DAA ASSIGNMENT-T2

Name of the algorithhm:Find the longest path in a Directed Acyclic Graph (DAG)

Description:The longest path problem is the problem of finding a simple path of maximal length in agraph; in other words, among all possible simple paths in the graph, the problem is to find the longest one. For an unweighted graph, it suffices to find the longest path in terms of the number of edges; for a weighted graph, one must use the edge weights instead. To explain the process of developing this algorithm, we'll first develop an algorithm for computing the single-source longest path for an unweighted directed acyclic graph (DAG), and then generalize that to compute the longest path in a DAG, both unweighted or weighted.

Psuedo Code:

```
#include <iostream>
#include <vector>
#include <climits>
#include <iomanip>
using namespace std;

struct Edge
{
   int src, dest, weight;
};

class Graph
{
public:
   vector<vector<Edge>> adjList;

   Graph(vector<Edge> const &edges, int N)
   {

      adjList.resize(N);

      for (Edge const &edge : edges)
      {
         adjList[edge.src].push_back(edge);
      }
   }
};
```

```
int DFS(Graph const &graph, int v, vector<bool> &discovered, vector<int> &departure, int &time)
{

   discovered[v] = true;
   for (Edge e : graph.adjList[v])
   {
      int u = e.dest;

      if (!discovered[u])
      {
         DFS(graph, u, discovered, departure, time);
      }
   }

   departure[time] = v;
   time++;
}

void LongestSimplePath(Graph const &graph, int source, int N)
{
   vector<int> departure(N, -1);

   vector<bool> discovered(N);
   int time = 0;

   for (int i = 0; i < N; i++)
   {
      if (!discovered[i])
      {
         DFS(graph, i, discovered, departure, time);
      }
   }

   vector<int> cost(N, INT_MAX);
   cost[source] = 0;

   for (int i = N - 1; i >= 0; i--)
   {

      int v = departure[i];
      for (Edge e : graph.adjList[v])
      {

         int u = e.dest;
         int w = e.weight * -1;

         if (cost[v] != INT_MAX && cost[v] + w < cost[u])
         {
            cost[u] = cost[v] + w;
         }
      }
   }
```

```
    for (int i = 0; i < N; i++)
    {
        cout << "dist(" << source << ", " << i << ") = " << setw(2) << cost[i] * -1;
        cout << endl;
    }
}
```

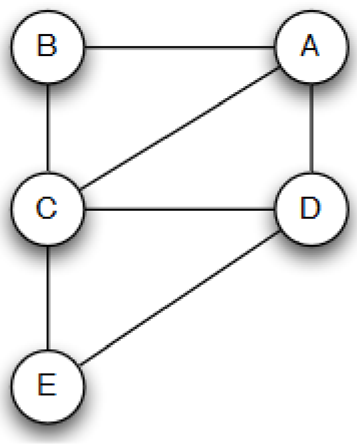
Sample Problem:**Given a Weighted Directed Acyclic Graph (DAG) and a source vertex in it, find the longest distances from source vertex to all other vertices in the given graph.**

```
->#include <iostream>
#include <vector>
#include <climits>
#include <iomanip>
using namespace std;

// Data structure to store a graph edge
struct Edge {
	int src, dest, weight;
};

// A class to represent a graph object
class Graph
{
public:
	// a vector of vectors to represent an adjacency list
	vector<vector<Edge>> adjList;

	// Graph Constructor
	Graph(vector<Edge> const &edges, int N)
	{
		// resize the vector to hold `N` elements of type vector<Edge>
		adjList.resize(N);

		// add edges to the directed graph
		for (Edge const &edge: edges) {
			adjList[edge.src].push_back(edge);
		}
	}
};

// Perform DFS on the graph and set the departure time of all
// vertices of the graph
int DFS(Graph const &graph, int v, vector<bool>
	&discovered, vector<int> &departure, int &time)
{
	// mark the current node as discovered
	discovered[v] = true;
```

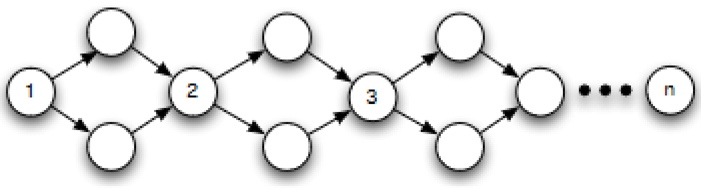
```
        // set arrival time – not needed
        // time++;

        // do for every edge `v —> u`
        for (Edge e: graph.adjList[v])
        {
                int u = e.dest;
                // if `u` is not yet discovered
                if (!discovered[u]) {
                        DFS(graph, u, discovered, departure, time);
                }
        }

        // ready to backtrack
        // set departure time of vertex `v`
        departure[time] = v;
        time++;
}

// The function performs the topological sort on a given DAG and then finds
// the longest distance of all vertices from a given source by running one pass
// of the Bellman–Ford algorithm
void findLongestDistance(Graph const &graph, int source, int N)
{
        // departure[] stores vertex number having its departure
        // time equal to the index of it
        vector<int> departure(N, -1);

        // to keep track of whether a vertex is discovered or not
        vector<bool> discovered(N);
        int time = 0;

        // perform DFS on all undiscovered vertices
        for (int i = 0; i < N; i++)
        {
                if (!discovered[i]) {
                        DFS(graph, i, discovered, departure, time);
                }
        }

        vector<int> cost(N, INT_MAX);
        cost[source] = 0;

        // Process the vertices in topological order, i.e., in order
        // of their decreasing departure time in DFS
        for (int i = N - 1; i >= 0; i--)
        {
```

```
        // for each vertex in topological order,
        // relax the cost of its adjacent vertices
        int v = departure[i];
        for (Edge e: graph.adjList[v])
        {
            // edge `e` from `v` to `u` having weight `w`
            int u = e.dest;
            int w = e.weight * -1;          // make edge weight negative

            // if the distance to destination `u` can be shortened by
            // taking edge `v —> u`, then update cost to the new lower value
            if (cost[v] != INT_MAX && cost[v] + w < cost[u]) {
                cost[u] = cost[v] + w;
            }
        }
    }

    // print the longest paths
    for (int i = 0; i < N; i++)
    {
        cout << "dist(" << source << ", " << i << ") = " << setw(2) << cost[i] * -1;
        cout << endl;
    }
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges =
    {
        {0, 6, 2}, {1, 2, -4}, {1, 4, 1}, {1, 6, 8}, {3, 0, 3}, {3, 4, 5},
        {5, 1, 2}, {7, 0, 6}, {7, 1, -1}, {7, 3, 4}, {7, 5, -4}
    };

    // total number of nodes in the graph
    int N = 8;

    // build a graph from the given edges
    Graph graph(edges, N);

    // source vertex
    int source = 7;

    // find the longest distance of all vertices from a given source
    findLongestDistance(graph, source, N);

    return 0;
}
```

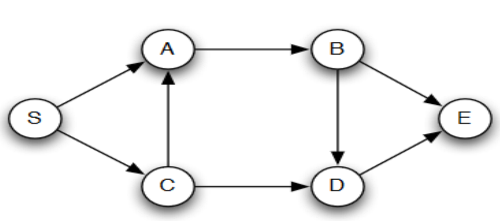
Proof of Correctness:

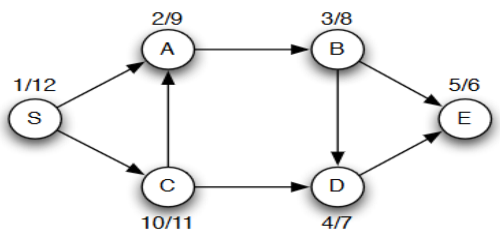


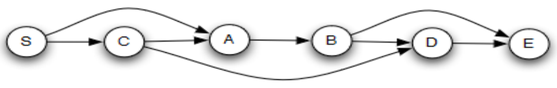
When it comes to finding the
longest path however, we find that the problem does not
have optimal substructure, and we lose the ability to use dynamic programming (and of course greedy strategy as well!). To see that the longest path problem does not have optimal substructure, consider the graph in above figure. The longest path from A to E is A–B–C–D–E This goes through B, but the longest path from B to E is not B–C–D–E. It's B–C–A–D–E. This problem does not have optimal substructure, which means that we cannot solve it using dynamic programming. Well, why not simply enumerate all the paths, and find the longest one? Because, there can be exponentially many such paths! For example, in the graph shown in below image there are 2^n different paths from vertex 1 to n.



Let's start with
the DAG G, shown in figure below, as an example and see how we can develop
a dynamic programming solution for the single-source longest path problem. Remember that a DAG can always be topologically sorted or linearized, which allows us to traverse the vertices in linearized order from left to right. To see how this helps, look at the linearized version of G, whose vertices, taken in the linearized order, are then: {S,C,A,B,D,E}.







before we tackle the problem of the finding the longest path in the
graph, let's first consider the problem of finding the longest path from vertex S to any other vertex. Consider the vertex Din the linearized graph — the only way to get to D from Sis

through one of its predecessors: Band C. That means, to compute the longest path from S to D, one must first compute the longest path from S to B(up to the first predecessor), and the longest path from S to C(up to the second predecessor). Once we've computed the longest paths from S to these two predecessors, we can compute the longest path to D by taking the larger of these two, and adding 1. If dist(v) is the longest distance from S to vertex v, and α(v) is the actual path, then we can write the following recurrence for dist(D):

$$\text{dist}(D) = \max\{\text{dist}(B) + 1, \text{dist}(C) + 1\}$$

Note the sub problems here: dist(B) and dist(C), both of which are "smaller" than dist(D). Similarly, we can write the recurrences for dist(B) and dist(C) in terms of its subproblems:

$$\text{dist}(B) = \text{dist}(A) + 1$$
$$\text{dist}(C) = \text{dist}(S) + 1$$

and so on for each vertex. For a vertex with no incoming edges, we take the maximum over an empty set, which results in an expected distance of 0. Or, we can explicitly set the base case dist(S) and α(S) for the source vertex S, as follows:,

$$\text{dist}(S) = 0$$
$$\alpha(S) = \{S\}$$

For G, by inspection, dist(B) = 3 and dist(C) = 1, which tells us that dist(D) = max{3 + 1,1 + 1}= 4, a fact easily verified by inspection. The corresponding actual longest path from S to D, through B, is α(D) = α(B) ∪{D}= {S,C,A,B,D}, again easily verified by inspection. We are now ready to write the recurrence for the longest path from S to any other vertex v in G, which involves first computing the longest paths to all of v's predecessors from S(these are the subproblems).

$$\text{dist}(v) = \max_{(u,v)\in E}\{\text{dist}(u) + 1\}$$

And we see how we can start from S, find the longest paths to all the vertices by traversing from left to right in linearized order. For a graph G= (V,E), the algorithm is shown below.
For each vertex v∈Vin linearized order:

$$\text{do dist}(v) = \max_{(u,v)\in E}\{\text{dist}(u) + 1\}$$

If there are no edges into a vertex (ie., if in-degree[v] = 0), we take the maximum over an empty set, so dist(S) will be 0 as expected. To implement this algorithm, we need to know the predecessor of each vertex, and the easy way to have that information to compute G_{rev}, which is G with each edge reversed. The algorithm is obviously O(|V|+ |E|).
Now that we've computed the longest path from S to all the other vertices v∈V, let's now look at our original problem of finding the longest path in G. It so happens that the subproblems {dist(v),v∈V}that we solve give us the longest path ending at some vertex v∈V. The longest path in G then is simply the largest of all

$$\{\text{dist}(v), v\in V\}!$$
$$\text{longest-path}(G) = \max_{v\in V}\{\text{dist}(v)\}$$

And, we can compute this bottom-up for each vertex v∈V taken in a linearized order. The final algorithm is shown below :

longest-path(G)
Input: Unweighted DAG G= (V,E)

```
Output: Largest path cost in G
Topologically sort G
for each vertex $v \in V$ in linearized order
do $dist(v) = \max_{(u,v) \in E} \{dist(u) + 1\}$
return $\max_{v \in V} \{dist(v)\}$
```

Time space comlexity : Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time complexity of this algorithm is O(V+E).