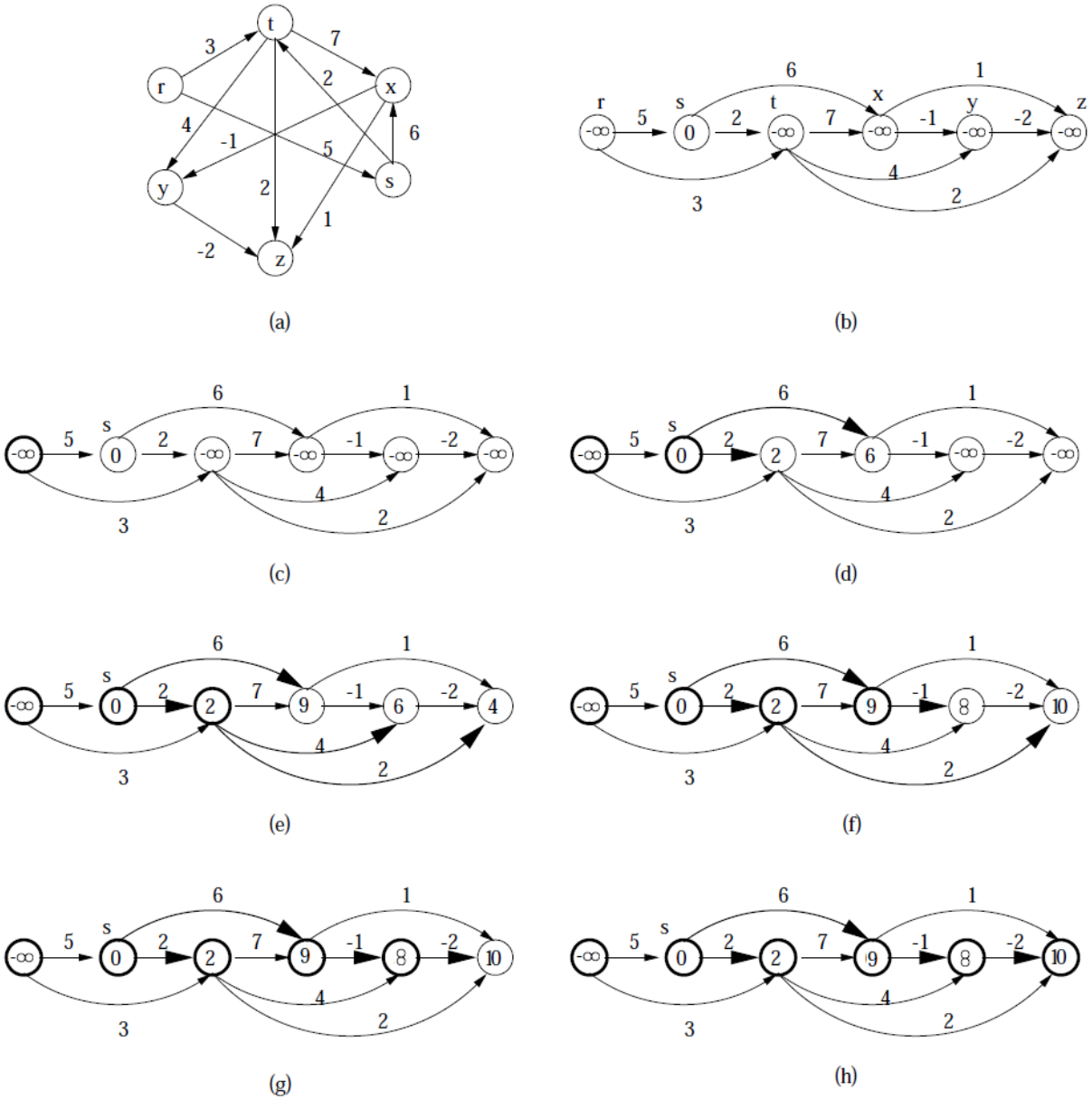
Longest Path in a Directed Acyclic Graph

Given a Weighted **D**irected **A**cyclic **G**raph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn't have optimal substructure property. In fact, the Longest Path problem is NP-Hard for a general graph. However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to linear time solution for shortest path in a directed acyclic graph., we use Tological Sorting.

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find atopological sorting of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure shows step by step process of finding longest paths.



Following is complete algorithm for finding longest distances.

1) Initialize dist[] = {NINF, NINF, ….} and dist[s] = 0 where s is the source vertex. Here NINF means negative infinite.

2) Create a toplogical order of all vertices.

3) Do following for every vertex u in topological order.

……..Do following for every adjacent vertex v of u

……..……if (dist[v] < dist[u] + weight(u, v))

……..……..……dist[v] = dist[u] + weight(u, v)

Following is C++ implementation of the above algorithm.

```
// A C++ program to find single source longest distances in a DAG
#include <iostream>
#include <list>
#include <stack>
#include <limits.h>
#define NINF INT_MIN
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int _v, int _w)  { v = _v;  weight = _w;}
    int getV()       {  return v;  }
    int getWeight()  {  return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by longestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds longest distances from given source vertex
    void longestPath(int s);
};

Graph::Graph(int V) // Constructor
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by longestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
```

```
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find longest distances from a given vertex. It uses
// recursive topologicalSortUtil() to get topological sorting.
void Graph::longestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = NINF;
    dist[s] = 0;

    // Process vertices in topological order
    while (Stack.empty() == false)
    {
        // Get the next vertex from topological order
        int u = Stack.top();
        Stack.pop();

        // Update distances of all adjacent vertices
        list<AdjListNode>::iterator i;
        if (dist[u] != NINF)
        {
          for (i = adj[u].begin(); i != adj[u].end(); ++i)
             if (dist[i->getV()] < dist[u] + i->getWeight())
                dist[i->getV()] = dist[u] + i->getWeight();
        }
    }

    // Print the calculated longest distances
    for (int i = 0; i < V; i++)
        (dist[i] == NINF)? cout << "INF ": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram.  Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
```

```
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 5, 1);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are longest distances from source vertex " << s <<" \n";
    g.longestPath(s);

    return 0;
}
```

Run on IDE

Output:

```
Following are longest distances from source vertex 1
INF 0 2 9 8 10
```

Time Complexity: 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).