خوارزمية ديجسترا لإيجاد أقصر مسارات من مصدر أحادي (Dijkstra ‘s algorithm) :

تطبق الخوارزمية من أجل إيجاد أقصر مسارات من رأس مصدر أحادي إلى جميع الرؤوس الأخرى في بيان موجه و موزون .

```
/* Dijkstra algorithm for shortest paths from node 1 */
#include <stdio.h>

typedef struct
{
 int weight;
 int v1;
 int v2;
}edge;

#define N 5
#define INF 1000

void dijkstra(int n, int(*W)[N + 1], edge* F);

int f_index = -1;

int main()
{
int W[N+1][N+1]={{0,0,0,0,0,0},
                 {0,INF,2,INF,INF,10},
                 {0,INF,INF,3,INF,7},
                 {0,INF,INF,INF,4,INF},
                 {0,INF,INF,INF,INF,5},
                 {0,INF,INF,6,INF,INF}};

 edge F[N];
    printf("shortest distance from vertex 1 to other
vretices:\n");

    dijkstra(N, W, F);
    printf("edges of shortest path tree:\n");

 for (int i = 0; i <= f_index; i++)
 {
     printf("%d - %d distance:%d\n", F[i].v1,
F[i].v2,W[F[i].v1][F[i].v2]);
```

```
 }
}

void dijkstra(int n, int (*W)[N+1], edge *F)
{
 int vnear;
 edge e;
 int touch[N + 1];
 int length[N + 1];

 for (int i = 2; i <= n; i++)
 {
      touch[i] = 1;
      length[i] = W[1][i];
 }
 while (1)
 {
      int min = INF;
      //find vertex with minimum length
      for (int i=2; i <= n; i++)
      {
           if (0 <= length[i] && length[i]<= min)
           {
                min = length[i];
                vnear = i;
           }
      }
      //edge from touch[vnear] to vnear
      e.v1 = touch[vnear];
      e.v2 = vnear;
      F[++f_index] = e;

      for (int i = 2; i <= n; i++)
      {
           if (length[vnear] + W[vnear][i] < length[i])
           {
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
           }
      }
      printf("distance of vertex %d :%d\n", vnear,
length[vnear]);
      //vnear is visited
      length[vnear] = -1;
      if (f_index == n - 2)
           break;
 }
}
```

خوارزمية إيجاد المكونات المتصلة بقوة في بيان موجه

(Kosaraju 's algorithm for strongly connected components) :

يعرف المكون المتصل بقوة في بيان موجه بأنه بيان جزئي يوجد لكل زوج من الرؤوس فيه v,w مسار من v إلى w و مسار آخر من w إلى v.

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;     // No. of vertices
    list<int> *adj;     // An array of adjacency lists

    /* Fills Stack with vertices (in increasing order of
finishing times). The top element of stack has the maximum
finishing time */
    void fillOrder(int v, bool visited[], stack<int> &Stack);

    // A recursive function to print DFS starting from v
     void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);

    /* The main function that finds and prints strongly
connected  components */
    void printSCCs();

    //Function that returns reverse (or transpose) of this graph
    Graph getTranspose();
};

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

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
```

```
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

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

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

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);
    //All vertices reachable from v are processed by now, push v
    Stack.push(v);
}

/* The main function that finds and prints all strongly
connected components */
void Graph::printSCCs()
{
```

```
    stack<int> Stack;
    // Mark all the vertices as not visited (For first DFS)
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing times
    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, Stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;
    // Now process all vertices in order defined by Stack
    while (Stack.empty() == false)
    {
        // Pop a vertex from stack
        int v = Stack.top();
        Stack.pop();
      // Print Strongly connected component of the popped vertex
        if (visited[v] == false)
        {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
}
// Driver program to test above functions
int main()
{
    // Create a graph
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Following are strongly connected components in "
            "given graph \n";
    g.printSCCs();

    return 0;
}
```

: Topological sorting خوارزمية

تستخدم الخوارزمية لزيارة رؤوس البيان الموجه بترتيب يقتضي أنه من أجل كل حافة vw، يتم المرور على الرأس v أولاً ثم على الرأس w .

```
// A C++ program to print topological
// sorting of a graph using indegrees.
#include <bits/stdc++.h>
using namespace std;

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

    // Pointer to an array containing
    // adjacency listsList
    list<int>* adj;

public:
    // Constructor
    Graph(int V);
     // Function to add an edge to graph
    void addEdge(int u, int v);

    // prints a Topological Sort of
    // the complete graph
    void topologicalSort();
};

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

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
}

// The function to do  Topological Sort.
void Graph::topologicalSort()
{
    /* Create a vector to store indegrees of all
     vertices. Initialize all indegrees as 0*/
    vector<int> in_degree(V, 0);
```

```
/* Traverse adjacency lists to fill indegrees of
 vertices.  This step takes O(V+E) time */
for (int u = 0; u < V; u++) {
    list<int>::iterator itr;
    for (itr = adj[u].begin();itr != adj[u].end(); itr++)
        in_degree[*itr]++;
}

// Create an queue and enqueue
// all vertices with indegree 0
queue<int> q;
for (int i = 0; i < V; i++)
    if (in_degree[i] == 0)
        q.push(i);

// Initialize count of visited vertices
int cnt = 0;
/* Create a vector to store
 result (A topological
 ordering of the vertices)*/
vector<int> top_order;

/* One by one dequeue vertices from queue and enqueue
 adjacents if indegree of adjacent becomes 0 */
while (!q.empty()) {
    /* Extract front of queue (or perform dequeue)
     and add it to topological order */
    int u = q.front();
    q.pop();
    top_order.push_back(u);

    /* Iterate through all its
     neighbouring nodes of dequeued node u and
     decrease their in-degree by 1 */
    list<int>::iterator itr;
    for (itr = adj[u].begin();itr != adj[u].end(); itr++)

        // If in-degree becomes zero,
        // add it to queue
        if (--in_degree[*itr] == 0)
            q.push(*itr);

    cnt++;
}

// Check if there was a cycle
if (cnt != V) {
```

```
        cout << "There exists a cycle in the graph\n";
        return;
    }

    // Print topological order
    for (int i = 0; i < top_order.size(); i++)
        cout << top_order[i] << " ";
    cout << endl;
}

// main program to test above functions
int main()
{
    /* Create a graph given in the
     above diagram */
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of\n";
    g.topologicalSort();

    return 0;
}
```