**Assignment No: 04**

**Name: S. M. Salah Uddin Kadir** **ID: 1800503**

**-----------------------------------------------------------------------------------------------------**

**[1] (Directed Graph, Programming) Write a C++ program to compute the longest path of a dag. As always, we are looking for the most efficient way. Please state the complexity of your algorithm.**

**Solution:**

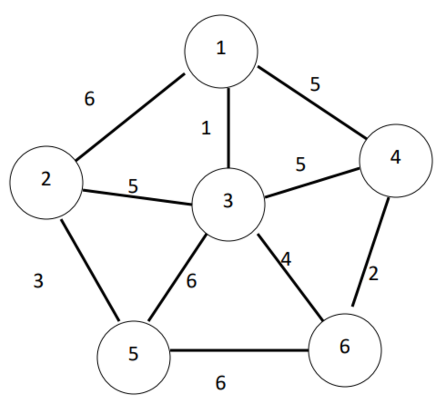
The cost for the main algorithm that finding out the longest path length,
Space cost = O (V + E).
Time complexity = O(V) * O (V + E)

Besides this, I used a matrix to keep track of the path, so the space cost for this, O($n^2$).
We can optimize it to O(path_length).

**[3] (Undirected Graph, Non-programming) (a) Describe an algorithm to enumerate all simple cycles of an undirected graph G. (b) How many such cycles can there be? (c) What is the complexity of your algorithm? To answer part (c), you may have to describe the data structure that you are using. Justification is needed to show why your algorithm works.**

**Solution:**

**(a)**




graph = {(1, 2), (1, 3), (1, 4), (2, 3), (3, 4), (3, 6), (3, 5), (2, 5), (5, 6), (6, 4)}
cycles = {}

Let,
'graph' is the set of edges and 'cycle' is the set of all simple cycles.

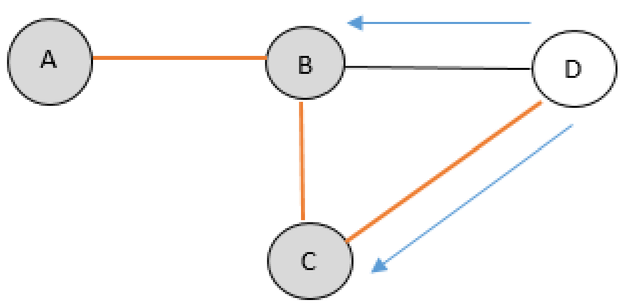
```
procedure find_new_cycle(path):
   start_node = path[0]
   next_node= None
   sub = []

   for each edge in graph:        // Visit each edge and each node of each edge
      node1, node2 = edge
      if start_node in edge:
            if node1 == start_node:
               next_node = node2
            else:
               next_node = node1
      if not visited(next_node, path):        // Neighbor node not on path yet
            sub <= next_node
            sub.extend(path)
            find_new_cycle(sub);          // Explore extended path
      elif len(path) > 2  and next_node == path[-1]:     // Cycle found
            p = rotate_to_smallest(path);
            inv = invert(p)
            if is_new_cycle(p) and is_new_cycle(inv):
               cycles.append(p)

procedure invert(path):
   return rotate_to_smallest(path:-1)

//  rotate cycle path such that it begins with the smallest node
procedure rotate_to_smallest(path):
   n = path.index(min(path))
   return path[n]

procedure visited(node, path):
   return node in path

procedure is_new_cycle(path):
   return not path in cycles
```

```
procedure main():
   for each edge in graph:
      for each node in edge:
         find_new_cycle(node)      // node – first node of the path
   for each cy in cycles:
      path = ",".join(cy)
      print(path)
```

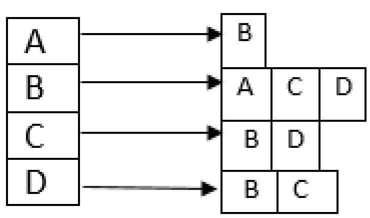
**(b)**

The number of cycles of a graph can vary based on the graph structure. It is possible making a graph with exponential number of cycles.

**(C)**




We can represent the graph as,




So, space cost = O (V + 2E)

We are visiting the graph by traversing each node, and also we need to check all arcs each time from both direction. So, the time complexity is, Θ ($E^2$)

<u>Why algorithm works?</u>

For current vertex u, start visiting for each adjacent vertex v of u,

- During visiting the graph, for any current vertex u, here D, if there is an adjacent vertex v, B, which is already visited, then the already visited vertex is the ancestor of u. As there is another way to reach the ancestor vertex from u, so there is a cycle in the graph.

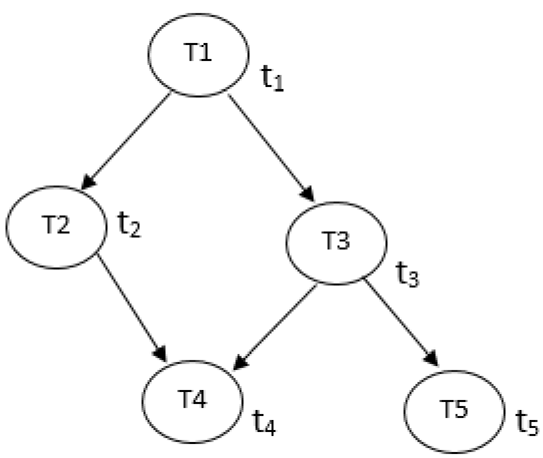
- But, if the already visited vertex v, here C, is the parent vertex of u then it is not a cycle, and will not be counted.

**[4] (Directed Graph, Non-programming) Describe a mathematical model for the following scheduling problem. Given tasks $T_1$, $T_2$, …, $T_n$ , which require times $t_1$, $t_2$, …, $t_n$ to complete, and a set of constraints, each of the form "$T_i$ must be completed prior to the start of $T_j$," find the minimum time necessary to complete all tasks. Justification is needed to show why your algorithm works.**

**<u>Solution:</u>**

Let, there are 5 tasks T1, T2, T3, T4, and T5 with required time t1, t2, t3, t4, and t5. But, there are some dependencies. T2 and T3 can start their tasks only after completing task T1. Similarly, T4 can start after completing both task T2 and T3, and T5 can start after T3.

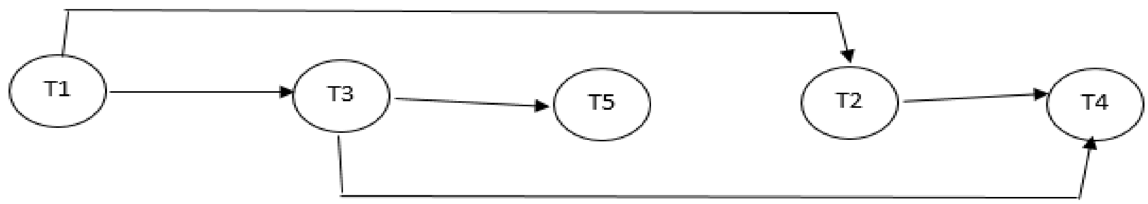
If we represent these tasks as a DAG,




Now, we can find the minimum required time to complete all tasks by

$= \text{Max}(t_1 + t_2 + t_4, t_1 + t_3 + t_4, t_1 + t_3 + t_5)$

But, how can we find this value procedurally,
If we apply topological sorting algorithm on the above graph, then it can be like this,




To find out the minimum required time, we need to follow the following algorithm,

Initialize,
time[] = {NINF, NINF, ….} and // NINF – negative infinite
time[source] = t1 // In this graph, T1
Apply topological sorting algorithm on the graph.
for every vertex u in topological order.
for every adjacent vertex v of u
if (time[v] < time[u] + time[v])
time[v] = time[u] + time[v]
Now, Find the maximum value from the time[] to get the minimum required time to complete all tasks.

But, how can I prove this algorithm will produce minimum required time to complete all tasks,
As, before calculating minimum required time, we have the graph as topologically sorted, so there is no issues with ordering.
and we have {T1, T3, T5, T2, T4}

Now, we will calculate the time step by step,
First we will pick T1 from the topologically ordered list. T1 has 2 adjacent vertices, T2 and T3
So, according to the algorithm,
time[1] = t1
time[2] = t1 + t2 and
time[3] = t1 + t3

Now, T3 has 2 adjacent vertices, T4 and T5

So,

time[4] = t4 + time[3] = t1 + t3 + t4

time[5] = t5 + time[3] = t1 + t3 + t5

Then, T5 has no adjacent vertices, so no update.

T2 has one adjacent vertex, T4

time[4] will be updated by (t1 + t2 + t4)

Only if

(t1 + t2 + t4) > (t1 + t3 + t4)

So, time[4] will be updated with the maximum time.

And T4 has no adjacent vertices.

Now, if we find the maximum time from the array, then it will be the required time for the longest thread to be dead. So, it is the minimum required time to complete all tasks.