interest) has been given n tokens (1 for each of the n bridges) and his desire is to cross all the bridges exactly once for which he needs to know which path to travel.

This problem can be considered as a graph problem where the islands are vertices and the bridges are edges and they form a connected graph.
The problem is to find a path in the graph such that we traverse all the edges exactly once.
If the graph contains a **Eulerian path** or a **Eulerian cycle** then that path or cycle will be the corresponding path which John needs to traverse to win, if there is no **Eulerian path** or **Eulerian cycle** in the graph, then John can't win.
Here is the Source for the proof of the above analogy and claim.

As per the source, for a graph to contain a **Eulerian path**, the number of odd degree vertices must be 0 or 2.
First we will check the above condition and if the above condition is not satisfied then John can't win as an Eulerian Path does not exist.
In an undirected graph, it is not possible to have only 1 node with odd degree, so we will use following algorithm to check if graph is **Eulerian** or not.

Algorithm 6 Eulerian Graph or Not

```
1:  procedure IsEulerian(Graph g)                    ▷ returns true or false
2:      $count \leftarrow 0$
3:      for all $v \in g$ do                         ▷ Here $v$ are vertices of graph
4:          if deg[v] is odd then                    ▷ deg[v] gives the degree of $v$
5:              $count{+}{+}$
6:          end if
7:      end for
8:      if $count > 2$ then
9:          return false
10:     end if
11:     return true
12: end procedure
```

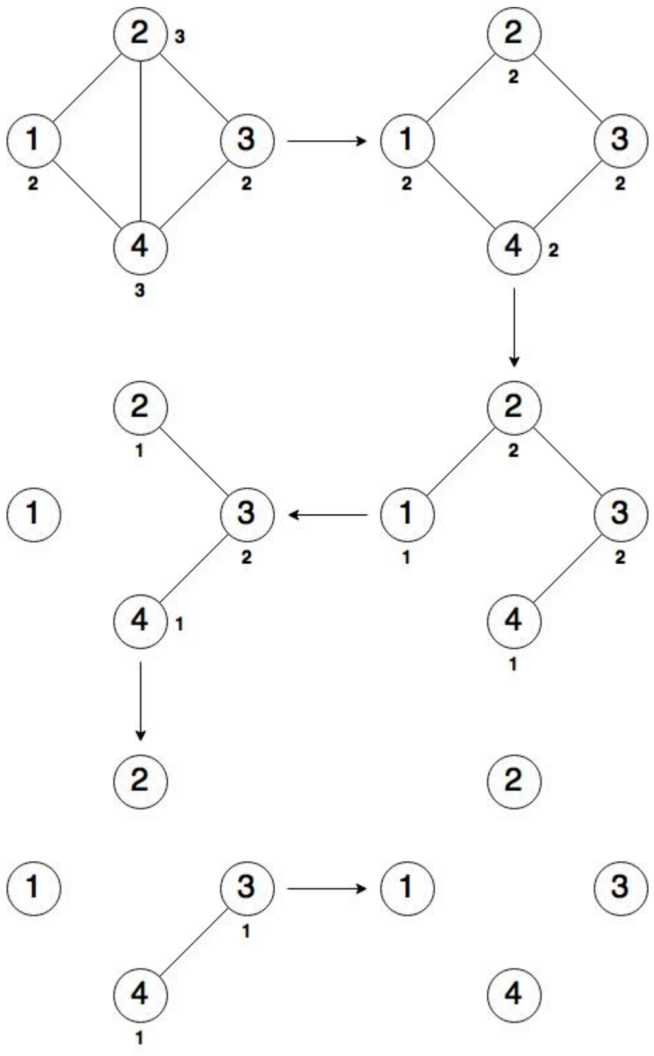
If the above procedure returns **false**, then John can't win but if it returns **true**, then John can win and we can use the following algorithm to find the path which he sould follow.

We will have to choose an initial vertex to begin with, i.e. the vertex from where we start traversing.
If there are 2 vertices with odd degree then initial vertex will be one of them, otherwise any vertex can be chosen.

Now, while degree of any vertex is non-zero (i.e. Edges exist in graph), follow the following steps.

- Move to a neighbouring vertex having max degree.
- Note the Traversed edge.
- Remove the traversed edge and reduce the degrees of its ends by 1 each.

Consider the following example



The **Eulerian Path** for the above example will be

$$(2,4) \quad (4,1) \quad (1,2) \quad (2,3) \quad (3,4)$$

The algorithm for finding the **Eulerian Path** is

Algorithm 7 Eulerian Path

```
procedure EULERIANPATH(Graph g)                       ▷ Prints the Eulerian path (order of edges)
    $a_v \leftarrow$ initial vertex                    ▷ Depending on 0 or 2 odd degree vertices
    while $\exists v$ such that $deg[v] \neq 0$ do     ▷ As long as there are edges
        $deg[a_v] \leftarrow deg[a_v] - 1$
        $b_v \leftarrow adj[a_v]$                      ▷ $b_v$ is the adjacent vertex with max degree
        $deg[b_v] \leftarrow deg[b_v] - 1$
        print $(a_v, b_v)$
        $del(E(a_v, b_v))$                             ▷ Removing traversed edge
        $a_v \leftarrow b_v$
    end while
end procedure
```

Proof of Correctness

Proof. According to algorithm, we are choosing the next vertex to be the vertex with max degree which means the vertex with degree 1 will be chosen when we have no other choice left.
Also there can't be a situation where we have to choose between 2 vertices such that both have degree 1 as it is only possible when either graph is **not Eulerian** or we didn't start with a vertex of odd degree even when the graph had initially.
So, at every point we will have a vertex with degree greater than 1 to go to unless we reach the last edge, at which point we traverse the last edge and we have been to each and every edge as all the other vertices have degree 0 (at which point while loop terminates). ☐

Time Complexity
The time complexity for the above algorithm will be $O((V+E)^2)$ or $O(E^2)$ which can broken down to $O((V+E)*(V+E))$ where $(V+E)$ is the time taken by **DFS** on adjacency list and it is done 2 times to find which node to move to next time.