COT5405 Analysis of Algorithms
Assignment 1

October 12, 2021

Sankalp Pandey
UFID: 92878142

Section 1

[50 marks] As part of this problem, you have to design and implement an algorithm to find a cycle (just one cycle) in an undirected graph.

1. Design a correct algorithm and show it in pseudo-code [10]

Ans:
(bool array visited of size number of vertex in graph with all values false)
(stack cycle to fill all vertex of cycle deteted)

Algorithm 1 Algorithm to find cycle in an undirected graph

```
Function FindCycle(source,parent,visited,cycle,graph):
    visited[source] ← true
    push cycle, source
    for each vertex v ∈ graph[source] do
        if visited[v] == false then
            if FindCycle(v, source, visited, cycle, graph) == true then
                return true
            else if parent ≠ v then
                push cycle, v
                return true
    end
    pop cycle
    return false
End Function
```

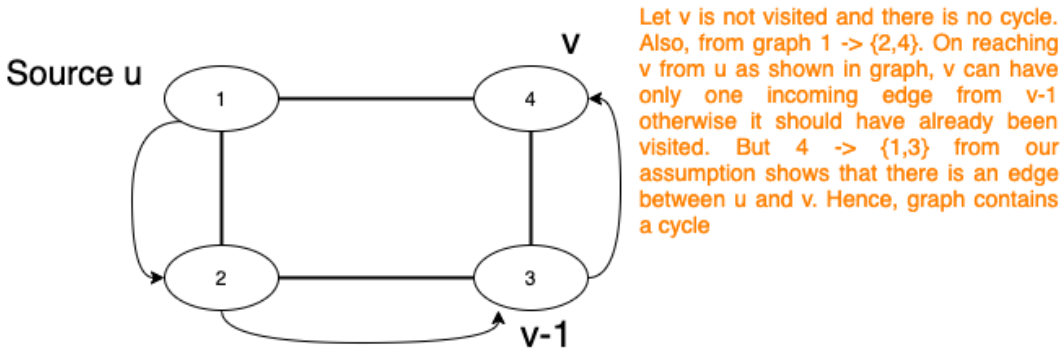

Figure 1: Proof of cycle detection visualization

2. Provide proof of the algorithm's correctness [10]

Ans:

(a) Proof by contradiction:
Suppose that the above graph G(V,E) does not contain cycle and $u \in adj(v)$ and $v \in adj(u)$ and $v - 1 \in adj(u)$
We start at a source vertex u and keep a boolean array to mark already visited vertices. We also maintain a parent pointer to keep track of parent of any vertex at any point of time. Now when we reach at vertex v, the parent contains v-1 if v is not visited. If v is not visited then no visited vertices before v-1 can have an edge with it. But we know that $u \in adj(v)$ from our representation. This means that graph G(V,E) has a cycle.

Proof of termination:

Case 1: If cycle exists
DFS algorithm will traverse each vertex once and push it to stack. If vertex is visited and the parent of previous vertex is not equal to current vertex, algorithm terminates with stack containing cycle nodes.

Case 2: If cycle doesn't exists
DFS algorithm traverse each vertex once and pushes the vertex in stack. If it doesn't finds cycle, it will pop each element and algorithm terminates with empty stack.

3. Find and prove the algorithm's running time [10]

Ans:
$|V|$ is number of vertex and $|E|$ is number of edges

Function `FindCycle`(*source,parent,visited,cycle,graph*)**:**

```
visited[source] ← true                                              O(1)
push cycle, source                                                  O(1)

for each vertex v ∈ graph[source]                                   O(|V| + |E|)
do
    if visited[v] == false then
        if FindCycle(v, source, visited, cycle, graph) == true then
            return true
        else if parent ≠ v then
            push cycle, v
            return true
end
pop cycle                                                           O(1)
return false
```

End Function

The running time of above algorithm is $\mathbf{O(|V| + |E|)}$. This is because in the above adjacency list representation of graph we discover each vertex and its neighbours exactly once. The time complexity to discover a vertex is O(1) and for its neighbours is O(degree(vertex)). In worst case, we discover all vertices to find a cycle. Hence, the time complexity will be:
$\Sigma(1 + degree(i))$ where $1 <= i <= V$
$\Sigma(degree(i)) = 2E$ as in an undirected graph there is incoming and outgoing edge from a vertex
Hence, time complexity will be $O(|V| + 2|E|) \sim O(|V| + |E|)$

4. Implement the algorithm in a compiled language and: [20]

Ans:

(a) Write a graph generator (Hint: use an existing graph generation library if you can)
Code attached in zip file

(b) Write test code to validate that the algorithm finds cycles
Code attached in zip file

(c) Test the algorithm for increasing graph sizes.
Code attached in zip file

(d) Plot the running time as a function of size to verify that the asymptotic complexity in step 3 matches experiments

How to run above the code in zip file

1. In terminal go to folder location where file *one_cycle.cpp* is present.
2. Run **make one_cycle**. This will generate executable file **one_cycle**.
3. Run **./one_cycle** to run the code.
4. Commandline will ask user to enter 1 for running algorithm and enter 2 to run the test code. Enter 1.
5. Next, commandline will ask to enter number of vertices. Enter 100.
6. Now commandline will ask to enter number of edges. Enter 200.
7. Algorithm will run to find a cycle. If cycle found, it will display the vertices of the cycle.
8. Run **./one_cycle** again. This time enter 2 to run testcode. Two test cases will run to show when cycle is present and other when cycle is not present.

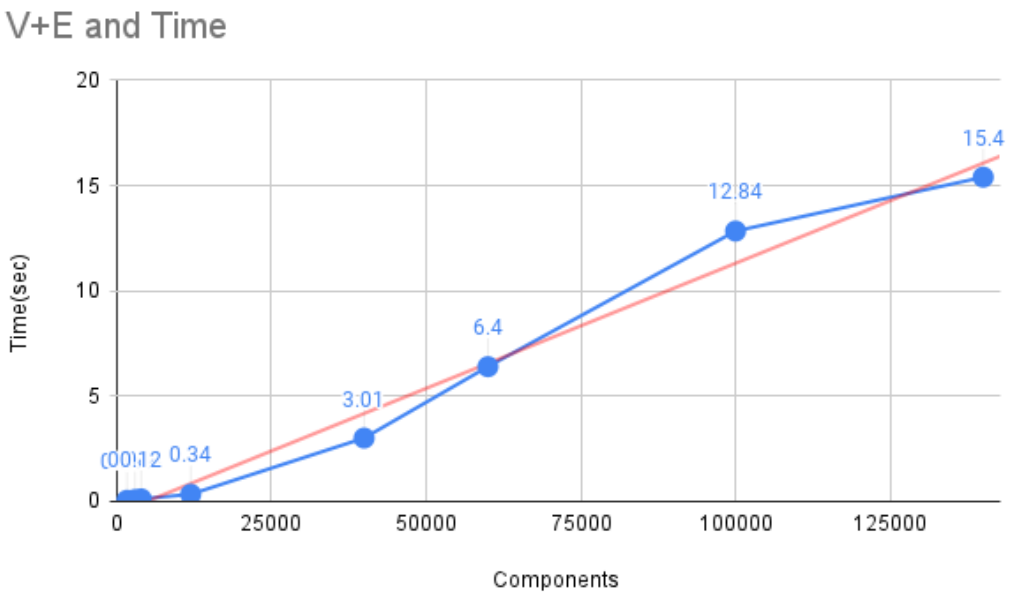

Figure 2: Plot of running time as a function of size

Section 2

For this problem, we consider undirected graphs that have n nodes and at most n+8 edges. For these graphs, you have to design an efficient algorithm that finds the minimum spanning tree.

1. Design a correct algorithm and show it in pseudo-code [10]

Ans:

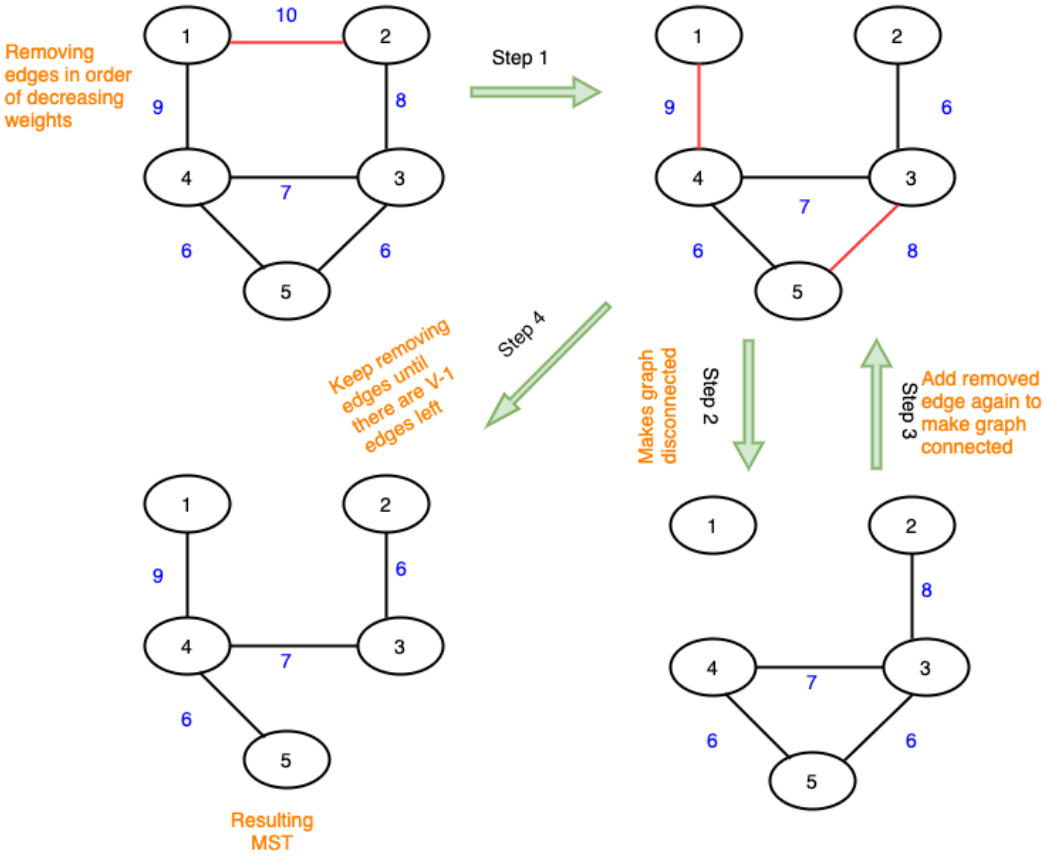

Figure 3: Algorithm to find minimum spanning tree

Algorithm 2 Algorithm to find minimum spanning tree

Function `FindMinimumSpanningTree`(*Edges*[] *E*, *vCount*)**:**

```
return if graph E is not connected
sort graph E based on decreasing order of edge weight
for i ← 0 ; i < size(E) ; i ← i + 1 do
    edge ← E[i]
    delete edge
    if graph is not connected then
        E[i] ← edge
end
return E
```

End Function

2. Provide proof of the algorithm's correctness [10]

Ans:
The provided algorithm deletes edges in order of decreasing cost. When any edge is removed the graph stays connected so it has be the most expensive edge on a cycle. The edges which are removed do not belong to any minimum spanning tree. The algorithm removes all cycles by deleting the most expensive edges. All edges that do not belong to a minimum spanning tree are removed.
When any edge is removed which causes the graph to become disconnected, it is replaced back in the graph. The algorithm does not remove any edges that causes graph to become disconnected. The graph produced by the algorithm has no cycles and is connected. The resulting graph is a tree and a minimum spanning tree.

Proof of correctness:
Part1: Resulting graph after edges deletion is a spanning tree The resulting subgraph from the algorithm is not disconnected as we are checking that in our algorithm. There is no cycle present since if it does then while iterating over the edges we would encounter the max edge in the cycle and we would delete that edge. Thus, resulting graph is a minimum spanning tree of original graph.

Part2: Minimality - Proof by induction:
If F is a set of edges remained after delete operations then there must be a spanning tree $T \subset F$. This is true as F contains all the edges and each weighted graph has a minimum spanning tree and thus it will be subset of F.
Let F be a non-final set of edges and $T \subset F$ be a minimum spanning tree, we need to show that after deleting an edge e in our algorithm, there exists a minimum spanning tree $T' \subset F$
1. If next deleted edge $e \notin T$ then $T = T' \subset F$. Hence, proved.
2. If $e \in T$. Deleting e causes T to become disconnected. Assume e separates T into sub-graphs S and V. Since the whole graph is connected after deleting e then there must exists a path between S and V so there must exist a cycle C in the F (before removing e). now we must have another edge in this cycle $f \notin T$ but $f \in F$ (since if all the cycle edges were in tree T then it would not be a tree anymore). So we show that T' = T - e + f is the minimum spanning tree $\subset$ F.

3. Find and prove the algorithm's running time [10]

Ans:

Function `FindMinimumSpanningTree(`*Edges*[] *E, vCount*`):`

return if graph is not connected	
sort graph E based on decreasing order of edge weight	$\mathbf{O}(\|E\|log(\|E\|)$
for $i \leftarrow 0$ *;* $i < size(E)$ *;* $i \leftarrow i+1$ **do**	
$edge \leftarrow E[i]$	$\mathbf{O}(\|E\|)$
delete edge	$\mathbf{O(1)}$
if *graph is not connected*	$\boldsymbol{O(}\|V\| + \|E\|\boldsymbol{)}$
then	
$E[i] \leftarrow edge$	$\mathbf{O(1)}$
end	
return E	

End Function

The running time of above algorithm is:
$\mathbf{O}(|E|log|E|)+\mathbf{O(1)}+\mathbf{O}(|E|)*(\mathbf{O(1)}+\mathbf{O}(|V|+|E|)+\mathbf{O(1)}) \sim \mathbf{O}(|E| \wedge 2)$
The time complexity of this algorithm can be made more efficient by improving the complexity of algorithm to find graph connectivity to $\mathbf{O}(mlogn(loglogn \wedge 3))$ as given by Mikkel Thorup

4. Implement the algorithm in a compiled language and: [20]

Ans:

(a) Write a graph generator (Hint: use an existing graph generation library if you can)
Code attached in zip file

(b) Write test code to validate that the algorithm finds cycles
Code attached in zip file

(c) Test the algorithm for increasing graph sizes.
Code attached in zip file

(d) Plot the running time as a function of size to verify that the asymptotic complexity in step 3 matches experiments

How to run above the code in zip file

1. In terminal go to folder location where file *mst.cpp* is present.
2. Run **make mst**. This will generate executable file **mst**.
3. Run **./mst** to run the code.
4. Commandline will ask user to enter 1 for running algorithm and enter 2 to run the test code. Enter 1.
5. Next, commandline will ask to enter number of vertices. Enter 5.
6. Now commandline will ask to enter number of edges. Enter 13.
7. Algorithm will run to find a mst and minimum cost. The algorithm will return minimum cost or -1 if graph is not connected.
8. Run **./mst** again. This time enter 2 to run testcode. Testcase will show that minimum cost is same as actual result.

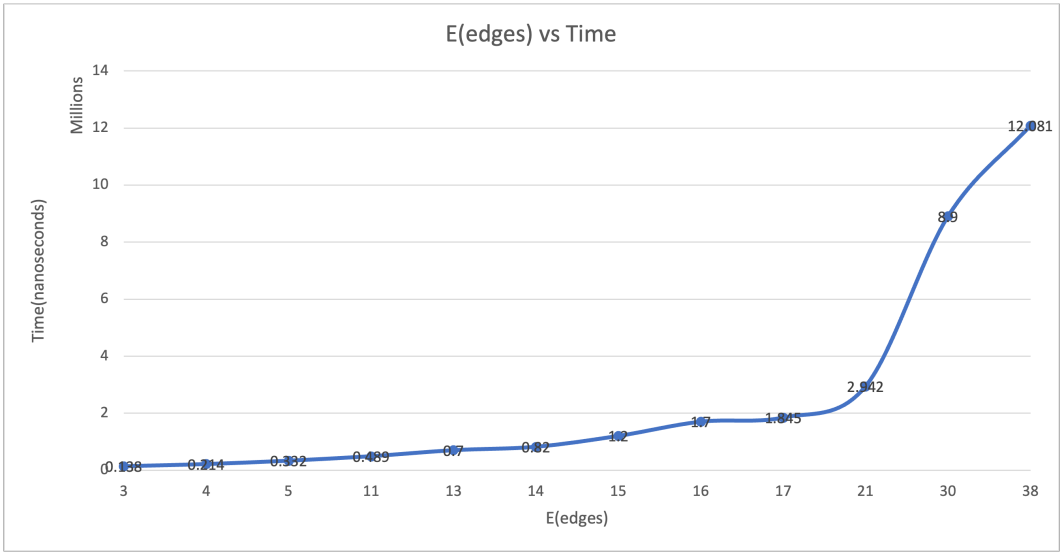

Figure 4: Plot of MST running time as a function of E(edges)