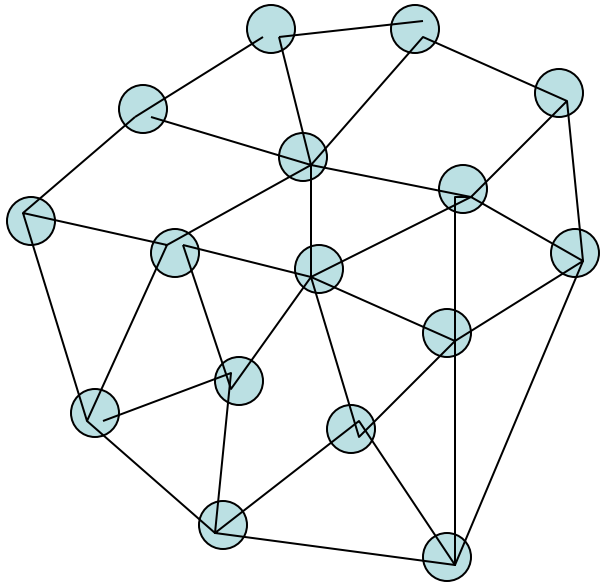
Graphs

Graphs

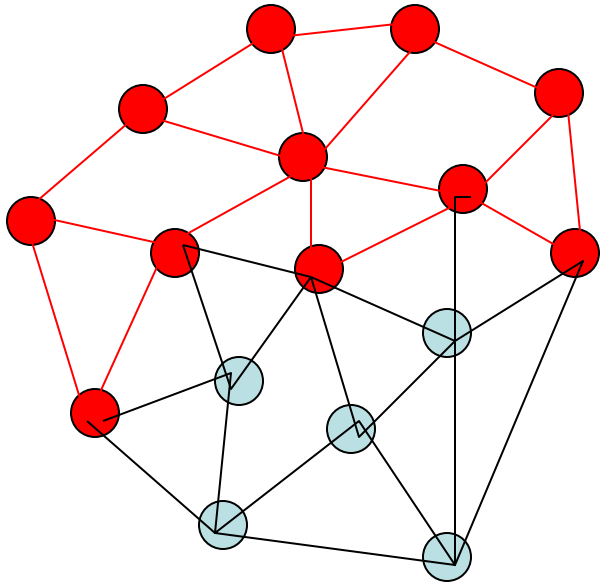
- ***Many interesting situations can be modeled by a graph.***

  Ex. Mass transportation system, computer network.

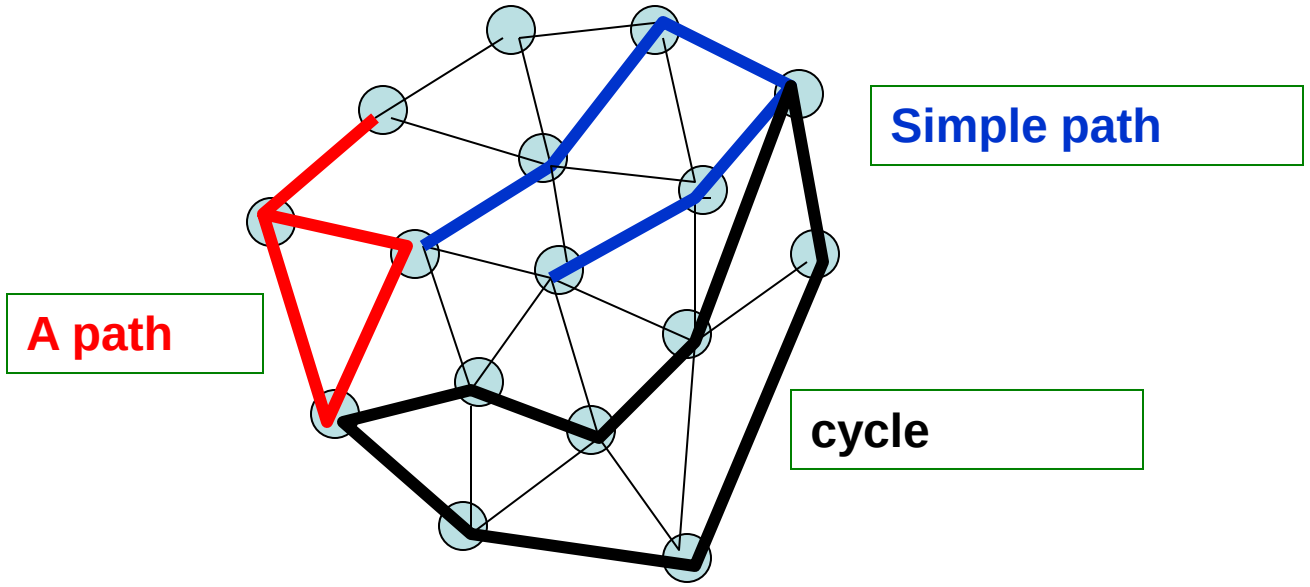


Terminology

- A graph consists of a set of nodes (vertices) and a set of edges.
- A node is basic component, which usually contains some information.
- An edge connects two distinct nodes (vertices).
- A subgraph is graph which consists of a subset of nodes (vertices) and a subset of edges of a graph.

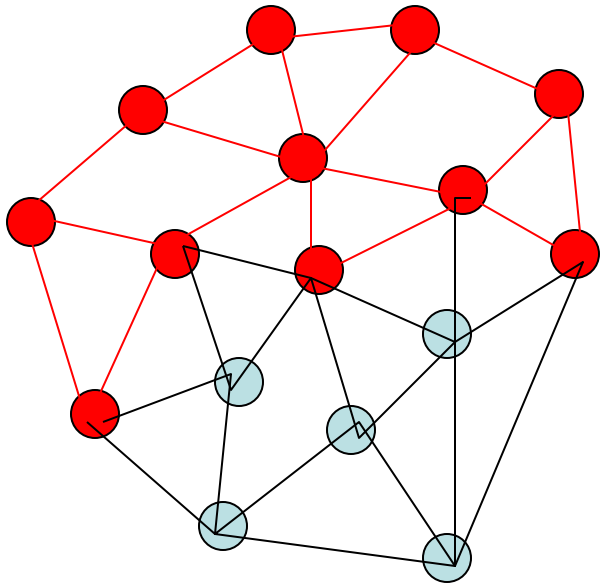


- A path is a sequence of nodes such that each successive pair is connected by an edge.
- A path is a simple path if each of its nodes occurs once in the sequence.

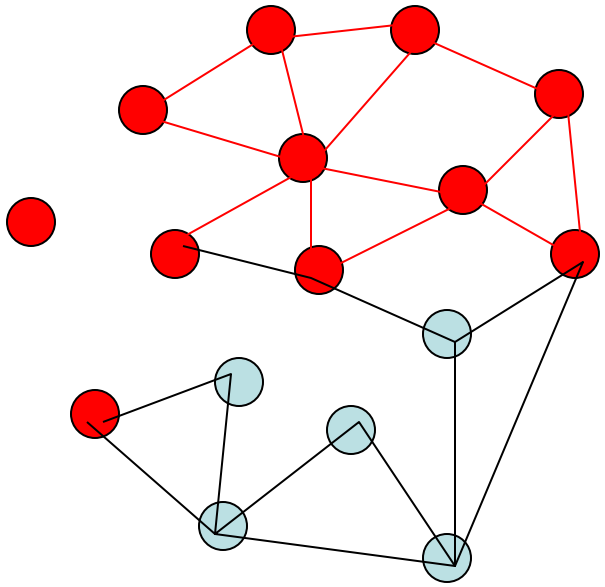


- A cycle is path that is a simple path except that the first and last nodes are the same.

- A graph is a connected graph if there is a path between every pair of its nodes.

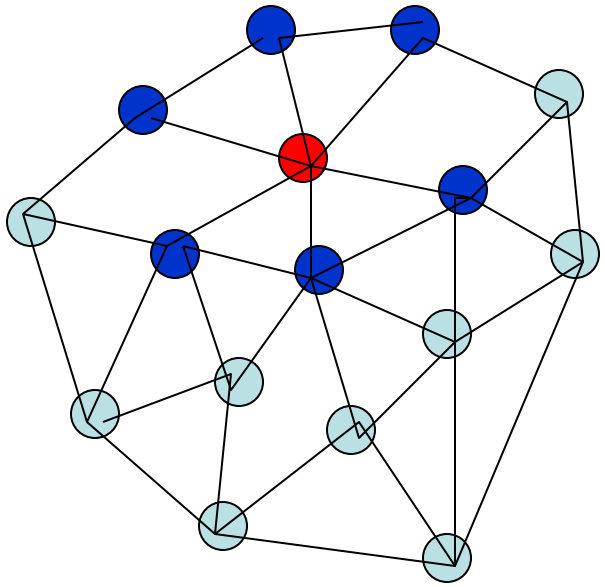


Connected graph



Not a connected graph

- Two nodes are adjacent nodes if there is an edge that connects them.

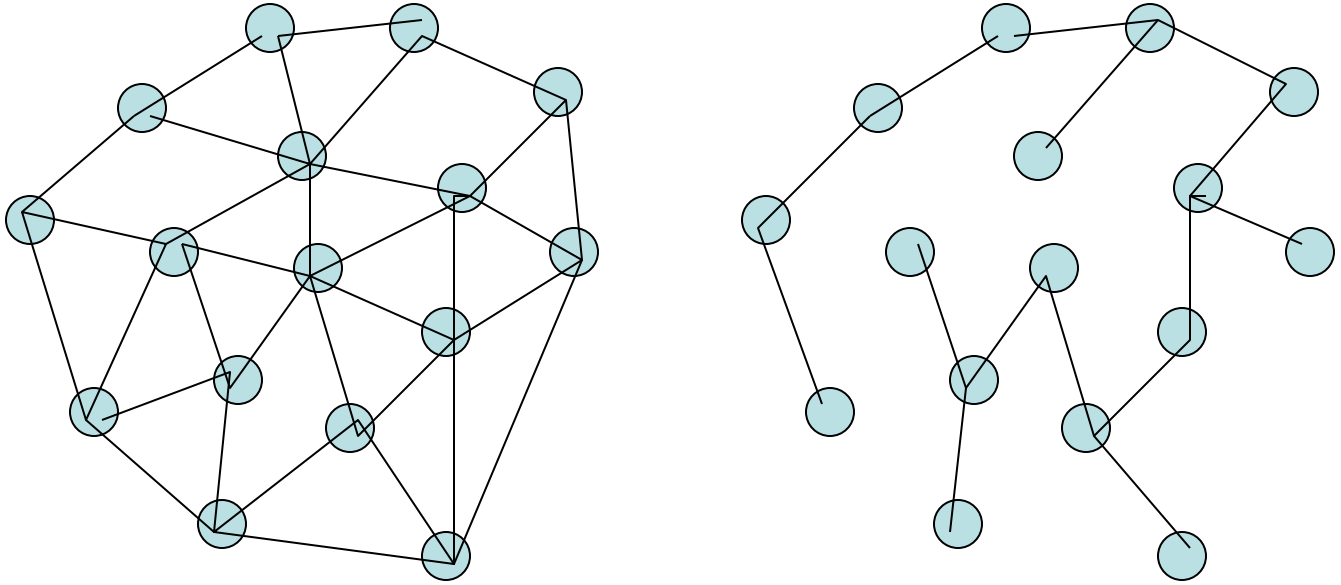


- Neighbors of a node are all nodes that are adjacent to it.
- A Tree is the special case of a graph that

  (i) is connected

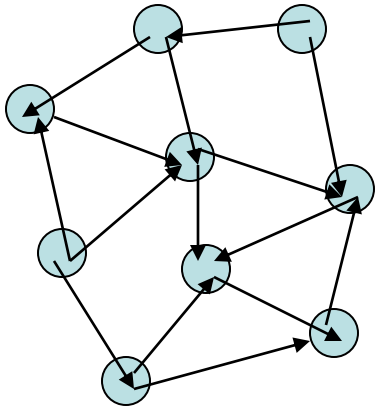
  (ii) has no cycle

- *If a connected graph has n nodes and n-1 edges, then it is a tree. This tree is called Spanning Tree.*

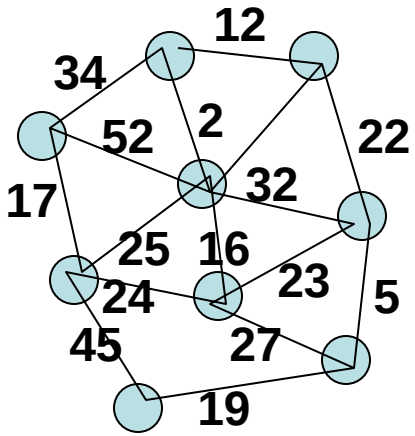


- *An oriented tree is a tree in which one node has been designated the root node.*

- *A directed graph or digraph is a graph in which each edge has an associated direction.*



- *A weighted graph is a graph in which each edge has an associated value.*



Specification of Graph

Elements: *A graph consists of nodes and edges*

Structure: *An edge is a one-to-one relationship between a pair of distinct nodes. A pair of nodes can be connected by at most one edge, but any node can be connected to any collection of other nodes.*

Domain: *The number of nodes (vertices) in a graph is bounded.*

Operations:

InsertNode (Type e)

Requires: *G is not full.*

Results: *If G does not contain an element whose key value is e.key then e is inserted in G and inserted is true, otherwise inserted is false.*

InsertEdge (*Key k1, k2*)

Requires: *G is not full and k1!=k2.*

Results: *If G contains two nodes whose key values are k1 and k2 then G contains an edge connecting those nodes. If the two nodes were connected by an edge before operation InsertEdge then inserted is false; otherwise inserted is true.*

DeleteNode (*Key k*)

Results: *G does not contain an element whose key value is k. if G contained a node before this operation with key value k then deleted is true and no edge that connected this node to an other node is in G; otherwise deleted is false.*

DeleteEdge (*Graph g, Key k1, k2*)

Results: *G does not contain an edge that connects nodes whose key values are k1 and k2. If G contained such an edge before this operation then deleted is true; otherwise deleted is false.*

Update (*T e*)

Results: *If G contained a node with key value e.key then the element in the node is e and updated is true. Otherwise updated is false.*

Retrieve (*key k, T e*)

Results: *If G contains a node whose key value is k before this operation then e is that element and retrieved is true; otherwise retrieved is false.*

Full ()

Results: *If G is full then Full returns true; otherwise Full returns false.*

Representation of a Graph

- *There are two approaches to representing graphs*

  (*i*) *Adjacency Matrix*

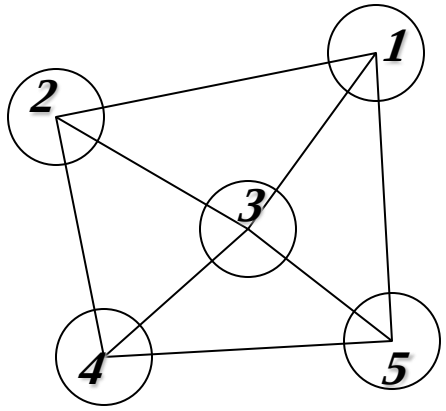
  (*ii*) *Adjacency List*

Adjacency matrix

A two dimensional array whose components are of type Boolean and whose index values correspond to the nodes.

EX.

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



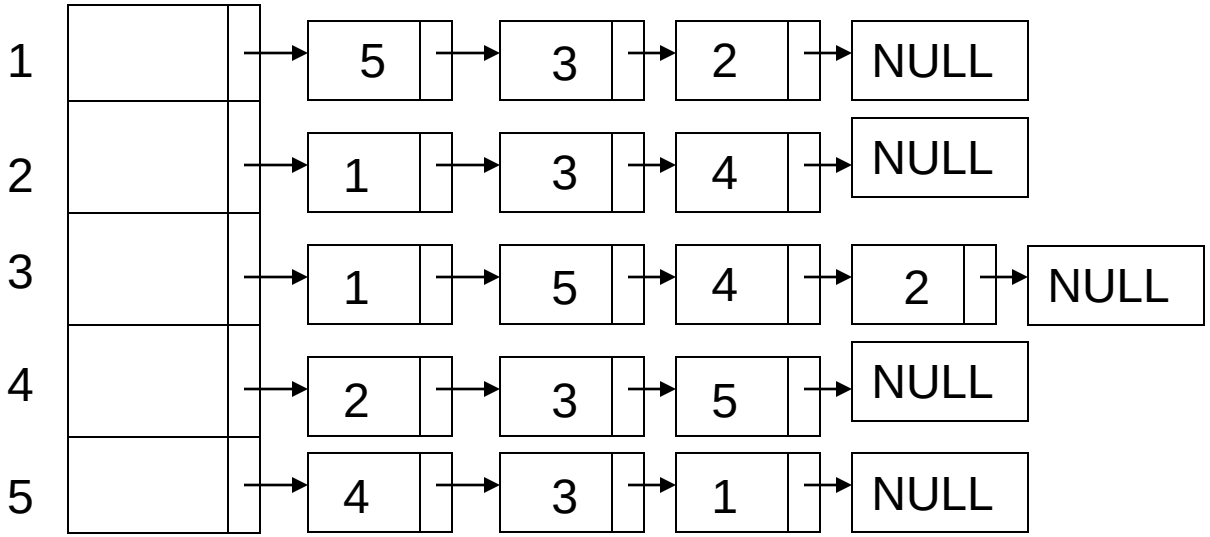
If A is the adjacency matrix corresponding to a graph G, then A_{ij} corresponds to the edge (i, j), and is true if there is an edge between the vertices i, and j.

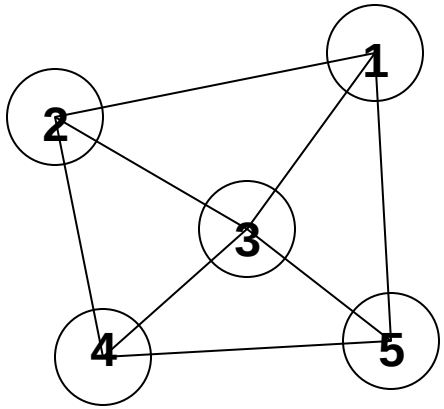
- *This representation is very simple, but the space requirement is $O(n^2)$ if the number of vertices is n.*
- *This representation is suitable only if the graph is dense with the number of edges i.e. $O(n^2)$, n being the number of vertices. But in most of the applications this is not true.*

Adjacency list

It is an array where each cell corresponds to a vertex of a graph and stores the header of a list of all adjacent vertices.

Ex. The following is an adjacency list corresponding to the graph on the right.





- *This is an ideal representation if the graph is not dense.*
- *In this case the space requirement is O(e + n) where e is the number of edges and n is the number of vertices.*

Traversal of a Graph

Process each node of the graph only once

- *There are two methods for graph traversal*

  (*i*) ***Breadth First Search***

  (*ii*) ***Depth First Search***

Breadth First Search (BFS)

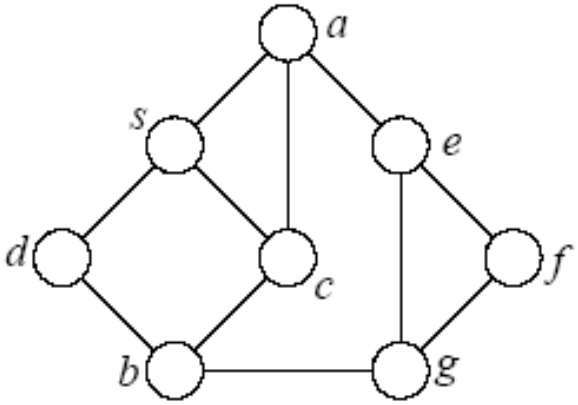
Start at some source node s and visit all its neighbors, then visit the neighbors of each neighbor of s and so on.

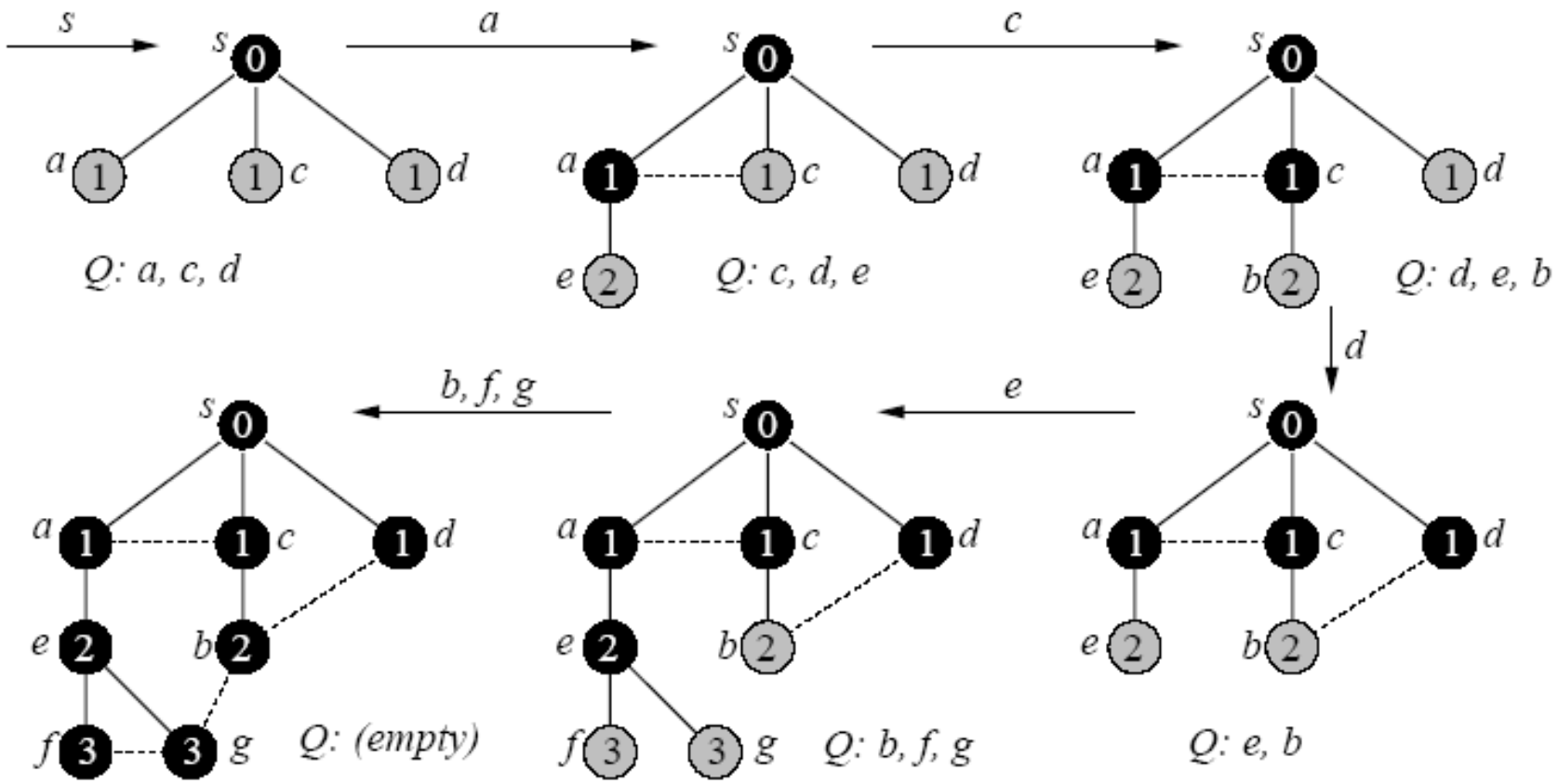
ALgorithm

1. *Assign the status of waiting to all nodes of a graph.*
2. *Start with source node s, and put it in queue*
3. *Dequeue one node from the queue, assign it the status of processed, and assign its neighbors the status of ready and put them in the queue.*
4. *Repeat Step 3 until the queue is empty.*

Example

Consider a graph with nodes a, b, c, d, e, f, g, s





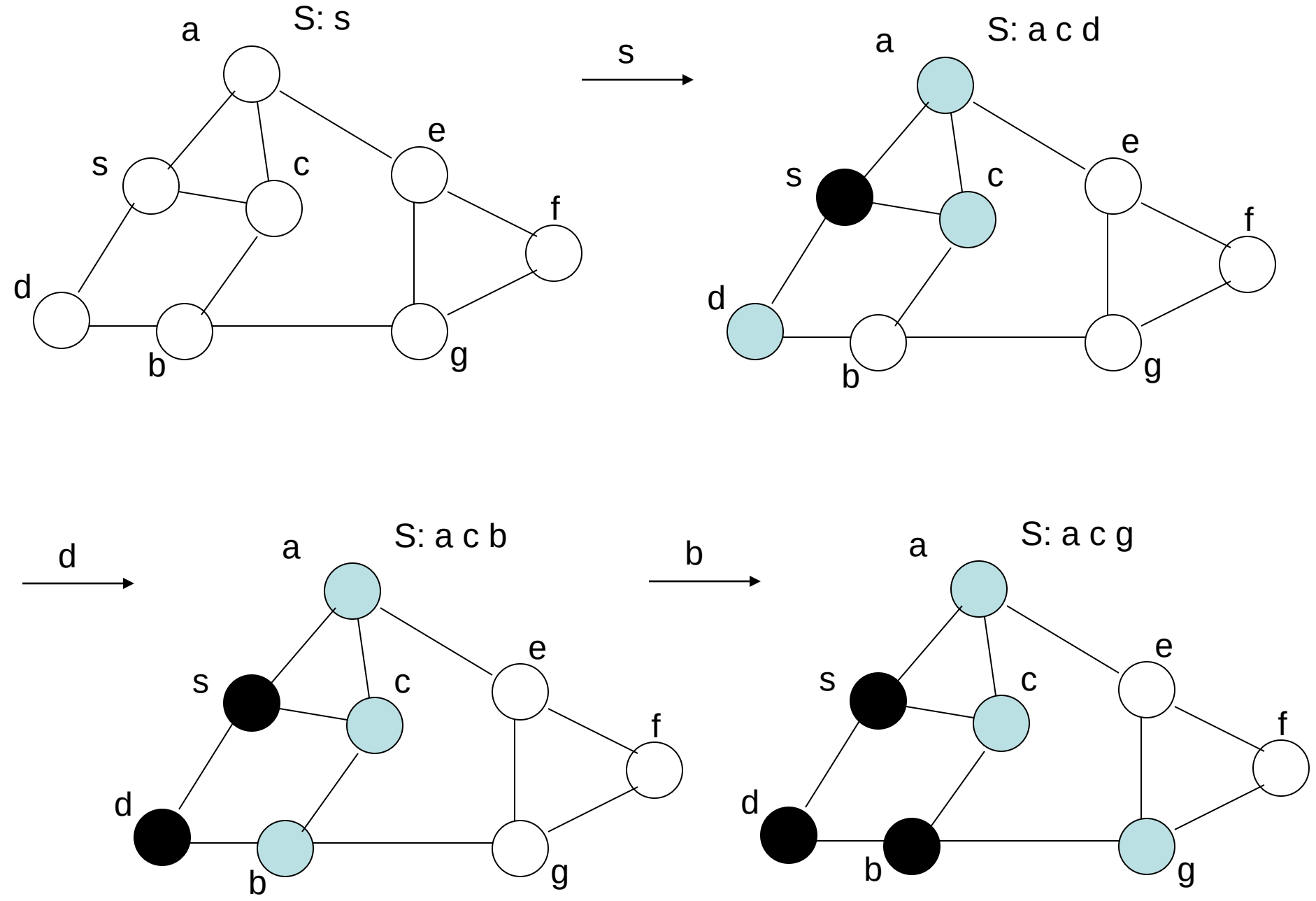
Pseudocode for BFS

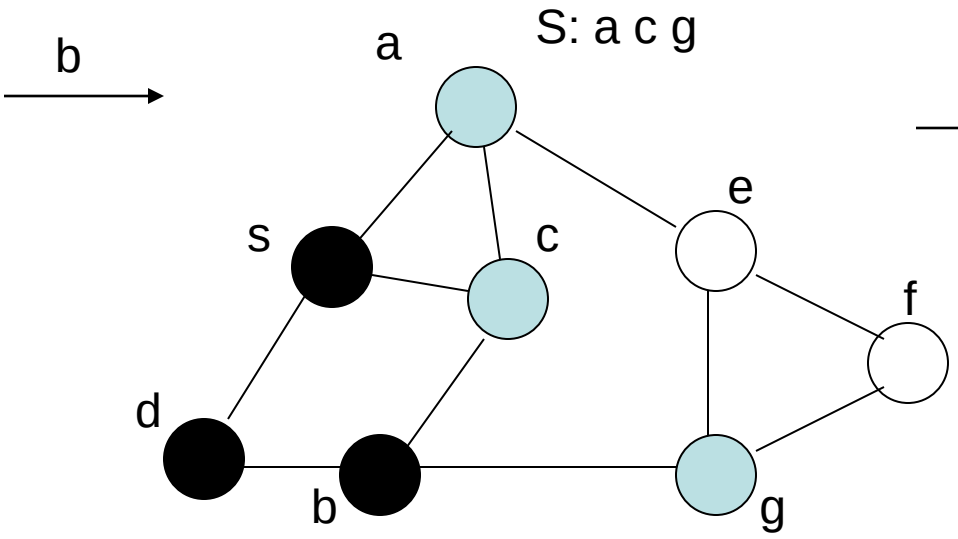
```
BFS(G, s)
{
      for each node u of G
            set status[u] = waiting;
      status[s] = ready;
      Q = {s};
      while( !Q.IsEmpty)
      {
            u = Q.dequeue();
            for each v in Neigh[u]
            {
                  if(status[v] = waiting)
                  {
                        staus[v] = ready;
                        Q.enqueue(v);
                  }
            }
            status[u] = processed;
      }
}
```

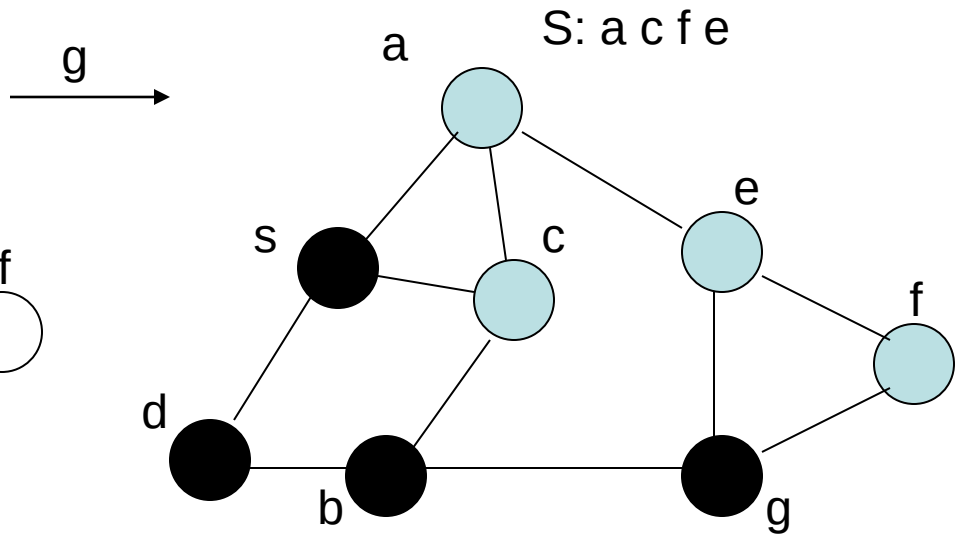
Depth First Search (DFS)

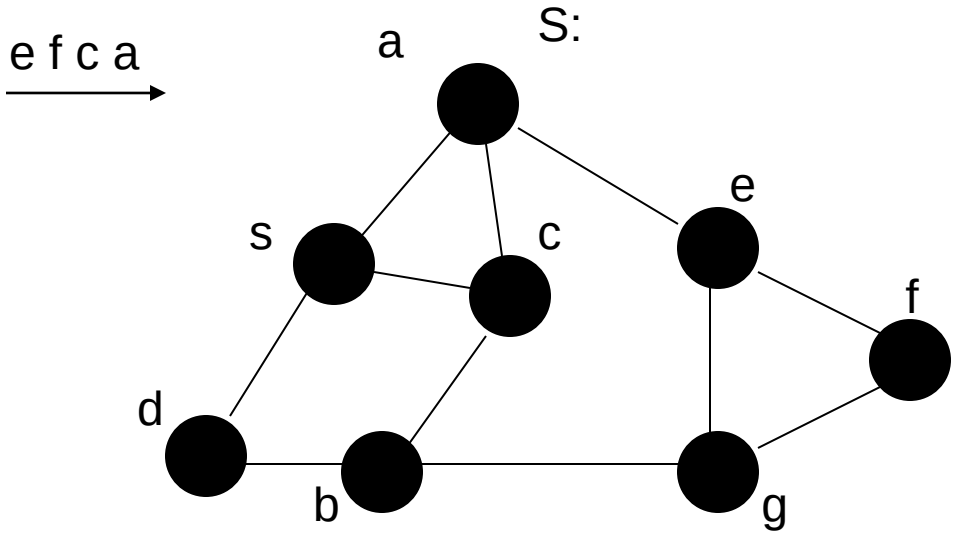
DFS visits one node and then visits one of its neighbors and puts the rest of its neighbors into a stack and so on.

1. *Assign the status of **waiting** to all nodes of a graph.*
2. *For each node of the graph, follow the following steps,*
    a) *Change the status of the node to be **ready** and put in a stack*
    b) *Repeat the following steps until the stack is empty*
    c) *Get a node from the stack, **process** it, change its status to **processed** and change the status from waiting to ready of its neighbors and put them in the stack*
    d) *Go to step b*


S: s
a
s
c
e
f
d
b
g
s
S: a c d
a
s
c
e
f
d
b
g
d
S: a c b
a
s
c
e
f
d
b
g
b
S: a c g
a
s
c
e
f
d
b
g


b
S: a c g
a
s
c
e
f
d
b
g


g
S: a c f e
a
s
c
e
f
d
b
g


e f c a
S:
a
s
c
e
f
d
b
g

Pseudocode for DFS

```
BFS(G)
{
        for each node u of G
                status[u] = waiting;
        for each node u of G
                if(status[u] == waiting)
                        Visit(u)
}
Visit(u)
{
        stack S;
        status[u] = ready;
        S.push(u)
        while( S.IsEmpty)
        {
                v = S.pop();
                for each w in Neigh[v]
                {
                        if(status[w] = waiting)
                        {
                                staus[w] = ready;
                                S.push(w);
                        }
                }
                status[v] = processed;
        }
}
```

Shortest Path (*unweighted*)

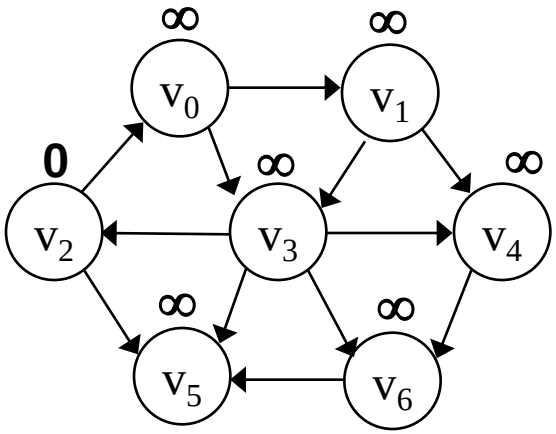
- *The problem*: *Find the shortest path from a vertex v to every other vertex in a graph G*
- *The unweighted path measures the number of edges, ignoring the edge's weights* (*if any*)

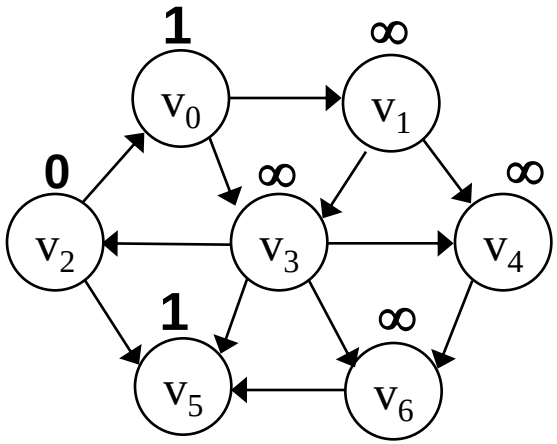
Finding Shortest Unweighted Path (Simple Algorithm)

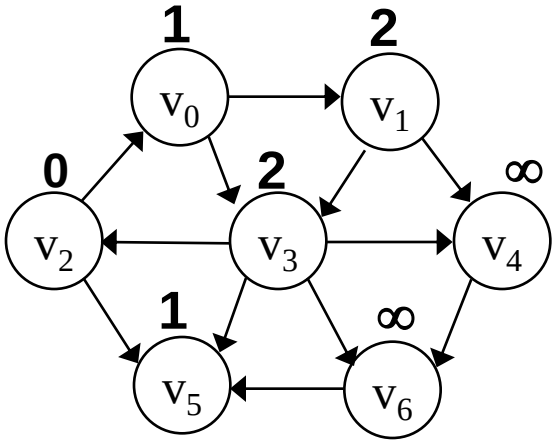
For a vertex v, d_v is the distance between a starting vertex and v

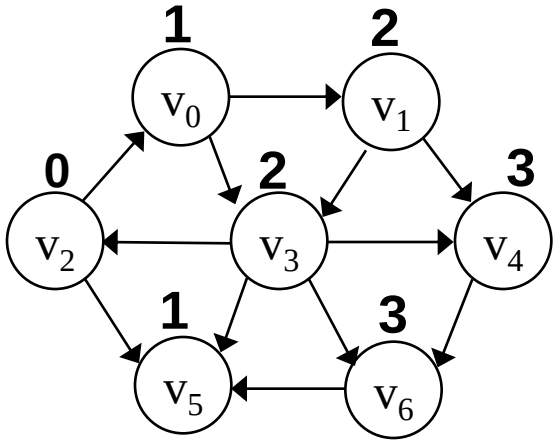
1. *Mark all vertices with d_v = infinity*
2. *Select a starting vertex s, and set $d_s = 0$, and set shortest = 0*
3. *For all vertices v with d_v = shortest, scan their adjacency lists for vertices w where d_w is infinity*
   - *For each such vertex w, set d_w to shortest+1*
4. *Increment shortest and repeat step **3**, until there are no vertices w*

Simple algorithm: example



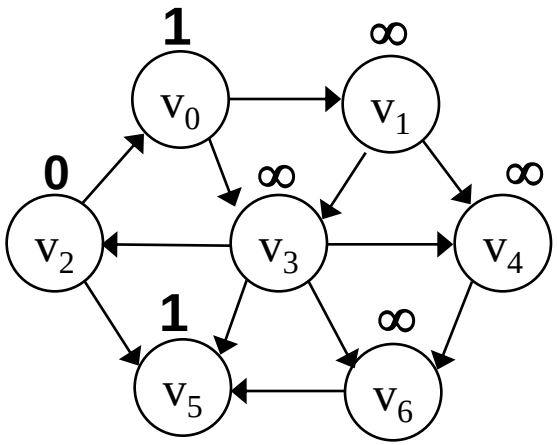




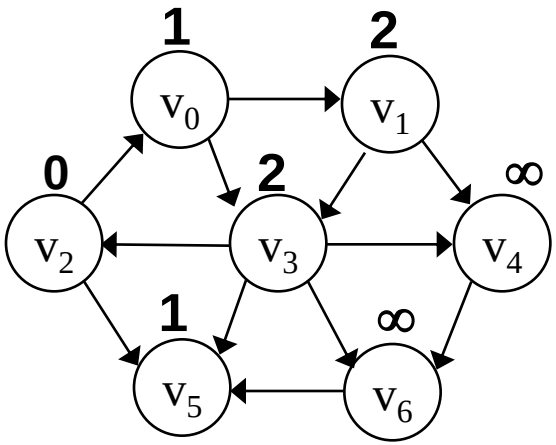


Simple algorithm: using queues

- *In step **3** of the simple algorithm, we search through the entire graph for vertices v with d_v = shortest*
- *Instead, because we deal with exactly these vertices in the previous iteration of step **3**, we can queue them*



Queue: v_0, v_5



Queue: v_5, v_1, v_3

Shortest weighted path: Dijkstra's (Dike-Struh) algorithm

- *What is the lowest-cost path between two vertices in a weighted graph?*
- *Only works on graphs all of whose weights are positive*
- *Works in a similar way to the unweighted algorithm, except we possibly adjust the values in the vertices more than once*

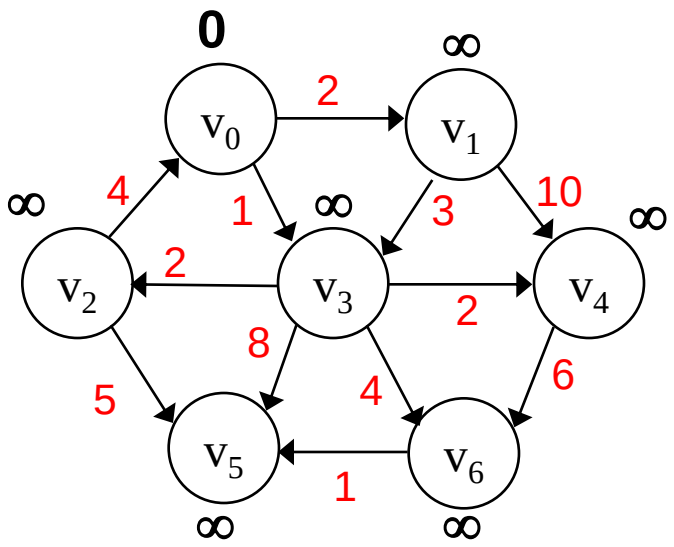
Dijkstra's (Dike-Struh) Algorithm

Initialise all costs to infinity
Select a start vertex
current_v = start vertex
Repeat
  For every adjacent vertex ***v*** *of* ***current_v***
    If (cost of ***current_v*** *+ weight of edge to* ***v****) < cost of* ***v***
      Reduce cost of ***v*** *to (cost of* ***current_v*** *+ weight of edge* ***v****)*
  Select lowest cost vertex as ***current_v*** *and examine adjacent vertices*
Until all vertices have been processed

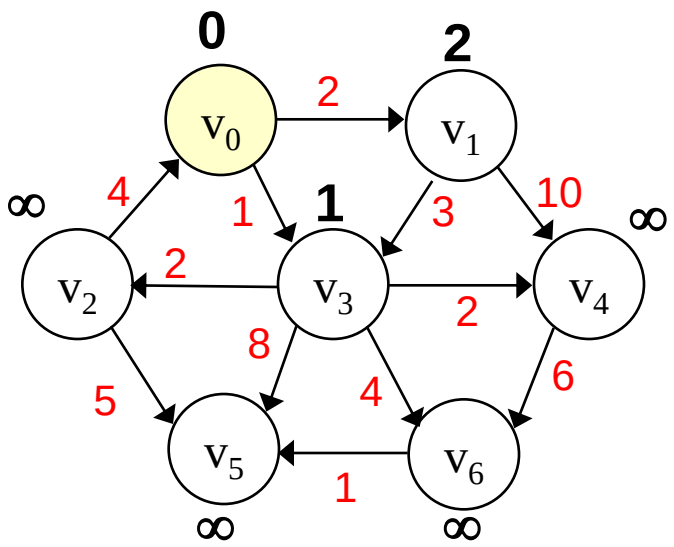
NOTE:

- *We can use an 'inverse' priority queue, in order to select the vertex with lowest cost*
- *Algorithm based on breadth-first traversal*

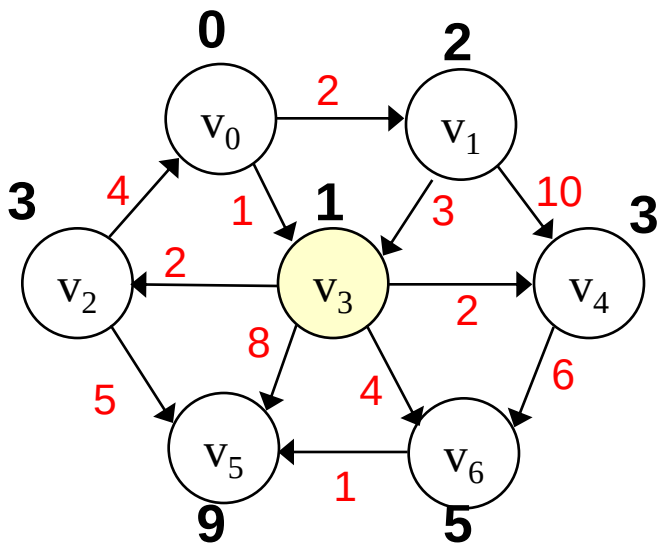
Example



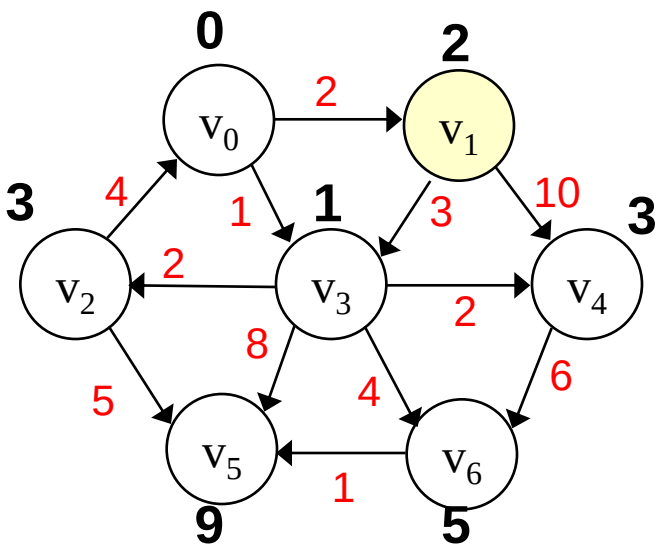
Queue: v_0



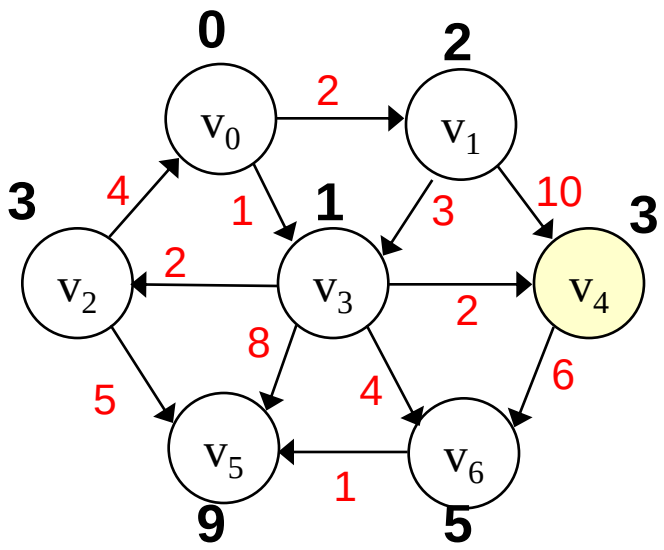
Queue: v_3, v_1



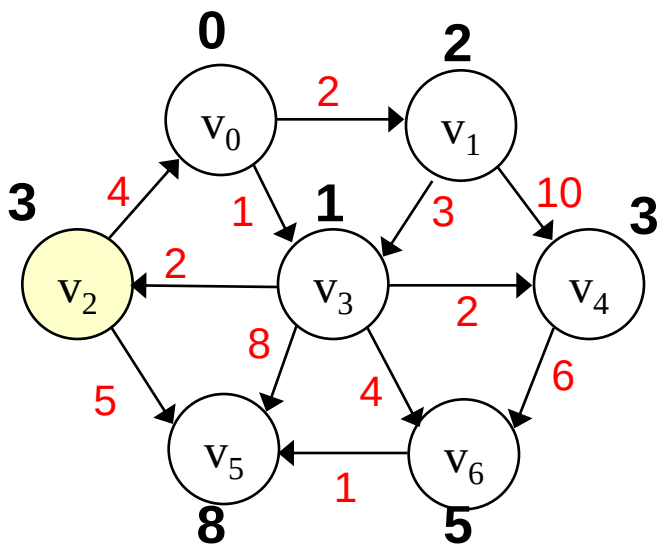
Queue: v_1, v_4, v_2, v_6, v_5



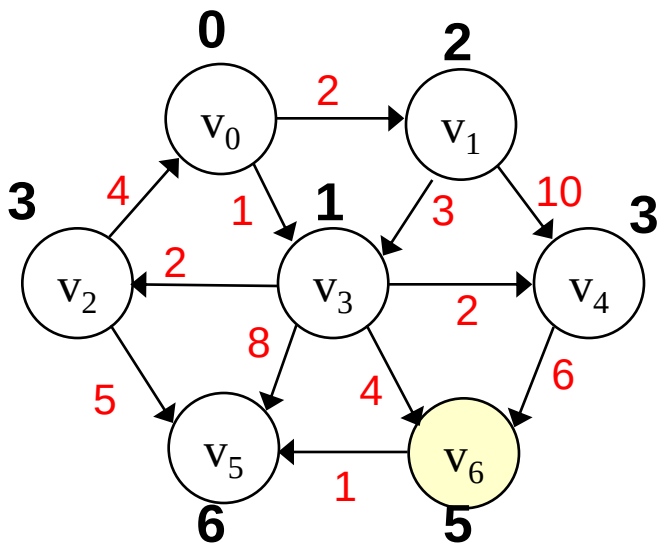
Queue: v_4, v_2, v_6, v_5

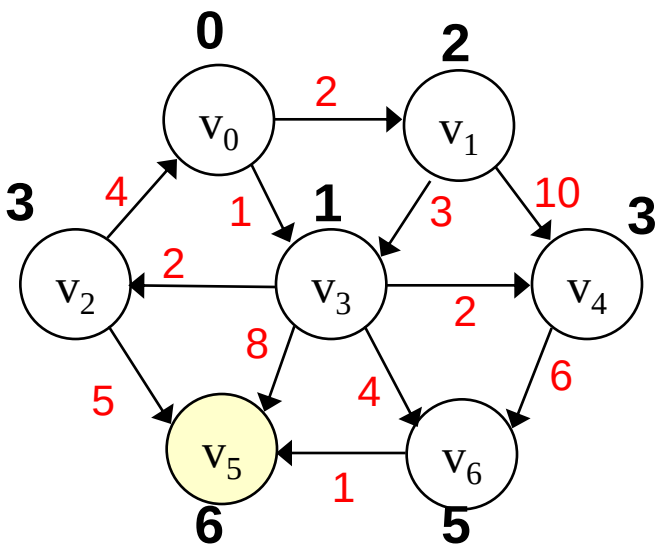


Queue: v_2, v_6, v_5



Queue: v_6, v_5





Queue: v_5

Queue:

The shortest path from v_0 to v_6 is $v_0v_3v_6$ and the shortest path from v_0 to v_4 is $v_0v_3v_4$

Complexity of Dijkstra's algorithm

- The time complexity is $O(|V|^2 + |E|)$
- For dense graphs where $|E| = O(|V|^2)$, the algorithm is optimal and the time complexity is $O(|V|^2)$
- For sparse graphs, it can be improved to give a time complexity of $O(|E| \log |V|)$