Graph Algorithms

Graphs are very useful in representing arbitrary relationships among data objects. There are two kinds of graphs: Directed graphs (also known as digraphs) and undirected graphs (simply called graphs). In this lecture and the next one we discuss some important graph algorithms and the related data structures.

Undirected graphs

An *undirected graph* (or simply a *graph*) G = (V, E) consists of a set of *nodes* (also called *vertices*) V, and a set of *edges* E. Each edge in E is a pair {x, y} where x, y ∈ V.

A pictorial representation of the graph G = (V, E), where V = {0, 1, 2, 3} and E = {{1, 2}, {1, 3}, {2, 0}, {3, 2}, {0, 3}} is shown in Figure 1.

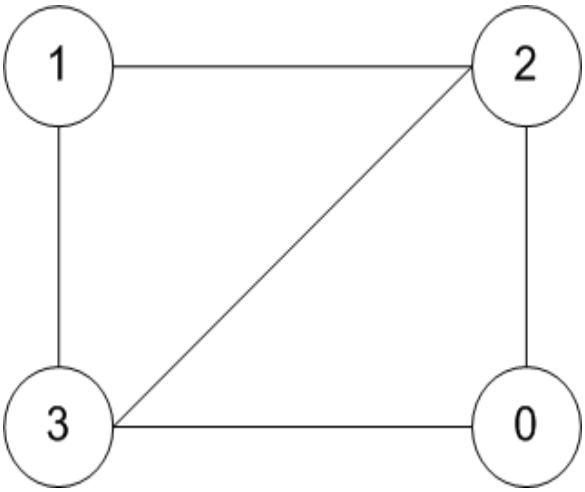

Figure 1: A graph with 4 nodes and 5 edges

Representation of Graphs

There are two important methods for storing graphs.

From now on we use n = |V |, for the number of nodes and e = |E|, for the number of arcs of G.

1. Adjacency Matrix: The matrix is of order (n × n) where n is the number of vertices. In the matrix cell a (i, j) = 1 if the edge {i, j} ∈ E. Otherwise the cell (i, j) = −1. The adjacency matrix for the graph in Figure 1 is

$$\begin{bmatrix} -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix}$$

Although the adjacency matrix representation is simple, it requires n^2 storage space.

2. Adjacency List: Figure 2 illustrates the adjacency lists representation for the graph of Figure 1.

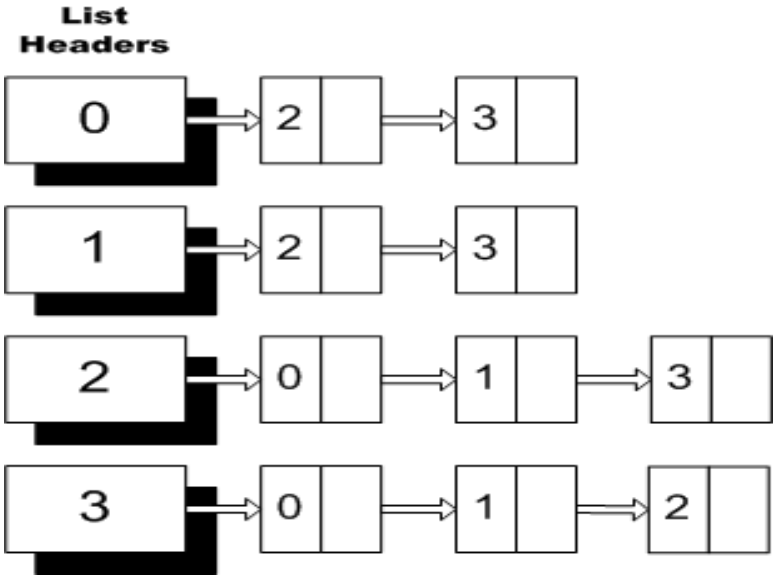

Figure 2: Adjacency list representation of the graph of Figure 1

The Adjacency lists representation requires n + e storage space. Since $0 \le e \le n^2$, the adjacency lists representation often requires much smaller space that the adjacency matrix representation. Most graphs in the real world applications have far too few edges as compared to n^2. In all such cases, it is more efficient to use the adjacency lists representation. We will see that many graph problems can be solved in linear time n + e if the adjacency lists representation is used. Note that any algorithm using the matrix representation must have running time n^2.

A C++ Code Example for Creating Adjacency Lists of an Undirected Graph

An adjacency lists representation of a graph can viewed as a Vector object. Each cell in the vector is a linked list of type either SList (singly linked list) or a DList (doubly linked list) depending on the application. The adjacency lists representation illustrated in Figure 2 uses singly linked lists. Thus the representation in Figure 2 can be considered as an object of type Vector<SList<int>>.

In this section, we show how we can create the adjacency lists using the example graph of 1.

Suppose the graph of Figure 1 is given to us in a file (say infile.dat) containing the following:

```
4  5
0  2
0  3
1  2
1  3
2  3
```

The first line in the above file has two integers, namely 4 and 5. We interpret these as the number of nodes (4) and the number of edges (5) of the given graph. So, after reading these two numbers, we know that the graph has 5 edges and we read 5 times (one reading for each edge) to get the end nodes of each edge.

The following code creates an adjacency lists object for a graph whose input is available in the file infile.dat.

```
#include <fstream>
#include "Vector.h"
#include "SList.h"

using namespace std;

ifstream input;

void main() {
      input.open("infile.dat");

      Vector<SList<int> > graph;
      int nodes, edges;

      input >> nodes >> edges;

      for (int i = 0; i < nodes; i++) {
            graph.AddLast(SList());
      }

      int edge_end_node_1, edge_end_node_2;

      for (i = 0; i < edges; i++) {
            input >> edge_end_node_1 >> edge_end_node_2;
            graph[edge_end_node_1].AddLast(edge_end_node_2);
            graph[edge_end_node_2].AddLast(edge_end_node_1);
      }
```

```
        input.close();

} // end main()
```

Directed Graphs

A directed graph or digraph G = (V, E) is a set V = {1, 2, . . . , |V |} of nodes V , and a set E = {(i, j)|i ∈ V, j ∈ V } of directed edges (also called arcs). A pair (i, j) ∈ E is a directed edge (arc) from i to j.

A pictorial representation of the digraph G = (V, E), where V = {1, 2, 3, 4} and E = {(1, 2), (1, 3), (2, 0), (3, 2), (0, 3)} is shown in Figure 3.

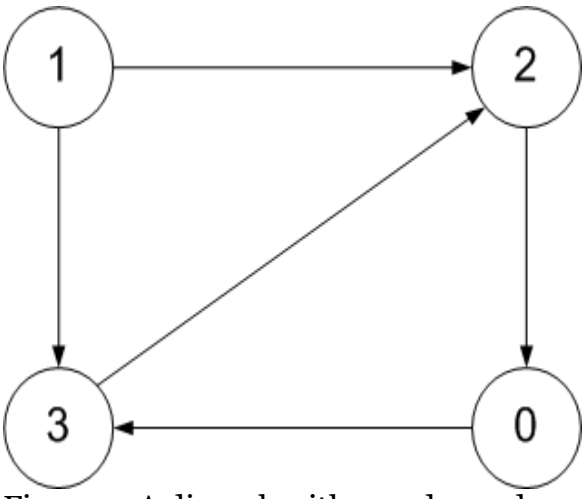

Figure 3: A digraph with 4 nodes and 5 arcs

As in the case of graphs, digraphs can be represented using the adjacency matrices as well as adjacency lists. The adjacency matrix representation uses a matrix of order (n × n) where n is the number of vertices. The adjacency matrix for the digraph in Figure 3 is

$$\begin{bmatrix} -1 & -1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ -1 & -1 & 1 & -1 \end{bmatrix}$$

Figure 4 illustrates the adjacency lists representation for the digraph of Figure 3.

Again, the adjacency matrix and the lists representation of digraphs need n^2 and n + e space respectively.

Note that in the digraph the adjacency lists, an edge (i, j) appears in the adjacency list i only but not in the list j. However, as seen before in the case of graphs each edge {i, j} appears in list i as well as list j.

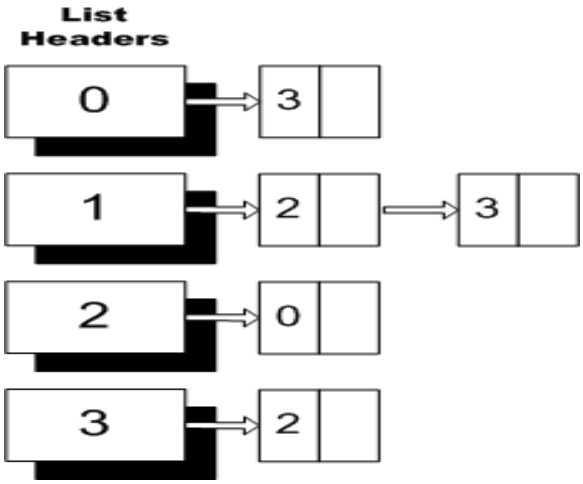

Figure 4: Adjacency list representation of the digraph of Figure 3

The following code creates an adjacency lists object of type Vector<SList<int> > for digraphs. As before we assume that the input digraph is available in the file infile.dat.

```
#include <fstream>
#include <vector>
#include "SList.h"

using namespace std;

ifstream input;

int main() {
	input.open("infile.dat");

	vector<SList<int> > digraph;
	int nodes, edges;

	input >> nodes >> edges;

	for (int i = 0; i < nodes; i++) {
		digraph.push_back(SList());
	}

	int edge_end_node_1, edge_end_node_2;

	for (i = 0; i < edges; i++) {
		input >> edge_end_node_1 >> edge_end_node_2;
		graph[edge_end_node_1].push_back(edge_end_node_2);
	}

}
```

Some more definitions on Graphs and Digraphs

In a graph G = (V, E), we say two nodes x and y are adjacent (to each other) if {x, y} is an edge in E. In a digraph, if there is an arc (x, y), we say y is adjacent to x in G. In a graph (or digraph), a path from node v to w, is a sequence v1, v2, . . . , vk of nodes such that v1 = v, vk = w, and vi is adjacent to vi+1 for all $1 \leq i < k$. The length of such a path is the number of edges on the path, which is equal to k – 1. Note that there is always the path of length zero from v to v. A path v1, v2, . . . , vk is said to be simple if all the the edges on the path and the nodes of the path are distinct, except possibly the first node v1 and the last node vk. A path v1, v2, . . . , vk, k > 1, and v1 = vk is a cycle.

Directed Acyclic Graphs (DAG)

A digraph is called an acyclic digraph if it has no cycles. Clearly the digraph of Figure 3 is not an acyclic graph since the sequence of nodes 2, 0, 3, 2 constitute a directed cycle of the digraph. However, the digraph of Figure 5 is an acyclic digraph.

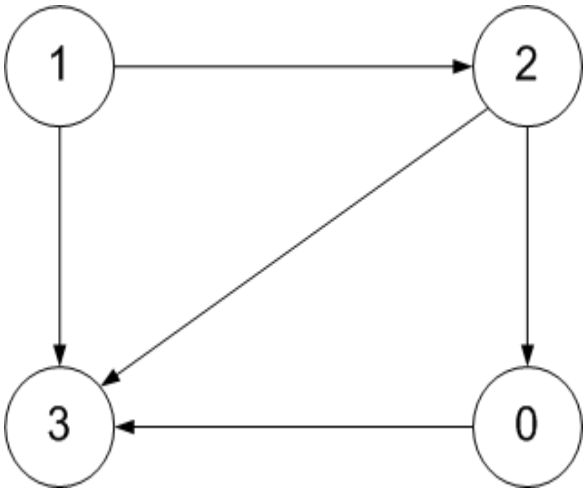

Figure 5: An acyclic digraph

The *indegree* of a node v, denoted by id(v), is the number of arcs directed into v. Likewise the *outdegree* of a node v, denoted by od(v), is the number of arcs directed out of v.

Forests and Trees

A digraph is a directed forest if it is acyclic and $id(v) \leq 1$ for all nodes v of the digraph.

In a directed forest, a node v with id(v) = 0 is a root.

Since a directed forest is an acyclic digraph, it has at least one root.

A directed forest with exactly one root is a directed tree.

A subgraph T of a digraph G is a spanning directed forest if T is a directed forest with V (T) = V (G). If T is a directed tree then it is a spanning directed tree of G.

A directed forest of Figure 5 is shown in Figure 6.

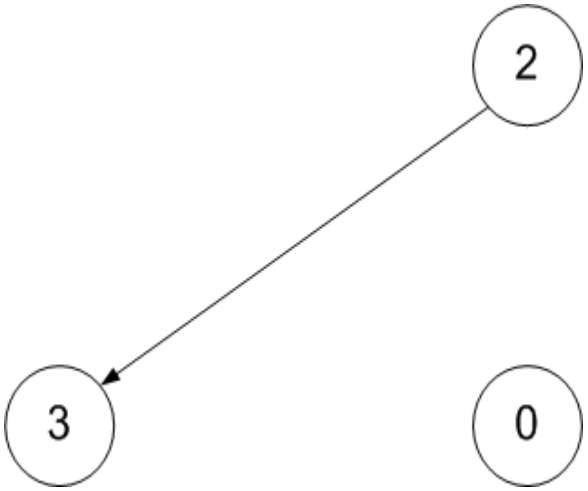

Figure 6: A directed forest of the digraph of Figure 5

Note that the directed forest of Figure 6 is not a spanning forest, since it does not contain all the nodes of the digraph. The digraphs shown in Figures 7 and 8 illustrate two spanning directed forests of the digraph of Figure 5. Indeed, the digraph of Figure 8 is also a spanning directed tree of the digraph of Figure 5.

Note that every acyclic digraph has a node of indegree = 0 and a node of outdegree = 0. In the acyclic digraph of Figure 5, node 1 has indegree = 0, and node 3 has outdegree = 0.

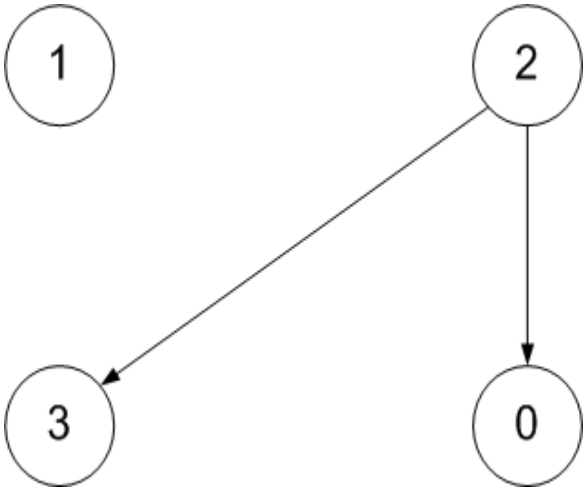

Figure 7: A spanning directed forest of the digraph of Figure 5

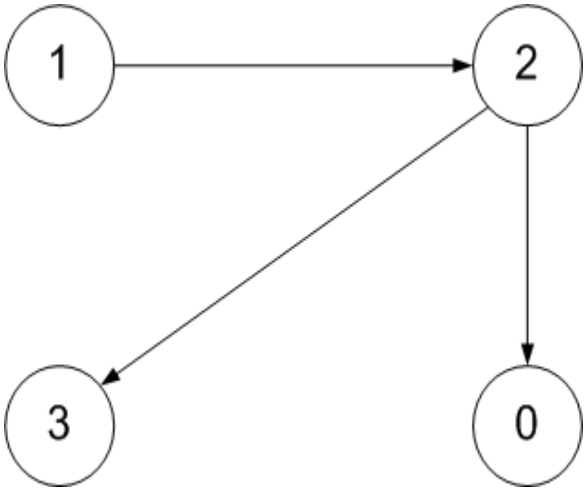

Figure 8: A spanning directed tree of the digraph of Figure 5

DAGs and Topological Sorting

Directed acyclic graphs (DAGs) are useful in many applications such as in the representation of task graphs and precedence relations in scheduling problems.

Suppose that G is digraph with n nodes. A topological sort is an assignment of integer numbers (from 1 to n) to the nodes of the digraph, such that for every edge (i, j) the number assigned to i is less that the number assigned to j.

Thus a topological sort consists of an ordering of the nodes of a digraph, such that, for any path, say from node vi to node vj , the node vj appears after the node vi in the ordering. The DAG of Figure 9 represents the course prerequisite structure of some of the graduate courses in the Computer Science department at Stevens. A directed edge from node v to node u indicates that course v must be completed before course u is taken. A topological ordering of these courses is any course sequence that does not violate the prerequisite requirements.

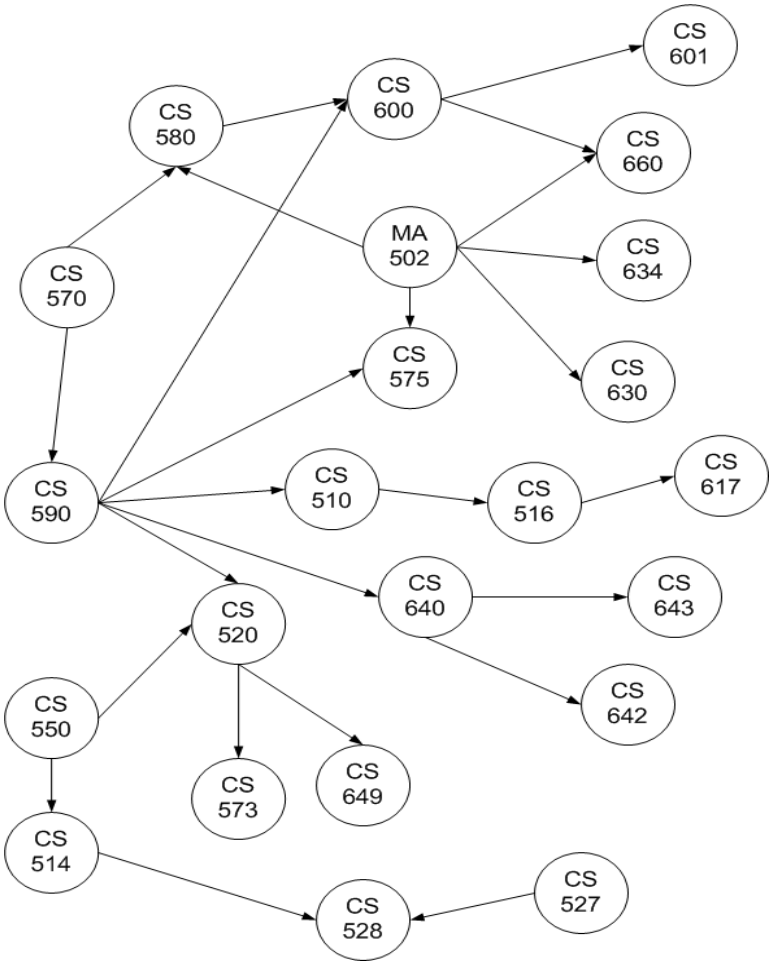

Figure 9: A DAG representing course prerequisites

It is easy to see that a topological ordering is impossible for a digraph containing directed cycles. To see this, consider any two nodes v and u on a directed cycle. Then the directed path from v to u indicates that v precedes u in the ordering, but the directed path from u to v requires that u must precede v and this is impossible.
Now the question is – can we obtain a topological ordering of the nodes of a DAG? The answer is yes. Indeed, a topological sort exists for a digraph D if and only if D is acyclic.

Consider the DAG of Figure 10.

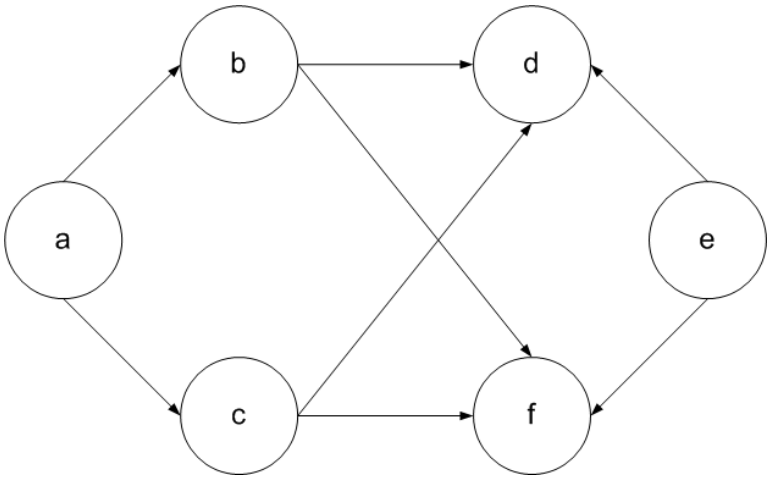

Figure 10: An acyclic digraph with 6 nodes

A topological sort of nodes of the graph of Figure 10 is shown in Figure 11.

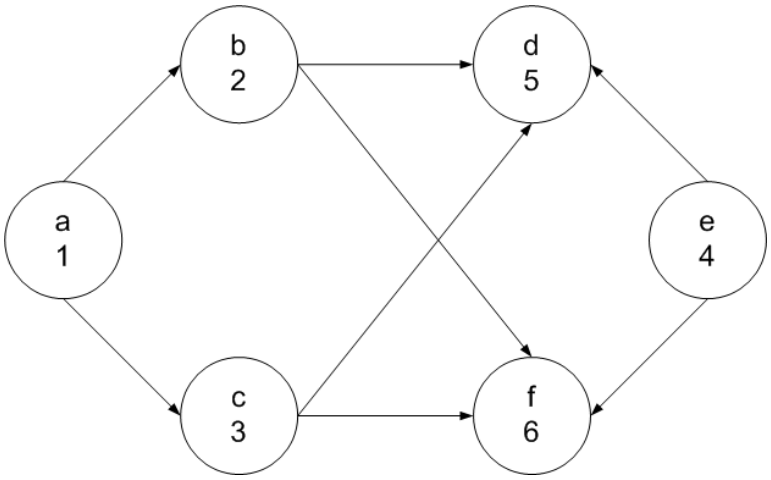

Figure 11: A topological ordering of the nodes of the digraph of Figure 10

A topological sort of a DAG is not necessarily unique. For example Figure 12 illustrates another topological sort for the DAG of Figure 10.

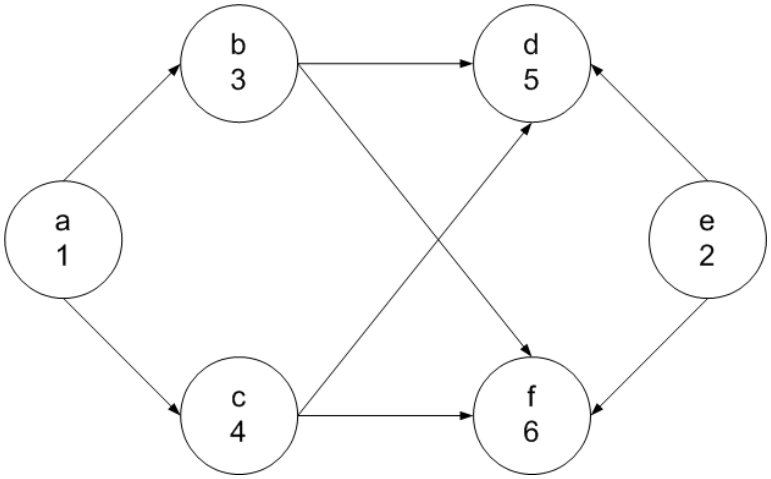

Figure 12: Another topological ordering of the digraph of Figure 10

A simple algorithm to find a topological ordering of the nodes of a digraph is the following:

TopologicalSort
Input: A digraph D.
Output: If D is acyclic: a topological order number for each node of D. Else: A message that D is cyclic.

```
int i = 0;

while (D is not empty) {
      find a node u with indegree == 0;
      if (no such node exists)
            return with the message "D is not acyclic";
      i = i+1;
      delete node u; // (of course all arcs directed out of u as well)
      topological order number of u = i;
}
```

Single Source Shortest Paths Problem

Input: A digraph G = (V, E) in which each arc has a non-negative cost and a specified node s ∈ V as the source node.

Output: The cost of the shortest path from the source to every other node in V (where the length of a path is the sum of the costs of the arcs on the path). In 1959, Dijkstra discovered an n^2 algorithm to solve this problem.

Dijkstra's Algorithm

The algorithm works by maintaining a set S of selected nodes whose shortest distance

from the source is already known. At each step, a new node v from V – S is selected and the selected node v is added into S. The selection of v from V – S is done by a greedy approach. Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.

Let C[i, j] = { cost of the arc (i, j) if the arc (i, j) exists };
C[i, j] = ∞ otherwise.

Now we are ready to present the Dijkstra's algorithm.

```
S = {s};

for (each node v in V) {
    D[i] = C[s,i];
}

while (V-S is not empty) {
     select a node v in V-S such that D[v] is a minimum; delete
     v from V-S;
     add v to S;
     for (each w in V-S) {
          D[w] = min(D[w], D[v] + C[v,w]);
     }
}
```

The above algorithm gives the costs of the shortest paths from source vertex to every other vertex. The actual shortest paths can also be constructed by modifying the above algorithm.

We illustrate the Dijkstra's Algorithm using the digraph of Figure 13.

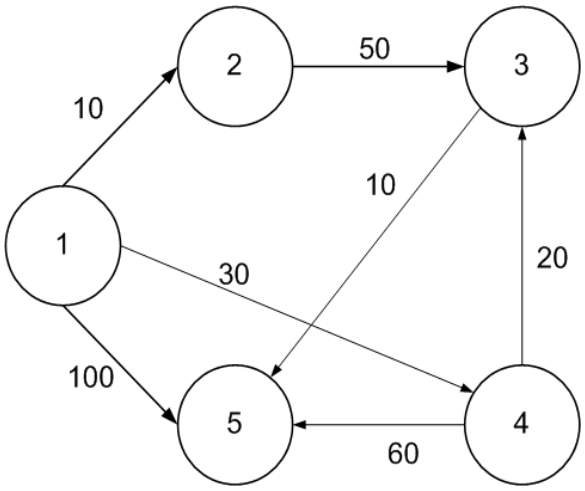

Figure 13: A digraph example for Dijkstra's algorithm

Suppose that the source node s = 1. Then
D[2] = 10, D[3] = ∞, D[4] = 30, D[5] = 100.

Iteration 1
Select v = 2, so that S = {1, 2}

$$\begin{aligned} D[3] &= \min(\infty, D[2] + C[2,3]) = 60 \\ D[4] &= \min(30, D[2] + C[2,4]) = 30 \\ D[5] &= \min(100, D[2] + C[2,5]) = 100 \end{aligned}$$

Iteration 2
Select v = 4, so that S = {1, 2, 4}

$$\begin{aligned} D[3] &= \min(60, D[4] + C[4,3]) = 50 \\ D[5] &= \min(100, D[4] + C[4,5]) = 90 \end{aligned}$$

Iteration 3
Select v = 3, so that S = {1, 2, 4, 3}

$$D[5] = \min(90, D[3] + C[3,5]) = 60$$

Iteration 4
Select v = 5, so that S = {1, 2, 4, 3, 5}

$$\begin{aligned} D[2] &= 10 \\ D[3] &= 50 \\ D[4] &= 30 \\ D[5] &= 60 \end{aligned}$$