Graph Algorithms-1

Graph Algorithms-1

Topics

- Graph terminology

- Graph Representations
  - *-Matrix Representation*
  - *-Linked list Representation*

- Depth First Search Algorithm (DFS)
  - *- Strategy*
  - *- DFS Applications*
  - *- DFS Visualization*
  - *-DFS Analysis*

- Breadth First Search (BFS)
  - *- Strategy*
  - *- BFS Applications*
  - *- DFS Visualization*
  - *-BFS Analysis*

Graph Terminology

Graphs

Definition

A ***graph*** *G =(V,E)* is a set of vertices *V* and a set of edges *E*, where

$$E \subseteq V \times V$$ (*Subset of cross-product of vertices, called binary relation*)

Example: *Let G= (V, E)* where

V = {a, b, c, d}

E = {(a, b), (a, d), (b, c),(b, a), (d, c), (d, b), (d, d)}

The number of vertices and edges are given by the cardinalities $|V|$ and $|E|$ of the corresponding sets. The sample graph consists of four vertices and seven edges. Thus,

$|V|= 4$ and $|E|=7$

➢ Graphs are usually represented by pictorial diagrams. The vertices can be shown as labeled *circles* or *rectangles*. The edges are depicted as *arcs* or *lines*. Except for some applications, in which distances among vertices are important , the positions of the vertices are, in general, immaterial. Thus, graphs can be shown pictorially in several ways.

Graphs

Undirected Graph

A graph *G=(V,E)* with vertex set *V={v1, v2, v3,……vn }* is called ***undirected*** *if*

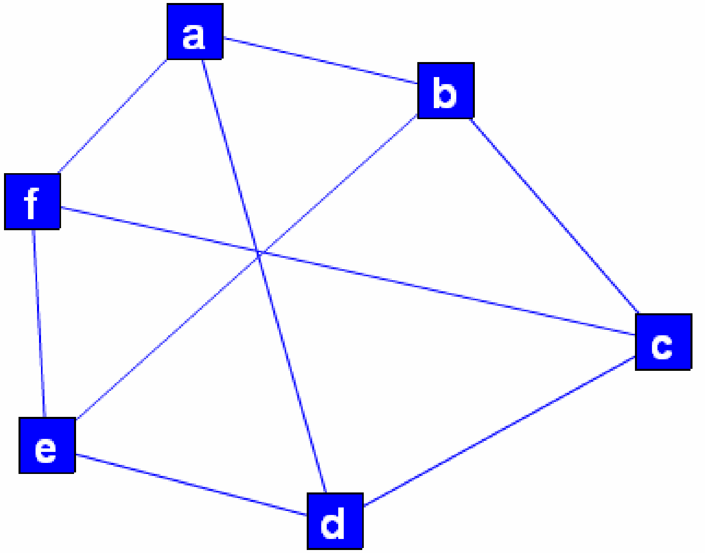
(vi, vj) = (vj, vi) for i ≠ j

An undirected graph is sometimes referred to as ***undigraph***.

In pictorial representation of undirected graph, the edges are not assigned any direction.

Example: Figure shows an undirected graph:

G=(V,E), V={a, b c, d, e, f}, E={(a,b), (a,d), (a,f), (b,c), (b,e), (c,f), (c,d), (d,e), (e,f)}



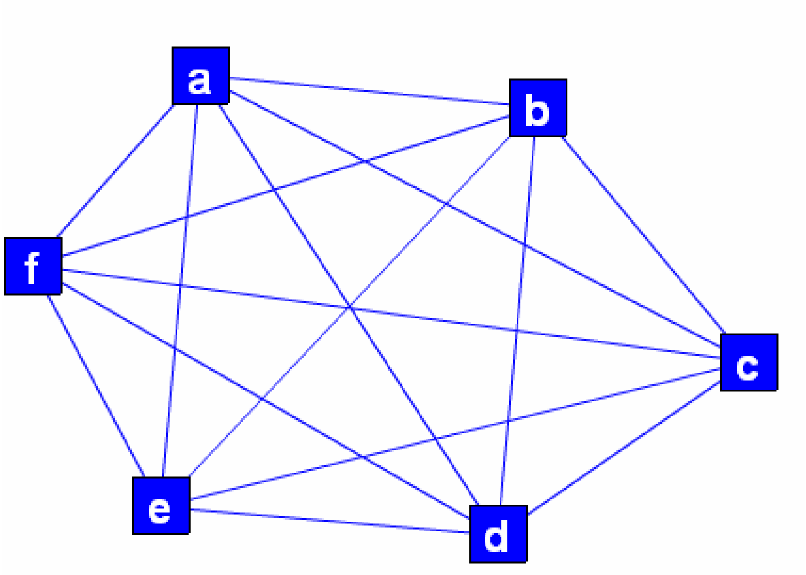
A sample undirected graph

Undirected Graphs

Complete Graph

A graph which has links among all of the vertices in a graph is referred to as ***complete.***

Example: The figure below shows a complete graph.



Sample undirected complete graph

- An ***undirected complete graph*** , with *n* vertices, has *n(n-1)/2 edges*. The ***space complexity*** of complete graph is $O(n^2)$. A complete graph is ***dense***. By contrast, a graph with space complexity *O(n lg n)* is called ***sparse***.

Graphs

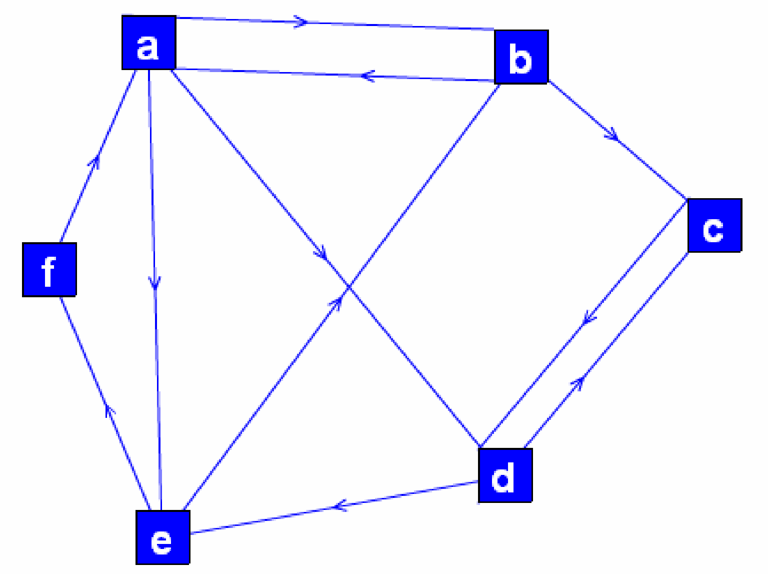
Directed Graph

A graph *G=(V,E)* with vertex set *V={v1, v2 ,v3,……vn }* is called ***directed*** *if*

(vi, vj) ≠ *(vj, vi) for i* ≠ *j*

In other words, the edges *(vi, vj)* and *(vj, vi),* associated with any pair of vertices *vi, vj* , are considered ***distinct***. In pictorial representation these are shown with arrows.

Example: The figure below shows a directed graph



A sample directed graph

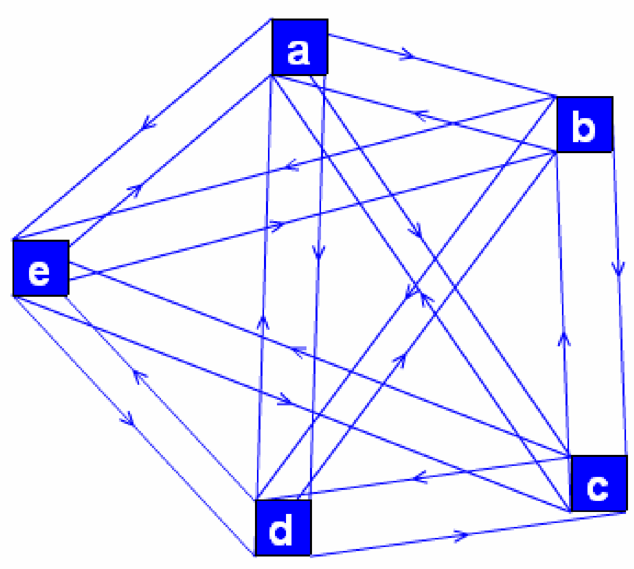
- The directed graph is often referred to ***digraph*** or ***network***.

Directed Graphs

Complete Graph

A ***complete directed graph*** has links among all of the vertices.

Example: The figure shows a directed complete graph



A sample directed complete graph

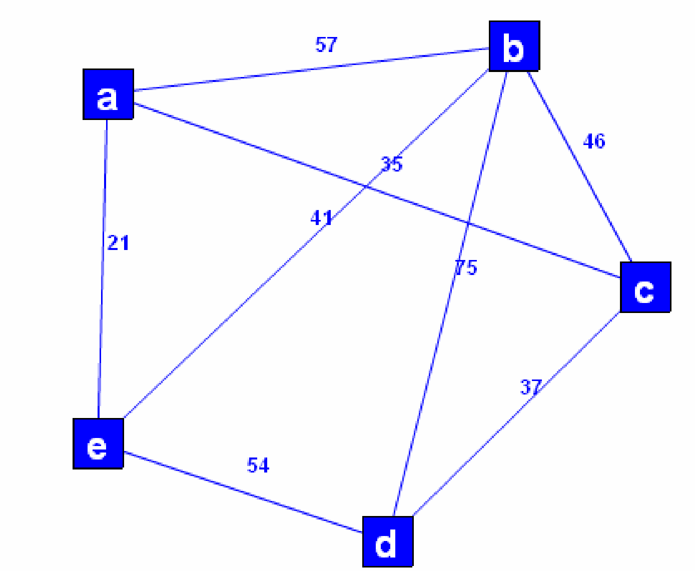
- A complete directed graph with *n* vertices has *n(n-1)* edges

Weighted Graphs

Undirected Graph

A graph in which ***labels*** or ***values*** *w1, w2, w3,…* …are associated with edges is called ***weighted*** graph. Weights are typically ***costs*** or ***distances*** in different applications of graphs.

Example: The figure shows an example of *weighted undirected* graph.



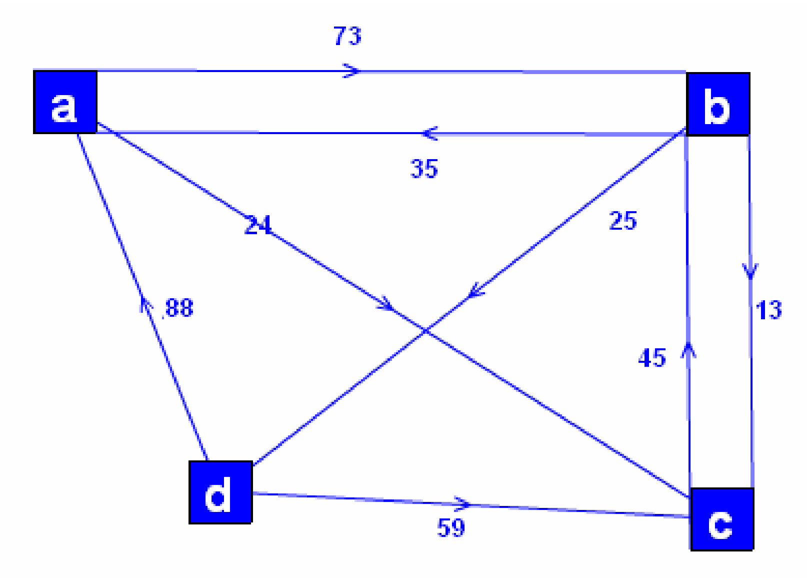
A sample undirected weighted graph

Weighted Graphs

Directed Graph

A weighted graph can also be ***directed***. The weights attached to the edges between the same pair of vertices may ,in general, be different

Example: The figure below shows an examples of the *directed weighted* graph. Note that weight of edged from vertex *a* to vertex *b* is 73 and that from vertex *b* to vertex *a* is 35



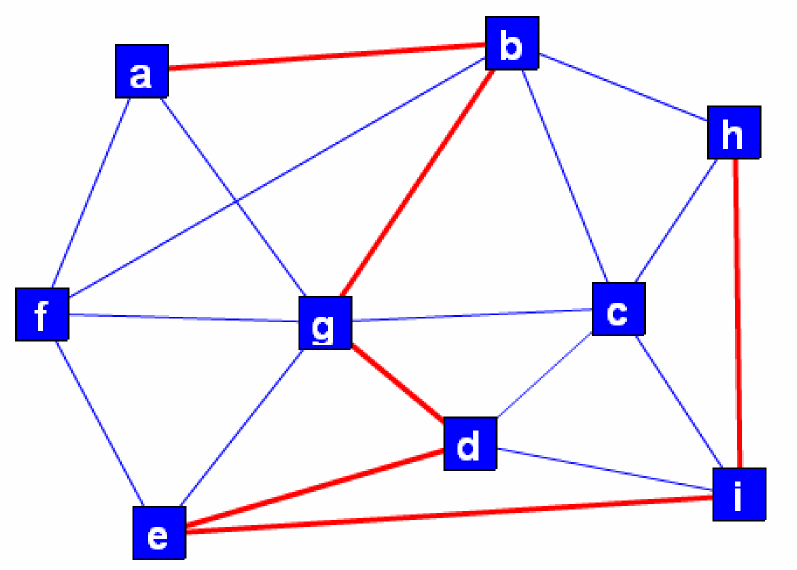
A sample directed weighted graph

Graph Paths

Definition

A ***path*** is a sequence of *adjacent vertices*. Two vertices are called adjacent if they have link or connecting edge. The path is denoted by enclosing the vertices in a pair of square brackets . In a path, the vertices may repeated

Example: The figure below depicts a path *P* = [*a, b, g, d, e, i, h*] in a sample graph. The path is shown in bold red lines.



Path P=[a, b, g, d, e, i, h]

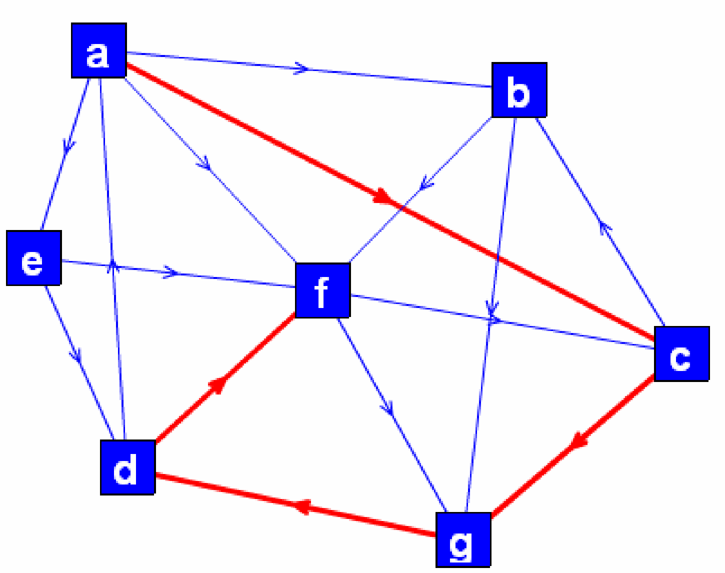
The number of edges connecting the vertices in a path is called *path length.* The path length in the above example is 6.

Graphs Paths

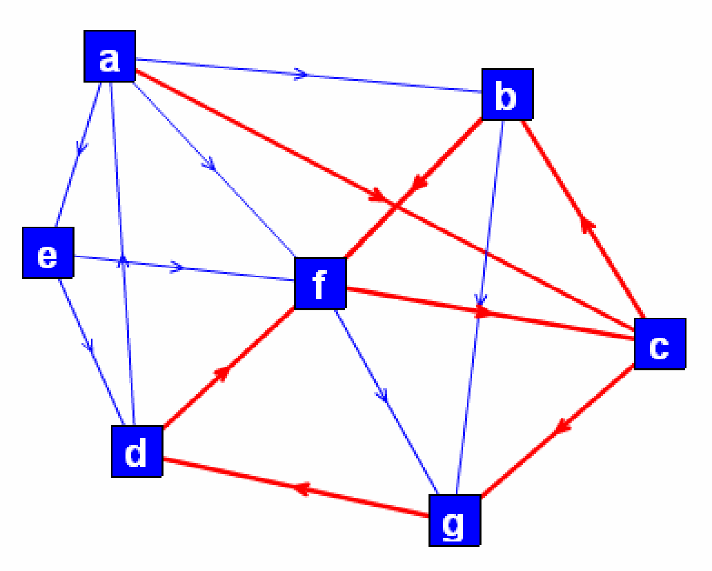
Simple Path

A path is called *simple* if no vertices are repeated; otherwise, the path is referred to as non-simple.

Example: The figures below show *simple* and *non-simple* paths in a graph. The path P_1=[*a, c, g, d, f*] is simple. The path P_2=[*a, c, g, d, f, c, b, f*] is non-simple, because it passes through vertices *c, f* twice



(a) A simple path P1=[a ,c ,g, d, f]



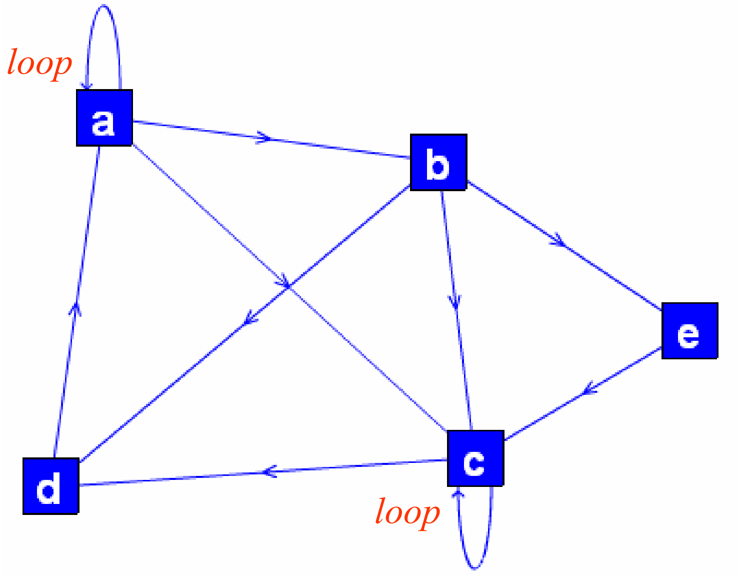
(b) A non- simple path P2=[a ,c ,g, d, f, c, b, f]

Graphs Paths

Loops

A ***loop*** is special path that originates and terminates at a single node, and does not pass through other vertices.

Example: The figure depicts two loops in a graph, at vertices *a* and *c*



A sample graph with two loops

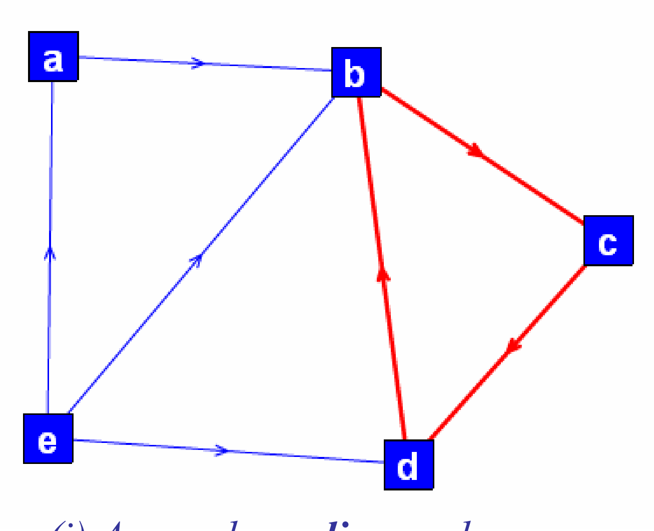
- The loops are important in certain some applications. For example, loops represent certain states in Finite State Automata

Graphs Paths

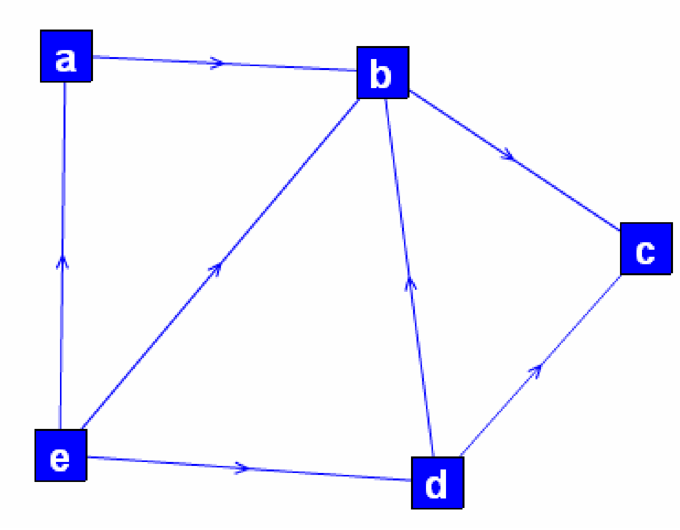
Cyclic and Acyclic Graphs

A path that originates and terminates at the ***same vertex***, and links ***two or more vertices***, is called ***cycle***. If a graph contains a cycle it is called ***cyclic***. By contrast, a graph which contains no cycles is known as ***acyclic***.

Example: In diagram *(i)*, the path *P=[b, c, d, a]* is a cycle. The diagram *(ii)* represents a acyclic graph.



*(i) A sample **cyclic** graph*



*(ii) A sample **acyclic** graph*

Graph Representations

Graphs Representation

Adjacency Matrix

One of the standard ways of representing a graph uses a matrix to denote links between pairs of vertices in a graph. The matrix is known as ***adjacency matrix.***

The adjacency matrix can be defined mathematically. Let $G=(V,E)$ be a graph, with $V=\{v1,v2,\ldots\ldots vn\}$, and $E=\{(v1,v2),(v1,v3)\ldots.\}$ where $n=|V|$. The adjacency matrix $A=[a_{ij}]$ as

$$a_{ij} = \begin{cases} 1 \text{ if } (vi,\ vj) \in E \\ 0 \quad \text{otherwise} \end{cases}$$

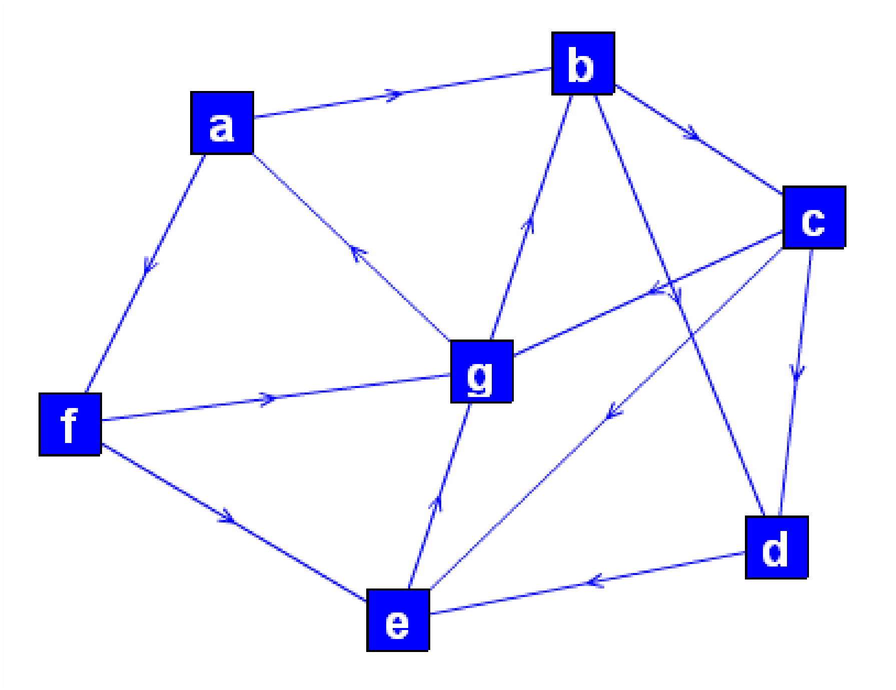
The definition implies that entry in the i^{th} row and j^{th} column of the matrix is 1 if a link exists between the i^{th} and j^{th} vertex; otherwise, it is 0.

- ➢ The size adjacency matrix is with vertex set V is $|V|x|V|$. Thus space complexity is $\theta(|V|^2)$

Graphs Representation

Adjacency Matrix

Example : Figure *(*i) shows a sample directed graph. Figure (*ii*) shows the adjacency matrix for the graph



(i) A sample graph

	a	b	c	d	e	f	g
a	0	1	0	0	0	1	0
b	0	0	1	1	0	0	0
c	0	0	0	1	1	0	1
d	0	0	0	0	1	0	0
e	0	0	0	0	0	0	1
f	0	0	0	0	1	0	1
g	1	1	0	0	0	0	0

(ii) Adjacency Matrix

Graphs Representation

Adjacency Linked List

A graph can also be represented using ***linked lists***. The list representation consists of an ***array of linked lists***, each corresponding to one vertex. The list stores all of the vertices that are linked to a given vertex.

Let G=(V,E) be a graph, and ***Adj(u)*** be the linked list corresponding to vertex *u,* then for all *v*,

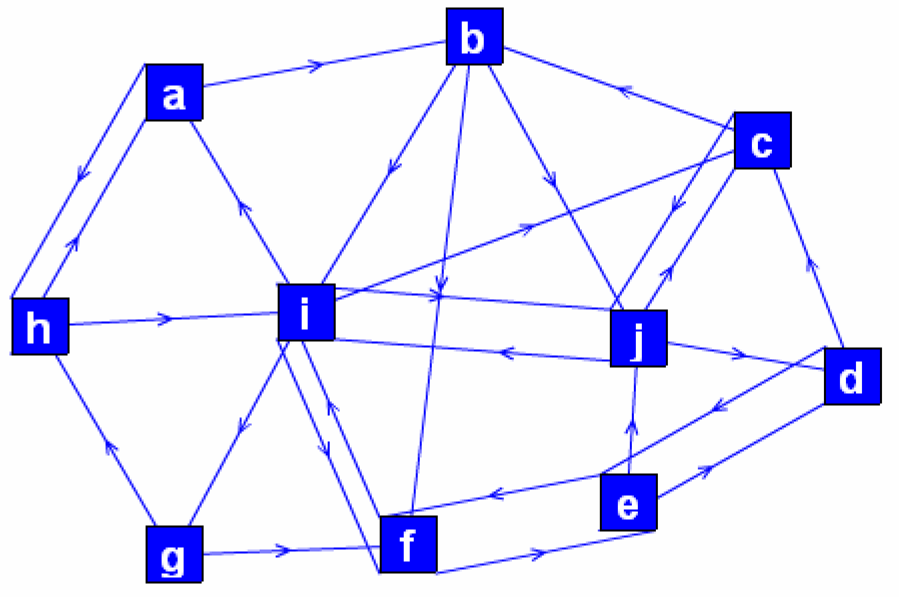
$$v \in Adj(u), \quad if \ (u, v) \in E.$$

- The vertices belonging to *Adj(u)* are called neighbors of vertex *u*, or ***adjacent vertices***.

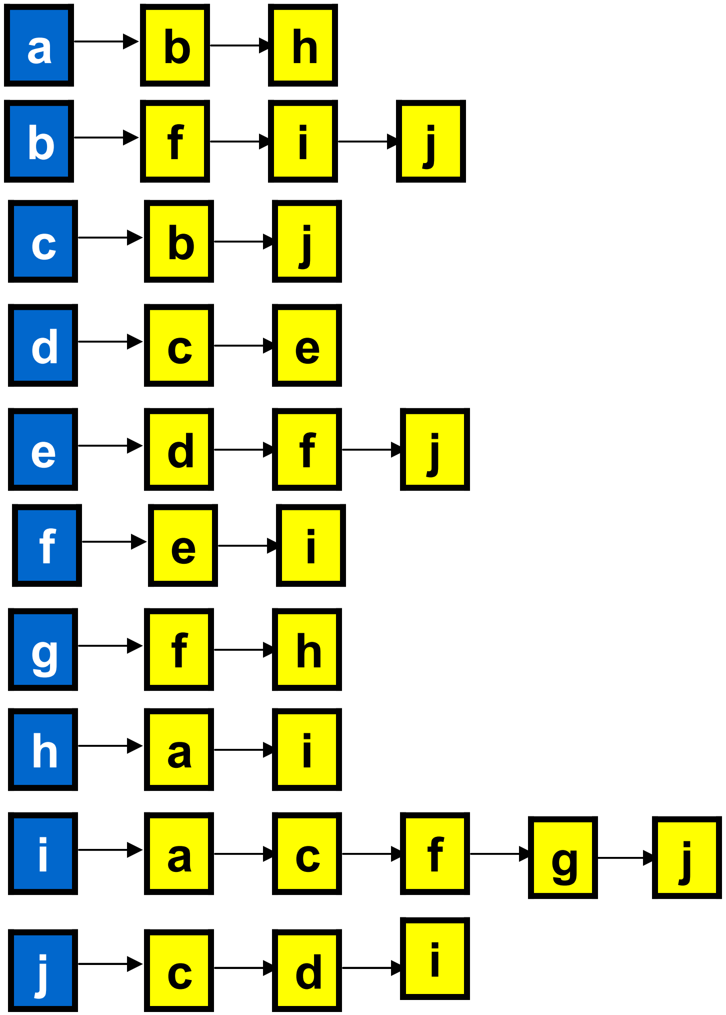
Graphs Representation

Linked List

Example: The diagrams below show a sample graph and its linked list representation



(i) Sample directed graph



a → b → h

b → f → i → j

c → b → j

d → c → e

e → d → f → j

f → e → i

g → f → h

h → a → i

i → a → c → f → g → j

j → c → d → i

(ii) Linked list Representation

Graph Traversal

Procedures

In graph traversal, ***all vertices are visited once***. Depending on the application, the data and keys are accessed and processed further. For example, the traversal procedure can be used to list keys stored in the vertices The two important procedures for graph traversal are:

(i) Depth First Search (DFS)

(ii) Breadth First Search (BFS)

- The DFS and BFS algorithms can be used to traverse both *directed* and *undirected* graphs

Depth First Search

Depth First Search

Strategy

The ***Depth First Search (DFS)*** systematically visits all vertices of a graph. Initially, a start vertex is chosen, and its neighbors are explored. Then neighbors and of neighbors are explored. This process is continued till the *remotest neighbor of the start* vertex is reached, which is called the *dead end*. The algorithm then backtracks to the start vertex and on the way back lists all of the vertices in *depth first search order*. The depth first search algorithm belongs to the class of *backtracking algorithms.*

➢ A ***stack*** is used to implement the DFS procedure. The stack keeps track of vertices traversed . For this purpose , the three states of vertices are designated as ***ready***, ***wait*** and ***processed,*** defined as follows :

i. All vertices are initially in the ***ready state***

ii. A vertex pushed on to the stack, is in the ***wait state***

iii. A vertex popped off the stack is in the ***processed state***

Depth First Search

Algorithm

The ***Depth First Search (DFS)*** for a connected graph consists of the following steps:

Step #1: *Initialize all vertices to mark as unvisited (ready state)*

Step #2: *Select an arbitrary vertex, push it to stack (wait state)*

Step #3: *Repeat steps # 4 through Step #5 until the stack is empty*

Step #4: *Pop off an element from the stack. Process and mark it as visited (processed)*

Step #5: *Push to stack adjacent vertices of the processed vertex. Go to Step # 3.*

The same procedure is applied to a subgraph, which may contain any unvisited vertices.

Depth First Search

Spanning Tree

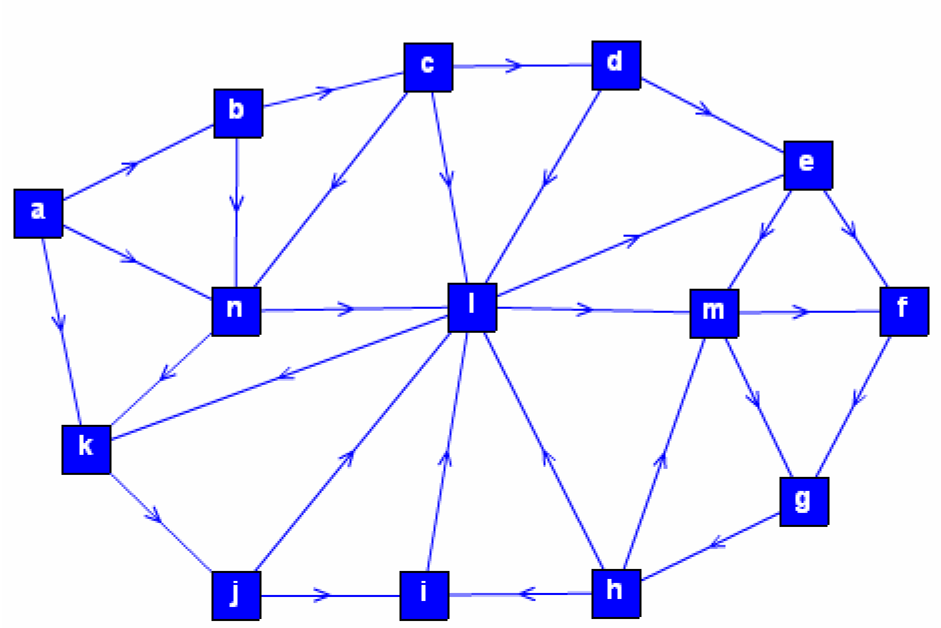
Apart from extracting information stored in vertices, the DFS can be used to construct special tree called ***DFS Spanning trees*** The Spanning trees are useful in investigating important properties of the graphs and subgraphs

During the DFS, a spanning tree is constructed by linking together a vertex that is visited for the ***first time to the vertex from which it is reached.*** The links are called ***tree edges***

Depth First Search

Example

Consider the sample directed graph shown below. The steps involved in Depth First Search are depicted in the next set of diagrams As the algorithm proceeds, the *depth first spanning tree* is also generated

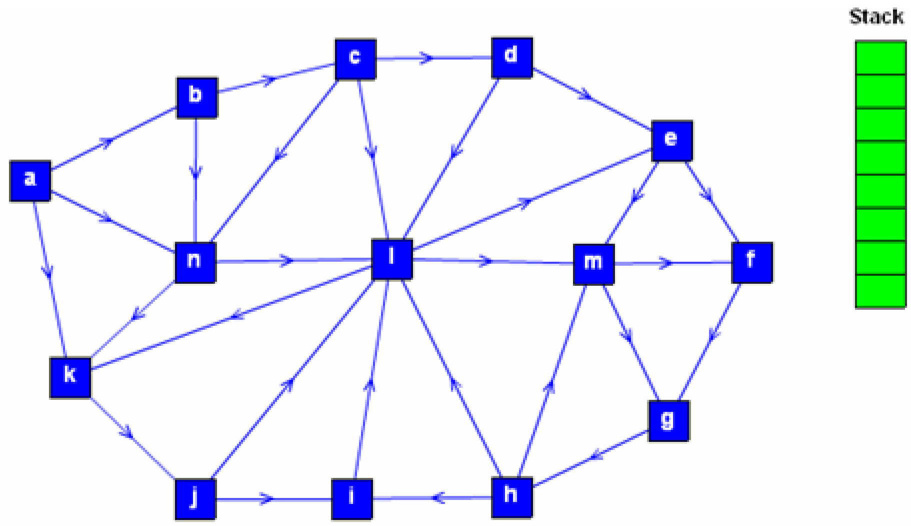


Sample directed graph

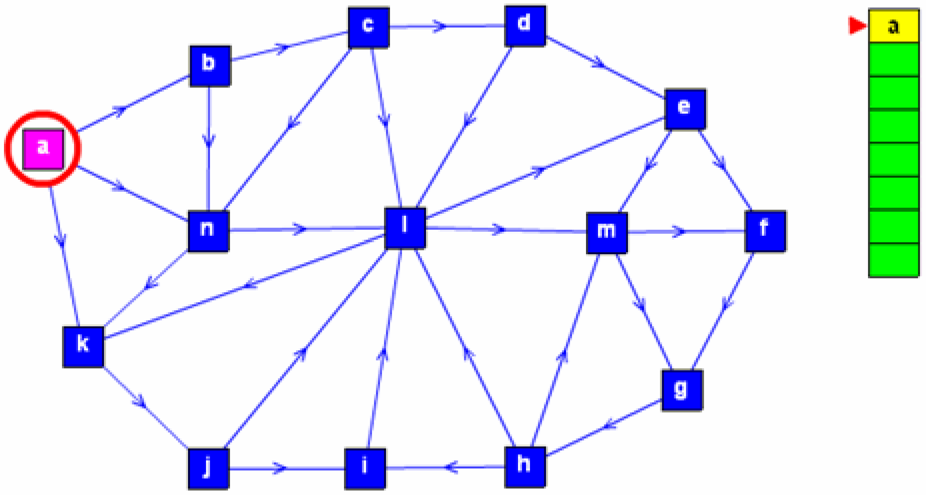
Depth First Search
Example

The following example illustrates the working of depth first search procedure

(1) For depth first search an auxiliary data structure *stack* is used as shown. It holds the vertices which are in *wait state*. Initially, the stack is empty, and all of the vertices are in *ready state*. These are shaded as blue



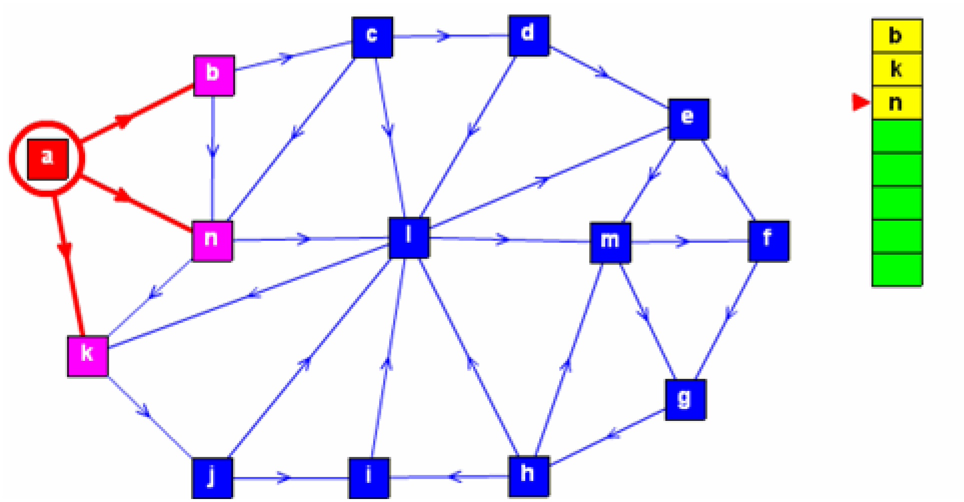
(2) The vertex *a* is chosen as the start vertex for depth first search. It is pushed on to the stack



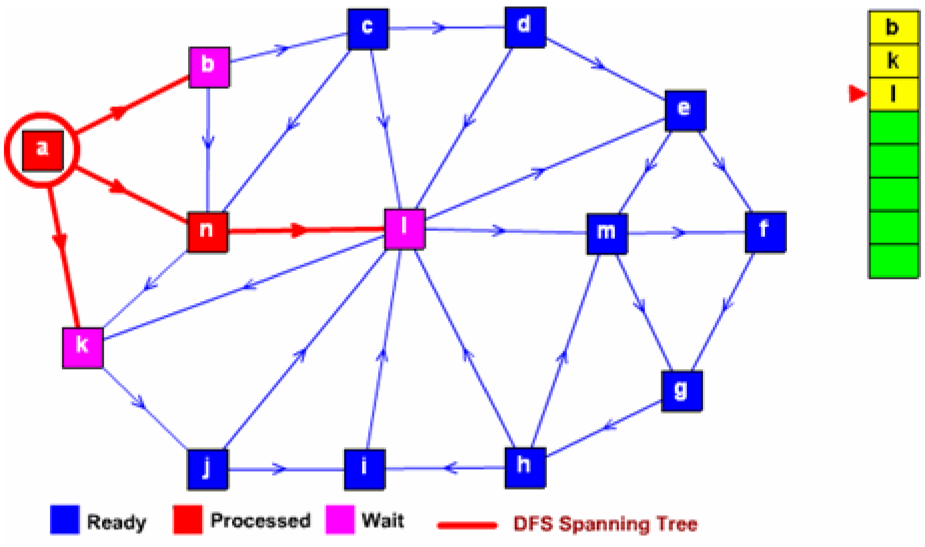
Depth First Search

Example

(3) The vertex *a* is popped. It is in the *processed state*. The neighbors *b, k, n* of the vertex *a* are pushed on to the stack.
The neighbors are now in the *wait state*
The vertex traversed so for is *a*

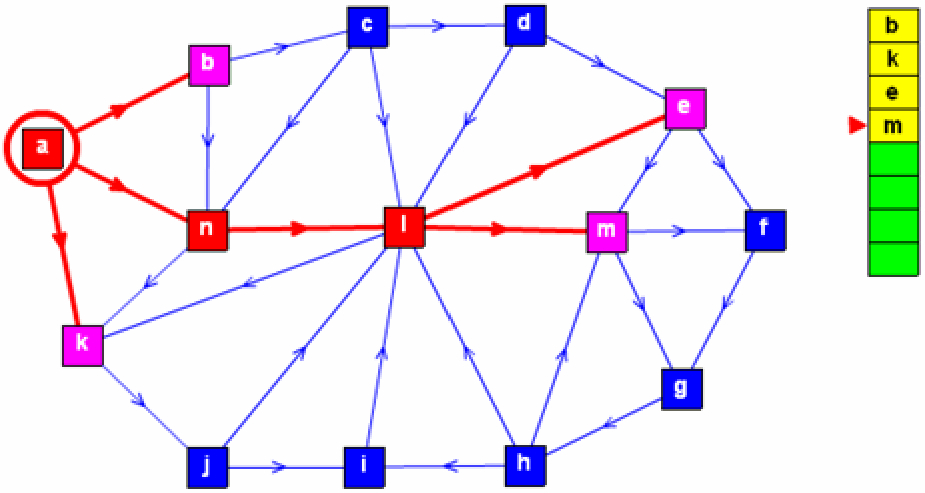


(4) Vertex *n* is popped. The neighbor *i* of *n* is pushed on to the stack.
The vertices traversed so for are: *a, n*

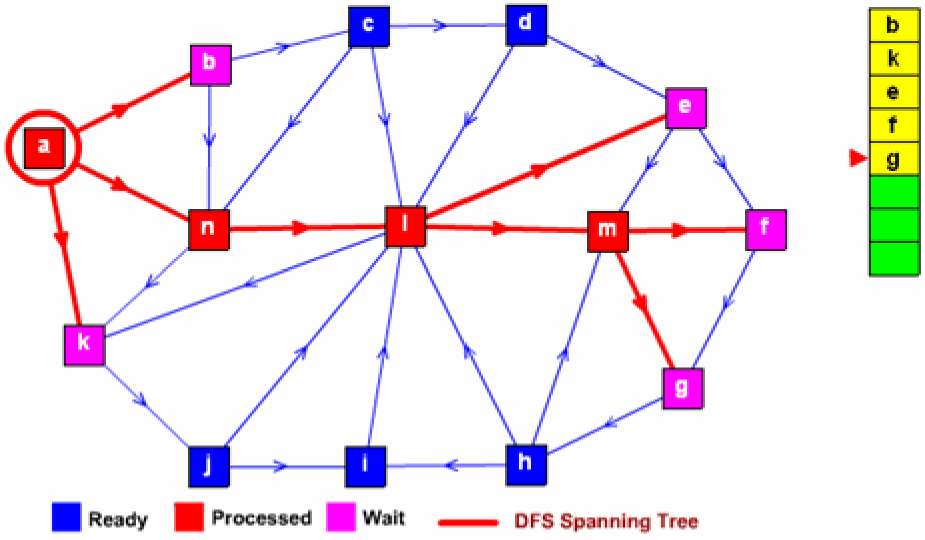


Depth First Search
Example

(5)The vertex *l* is popped . The neighbors *e, m* of *l* are pushed on to the stack
The vertices traversed so for are: *a,n,l*



(6)The vertex *m* is popped. The neighbors *f, g* of *m* are pushed on to the stack
The vertices traversed so for are: *a,n,l,m*

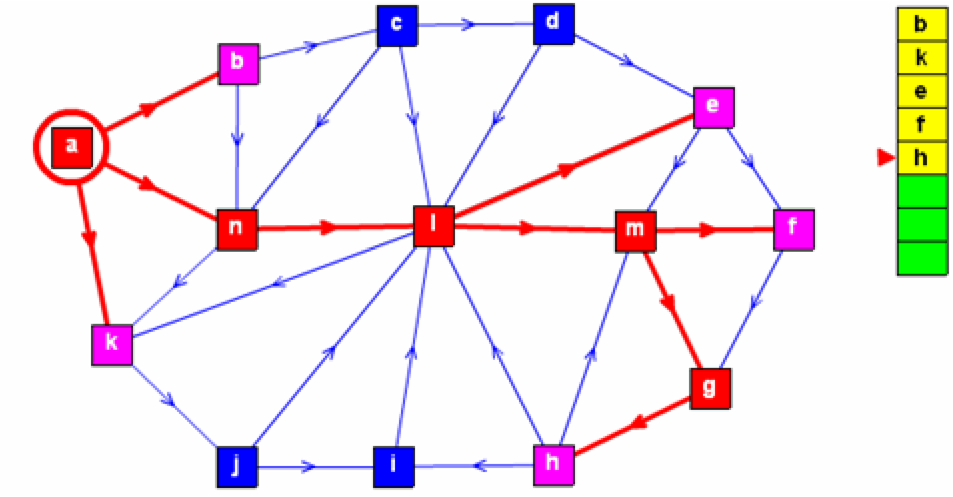


Depth First Search
Example

(7)The vertex *g* is popped and marked as processed. The neighbor *h* of *g* is pushed on to the stack

The vertices traversed so for are:

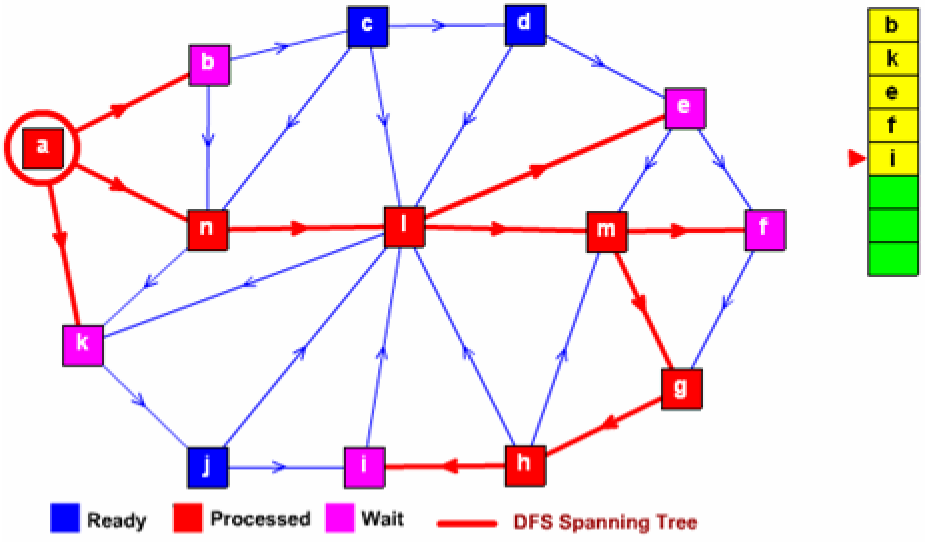
a,n,l,m,g



(8)The vertex *h* is popped and marked as processed. The neighbor *i* of *h* is pushed on to the stack

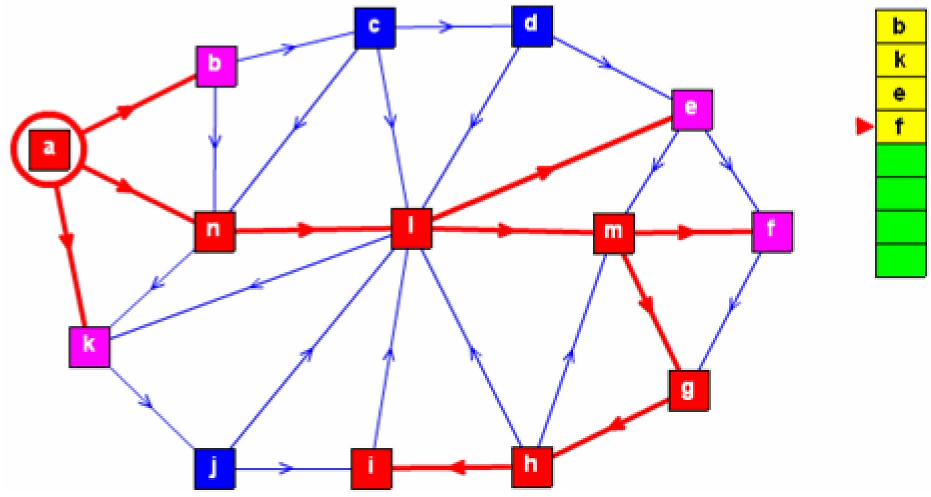
The vertices traversed so for are:

a,n,l,m,g,h

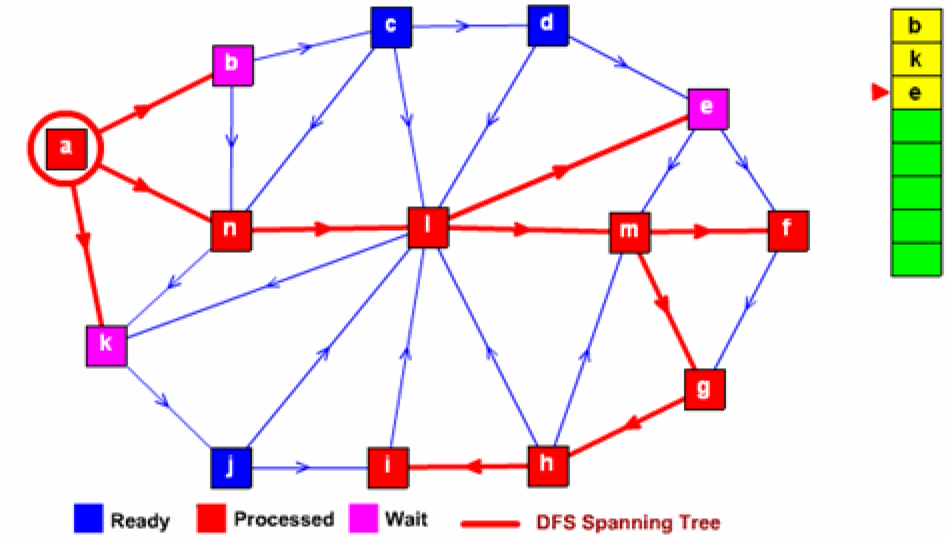


Depth First Search
Example

(9)The vertex *i* is popped and marked as processed. The popped vertex has no unvisited neighbors
The vertices processed so for are*:*
a,n,l,m,g,h,i



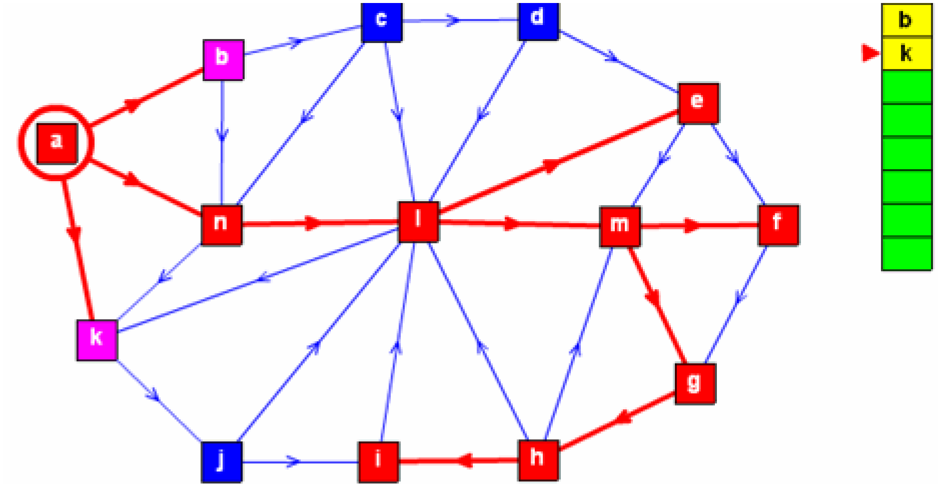
(10)The vertex *f* is popped and marked as processed. The popped vertex has no unvisited neighbors
The vertices traversed so for are:
a,n,l,m,g,h,i, f



Depth First Search
Example

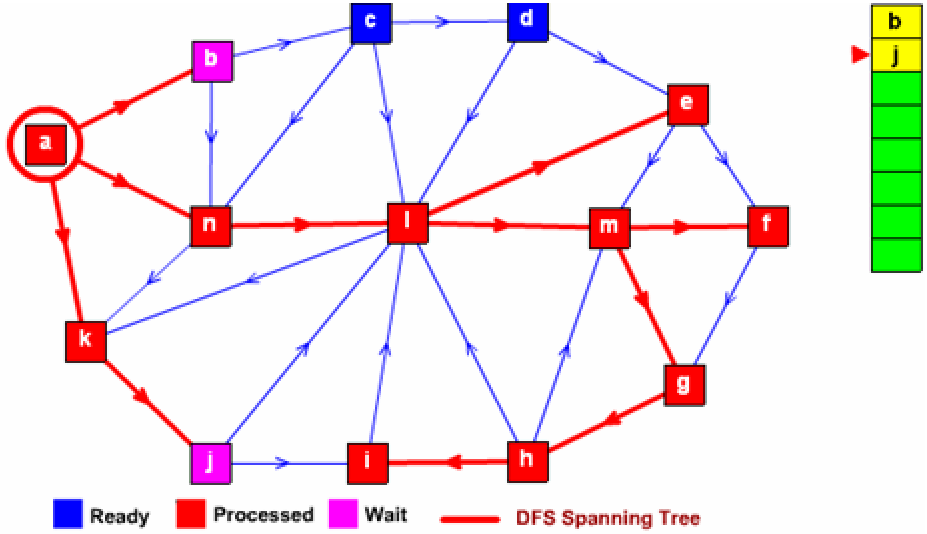
(11) The vertex *e* is popped and marked as processed. The vertex has no unvisited neighbors

The vertices traversed so for are: *a,n,l,m,g,h,i,f,e*



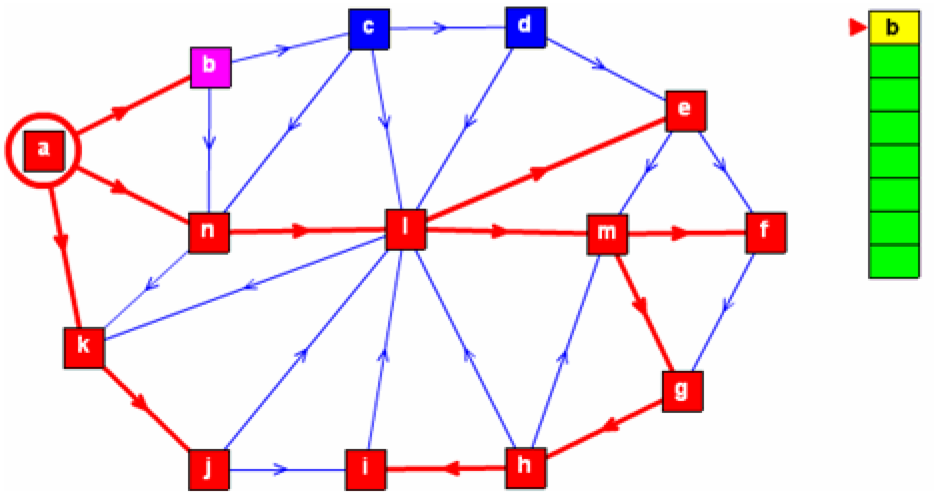
(12) The vertex *k* is popped and marked as processed. The vertex *j* which is neighbor of vertex *k* is pushed on to the stack

The vertices traversed so for are: *a,n,l,m,g,h,i,f,e,k*

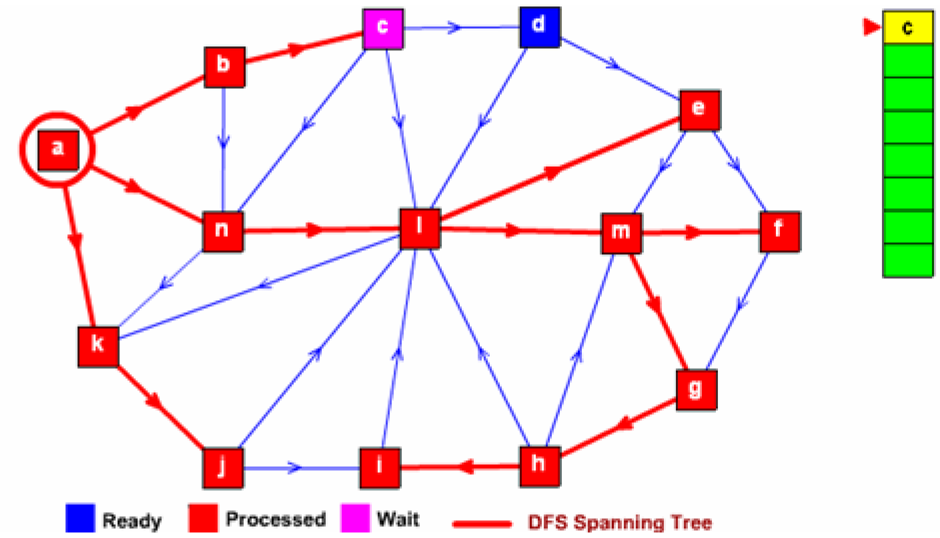


Depth First Search
Example

(13) The vertex *j* is popped and marked as processed. The vertex has no unvisited neighbors
The vertices traversed so for are:
a,n,l,m,g,h,i,f,e,k,j

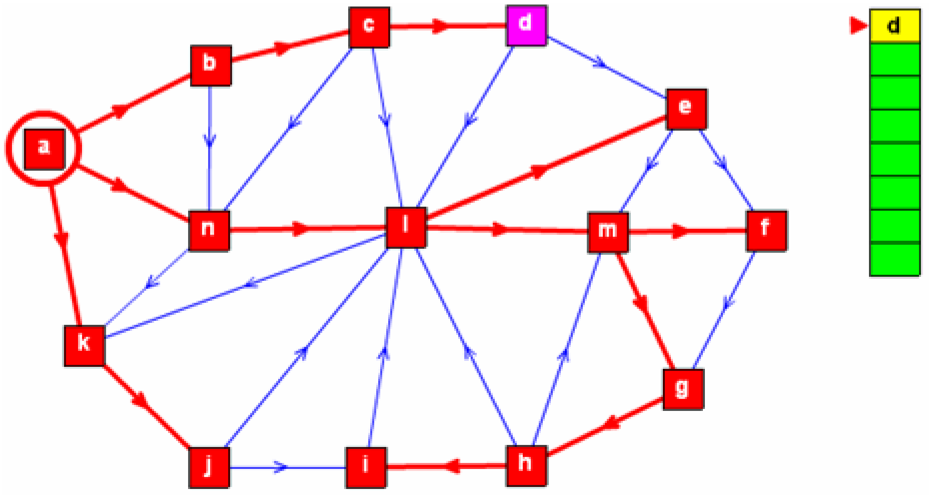


(14) The vertex *b* is popped and marked as processed. The vertex *c* which is neighbor of *b* is pushed on to the stack
The vertices traversed so for are :
a,n,l,m,g,h,i,f,e,k,j,b

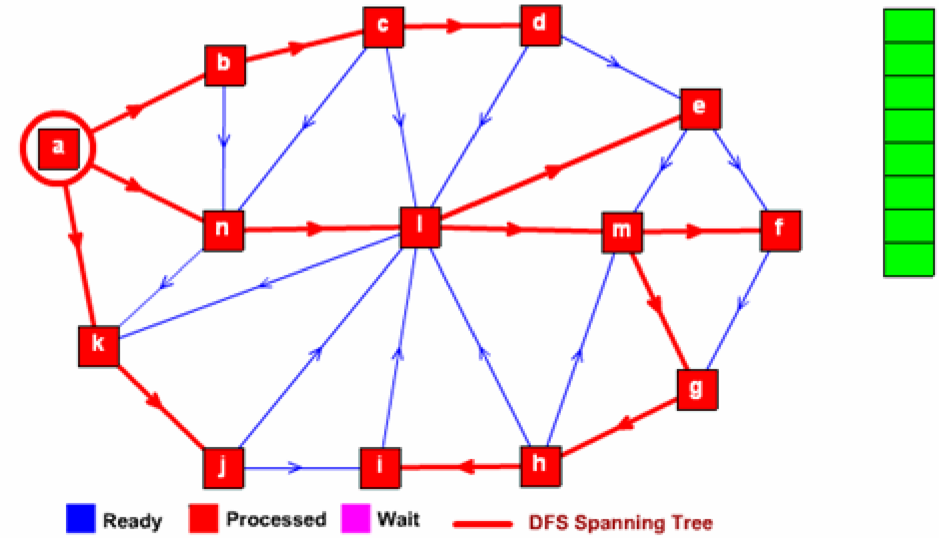


Depth First Search
Example

(15) The vertex *c* is popped and marked as processed. The vertex *d* which is neighbor of *c* is pushed on to the stack The vertices traversed so for are :
a,n,l,m,g,h,i,f,e,k,j,b,c



(16) The vertex *d* is popped and marked as processed. The vertex *d* has no unvisited neighbor. Further, the stack is *empty*. The depth first search algorithm terminates
The DFS order is finally:
a,n,l,m,g,h,i,f,e,k,j,b,c,d



Depth First Search

Applications

Some important applications of DFS are

(i) Checking connectivity of a graph

(ii) Checking accessibility of vertices from a given vertex

(iii) Checking cyclicity / acyclicity of a graph

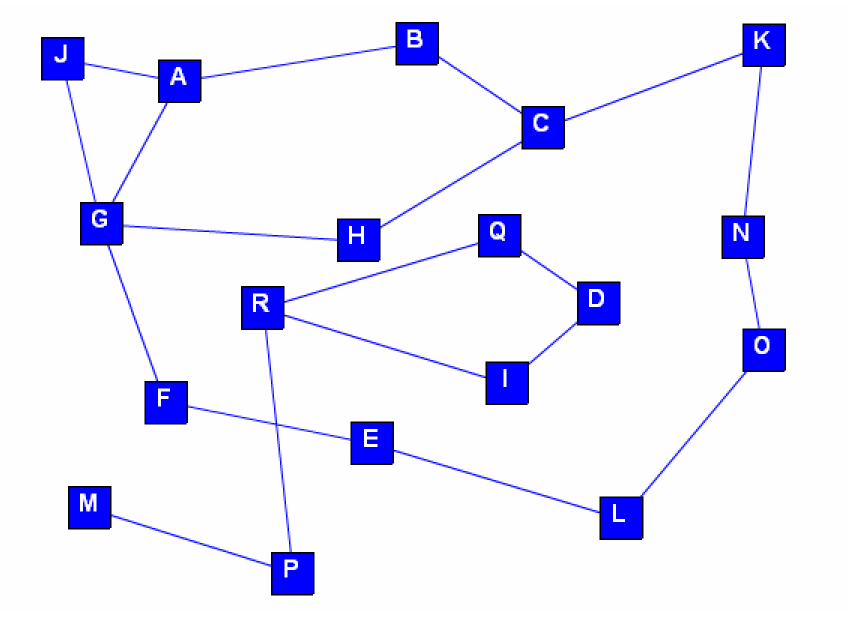
Graph connectivity and cyclicity are illustrated by the following examples

DFS Applications

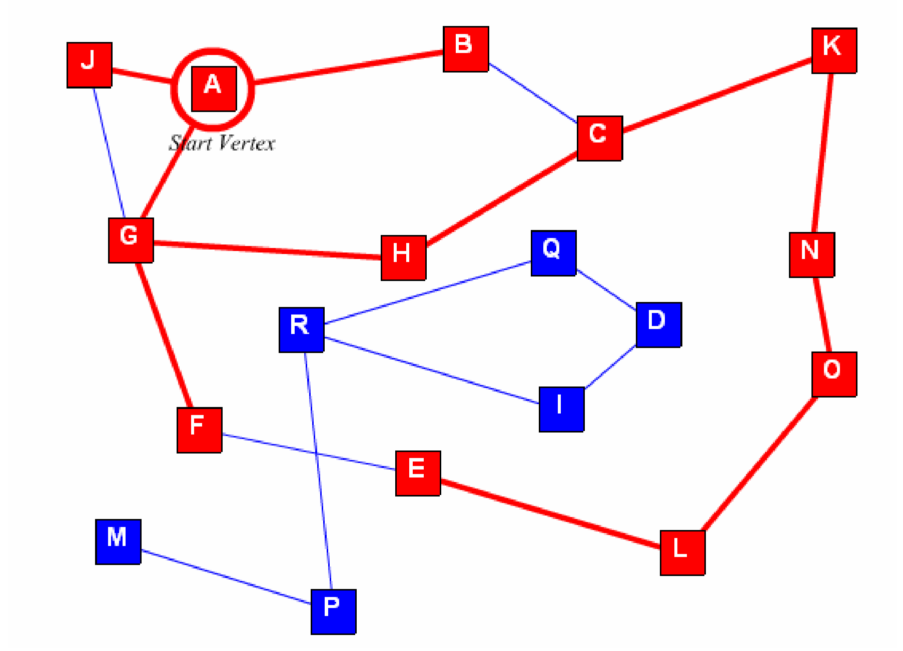
Graph Connectivity

An ***undirected graph is not connected if the DFS algorithm terminates before all of the vertices have been visited.*** The unvisited vertices form one or more subgrphs

Example: Figure (*i*) shows a sample graph. Figure (*ii*) provides a DFS tree. The vertices *M.P,R,Q,D* remain unvisited and constitute a *disconnected subgraph*



(i) Sample undirected graph



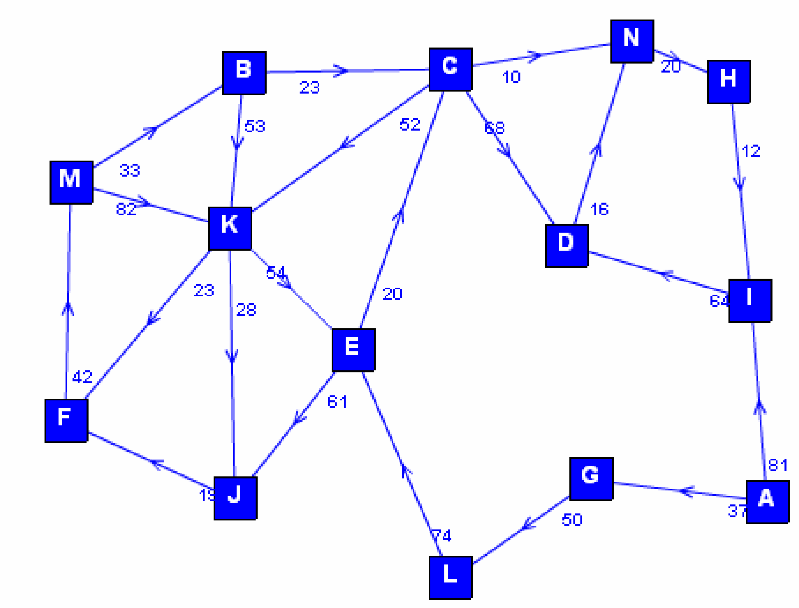
(ii) Unconnected subgraph identified by DFS algorithm

DFS Applications

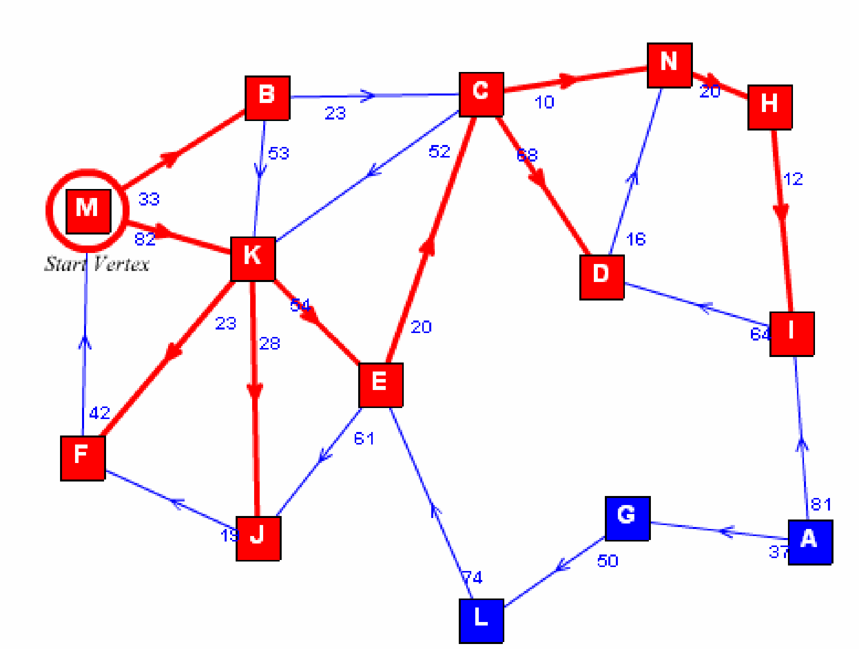
Subgraph Accessibility

In a ***directed graph***, all the vertices may have links, but may not be accessible from given vertex. The inaccessible vertices can be explored by using DFS

Example: Figure (*i*) shows a sample directed graph. Figure (*ii*) provides a DFS tree. The vertices *L,G,A* remain unvisited and constitute subgraph whose vertices are not accessible from initial vertex *M*



(i) Sample directed graph



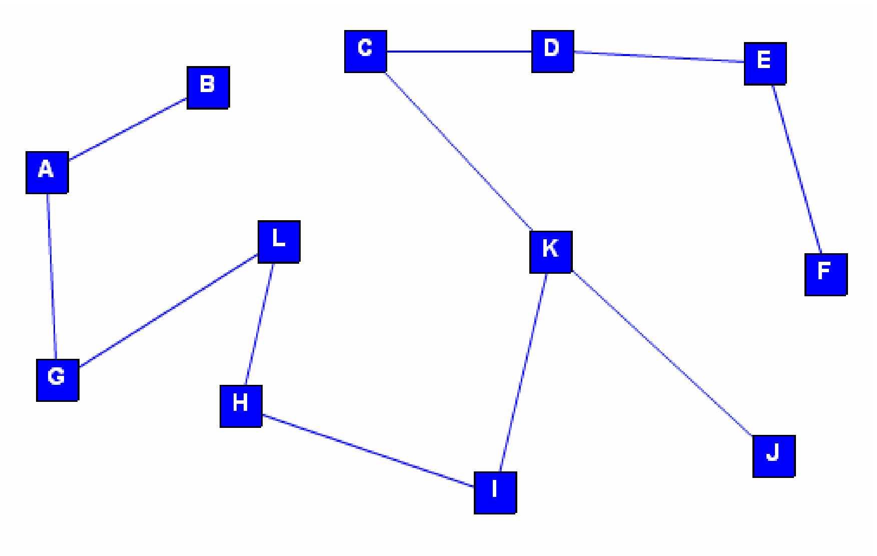
(ii) Inaccessible subgraph identified by DFS

DFS Applications

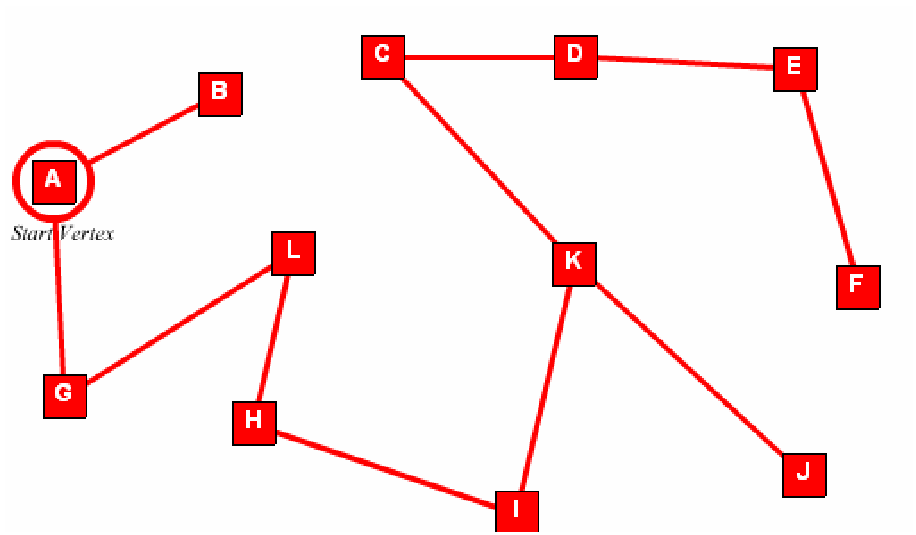
Acyclic Graph

An ***undirected graph is acyclic (has no cycles) if the DFS spanning has no back edges***

Example: Figure (*i*) shows a sample graph . Figure (*ii*) contains a DFS Spanning Tree, which has *no back edges*. The graph is therefore acyclic.



(i) Sample undirected graph.



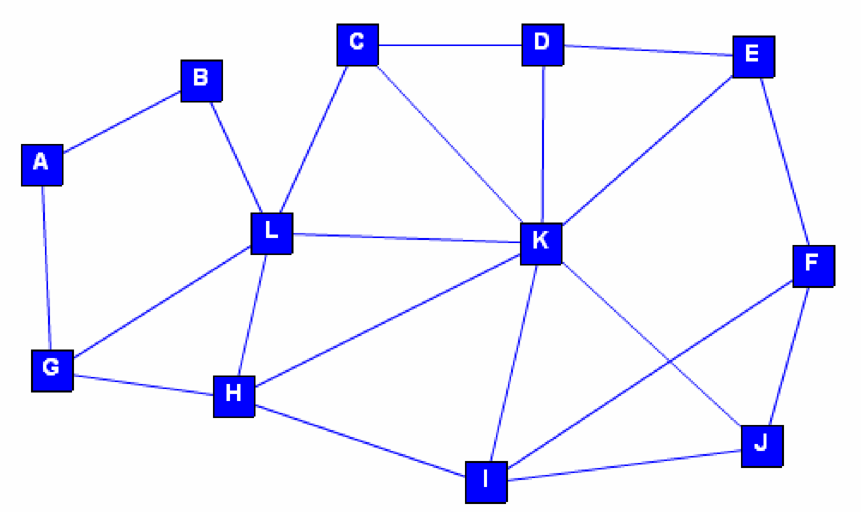
(ii) Undirected graph with DFS Tree.
There are no back edges.

DFS Applications

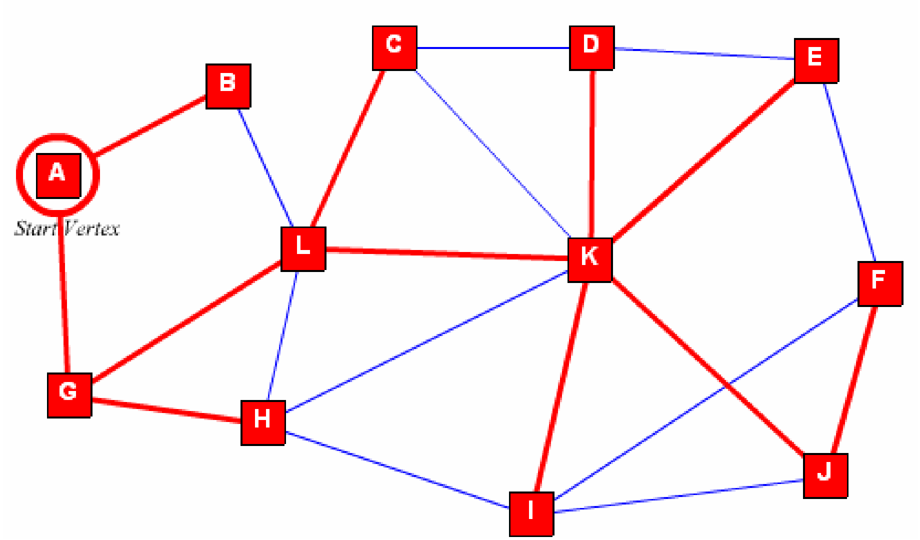
Cyclic Graph

An ***undirected graph is cyclic if the DFS spanning has back edges***

Example: Figure (i) shows a sample graph . Figure (*ii*) contains the DFS spanning tree, which ha *back edges shown in blue color*. The graph is therefore cyclic.



(i) Sample undirected graph.



(ii) Undirected graph with DFS Tree.
with back edges, shown in blue color

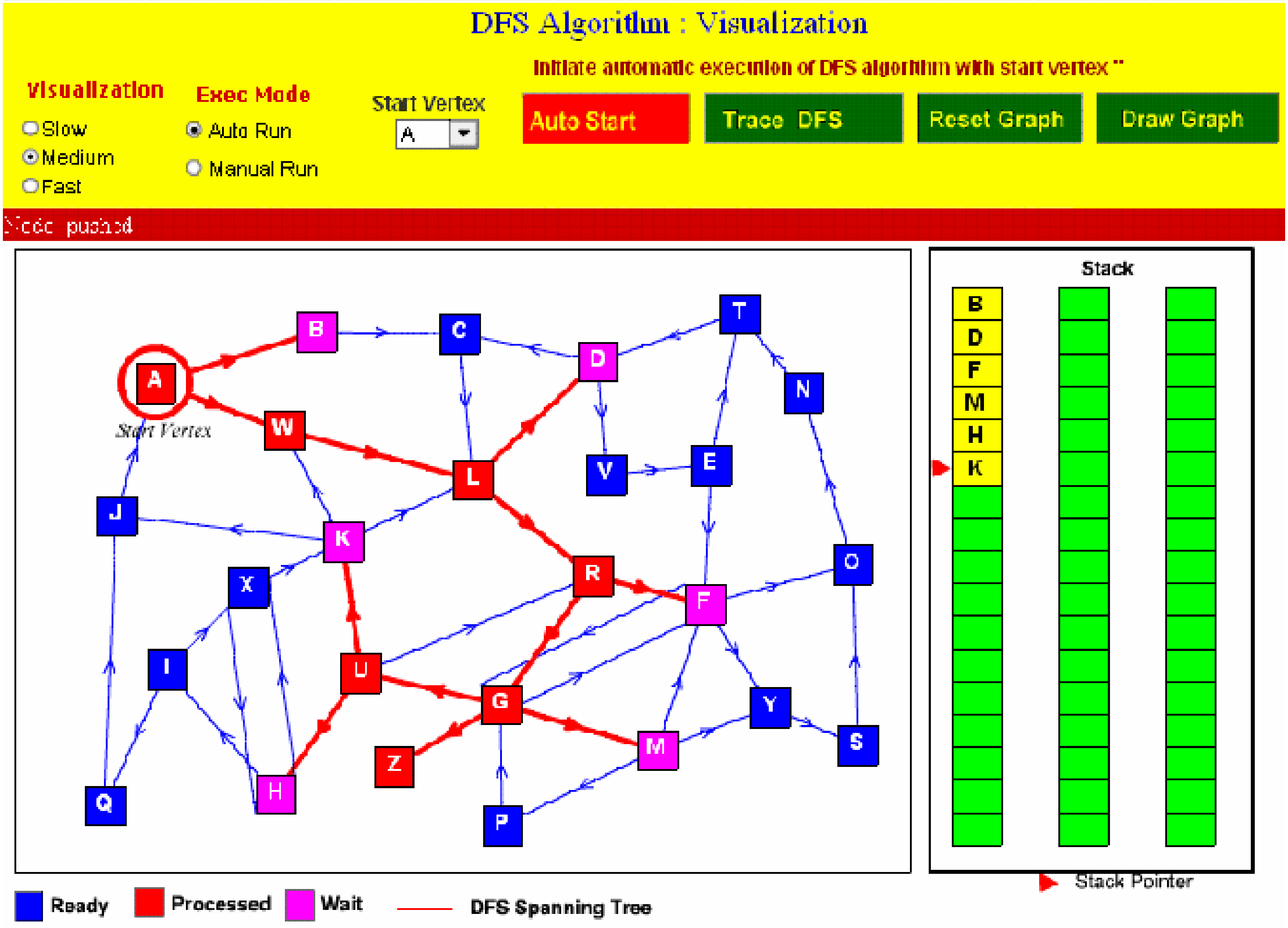
Depth First Search

Implementation

The DFS implements depth-first-search procedure by using a ***stack S***. The address **s** of first selected vertex is passed to the method. The implementation uses an array ***state[]*** which stores status of vertices as READY**,** WAIT, PROCESSED. The graph is represented by ***adjacency list Adj*** . The notation ***Adj[u]*** denotes a neighbor of vertex *u*

```
DFS(G, s)                                 ▶ s is address of starting vertex
for each vertex $u \in V[G] - \{ s \}$    ▶ V[G] is vertex set of graph G. V[G]-{s} is the
                                            difference set that excludes initial vertex s.
    do state[u] ← READY                   ▶ All unvisited vertices are assigned READY status
state[s] ← WAIT                           ▶ Vertex s is pushed to stack and assigned WAIT status
$P \leftarrow \varphi$                    ▶ Stack P is initialized as empty
PUSH(S,s)                                 ▶ The starting vertex s is added to the stack
while $S \neq \varphi$                    ▶ A loop is created to pop off vertices .
    do u←POP(S)                           ▶ u is the popped vertex
        for each $v \in Adj[u]$           ▶ All vertices v that are adjacent to vertex u are explored
            do if state[v] = READY        ▶ If the neighbor vertex v is not visited,
                then state[v]← WAIT         then it is assigned WAIT state
                     PUSH(S,v)              and pushed on to the stack S
    state[u] ← PROCESSED                  ▶ The vertex u popped off the stack is marked visited
                                            and assigned PROCESSED state
```

DFS Visualization



DFS Algorithm : Visualization

Initiate automatic execution of DFS algorithm with start vertex ''

Visualization
○ Slow
● Medium
○ Fast

Exec Mode
● Auto Run
○ Manual Run

Start Vertex: A

Node pushed

Stack

B
D
F
M
H
K

🟦 Ready 🟥 Processed 🟪 Wait —— DFS Spanning Tree

▶ Stack Pointer

Analysis of DFS

Running Time

The running time for DFS consists of following major components

T_i = *Initializing vertices T*

T_s = *Scanning Adjacency list*

- The Graph *G(V,E)* has $|V|$ vertices and $|E|$ edges. Each vertex is initialized once. Therefore,

$$T_i = |V|$$

- The scanning of Adjacency list takes place for each vertex. The total time is proportional to the number of edges scanned. Since $|E|$ is the number of edges in *G,* in ***worst case*** the algorithm explores all edges **,** therefore,

$$T_s = |E|$$

- Thus, the worst case running time is given by time

$$T_{DFS} = |V| + |E| = \boldsymbol{O(|V|+|E|)}$$

Breadth First Search

Breadth First Search

Strategy

The ***Breadth First Search (BFS)*** is one of the standard graph traversal methods. Each node is visited once only. The method proceeds by visiting all ***immediate neighbors*** (vertices one edge away), then visiting ***neighbor's neighbors*** (vertices two edges away), and so forth, until all reachable vertices have been visited.

If a graph has any subgraph (which is not reachable from the initial vertex), the procedure may be re-started by choosing a vertex from the unvisited subgraph.

A ***queue*** is used to implement the BFS procedure. It keeps track of vertices to be visited next. A vertex pushed to the queue is said to be in ***ready*** state. An ***enqueued*** vertex is said be in ***wait*** state. The ***dequeued*** vertex is referred to as ***processed.***

The queue is initialized with the address of a starting vertex.

Breadth First Search

Algorithm

The ***Breadth First Search (BFS)*** for a connected graph consists of following steps:

Step #1: *Initialize all vertices to mark as unvisited (ready state)*

Step #2: *Select an arbitrary vertex, add it to queue (wait state)*

Step #3: *Repeat steps # 4 through Step #5 until queue is empty*

Step #4: *Remove an element from the queue. Process and mark it as visited (processed state)*

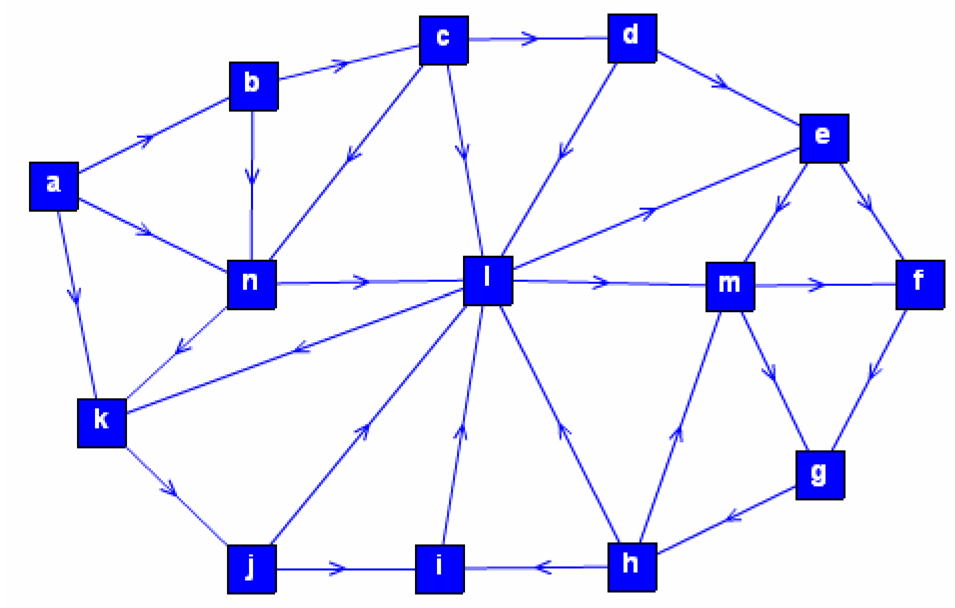
Step #5: *Enqueue all adjacent vertices of the processed vertex. Go to Step # 3.*

The same algorithm may be applied to a subgraph, which contains any unvisited vertices.

Breadth First Search

Example

Consider the sample directed graph shown below. The steps involved in performing *breadth first search* on the graph are depicted in the next set of diagrams. The execution of algorithm also generates BFS Spanning Tree, which is shown in bold red lines.



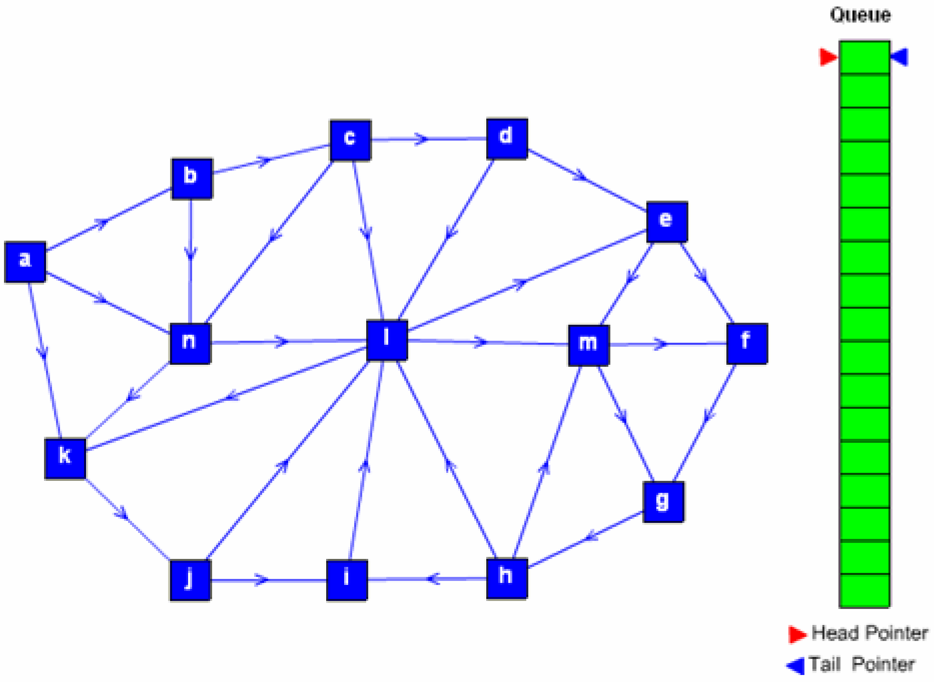
Sample directed graph

Breadth First Search

Example

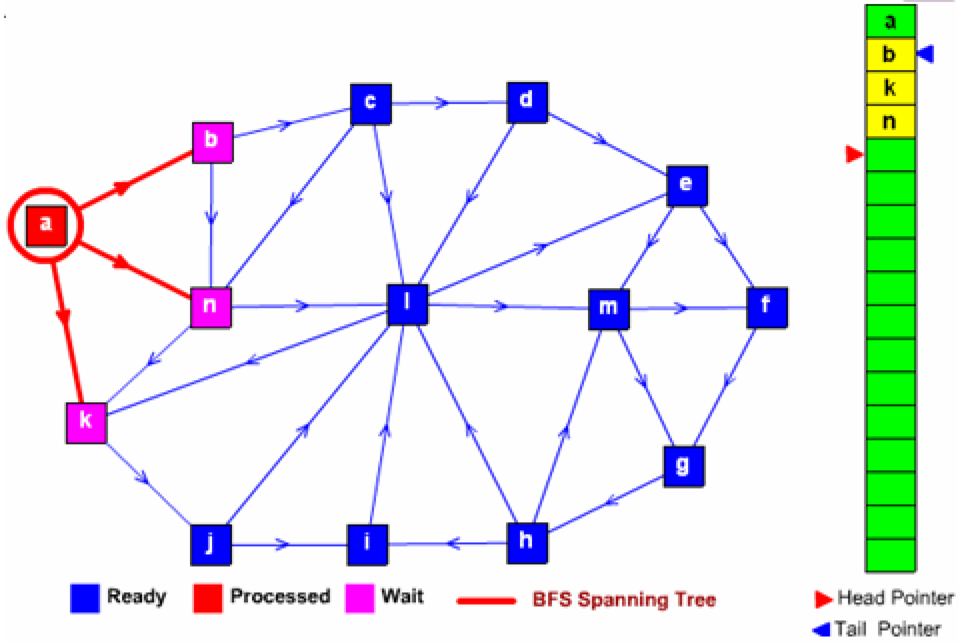
The following example illustrates the working of breadth first search procedure

(1)Initially, all the vertices are in *ready state*. For the execution of BFS algorithm, an auxiliary data structure *queue* is used, which holds the vertices in *wait state*



(2) The vertex *a* is chosen as the start vertex . The neighbors *b, n, k* of *a* are pushed to the queue. The current status of all of the vertices is given by the legend placed at bottom.

The vertex traversed so for is *a*



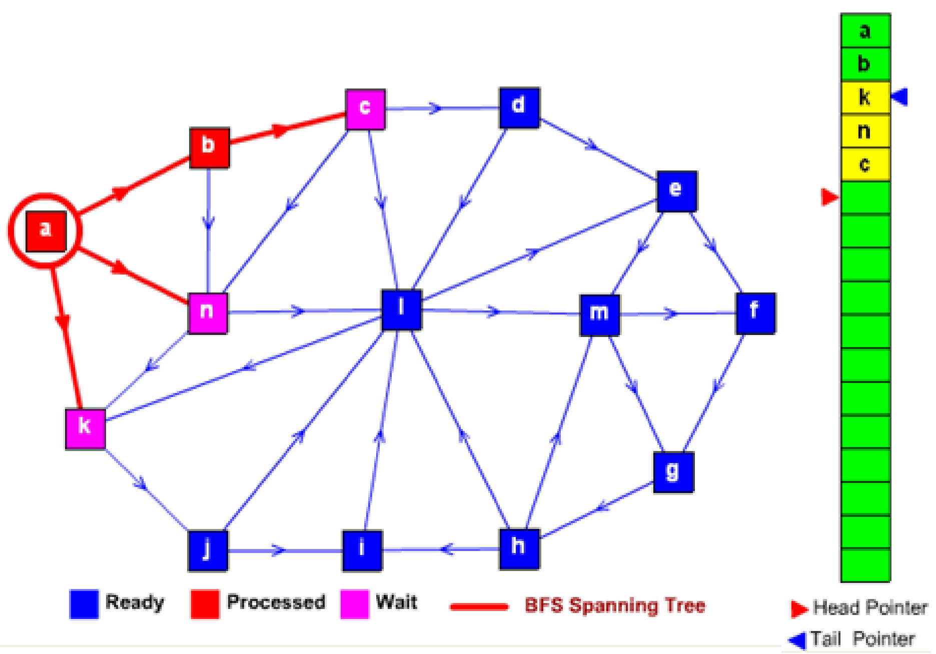
Breadth First Search

Example

(3) The vertex *b* is removed from the queue. The vertex *c,* neighbor of *b,* is pushed to the queue

The vertices traversed so for are:

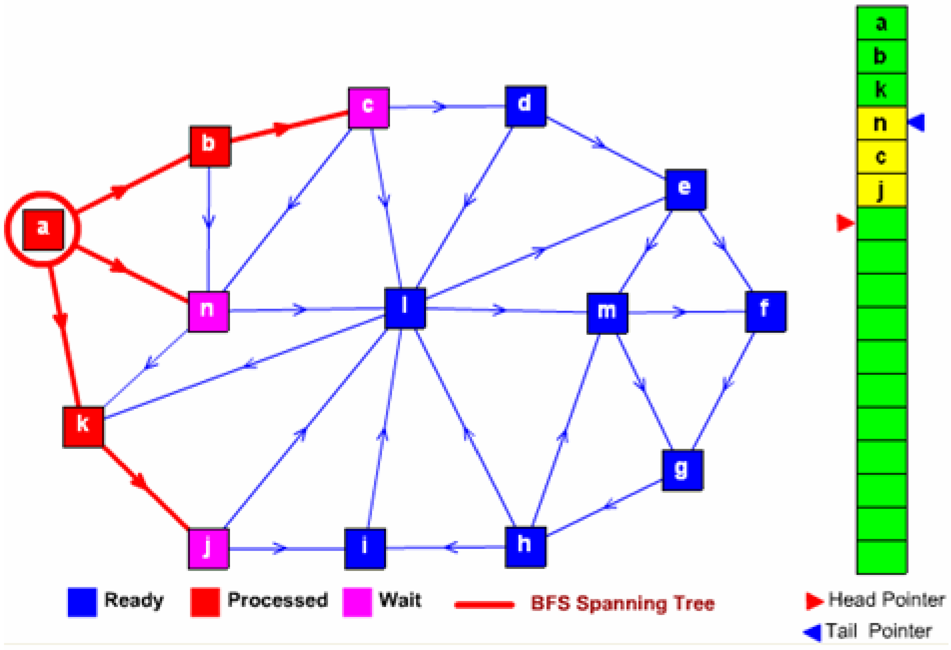
a,b



(4) The vertex *k* is removed from the queue. The neighbor *j* of vertex *k* is pushed to the queue

The vertices traversed so for are:

a,b,k



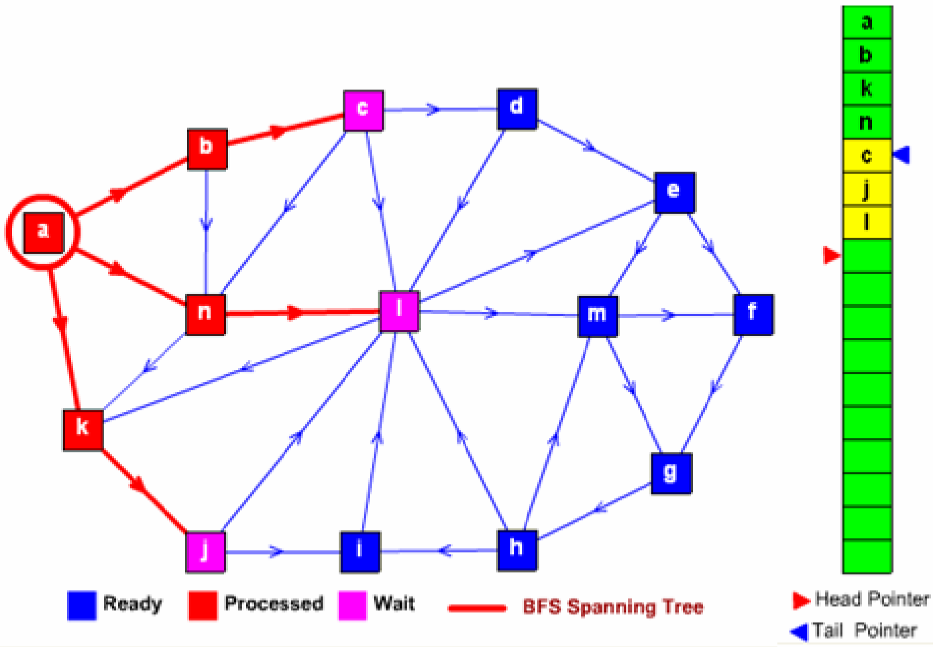
Breadth First Search

Example

(5) The vertex *n* is removed from the queue. The neighbor *l* of vertex *n* is pushed to the queue

The vertices traversed so for are:

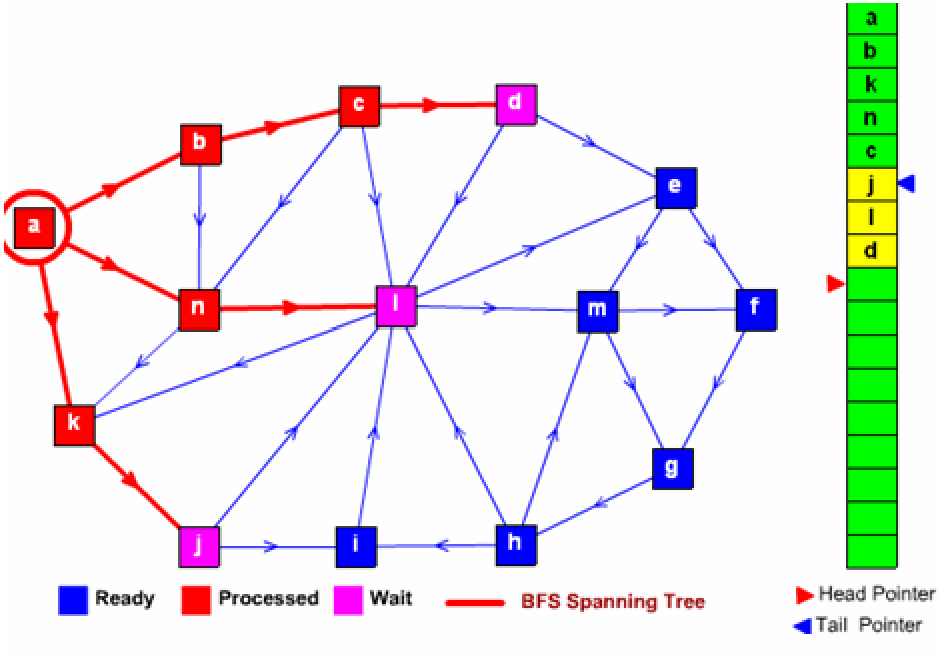
a,b,k,n



(6) The vertex *c* is removed from the queue. The neighbor *d* of vertex *c* is pushed to the queue

The vertices traversed so for are:

a,b,k,n,c



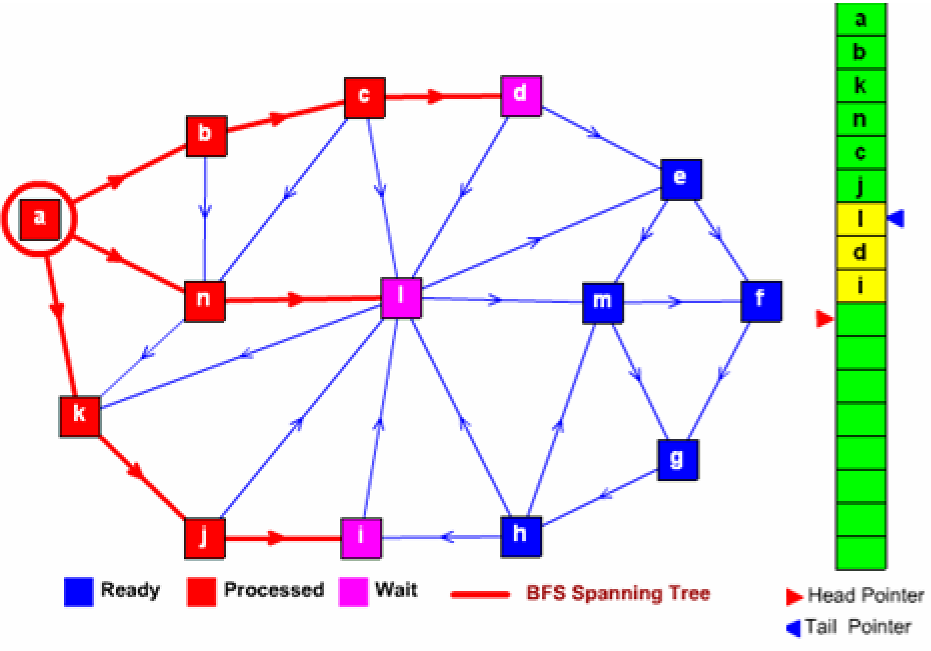
Breadth First Search

Example

(7) The vertex *j* is removed from the queue. The neighbor *i* of vertex *j* is pushed to the queue

The vertices traversed so for are:

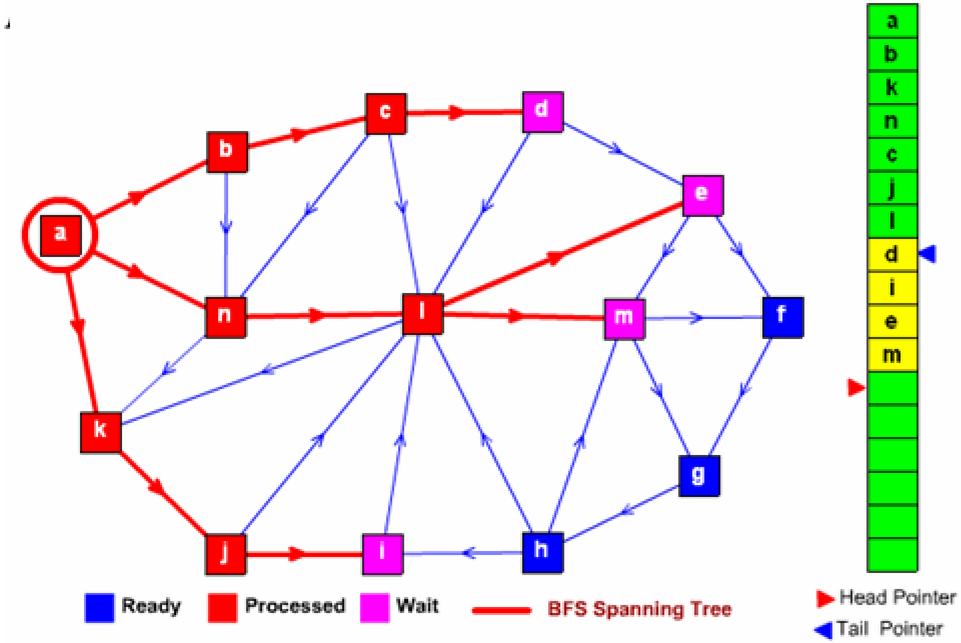
a,b,k,n,c,j



(8)The vertex *l* is removed from the queue. The neighbors *e, m* of vertex *l* are pushed to the queue

The vertices traversed so for are:

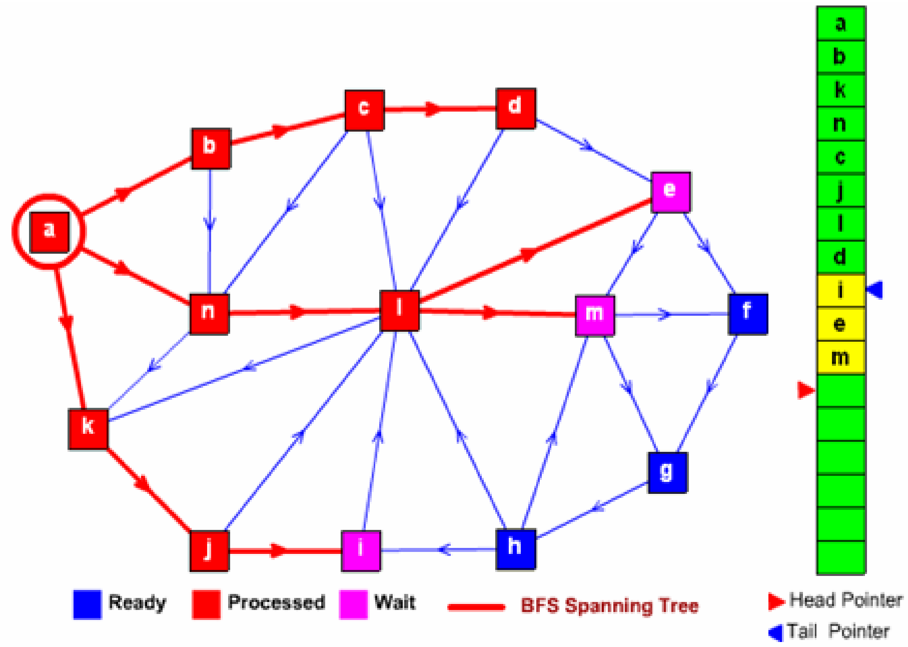
a,b,k,n,c,j,l



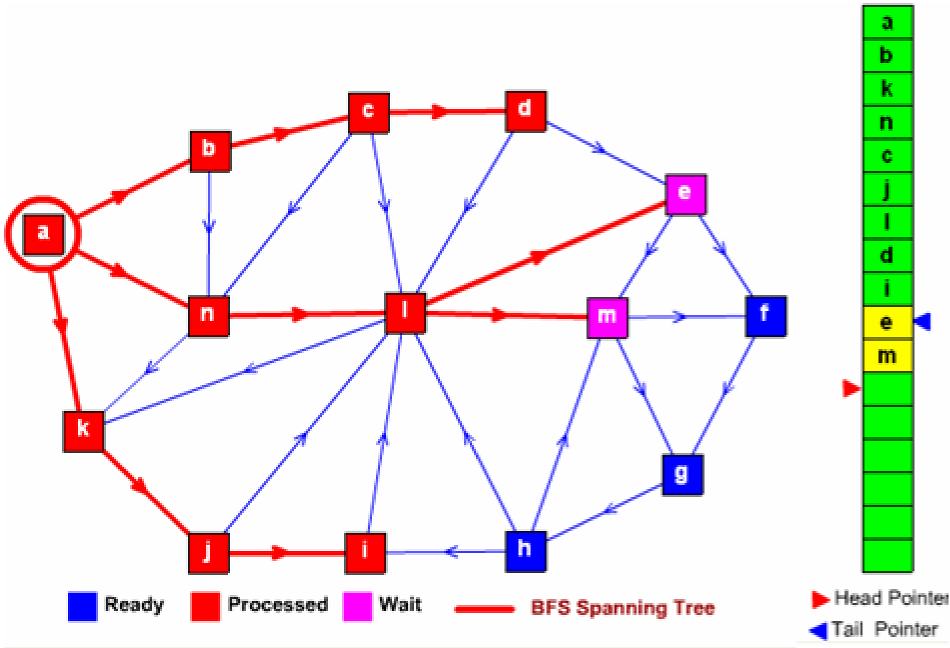
Breadth First Search

Example

(9)The vertex *d* is removed from the queue. The vertex *d* has no unvisited neighbor
The vertices traversed so for are:
a,b,k,n,c,j,l,d



(10)The vertex *i* is removed from the queue. The vertex *i* has no unvisited neighbor
The vertices traversed so for are:
a,b,k,n,c,j,l,d,i



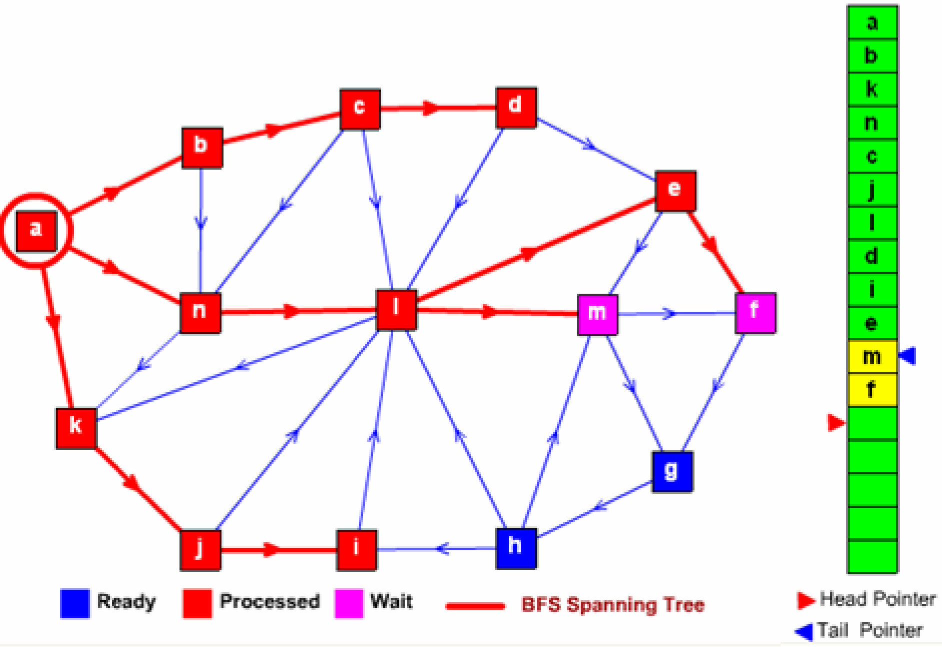
Breadth First Search

Example

(11)The vertex *e* is removed from the queue. The neighbor *f* of vertex *e* is pushed to the queue

The vertices traversed so for are:

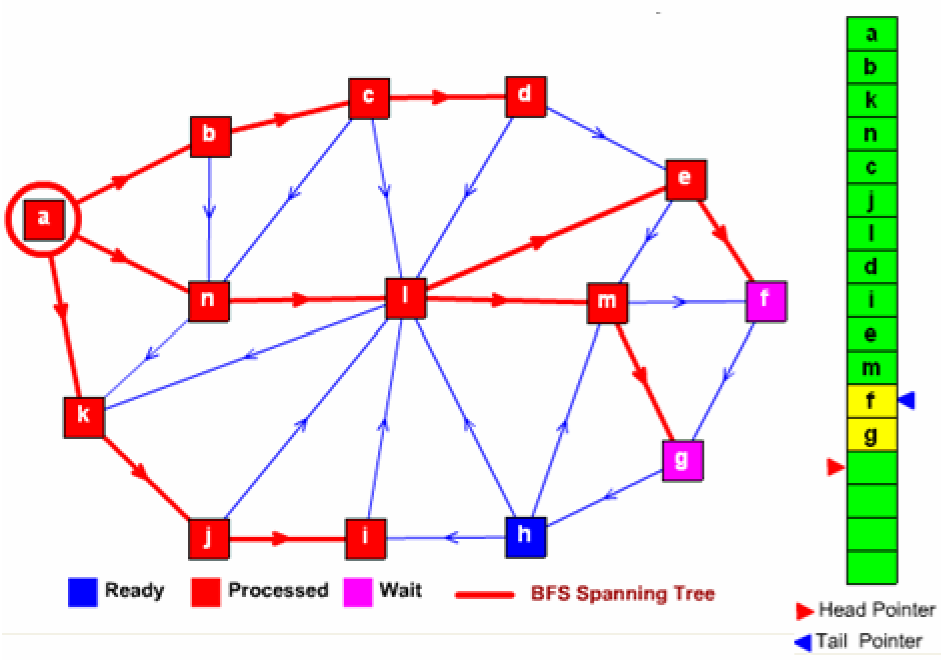
a,b,k,n,c,j,l,d,I,e



(12) The vertex *m* is removed from the queue. The neighbor *g* of vertex *m* is pushed to the queue

The vertices traversed so for are:

a,b,k,n,c,j,l,d,I,e,



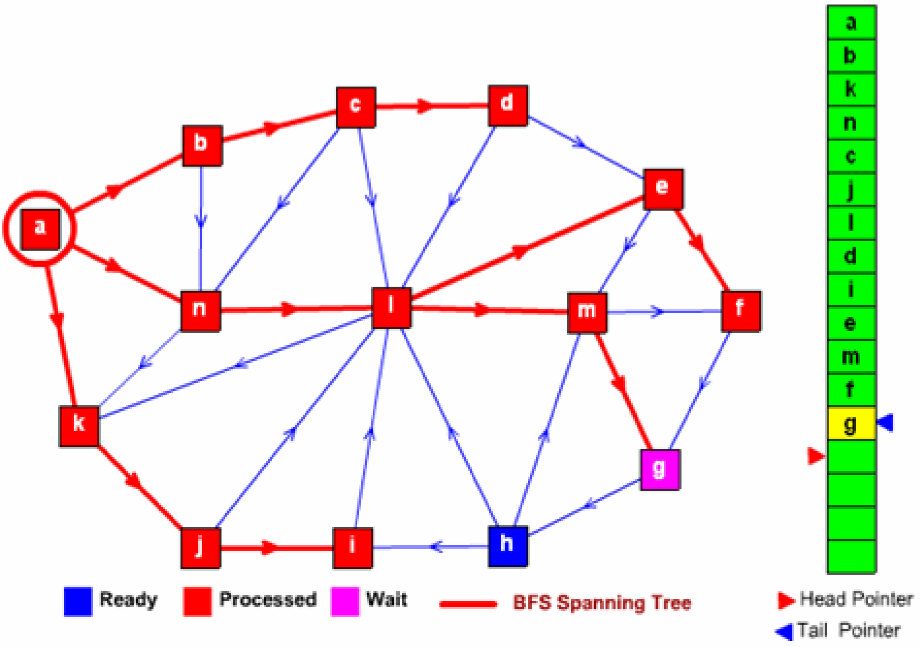
Breadth First Search

Example

(13) The vertex *f* is removed from the queue. The vertex *f* has no unvisited neighbor

The vertices traversed so for are:

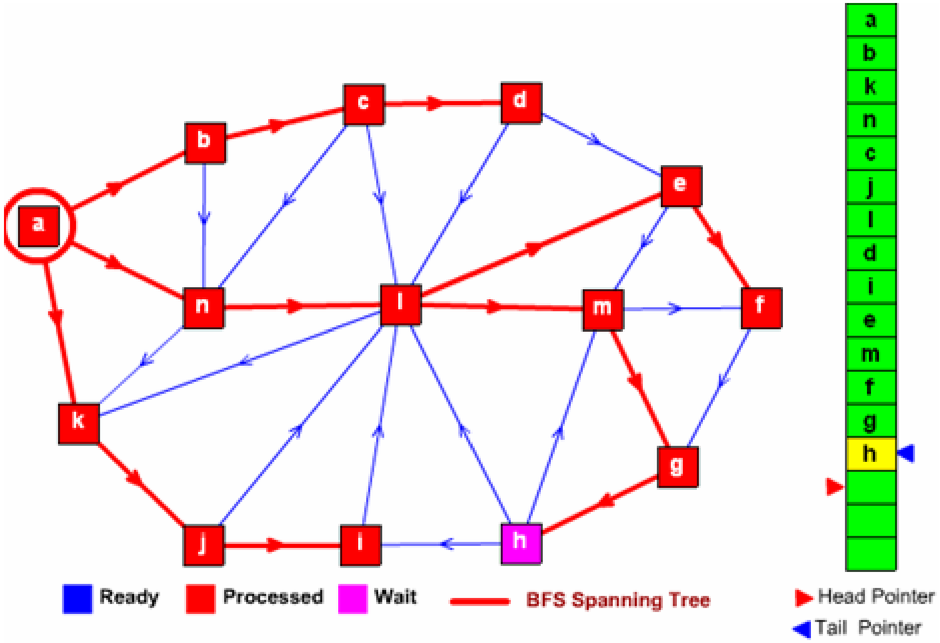
a,b,k,n,c,j,l,d,I,e,f



(14)The vertex *g* is removed from the queue. The vertex *h* neighbor of *g* is pushed to the queue

The vertices traversed so for are:

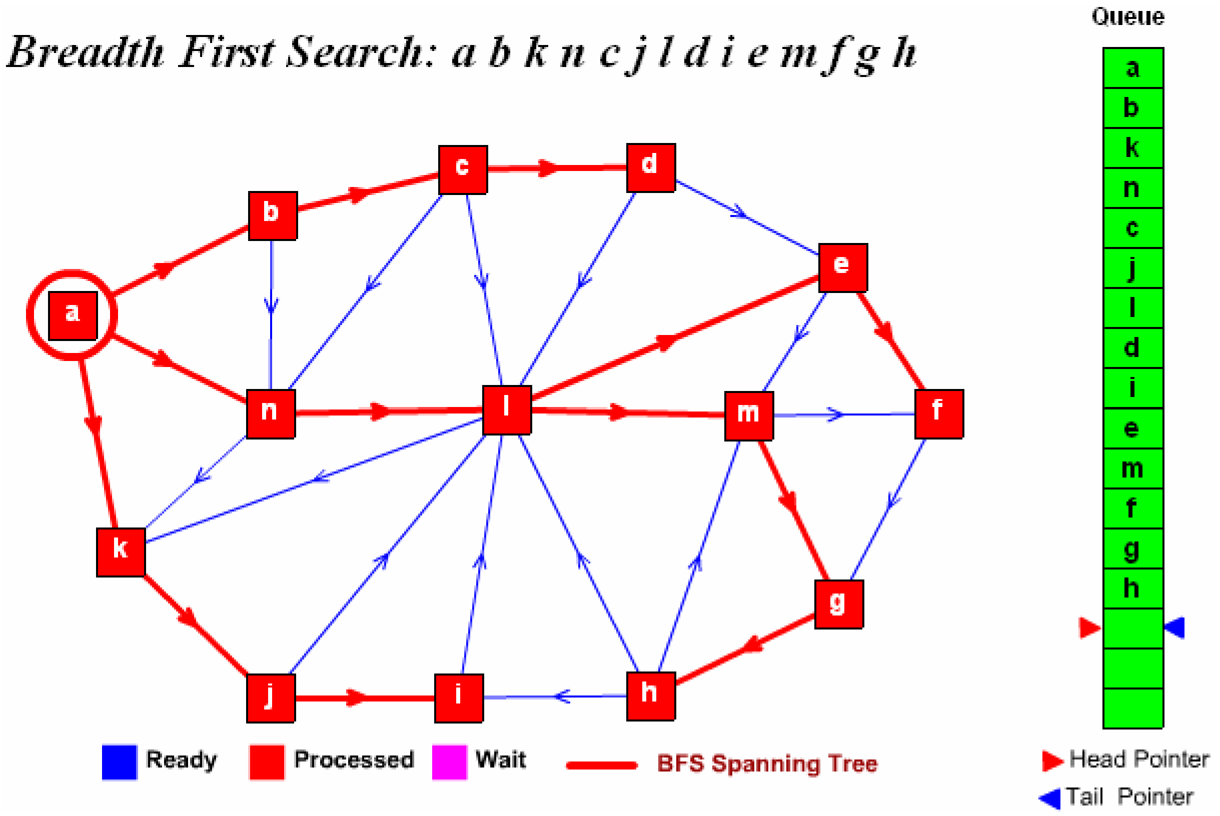
a,b,k,n,c,j,l,d,I,e,f,g



Breadth First Search

Example

(15)The vertex *h* is removed from the queue. The vertex *h* has no unvisited neighbors. Further, the queue is empty. The breadth first search algorithm terminates



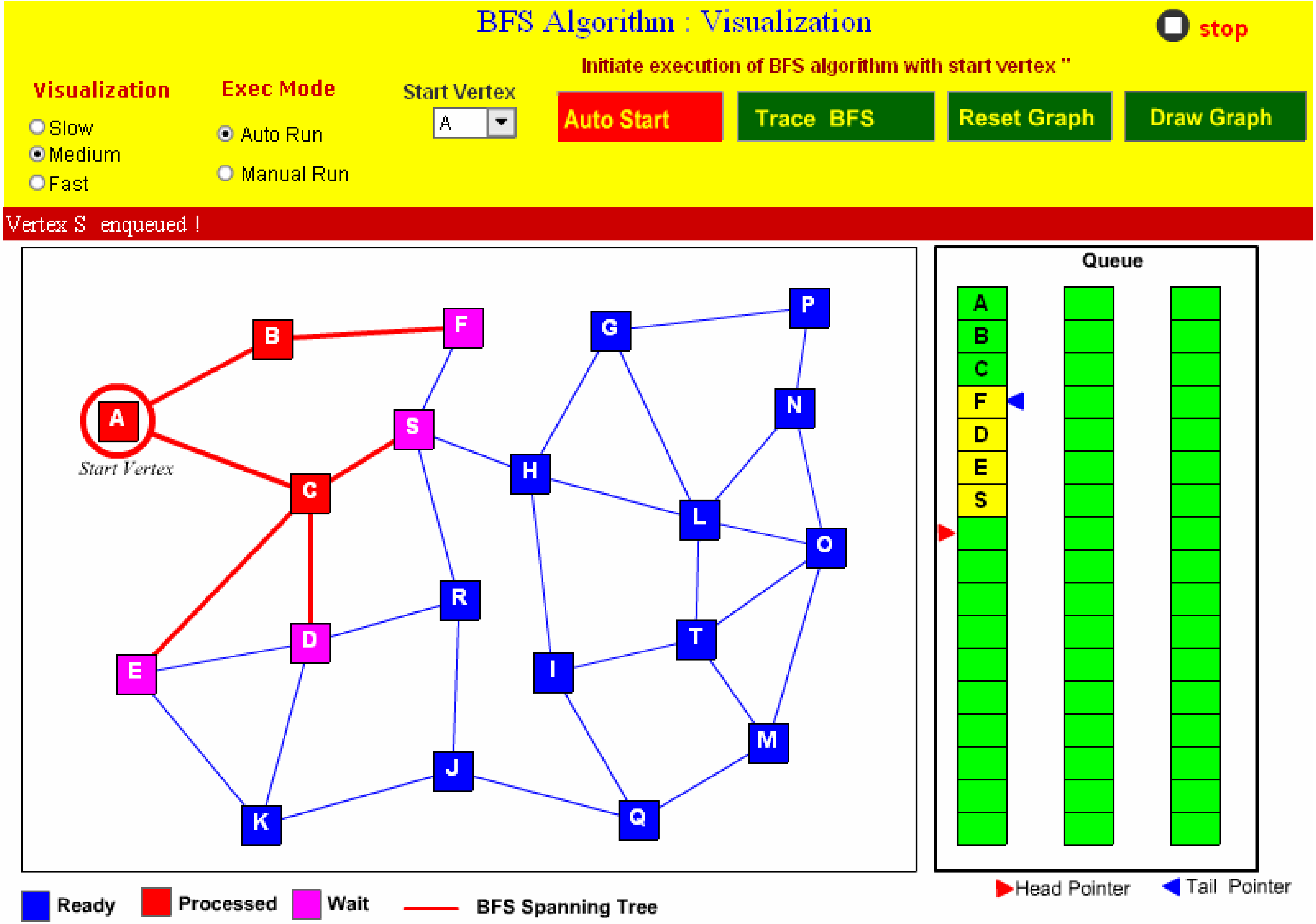
Breadth First Search

Implementation

The BFS implements ***breadth-first-search procedure*** by using a queue ***Q***. The address ***s*** of first selected vertex is passed to the method. The implementation uses an array ***state[]*** to keep track of the state of vertices: READY, WAIT, PROCESSED. The graph is represented by ***adjace***

```
BFS(G, s)                              ▶ s is address of starting vertex
for each vertex u ∈ V[G] – { s }       ▶ V[G] is vertex set of graph G. V[G]-{s} is the
                                       difference set that excludes initial vertex s in {s}.
    do state[u] ← READY                ▶ All unvisited vertices are assigned READY status
    state[s] ← WAIT                       Vertex s is being pushed to queue and assigned WAIT status
Q ← φ                                  ▶ Queue is initialized as empty
ENQUEUE(Q,s)                           ▶ The starting vertex s is added to the queue
while Q ≠ φ                            ▶ A loop is created to remove vertices from the queue.
    do u ← DEQUEUE(Q)                  ▶ u is the removed vertex
        for each v ∈ Adj[u]            ▶ All vertices v that are adjacent to vertex u are explored
            do if state[v] = READY     ▶ If the neighbor vertex v is not visited,
                then state[v] ← WAIT      it is assigned WAIT status
                     ENQUEUE(Q, v)           and added to the queue
state[u] ← PROCESSED                   ▶ The vertex u removed from the queue is marked visited
                                       and assigned PROCESSED status
```

BFS Visualization



BFS Algorithm : Visualization

Initiate execution of BFS algorithm with start vertex "

Visualization
○ Slow
● Medium
○ Fast

Exec Mode
● Auto Run
○ Manual Run

Start Vertex
A

Vertex S enqueued !

Analysis of BFS

Running Time

The running time for BFS consists of following major components

T_i = *Initializing vertices* T

T_s = *Scanning Adjacency list*

The Graph *G(V,E)* has $|V|$ vertices and $|E|$ edges. Each vertex is initialized once. Therefore,

$$T_i = |V|$$

The scanning of the Adjacency list takes place for each of the dequeued vertices. In ***BSF*** all links are are examined. The total time is proportional to the number of edges scanned. Since $|E|$ is the number of edges in G,

$$T_s = |E|$$

Thus time running time T_{BST} is given by

$$T_{BST} = |V| + |E| = \boldsymbol{O(|V|+|E|)}$$