Directed Graphs -- Digraphs

A Digraph is a graph is which each edge has a direction. Directed edges are called Arcs.
D = (V,A) is a Digraph, where V = Set of Vertices
and A = Set of Arcs, each arc is an ordered pair.

In effect, we have implemented (undirected) Graphs as Digraphs as each edge {i,j} (unordered pair) is represented by two arcs --directed edges, (i,j) and (j,i).

The In-degree of a vertex v is the number of arcs leading into v and the Out-degree of v is the number of arcs leading out of v.

Implementation of Digraph

A Digraph may be represented by an Adjacency Matrix or Adjacency Lists.

Traversing Digraphs

Just as in (undirected) Graphs we can traverse Digraphs by Depth First or Breadth First. The algorithms are the same as for (undirected) Graphs.

Let D be a Digraph.
The underlying Graph of D is the (undirected) graph where the arcs are viewed as (undirected) edges.

Path in D

A sequence $x_1, x_2, .. x_k$ ($x_1 \neq x_k$) of vertices is a path if each (x_1,x_2), (x_2,x_3) .. is an arc in D
If the vertices $x_1, x_2, .. x_k$ are distinct then it is called an elementary path.
If $x_1 = x_k$ then we have a circuit or elementary circuit if the path is elementary.

D is Connected iff the underlying graph of D is connected.

D is Strongly Connected iff for each pair of vertices (i,j) in D there is a path from i to j.

Directed Acylic Graph -- DAG

A DAG is a Digraph with no circuits. The underlying graph may have a cycle.
Note: A graph is a Tree if it has no cycles.

A Directed Tree is Digraph in which each vertex, except the root, has In-degree 1.

Vertices with Out-degree 0 are called Leaves.

Note:
In some circumstances a Binary Tree may be regarded as Directed Tree in which the max Out-degree (of all the vertices) is 2.

A Binary tree is different from a Directed tree as the 'children' are ordered i.e. we have a left and a right child.

We could associate with each Binary Tree a directed tree where the order of the 'children' is ignored.

Topological Sort

A directed acylic graph (DAG) D gives rise to a (strict) partial order on the vertices of D.

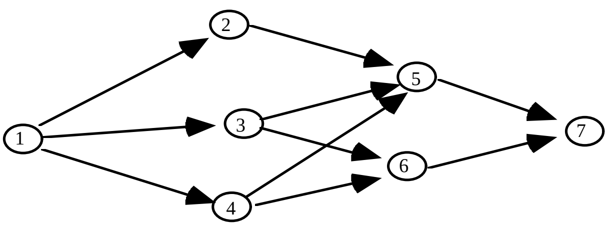
$i \rightarrow j$ "i can reach j" iff there is a path from i to j

The relation $\rightarrow$ is a (strict) partial order on D as it is

1. **Irreflexive: not($i \rightarrow i$), there is no path from i to itself**
2. **Asymmetric: $i \rightarrow j$ and $j \rightarrow i$ is impossible**
3. **Transitive: if $i \rightarrow j$ and $j \rightarrow k$ then $i \rightarrow k$**

Application of DAG

A DAG can be used to represent an Activity Network.
E.G.

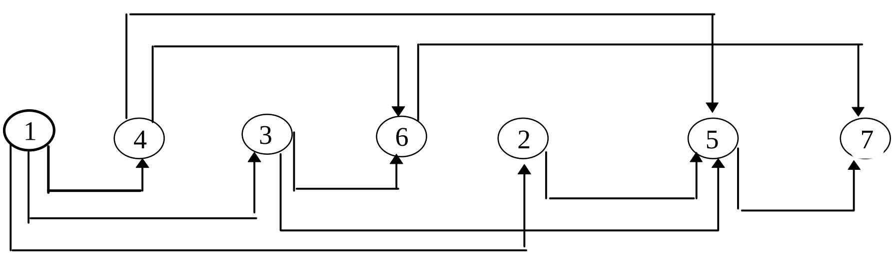


The project is completed when task 7 is done.
Task has to wait for 5 and 6 to be done.

In an activity Network,
$i \rightarrow j$ means that i must be completed before j begins.
and so $i \rightarrow i$ is impossible.

Topological Order

From a DAG, D, with PO (Partial Order) $\rightarrow$ ("reaches"), we can generate a sequence of vertices S with the following property;
if $i \rightarrow j$ in D then i precedes j in the S.
Such a sequence S is said to in Topological Order. For a given DAG there may be more than one such order. In effect, the partial order of the DAG is embedded in the sequence.
A topological order for the above DAG, is



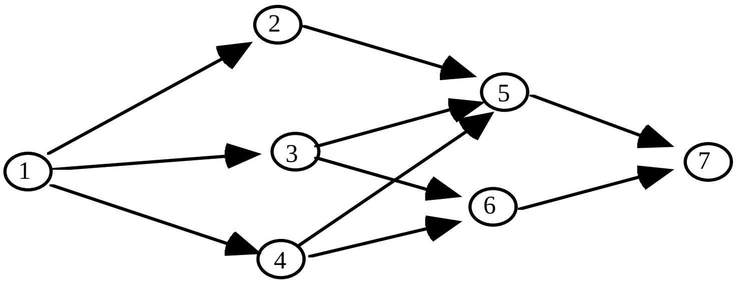
Alogrithm for Topological Sort

Given a DAG, write a routine that will ouput the vertices of D in a Topological Order.

Abstract algorithm:

```
until
        no more vertices
loop
        Select a vertex v, with in-degree 0 (i.e. no predecessors)
                output v
        Delete v (and all arcs leading from v)
        end
```

Example:



Reading in a Digraph for Topological Sort:

As well as keeping track of the neighbours of a vertex, we need to also need to know the In_Degree of the Vertex. A Digraph D is an array of VERTEX_D
i.e. D : ARRAY[VERTEX_D] where

```
class VERTEX_D   -- Vertex Properties
creation
        make
feature
        In_Degree : INTEGER
        Adj_L : LIST_SET[INTEGER]

        Degree_Set(n : INTEGER) is
        do
                In_Degree := n
        end -- Degree_Set

        make is
        do
                !!Adj_L
        end -- make

end -- VERTEX_D
```

To input a Digraph we assume the input is given as ordered pairs (the arcs) e.g. for the above the input could be

1 2 1 3 1 4
2 5
3 5 3 6
4 5 4 6
5 7
6 7

To read in a Digraph we can use,

```
Read_Digraph is
        local
                i,j,k,ind : INTEGER  -- i, j are vertices
                vx : VERTEX_D
        do
                 !!D.make(1,size)
                 from
                         k := 1
                 until
                         k > size
                loop
                         !!vx.make
                         D.put(vx,k)
                         k := k+1
                end
                from
                         io.read_integer
                until
                         io.end_of_file
                loop
                         i := io.last_integer
                         io.read_integer
                         j := io.last_integer
                         D.item(i).Adj_L.add(j)
                         ind := D.item(j).In_Degree
                         D.item(j).Degree_Set(ind+1)
                         io.read_integer
                 end
        end --Read_Digraph
```

Outputting a Digraph is as for (undirected) graphs.

```
Topol_Sort is
	local
		Zero_V : QUEUE[INTEGER]
		L : List_Set[INTEGER]
		k, z, it, degree : INTEGER
	do
		!!Zero_V.make
		from
			k := 1
		until
			k > size
		loop
			if D.item(k).In_Degree = 0 then
				Zero_V.add(k)
			end
			k := k+1
		end  -- Zero_V is a queue of vertices with in-degree 0
		from
		until
			Zero_V.Empty
		loop
			z := Zero_V.item
			Zero_V.remove
			io.put_int(z)
			io.put_string("  ")
			L := D.item(z).Adj_L
			from
				L.first
			until
				L.off
			loop
				it := L.item
				degree := D.item(it).In_Degree - 1
				D.item(it).Degree_Set(degree)
				if degree = 0 then
					Zero_V.add(it)
				end
				L.forth
			end
		end
	end -- Topol_Sort
```

This algorithm is very similar to Breadth First Traverse. Since the digraph is a DAG, there will be at least one vertex with In-degree 0. There may be more than one. The algorithm collects the these zero-in-degree vertices in a Queue, Zero_V. Until Zero_V is empty the (front) item of Zero_V is processed. The item is printed and then in effect deleted from the digraph by decrementing the in-degree of each of its neighbours. Any neighbour then with in-degree 0 is added at the end of the queue, Zero_V.