Algorithmics

Lecture 5

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 3 – Graphs and graph algorithms

Graph basics – recap

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Section 3 – Graphs and graph algorithms

Graph basics – recap

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Graph basics

(undirected) graph $G = (V,E)$

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

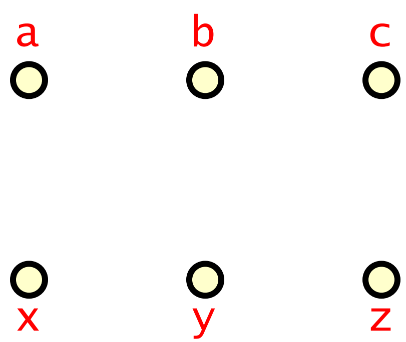
Graph basics

(undirected) graph `G = (V,E)`

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

Pictorially:

- a vertex is represented by a point



V= {a,b,c,x,y,z}

E= { {a,x},{a,y},{a,z},
{b,x},{b,y},{b,z},
{c,x},{c,y},{c,z} }

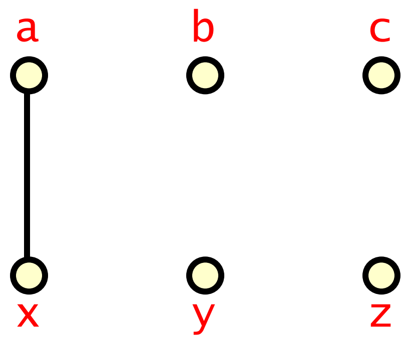
Graph basics

(undirected) graph `G = (V,E)`

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

Pictorially:

- a vertex is represented by a point
- an edge by a line joining the relevant pair of points



V= {a,b,c,x,y,z}

E= { **{a,x}**,{a,y},{a,z},
{b,x},{b,y},{b,z},
{c,x},{c,y},{c,z} }

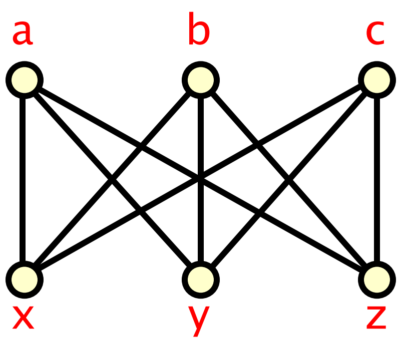
Graph basics

(undirected) graph $G = (V,E)$

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

Pictorially:

- a vertex is represented by a point
- an edge by a line joining the relevant pair of points



V= {a,b,c,x,y,z}

E= { {a,x},{a,y},{a,z},
{b,x},{b,y},{b,z},
{c,x},{c,y},{c,z} }

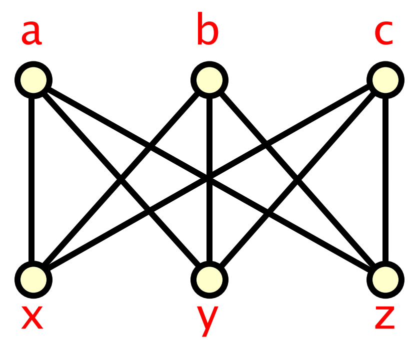
Graph basics

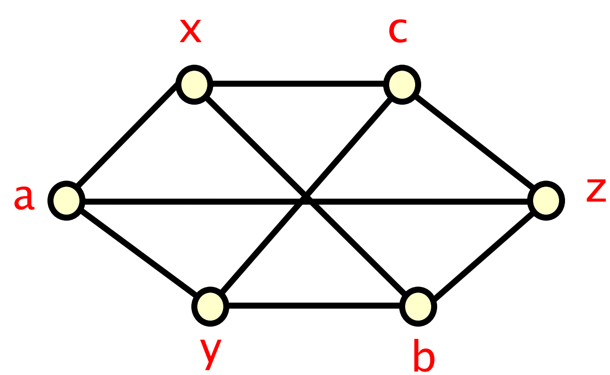
(undirected) graph `G = (V,E)`

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

Pictorially:

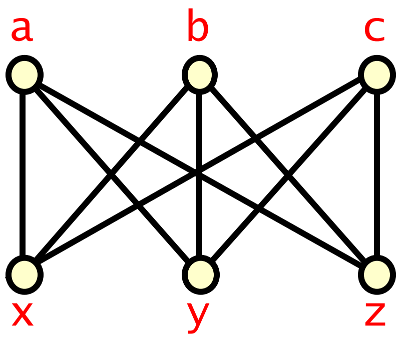
- a vertex is represented by a point
- an edge by a line joining the relevant pair of points
- a graph can be drawn in different ways
- e.g. two representations of the same graph

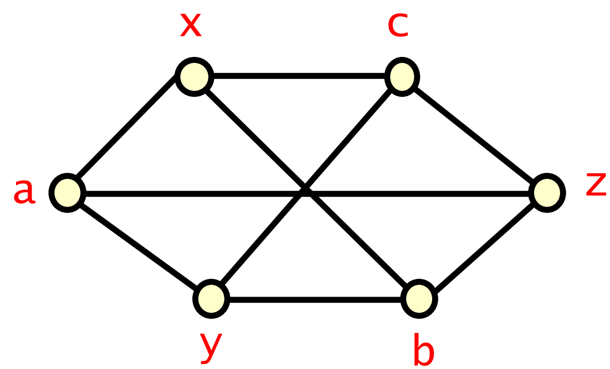





```
V= {a,b,c,x,y,z}
E= { {a,x},{a,y},{a,z},
     {b,x},{b,y},{b,z},
     {c,x},{c,y},{c,z} }
```

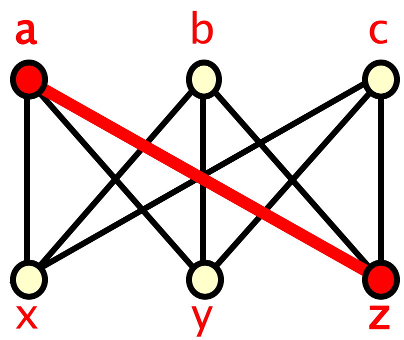
Graph basics

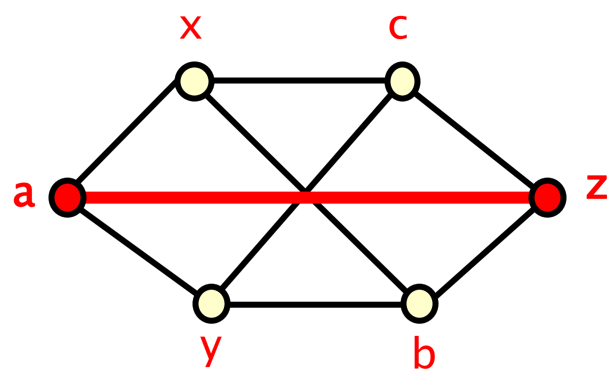




In this graph:

Graph basics

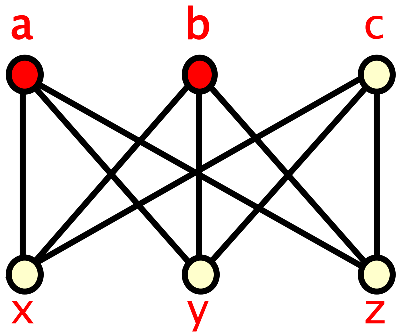


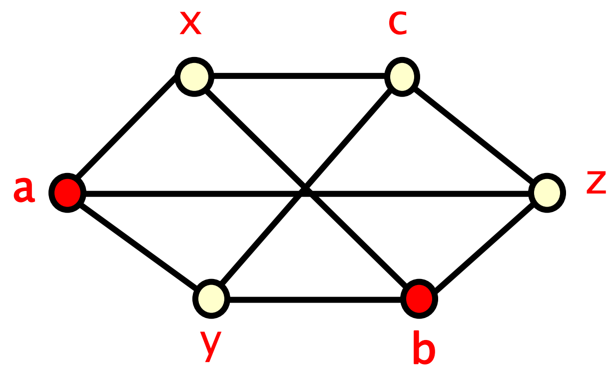


In this graph:

- vertices a & z are adjacent that is {a,z} is an element of the edge set E

Graph basics

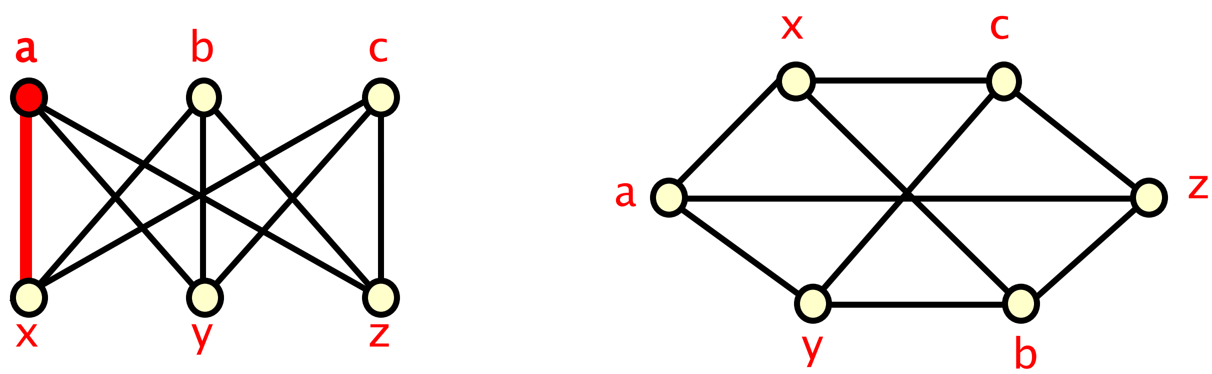




In this graph:

- vertices a & z are adjacent that is {a,z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a,b} is not an element of E

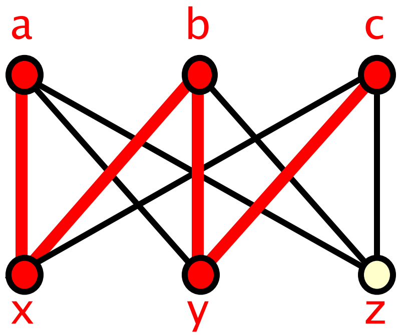
Graph basics

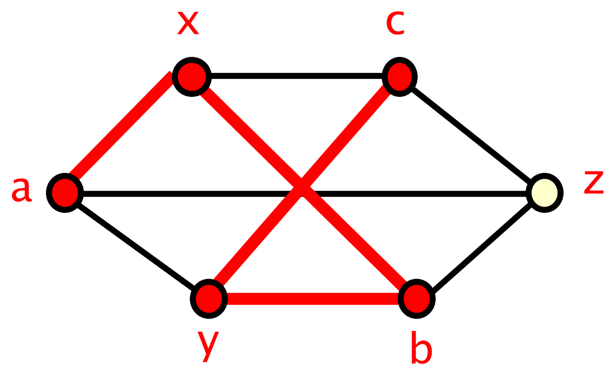


In this graph:

- vertices a & z are adjacent that is {a,z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a,b} is not an element of E
- vertex a is incident to edge {a,x}

Graph basics

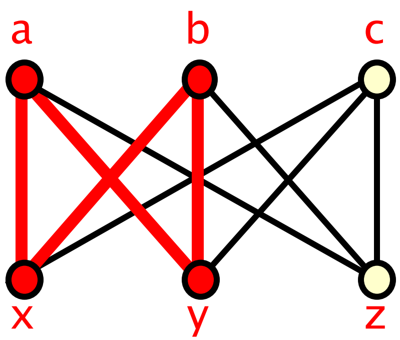


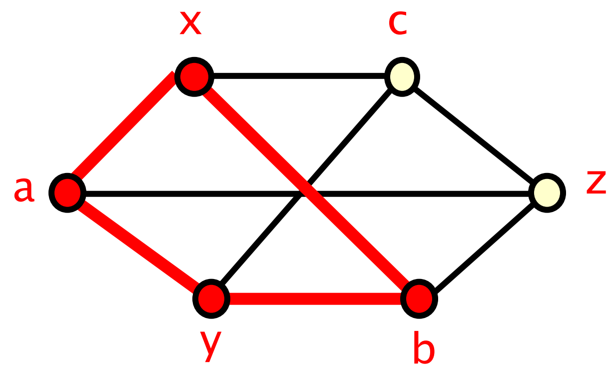


In this graph:

- vertices a & z are adjacent that is {a,z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a,b} is not an element of E
- vertex a is incident to edge {a,x}
- a→x→b→y→c is a path of length 4 (number of edges)

Graph basics

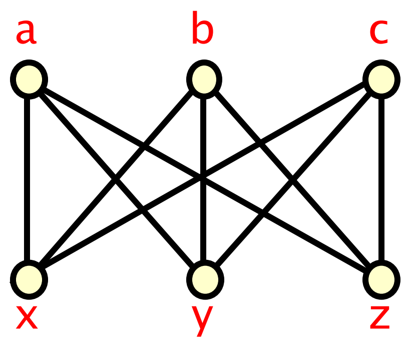


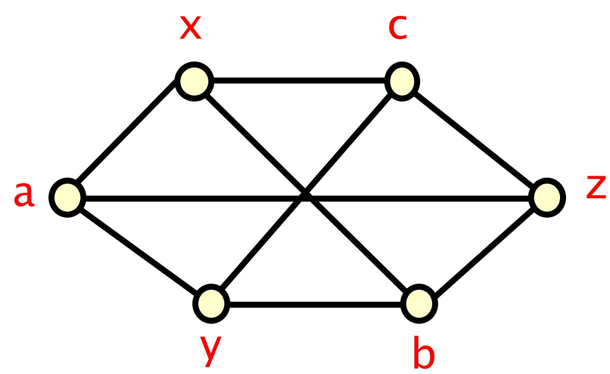


In this graph:

- vertices a & z are adjacent that is {a, z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a, b} is not an element of E
- vertex a is incident to edge {a, x}
- a→x→b→y→c is a path of length 4 (number of edges)
- a→x→b→y→a is a cycle of length 4

Graph basics



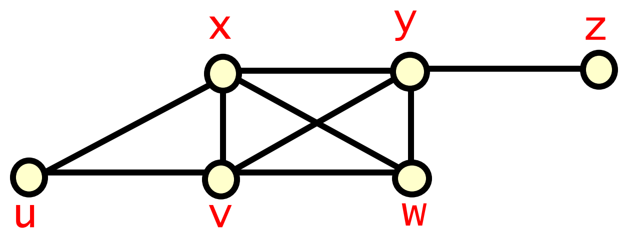


In this graph:

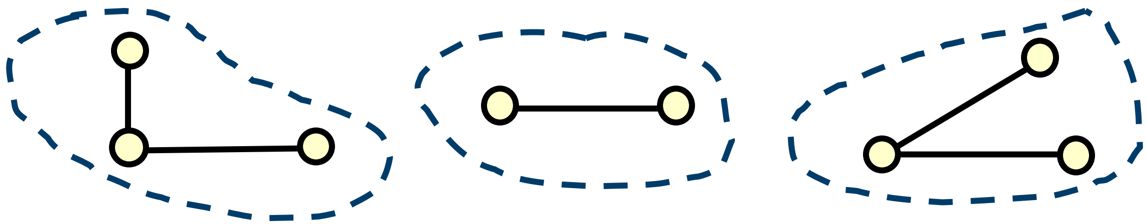
- vertices a & z are adjacent that is {a, z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a, b} is not an element of E
- vertex a is incident to edge {a, x}
- a→x→b→y→c is a path of length 4 (number of edges)
- a→x→b→y→a is a cycle of length 4
- all vertices have degree 3
  - i.e. all vertices are incident to three edges

Graph basics – Definitions

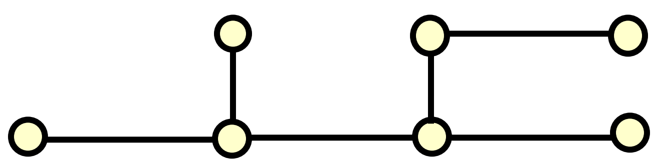
A graph is: connected, if every pair of vertices is joined by a path



A non-connected graph has two or more connected components



- A graph is a tree if it is connected and acyclic (no cycles)

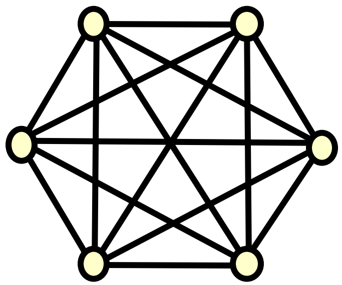


a tree with n vertices has n-1 edges
– at least n-1 edges to be connected
– at most n-1 edges to be acyclic

- A graph is a forest if it is acyclic and components are trees

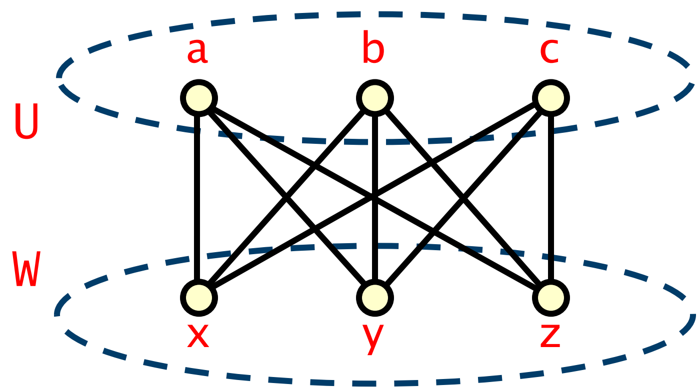
Graph basics – Definitions

A graph is **complete** (**a clique**) if every pair vertices is joined by an edge



K_6, the clique on 6 vertices

A graph is **bipartite** if the vertices are in two **disjoint** sets **U** & **W** and **every** edge joins a vertex in **U** to a vertex in **W**

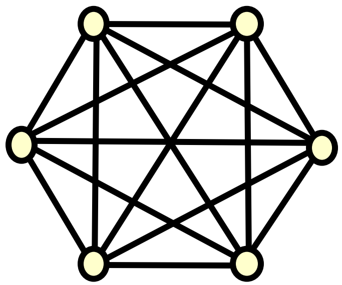


the complete bipartite graph $K_{3,3}$

it is complete since all edges between vertices in U and W are present

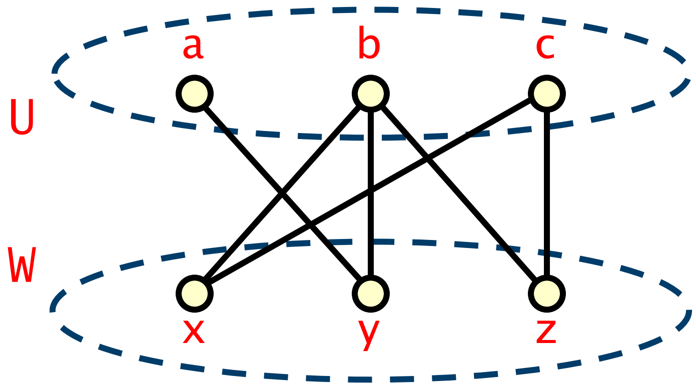
Graph basics – Definitions

A graph is **complete** (**a clique**) if every pair vertices is joined by an edge



K_6, the clique on 6 vertices

A graph is **bipartite** if the vertices are in two **disjoint** sets **U** & **W** and **every** edge joins a vertex in **U** to a vertex in **W**



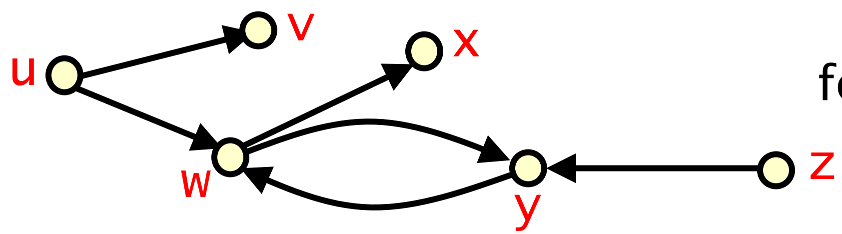
bipartite graphs do not need to be complete

Graph basics – Directed graphs

A **directed graph** (digraph) $D = (V,E)$

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



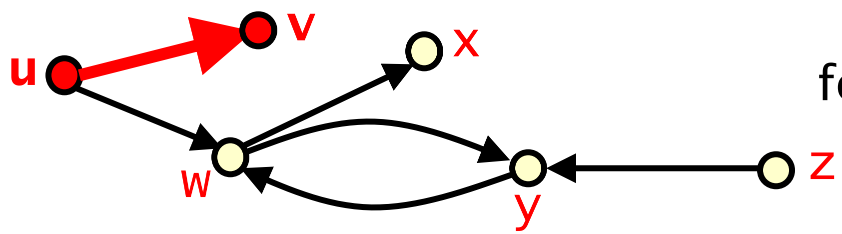
for example $(u,v),(w,y),(y,w) \in E$

Graph basics – Directed graphs

A directed graph (digraph) D = (V,E)

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example (u,v),(w,y),(y,w) $\in$ E

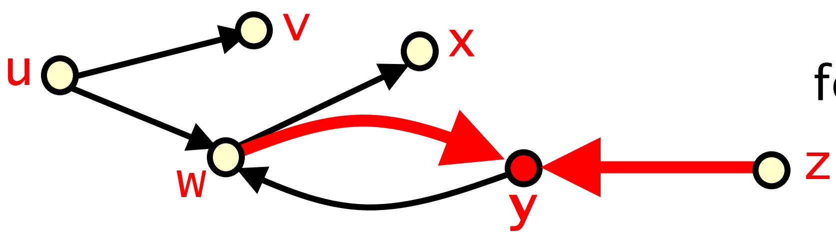
- u is adjacent to v and v is adjacent from u

Graph basics – Directed graphs

A directed graph (digraph) D = (V,E)

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example (u,v),(w,y),(y,w) $\in$ E

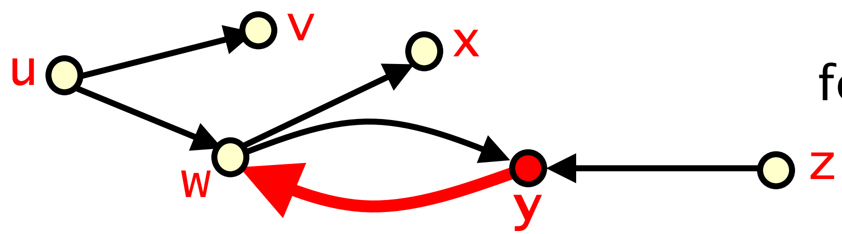
- u is adjacent to v and v is adjacent from u
- y has in-degree 2

Graph basics – Directed graphs

A directed graph (digraph) $D = (V,E)$

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example $(u,v),(w,y),(y,w) \in E$

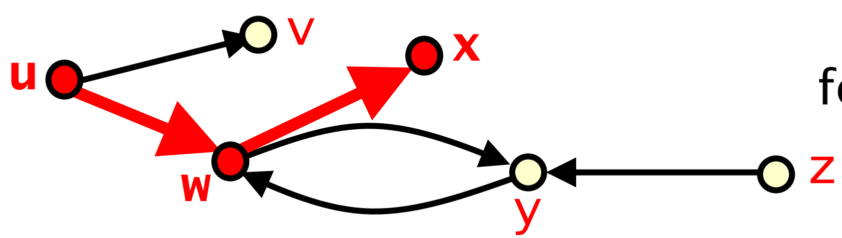
- u is adjacent to v and v is adjacent from u
- y has in-degree 2 and out-degree 1

Graph basics – Directed graphs

A directed graph (digraph) D = (V,E)

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example (u,v),(w,y),(y,w) $\in$ E

- u is adjacent to v and v is adjacent from u
- y has in-degree 2 and out-degree 1

In a digraph, paths and cycles must follow edge directions

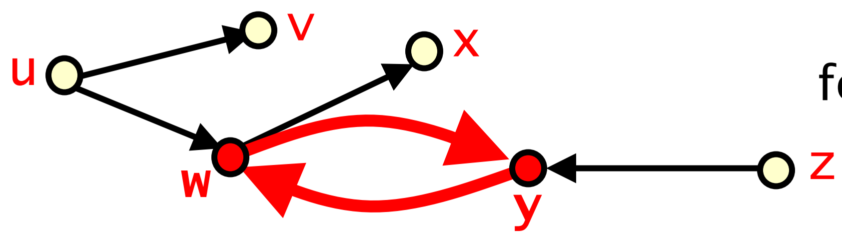
- e.g. u → w → x is a path

Graph basics – Directed graphs

A directed graph (digraph) `D = (V,E)`

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example (u,v),(w,y),(y,w) ∈ E

- u is adjacent to v and v is adjacent from u
- y has in-degree 2 and out-degree 1

In a digraph, paths and cycles must follow edge directions

- e.g. u → w → x is a path and w → y → w is a cycle

Section 3 – Graphs and graph algorithms

Graph basics

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- breadth/depth first search

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim-Jarnik and Dijkstra's refinement)

Topological ordering

Graph representations – Undirected graphs

Undirected graph: Adjacency matrix

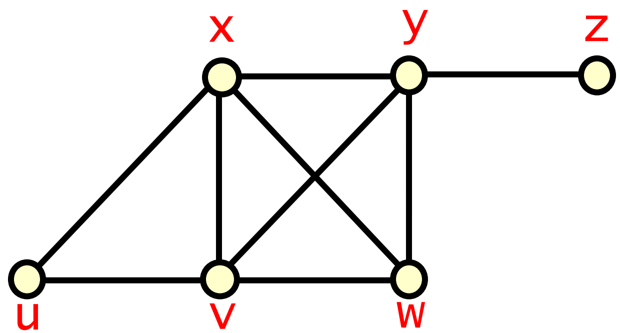
- one row and column for each vertex
- row `i`, column `j` contains a `1` if `i`th and `j`th vertices adjacent, `0` otherwise

Undirected graph: Adjacency lists

- one list for each vertex
- list `i` contains an entry for `j` if the vertices `i` and `j` are adjacent

Graph representations – Undirected graphs

Undirected graph **G**



Adjacency matrix for **G**

	u	v	w	x	y	z
u:	0	1	0	1	0	0
v:	1	0	1	1	1	0
w:	0	1	0	1	1	0
x:	1	1	1	0	1	0
y:	0	1	1	1	0	1
z:	0	0	0	0	1	0

$|V| \times |V|$ array

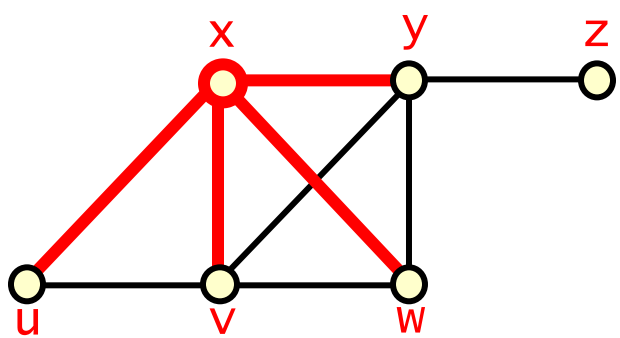
Adjacency lists for **G**

u: $v \rightarrow x$
v: $u \rightarrow w \rightarrow x \rightarrow y$
w: $v \rightarrow x \rightarrow y$
x: $u \rightarrow v \rightarrow w \rightarrow y$
y: $v \rightarrow w \rightarrow x \rightarrow z$
z: y

$2 \times |E|$ entries in all

Graph representations – Undirected graphs

Undirected graph G



Adjacency matrix for **G**

	u	v	w	x	y	z
u:	0	1	0	1	0	0
v:	1	0	1	1	1	0
w:	0	1	0	1	1	0
x:	**1**	**1**	**1**	**0**	**1**	**0**
y:	0	1	1	1	0	1
z:	0	0	0	0	1	0

$|V| \times |V|$ array

Adjacency lists for **G**

u: v→x
v: u→w→x→y
w: v→x→y
x: u→v→w→y
y: v→w→x→z
z: y

$2 \times |E|$ entries in all

Graph representations – Directed graphs

Directed graph: Adjacency matrix

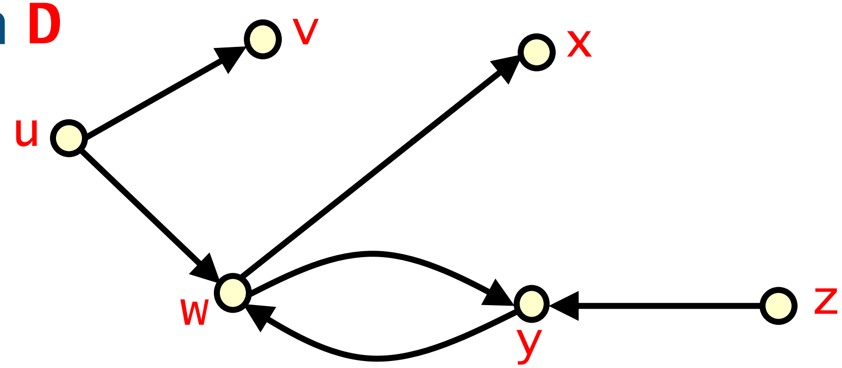
- one row and column for each vertex
- row `i`, column `j` contains a `1` if there is an edge **from** `i` **to** `j` and `0` otherwise

Directed graph: Adjacency lists

- one list for each vertex
- the list for vertex `i` contains vertex `j` if there is an edge **from** `i` **to** `j`

Graph representations – Directed graphs

Directed graph **D**



Adjacency matrix for **D**

	u	v	w	x	y	z
u:	0	1	1	0	0	0
v:	0	0	0	0	0	0
w:	0	0	0	1	1	0
x:	0	0	0	0	0	0
y:	0	0	1	0	0	0
z:	0	0	0	0	1	0

$|V| \times |V|$ array

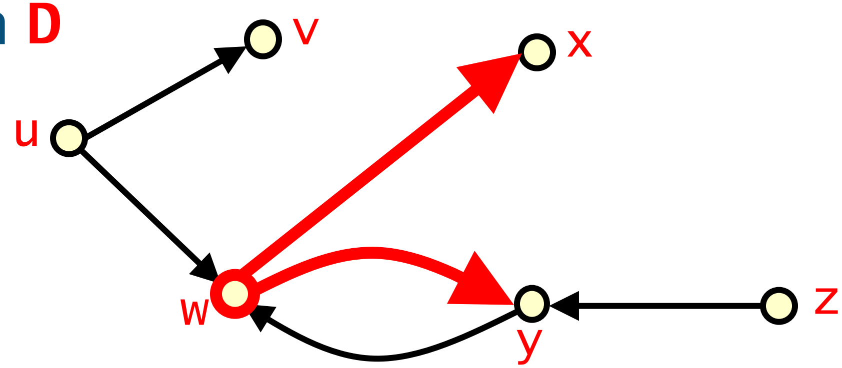
Adjacency lists for **D**

u: v→w
v:
w: x→y
x:
y: w
z: y

$|E|$ entries in all

Graph representations – Directed graphs

Directed graph D



Adjacency matrix for **D**

	u	v	w	x	y	z
u:	0	1	1	0	0	0
v:	0	0	0	0	0	0
w:	**0**	**0**	**0**	**1**	**1**	**0**
x:	0	0	0	0	0	0
y:	0	0	1	0	0	0
z:	0	0	0	0	1	0

$|V| \times |V|$ array

Adjacency lists for **D**

u: v→w
v:
w: x→y
x:
y: w
z: y

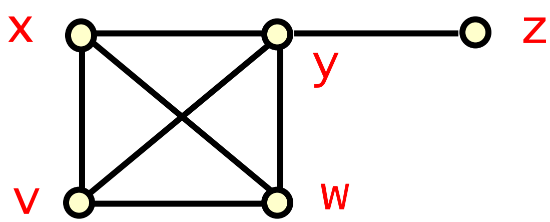
$|E|$ entries in all

Implementation – Adjacency lists

Recall adjacency list for an undirected graph

- one list for each vertex
- list i contains an element for j if the vertices i and j are adjacent

graph G



adjacency lists for G

v: w→x→y
w: v→x→y
x: v→w→y
y: v→w→x→z
z: y

Implementation: define classes for

- the entries of adjacency lists
- the vertices (includes a linked list representing its adjacency list)
- graphs (includes the size of the graph and an array of vertices)
  - array allows for efficient access using “index” of a vertex

Implementation – Adjacency lists

```
 /** class to represent an entry in the adjacency list of a vertex
in a graph */
 public class AdjListNode {

   private int vertexIndex; // the vertex index of the entry

   // possibly other fields, for example representing properties
   // of the edge such as weight, capacity, …

   /** creates a new entry for vertex indexed i */
   public AdjListNode(int i){
     vertexIndex = i;
   }
   public int getVertexIndex(){ // gets the vertex index of the entry
     return vertexIndex;
   }
   public void setVertexIndex(int i){ // sets vertex index to i
     vertexIndex = i;
   }
 }
```

Implementation – Adjacency lists

```
import java.util.LinkedList; // we require the linked list class

/** class to represent a vertex in a graph */
public class Vertex {

  private int index; // the index of this vertex
  private LinkedList<AdjListNode> adjList; // the adjacency list of vertex

  // possibly other fields, e.g. representing data stored at the node

  /** create a new instance of vertex with index i */
  public Vertex(int i) {
    index = i; // set index
    adjList = new LinkedList<AdjListNode>();// create empty adjacency list
  }

   /** return the index of the vertex */
  public int getIndex(){
    return index;
  }
```

Implementation – Adjacency lists

```
 // class Vertex continued

   /** set the index of the vertex */
   public void setIndex(int i){
    index = i;
   }
   /** return the adjacency list of the vertex */
   public LinkedList<AdjListNode> getAdjList(){
    return adjList;
   }
   /** add vertex with index j to the adjacency list */
   public void addToAdjList(int j){
     adjList.addLast(new AdjListNode(j));
   }
   /** return the degree of the vertex */
   public int vertexDegree(){
     return adjList.size();
   }
}
```

Implementation – Adjacency lists

```
import java.util.LinkedList; // again require the linked list class
// (to add graph algorithms we will need to access adjacency lists)
/** class to represent a graph */
public class Graph {

  private Vertex[] vertices; // array of vertices for easy access
  private int numVertices = 0; // number of vertices
  // possibly other fields representing properties of the graph

  /** Create a Graph with n vertices indexed 0,...,n-1  */
  public Graph(int n) {
    numVertices = n;
    vertices = new Vertex[n];
    for (int i = 0; i < n; i++) vertices[i] = new Vertex(i);
  }
  /** returns number of vertices in the graph */
  public int size(){
    return numVertices;
  }
}
```

Section 3 – Graphs and graph algorithms

Graph basics

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

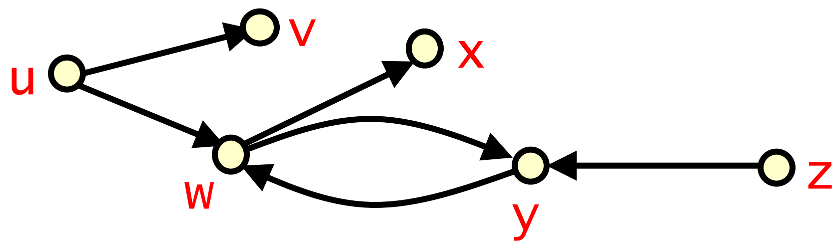
Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Graph search and traversal algorithms

Graph search and traversal algorithms

- a systematic way to explore a graph (when starting from some vertex)



Example: web crawler collects data from hypertext documents by traversing a directed graph **D** where

- vertices are hypertext documents
- (u,v) is an edge if document u contains a hyperlink to document v

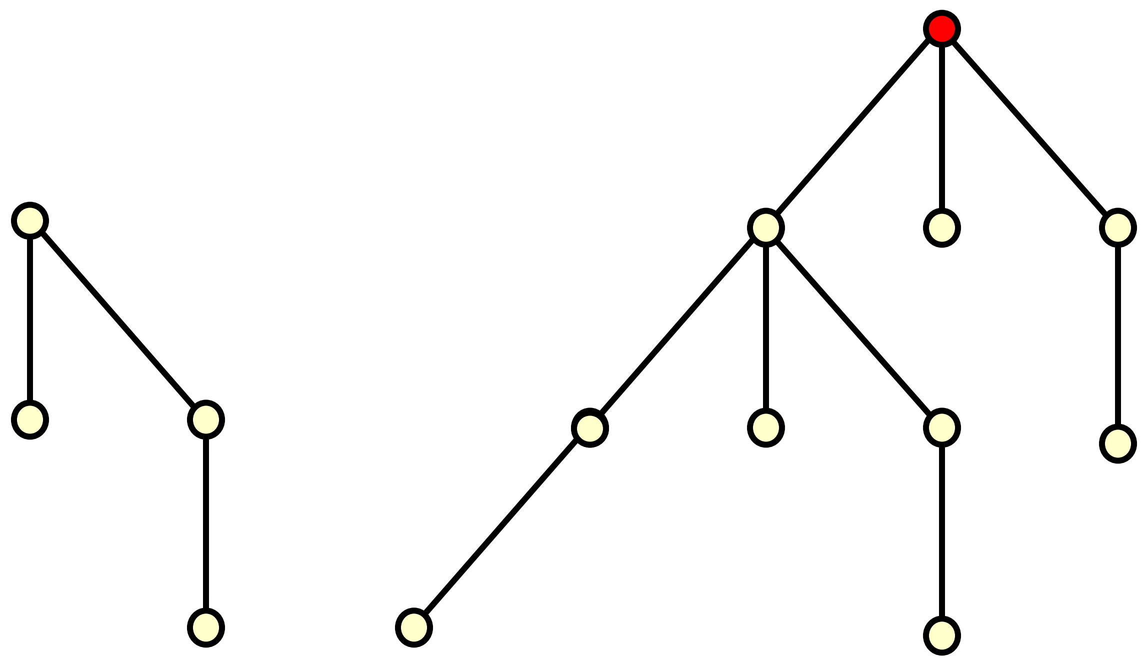
A search/traversal visits all vertices by travelling along edges

- traversal is efficient if it explores graph in $O(|V|+|E|)$ time

Depth first search/traversal (DFS)

From starting vertex

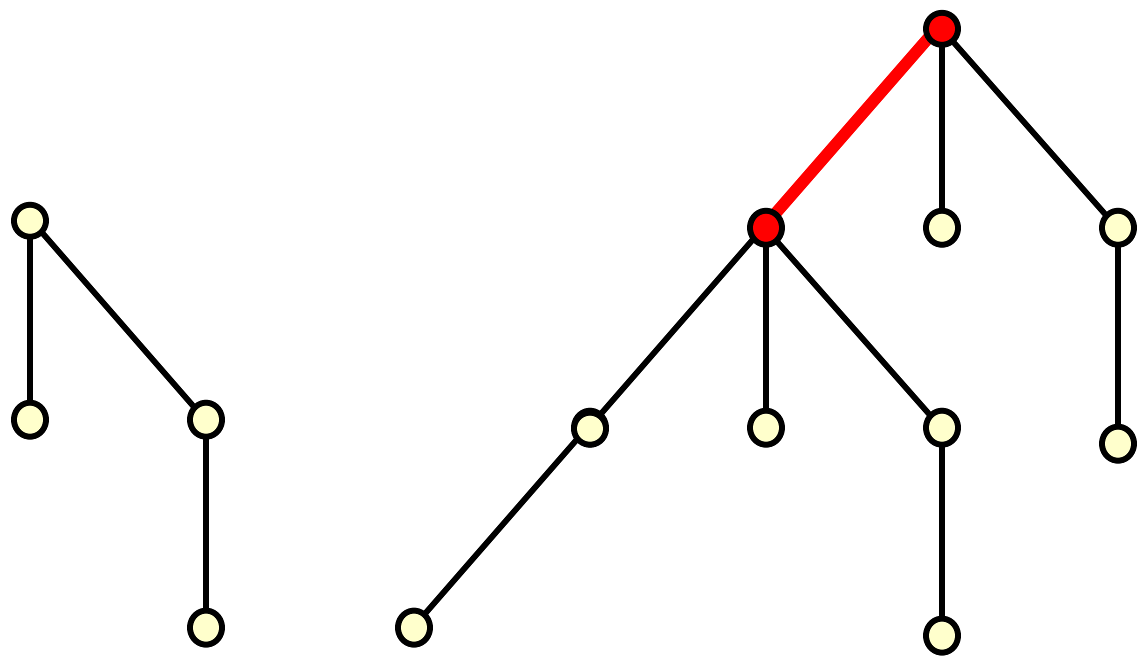
- follow a path of unvisited vertices until path can be extended no further



Depth first search/traversal (DFS)

From starting vertex

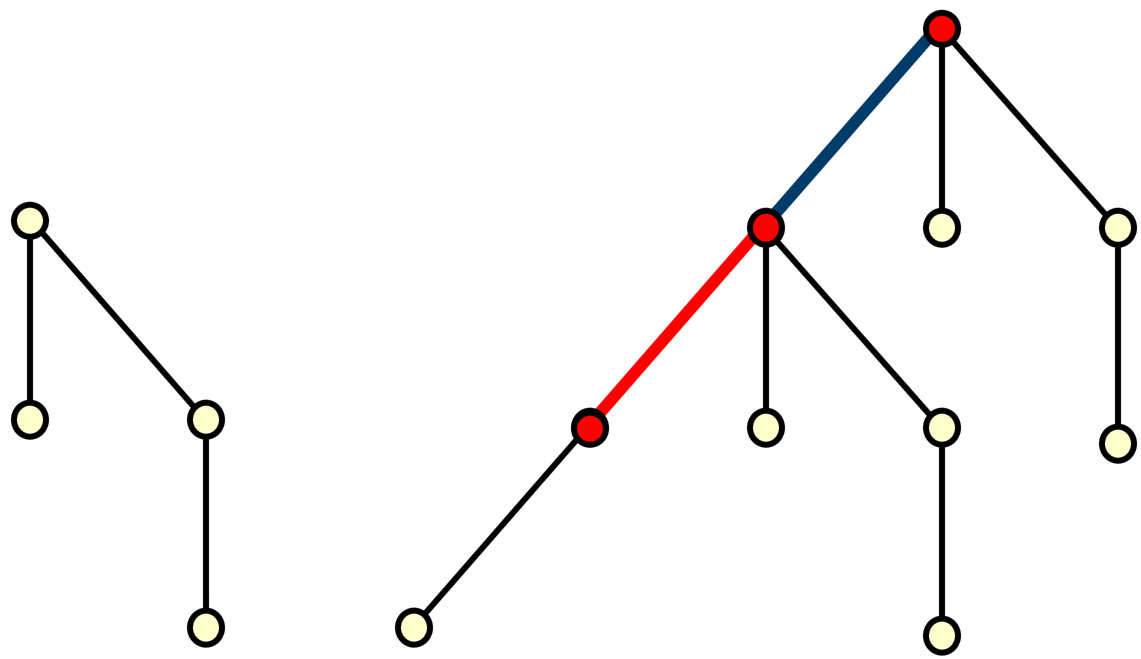
- follow a path of unvisited vertices until path can be extended no further



Depth first search/traversal (DFS)

From starting vertex

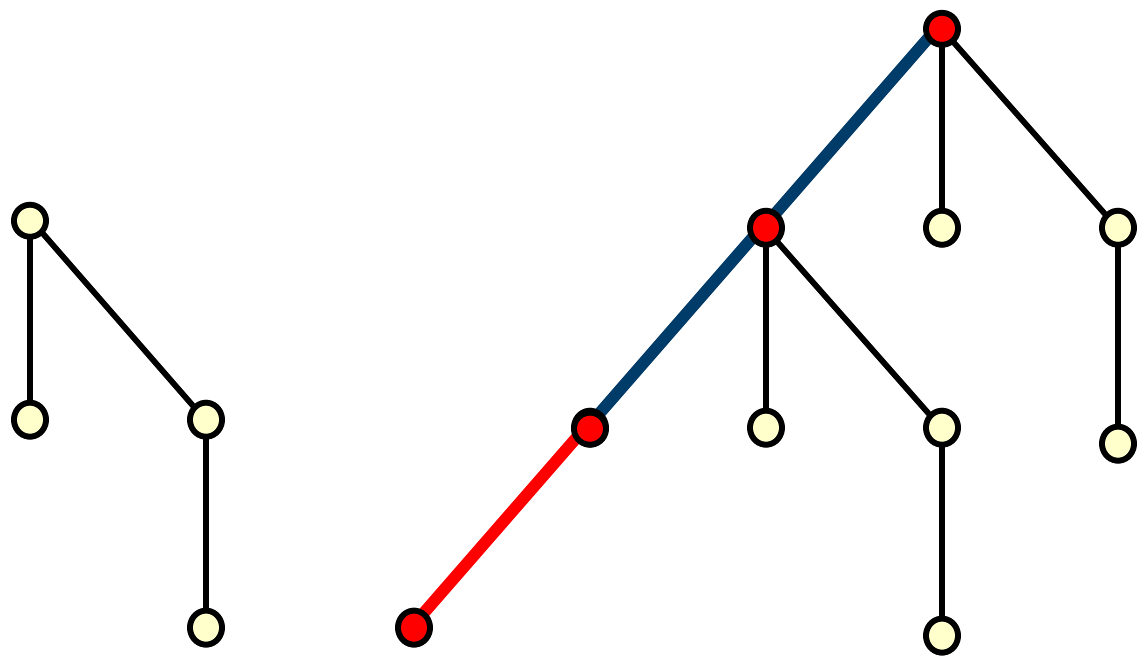
- follow a path of unvisited vertices until path can be extended no further



Depth first search/traversal (DFS)

From starting vertex

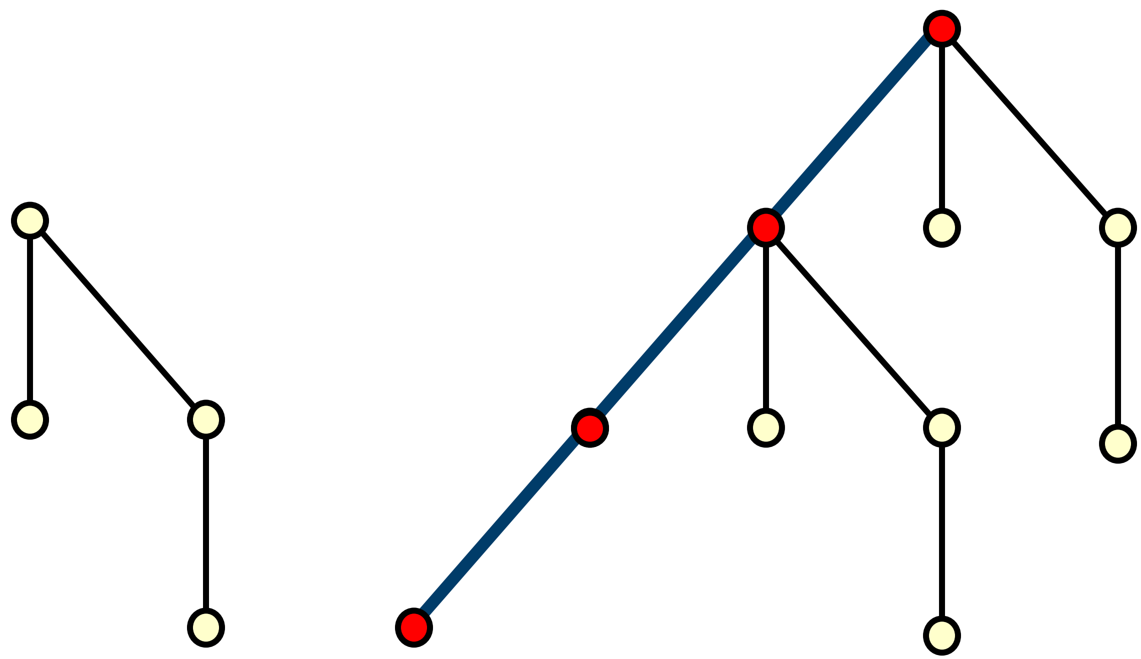
– follow a path of unvisited vertices until path can be extended no further



Depth first search/traversal (DFS)

From starting vertex

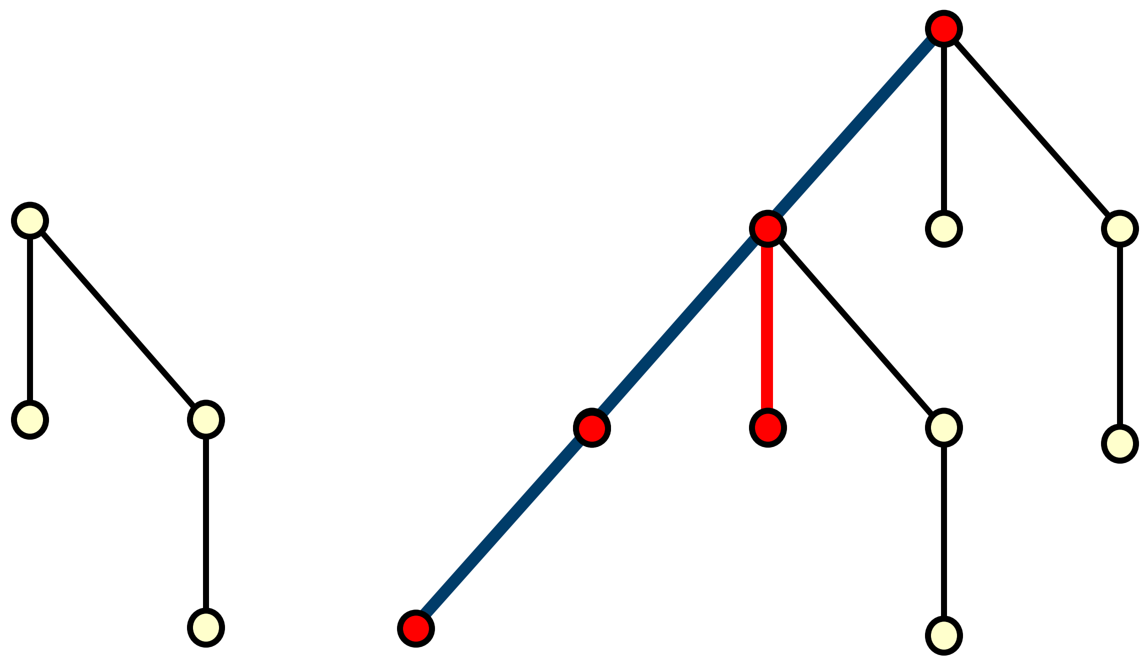
- follow a path of unvisited vertices until path can be extended no further



Depth first search/traversal (DFS)

From starting vertex

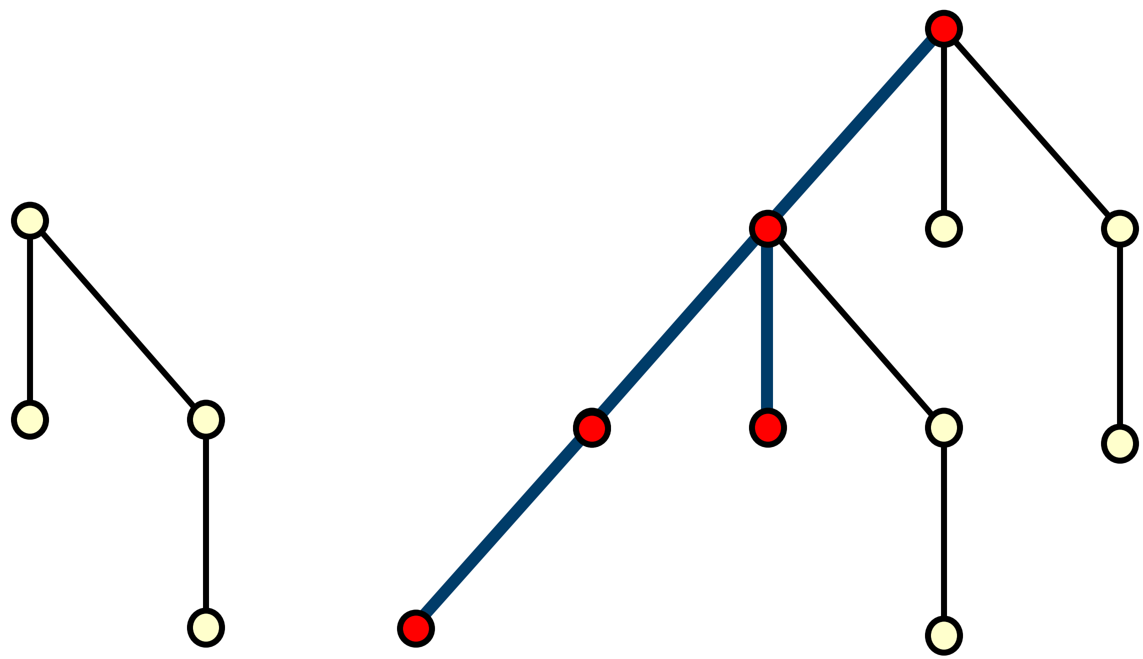
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached



Depth first search/traversal (DFS)

From starting vertex

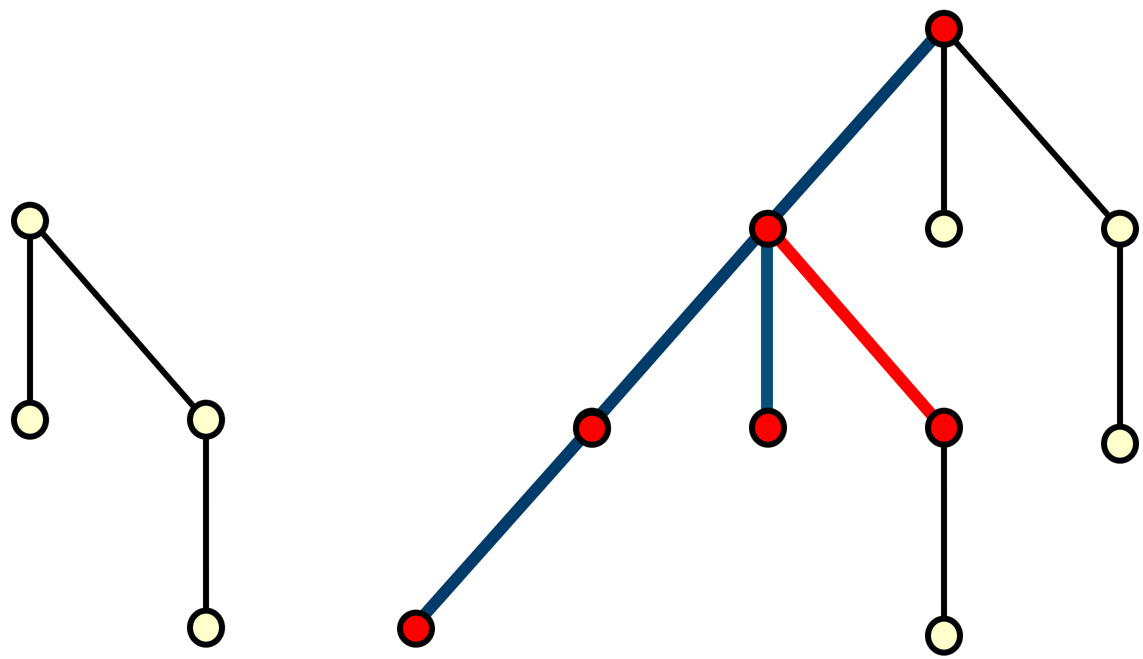
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached



Depth first search/traversal (DFS)

From starting vertex

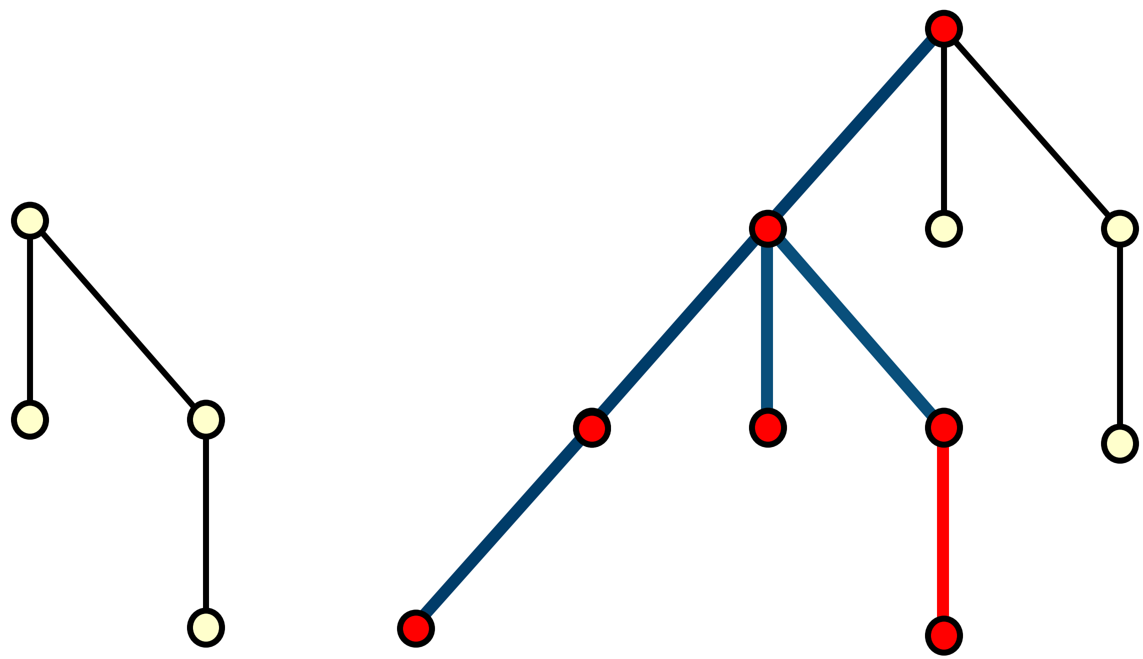
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices



Depth first search/traversal (DFS)

From starting vertex

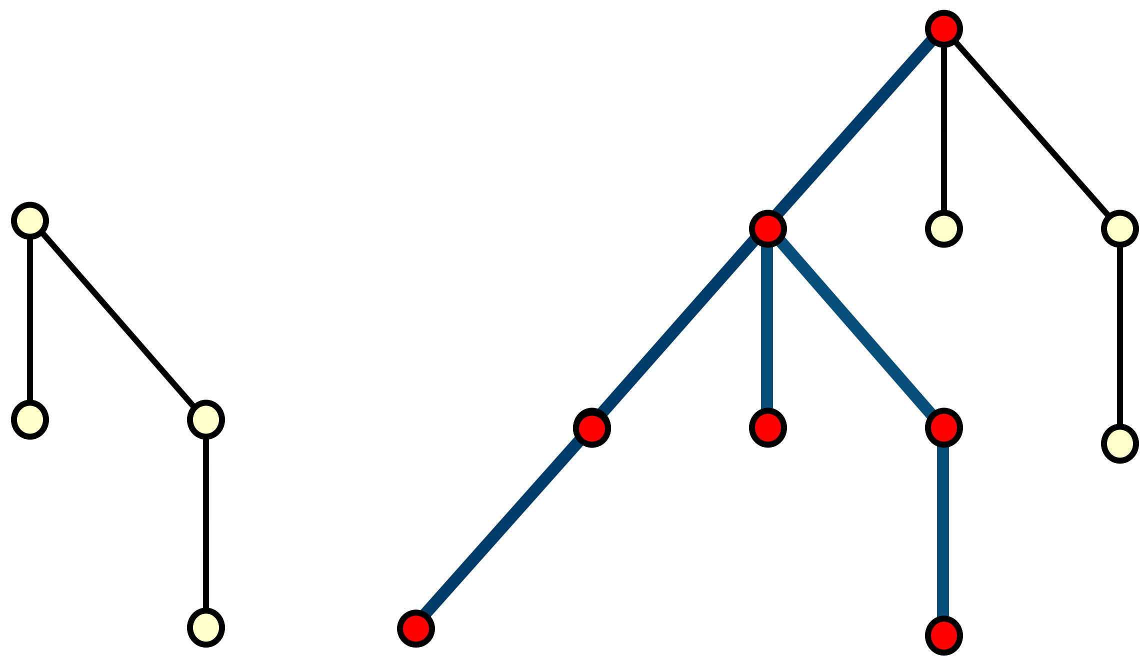
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices



Depth first search/traversal (DFS)

From starting vertex

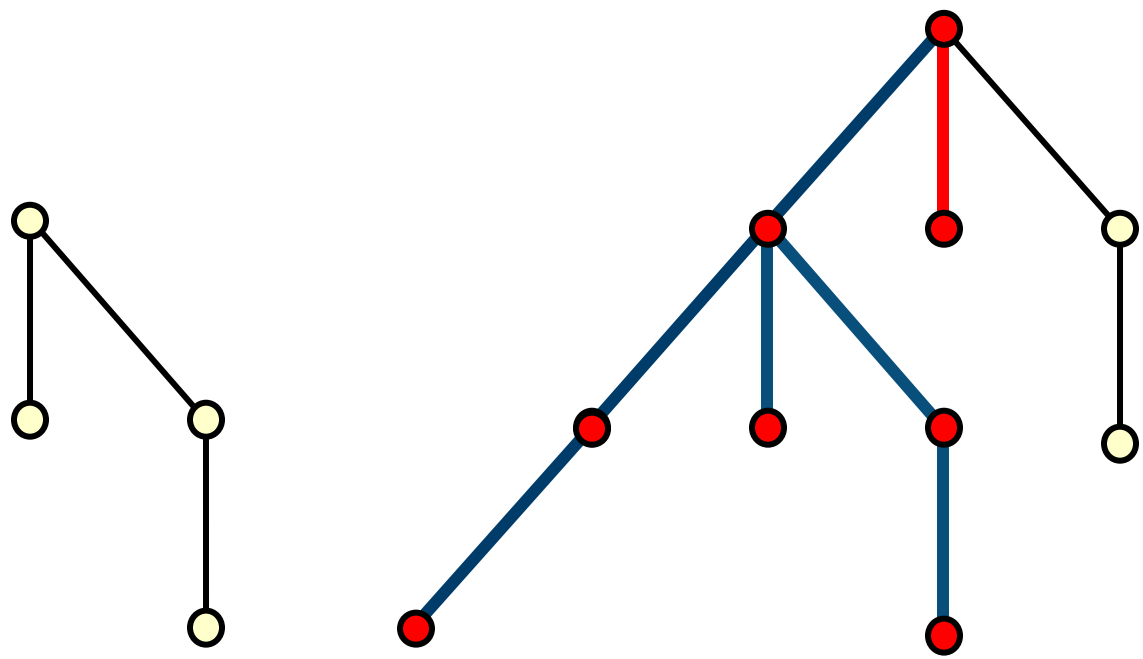
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices



Depth first search/traversal (DFS)

From starting vertex

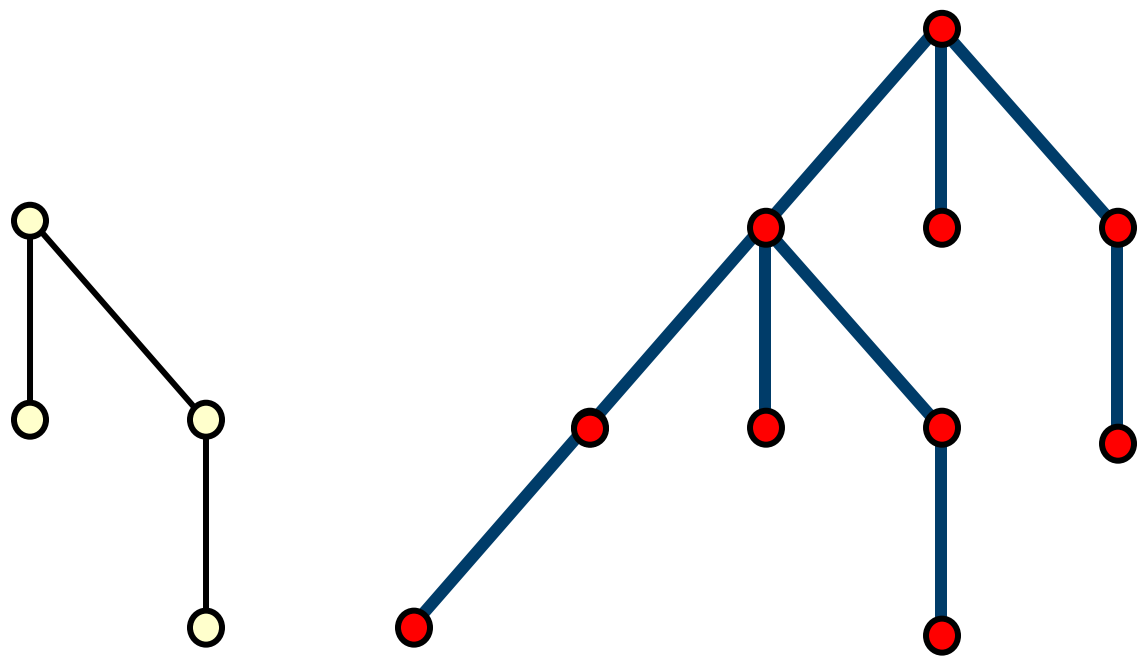
- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices



Depth first search/traversal (DFS)

From starting vertex

- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices

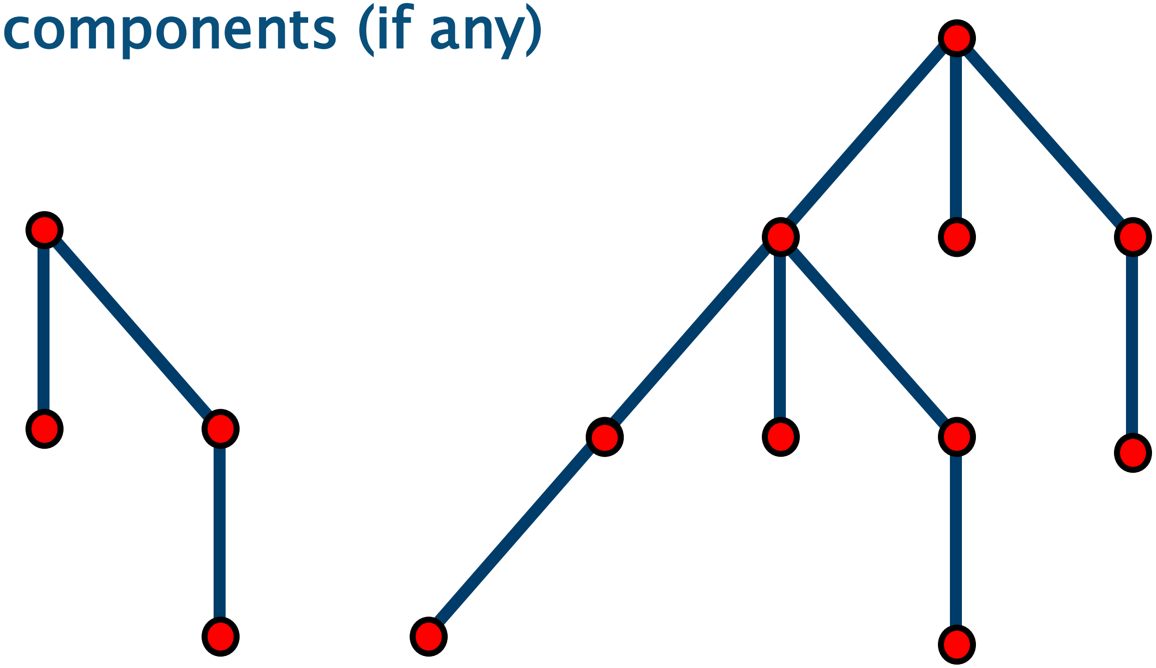


Depth first search/traversal (DFS)

From starting vertex

- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices

Repeat for other components (if any)



Depth first search/traversal (DFS)

From starting vertex

- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices

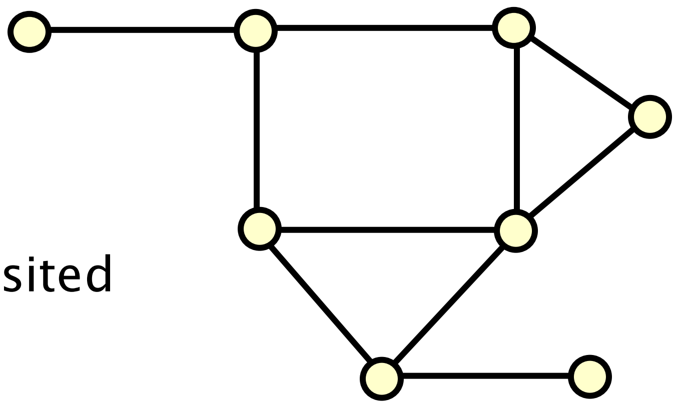
Repeat for other components (if any)

The edges traversed form a spanning tree (or forest)

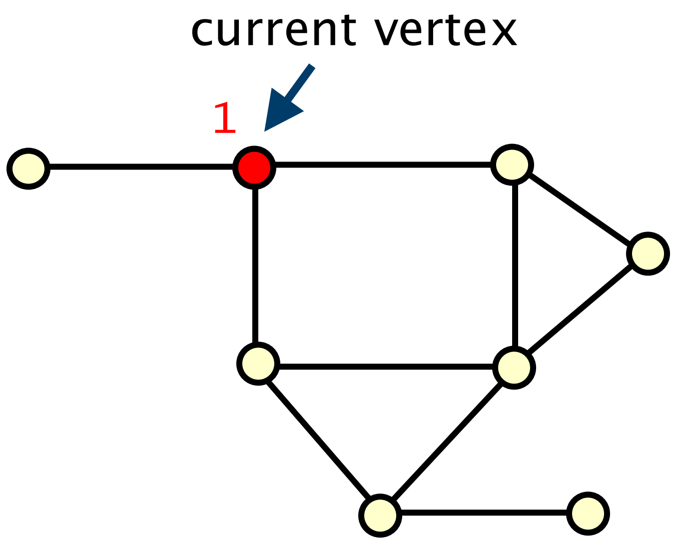
- a depth-first spanning tree (forest)
- spanning tree of a graph is a tree composed of all the vertices and some (or perhaps all) of the edges of the graph

Depth first traversal – Example

Undirected graph G

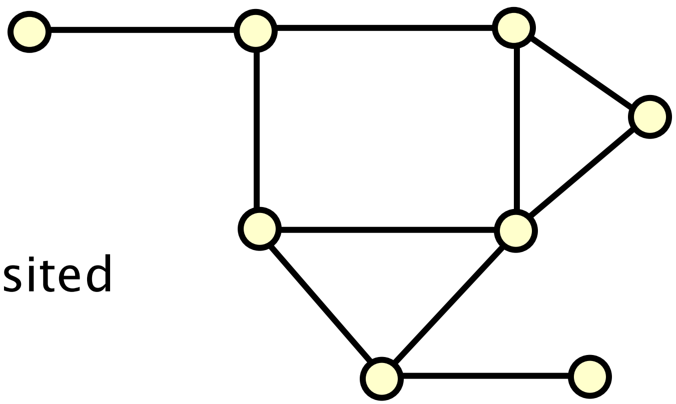


denotes vertex has been visited

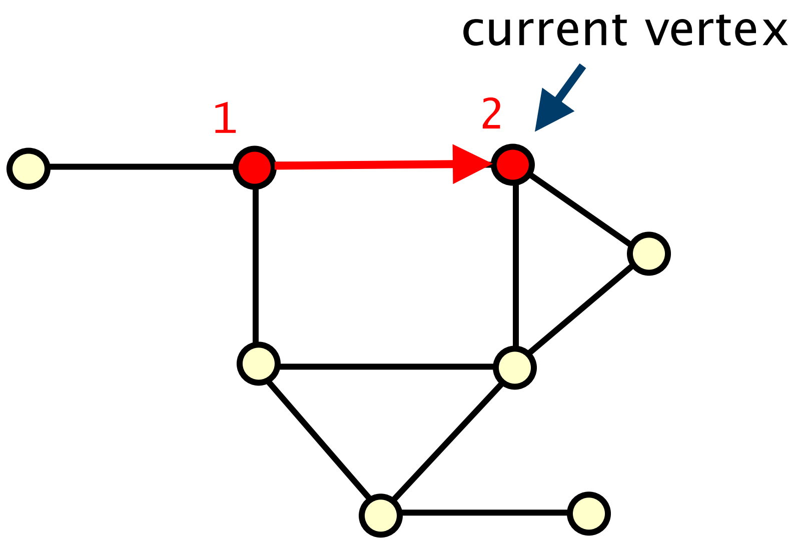


Depth first traversal – Example

Undirected graph G

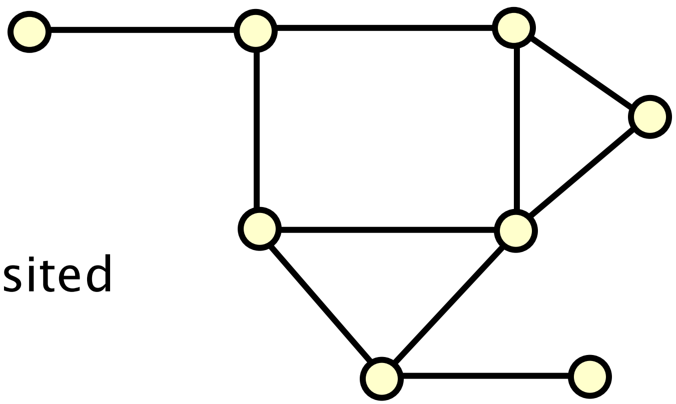


denotes vertex has been visited

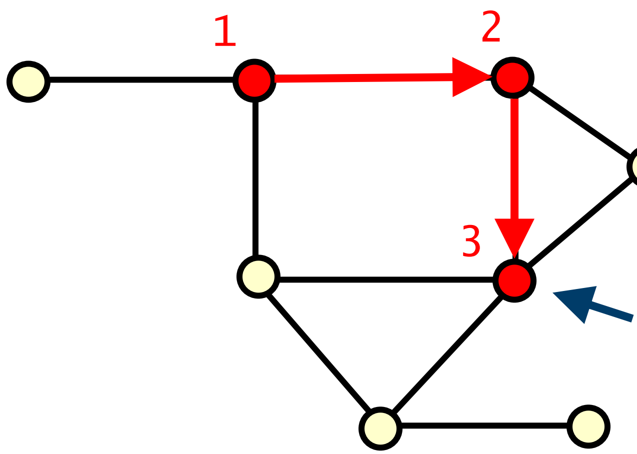


Depth first traversal – Example

Undirected graph G



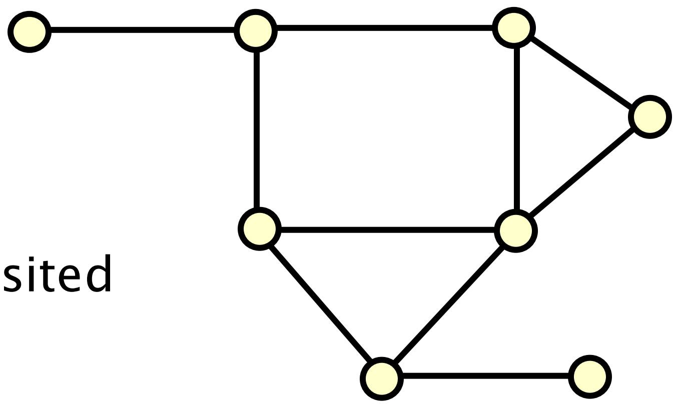
denotes vertex has been visited



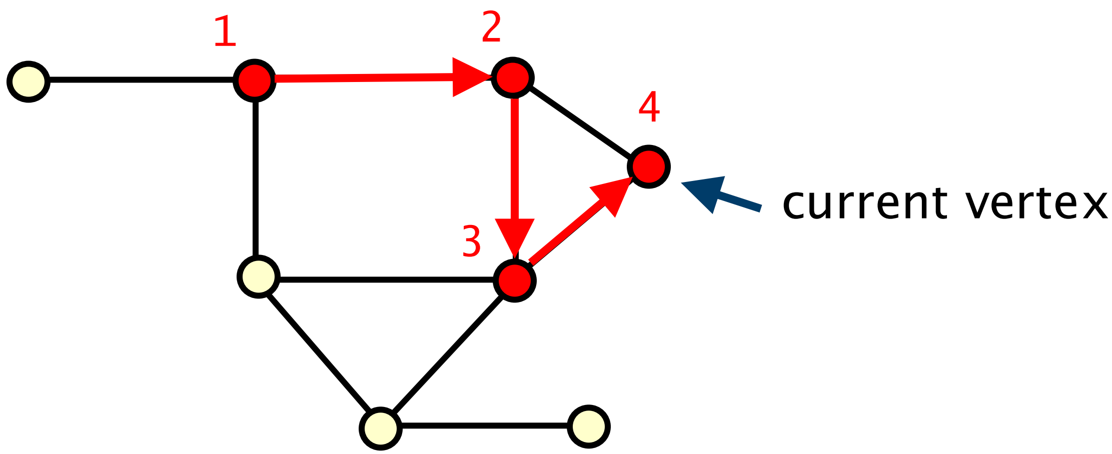
current vertex

Depth first traversal – Example

Undirected graph G

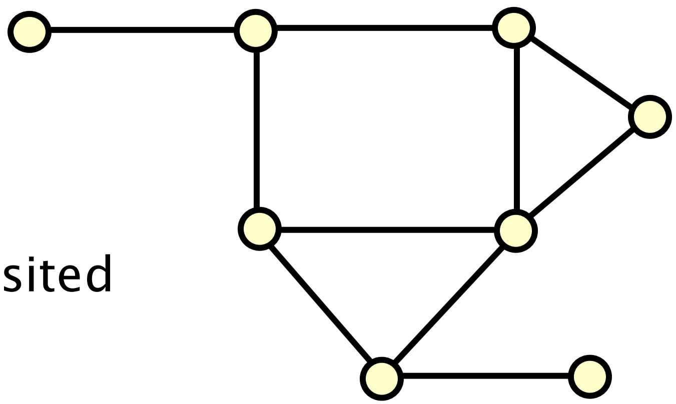


denotes vertex has been visited

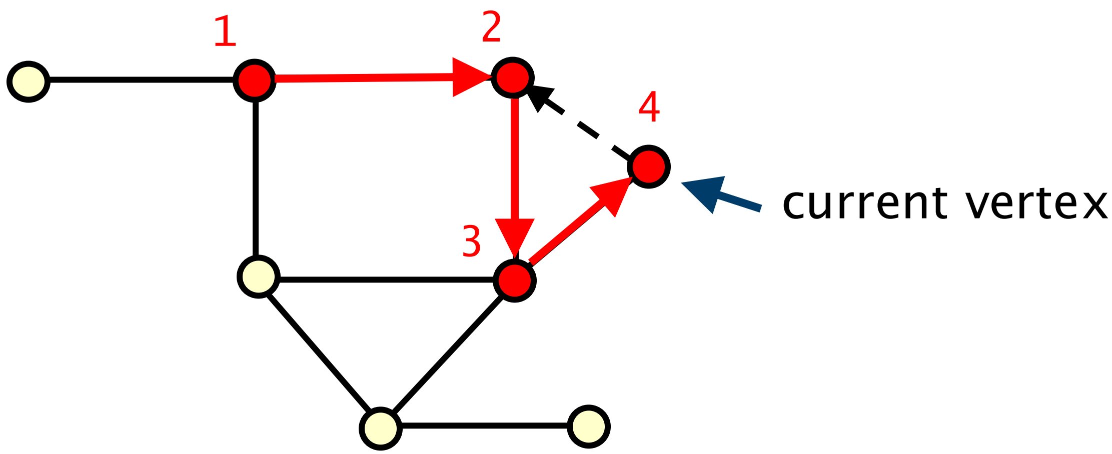


Depth first traversal – Example

Undirected graph G

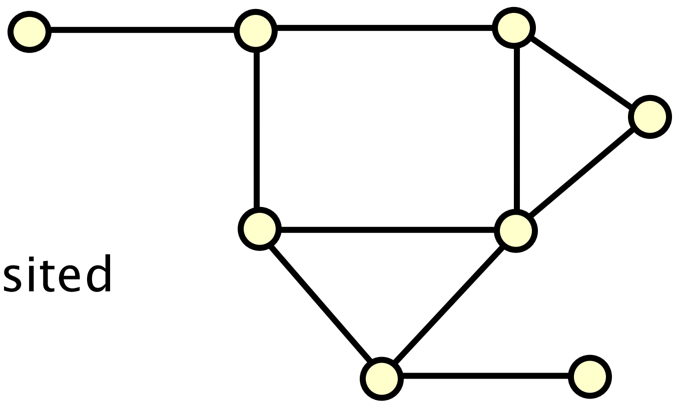


denotes vertex has been visited

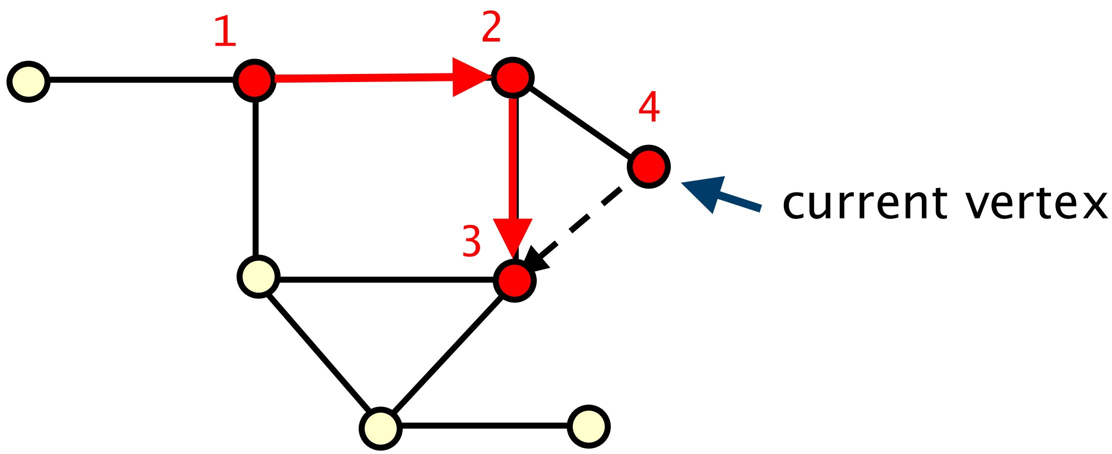


Depth first traversal – Example

Undirected graph G

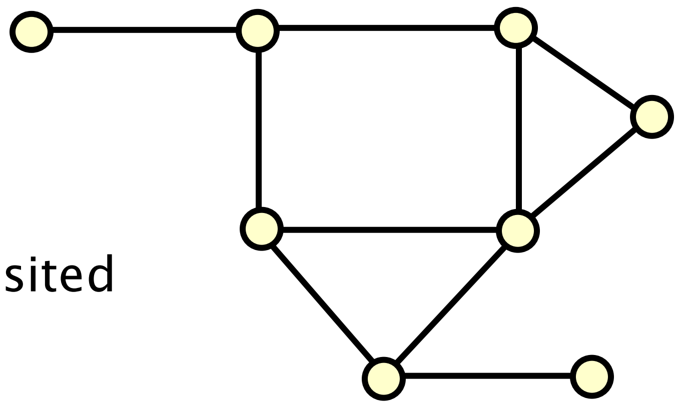


denotes vertex has been visited



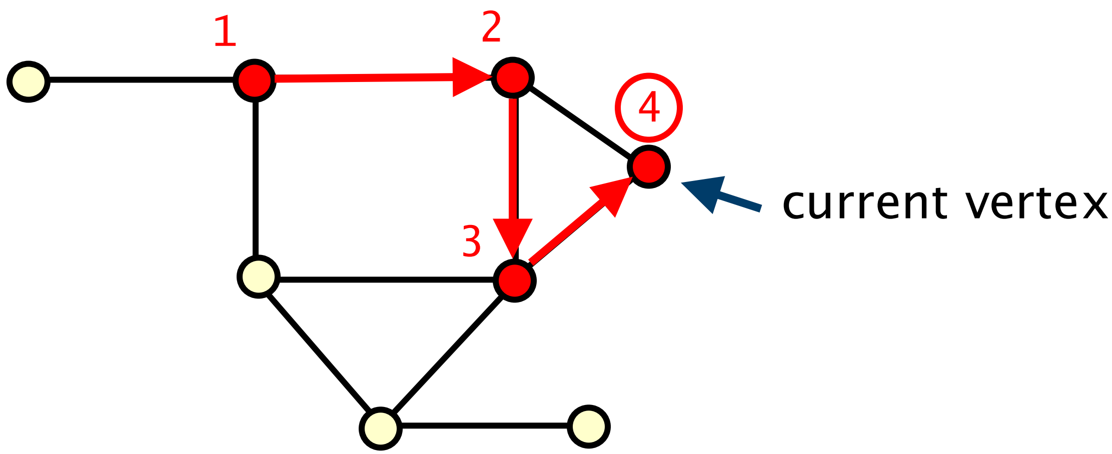
Depth first traversal – Example

Undirected graph **G**



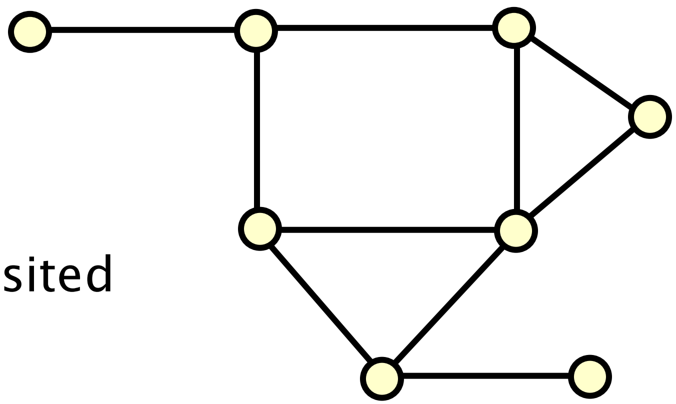
denotes vertex has been visited

i means all adjacent vertices of i have been considered



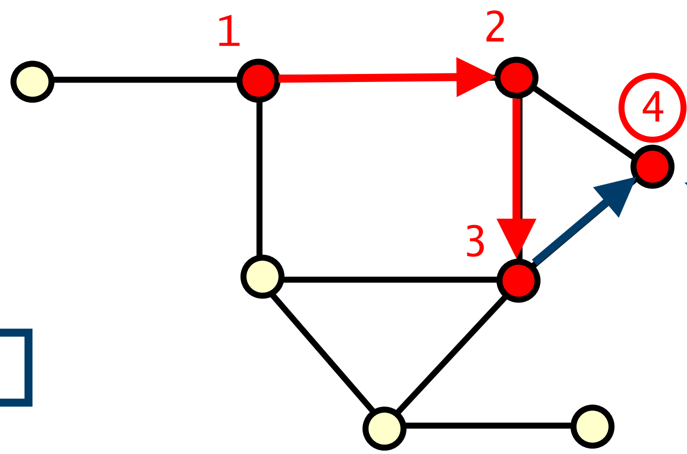
Depth first traversal – Example

Undirected graph G



denotes vertex has been visited

i means all adjacent vertices of i have been considered

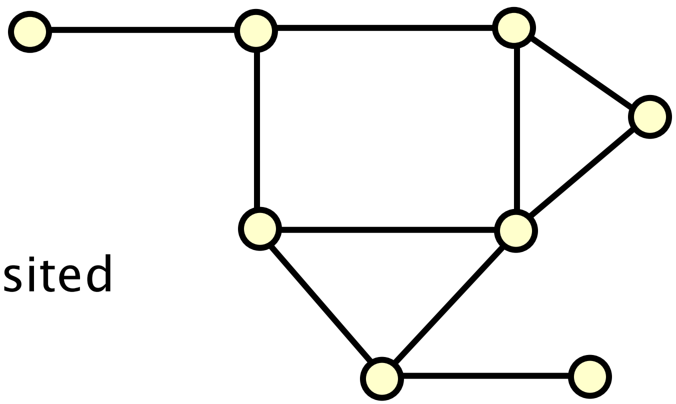


current vertex

backtrack

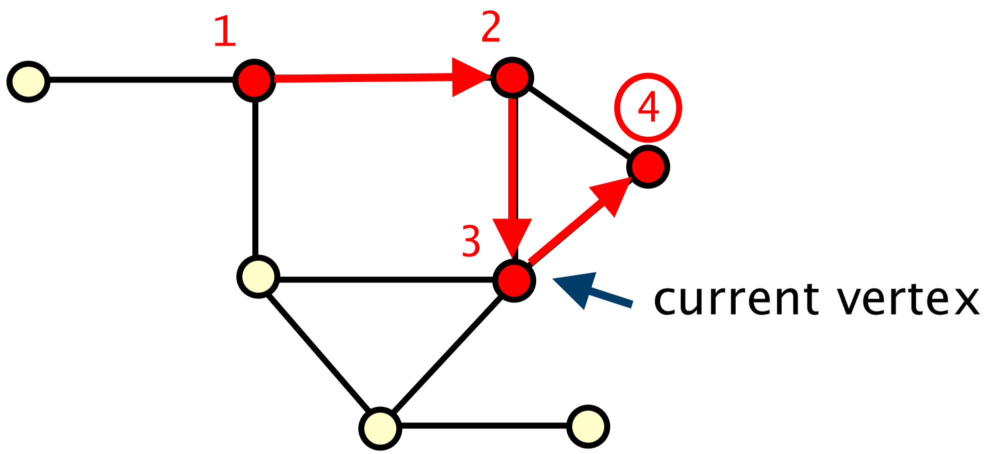
Depth first traversal – Example

Undirected graph G



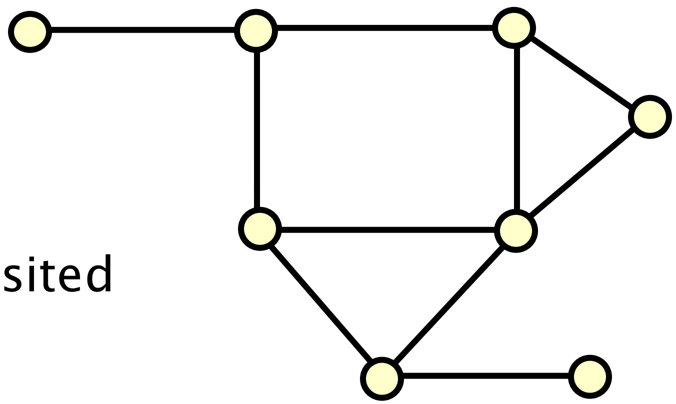
denotes vertex has been visited

means all adjacent vertices of i have been considered



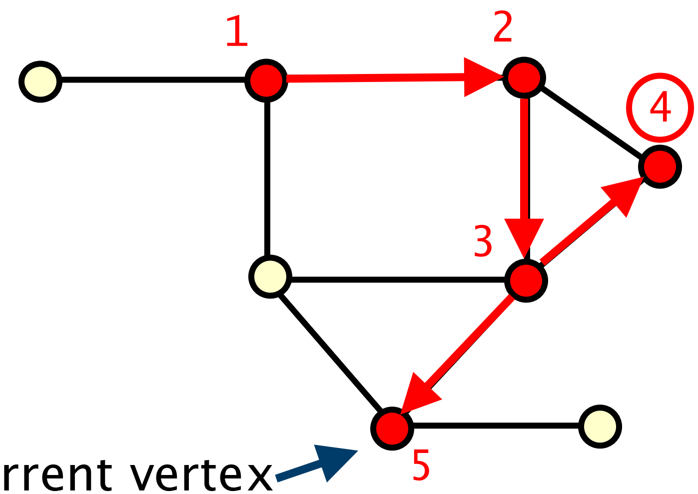
Depth first traversal – Example

Undirected graph G



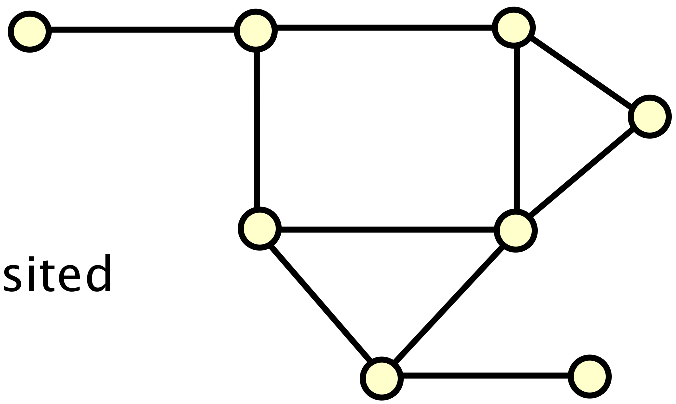
denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



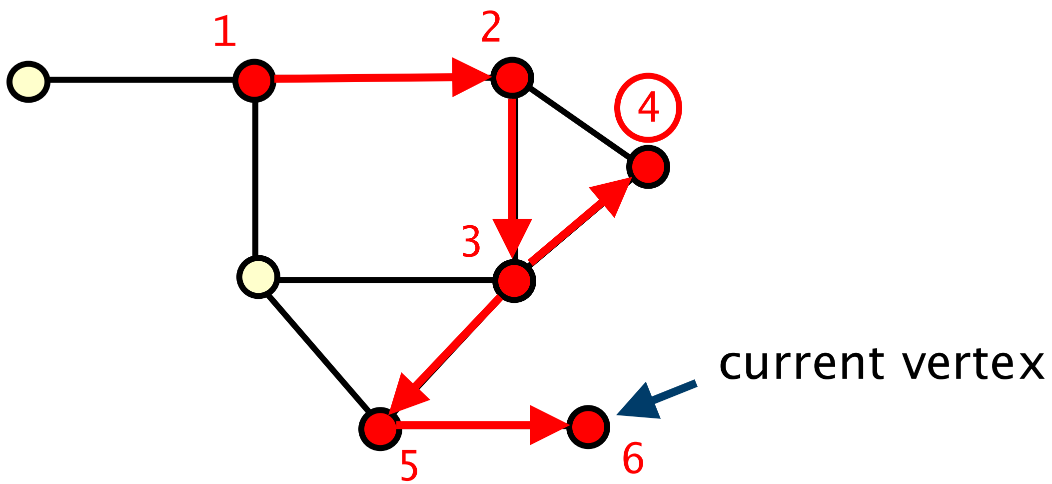
Depth first traversal – Example

Undirected graph G



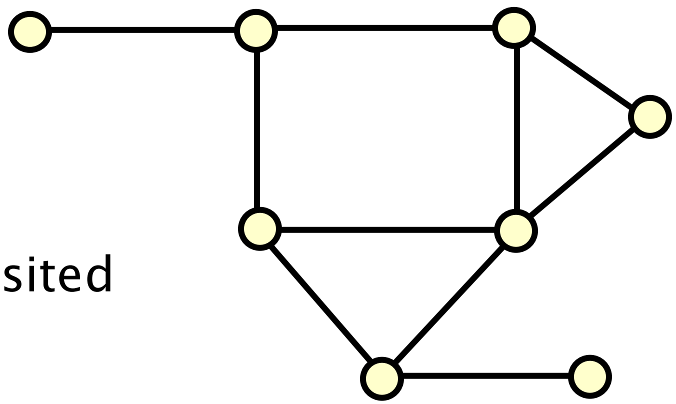
● denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



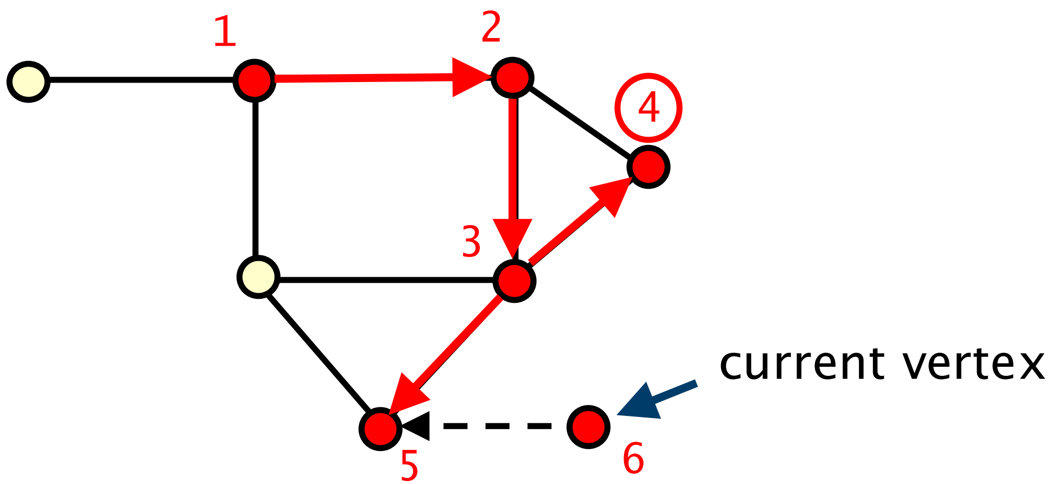
Depth first traversal – Example

Undirected graph G



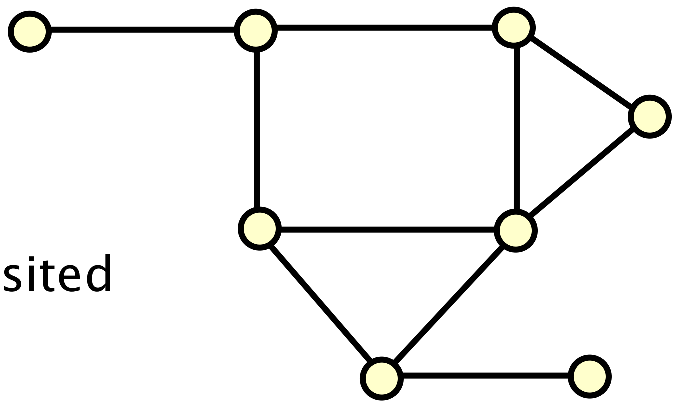
denotes vertex has been visited

i means all adjacent vertices of i have been considered



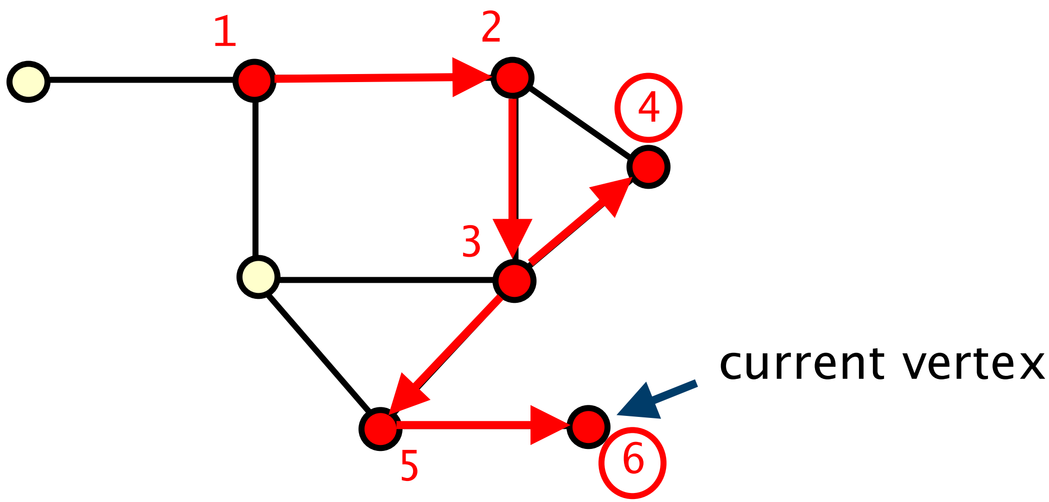
Depth first traversal – Example

Undirected graph G



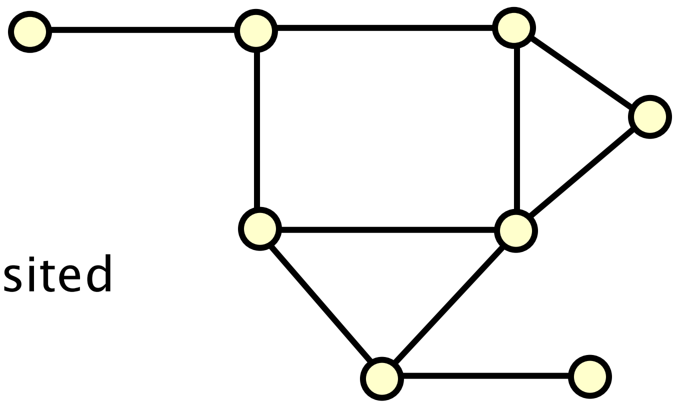
denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



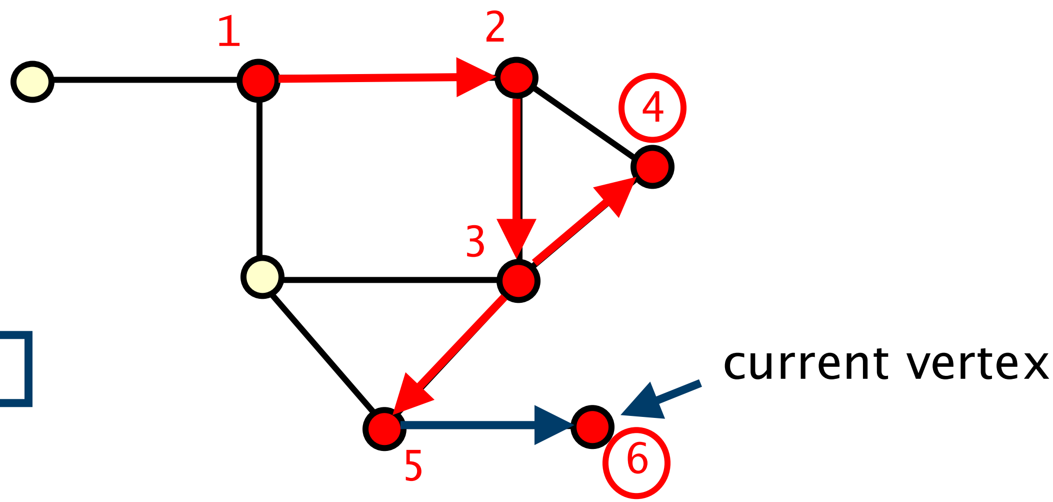
Depth first traversal – Example

Undirected graph G



denotes vertex has been visited

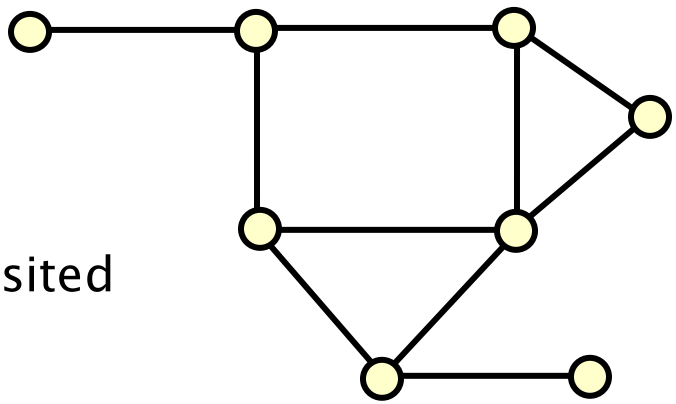
i means all adjacent vertices of i have been considered



backtrack

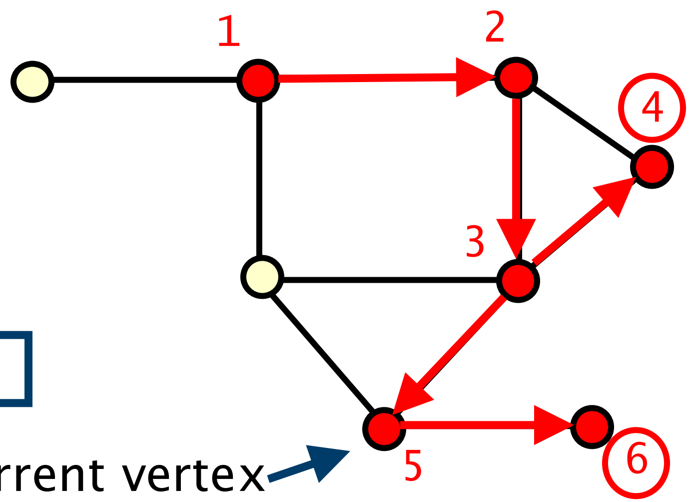
Depth first traversal – Example

Undirected graph G



● denotes vertex has been visited

(i) means all adjacent vertices of i have been considered

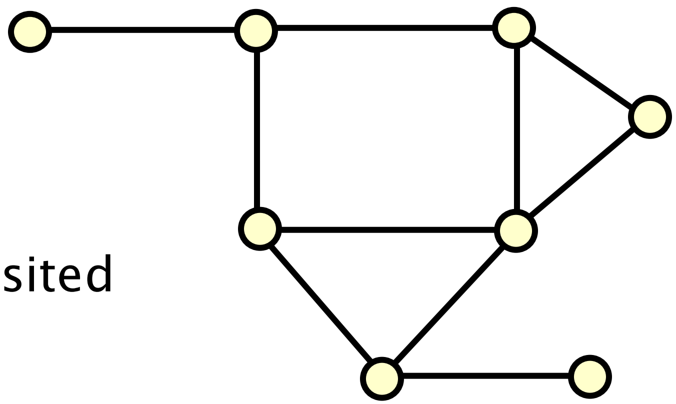


backtrack

current vertex

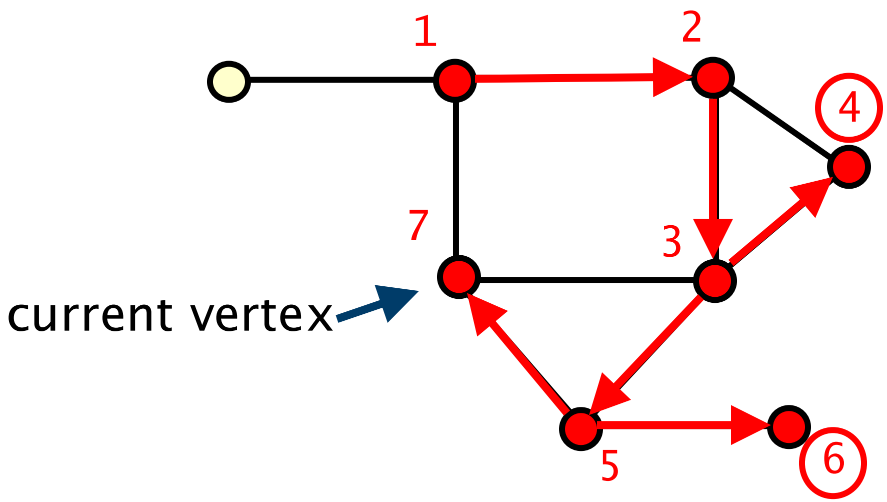
Depth first traversal – Example

Undirected graph G



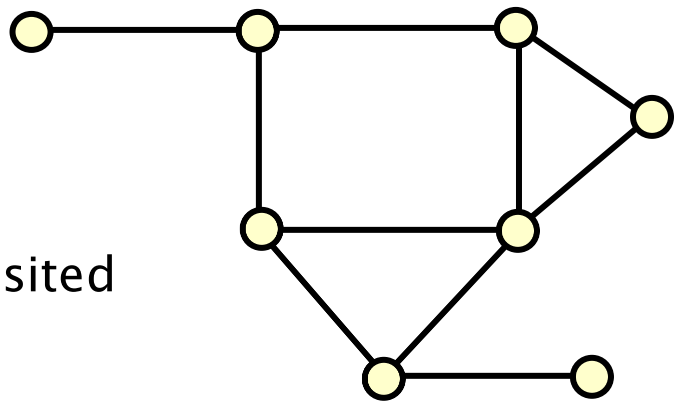
denotes vertex has been visited

i means all adjacent vertices of i have been considered



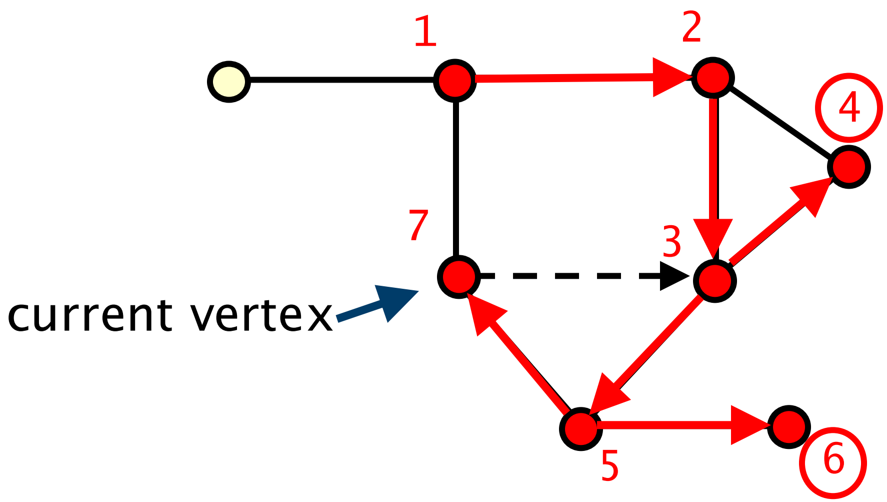
Depth first traversal – Example

Undirected graph G



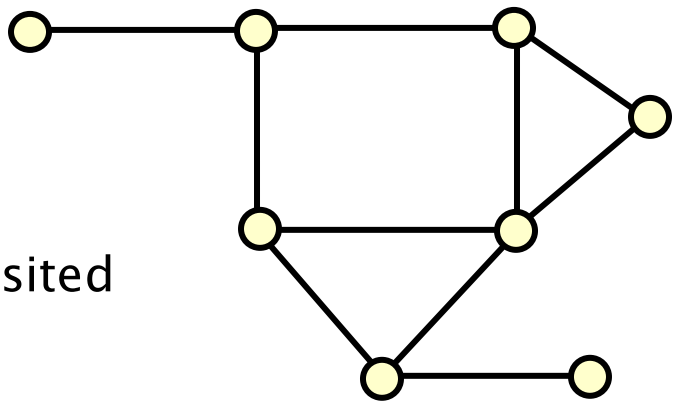
denotes vertex has been visited

i means all adjacent vertices of i have been considered



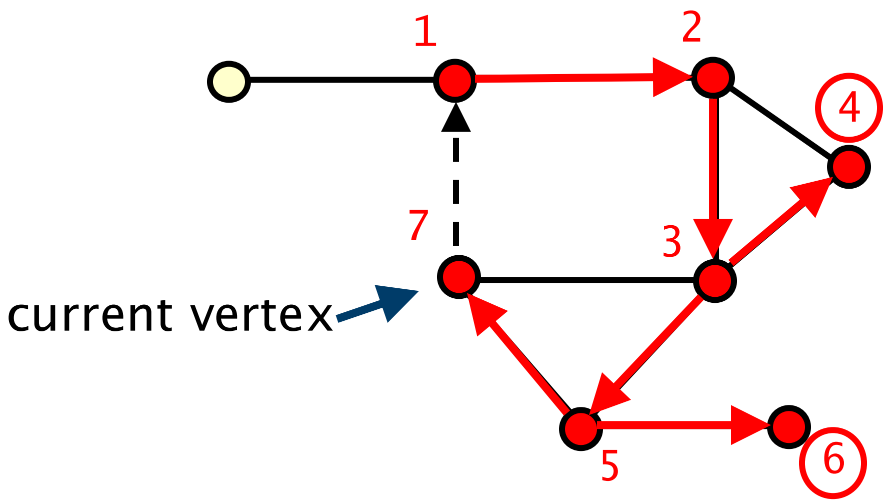
Depth first traversal – Example

Undirected graph **G**



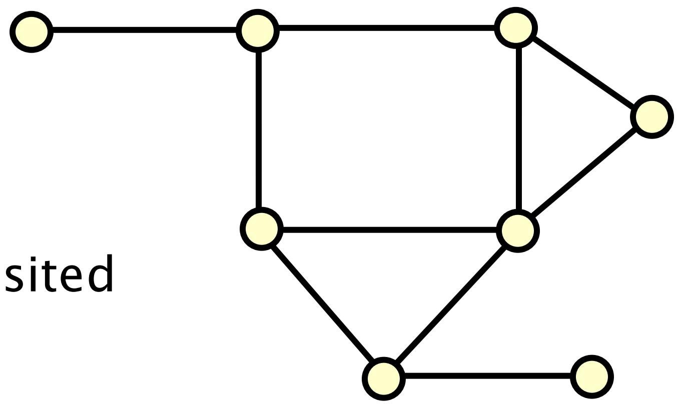
● denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



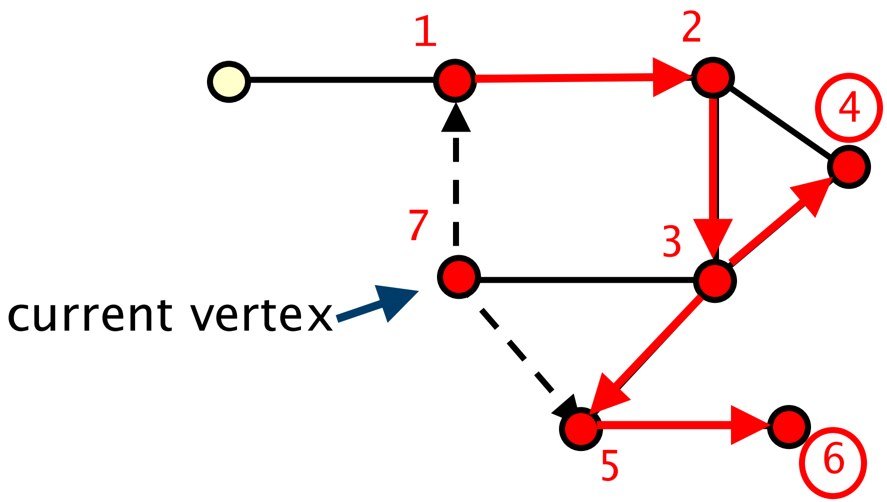
Depth first traversal – Example

Undirected graph **G**



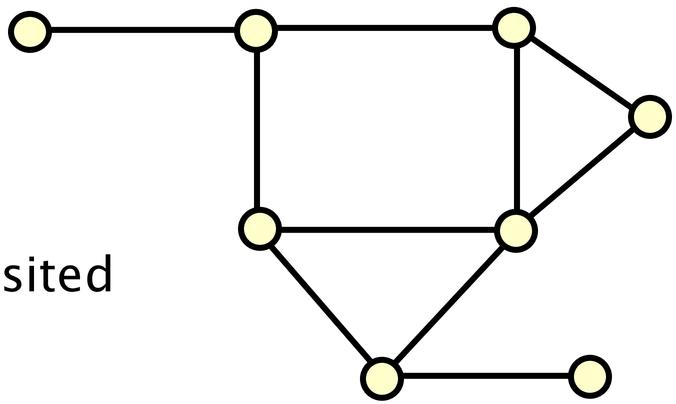
denotes vertex has been visited

i means all adjacent vertices of i have been considered



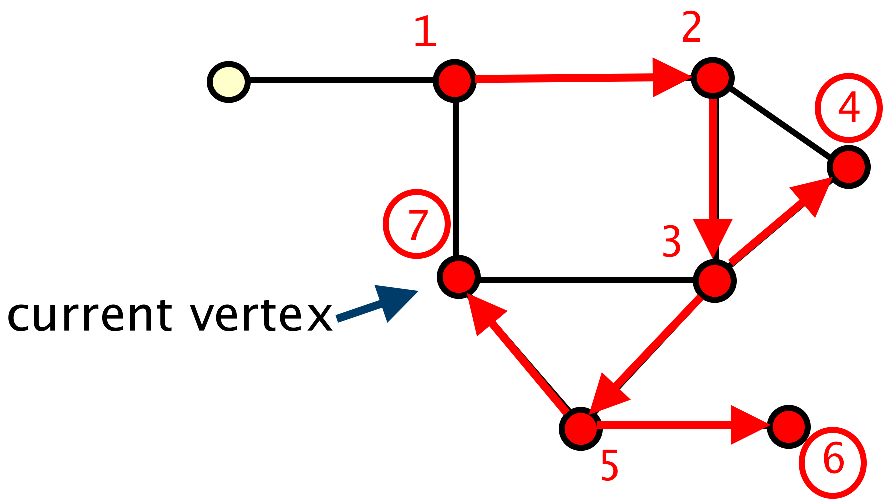
Depth first traversal – Example

Undirected graph G



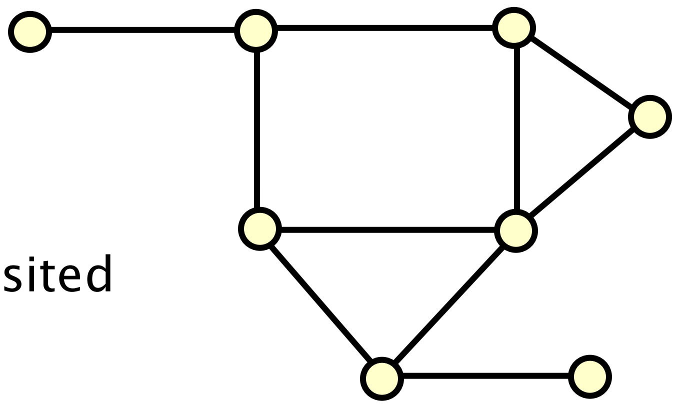
denotes vertex has been visited

i means all adjacent vertices of i have been considered



Depth first traversal – Example

Undirected graph G

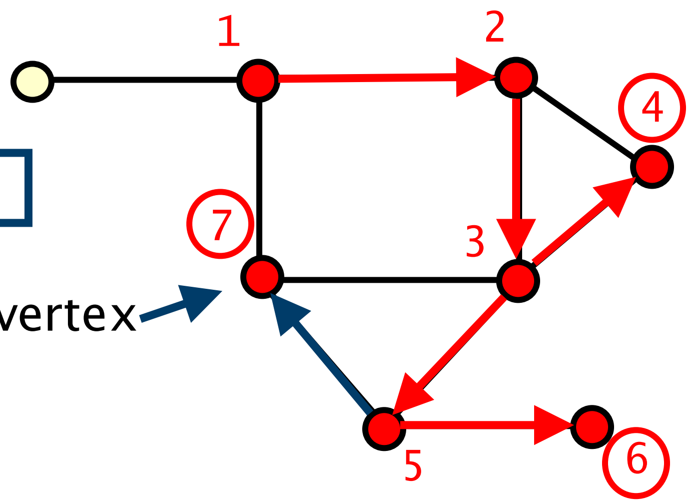


denotes vertex has been visited

i means all adjacent vertices of i have been considered

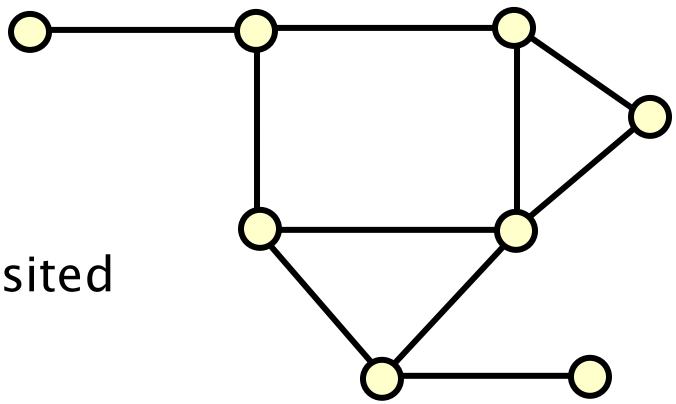
backtrack

current vertex



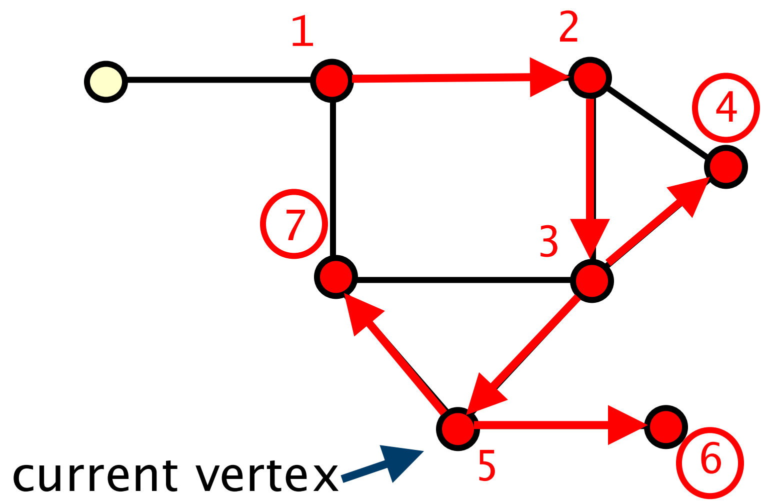
Depth first traversal – Example

Undirected graph G



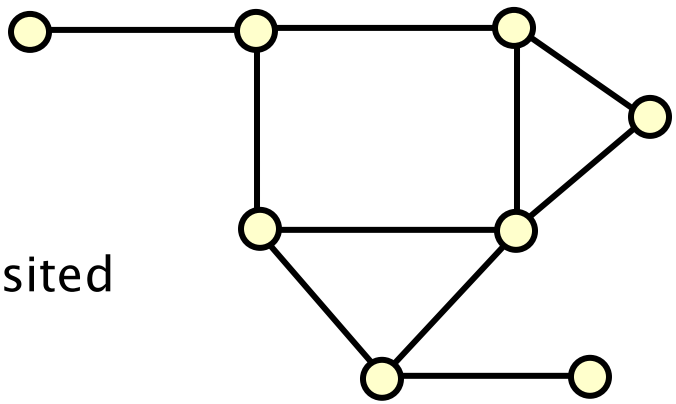
denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



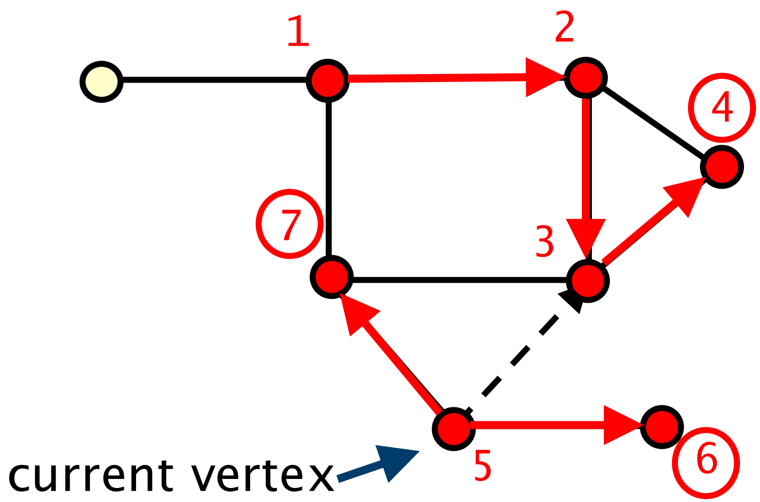
Depth first traversal – Example

Undirected graph G



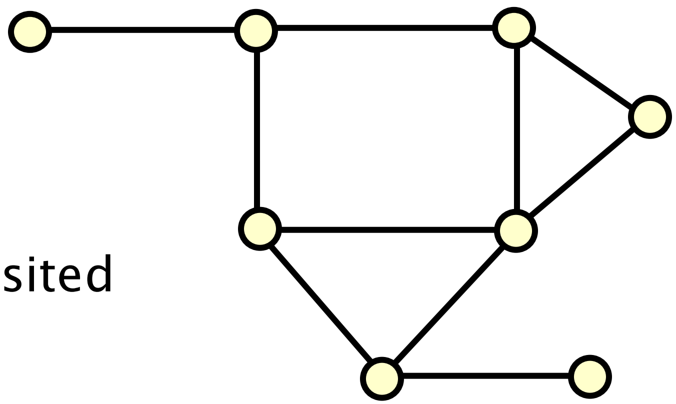
denotes vertex has been visited

i means all adjacent vertices of i have been considered



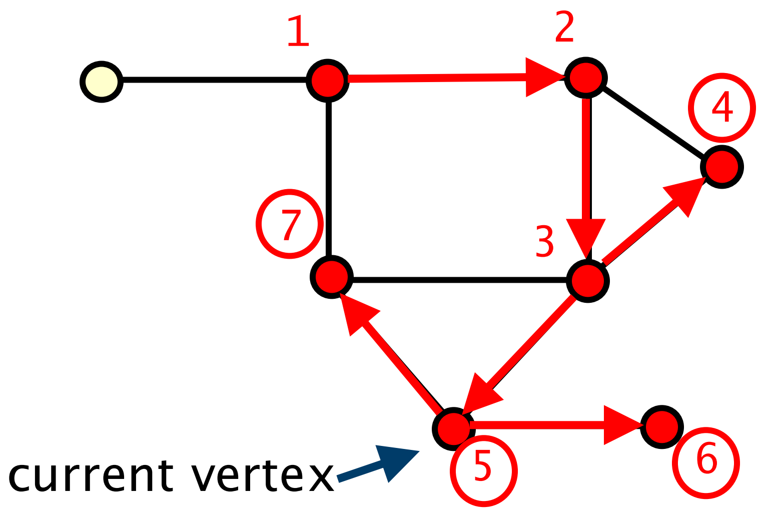
Depth first traversal – Example

Undirected graph G



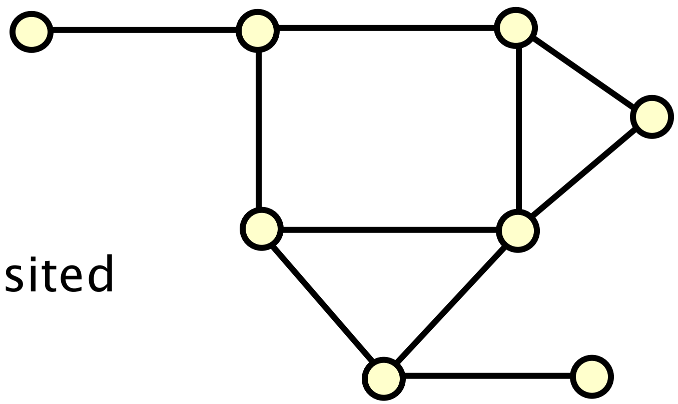
denotes vertex has been visited

i means all adjacent vertices of i have been considered



Depth first traversal – Example

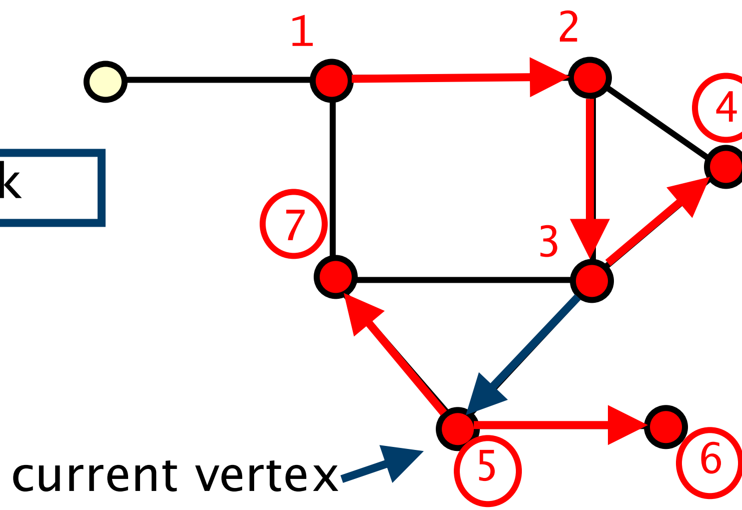
Undirected graph G



denotes vertex has been visited

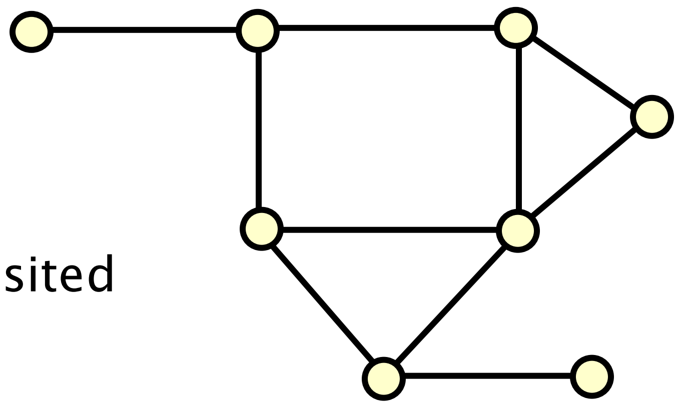
(i) means all adjacent vertices of i have been considered

backtrack



Depth first traversal – Example

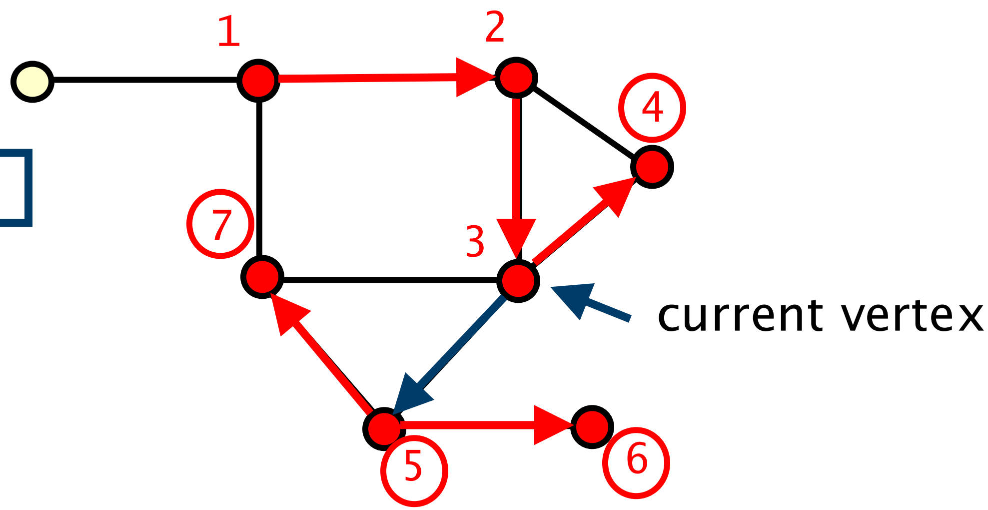
Undirected graph **G**



● denotes vertex has been visited

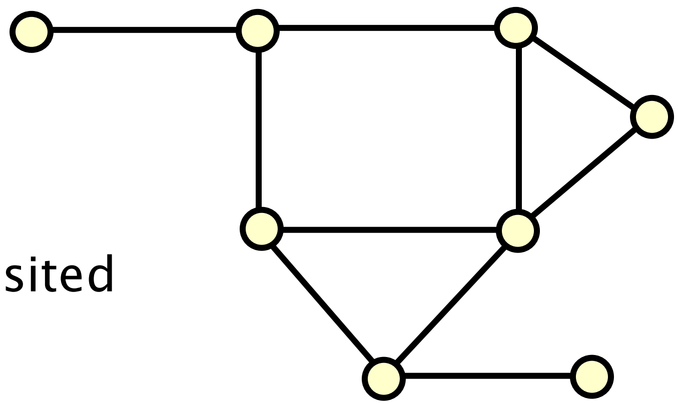
(i) means all adjacent vertices of i have been considered

backtrack



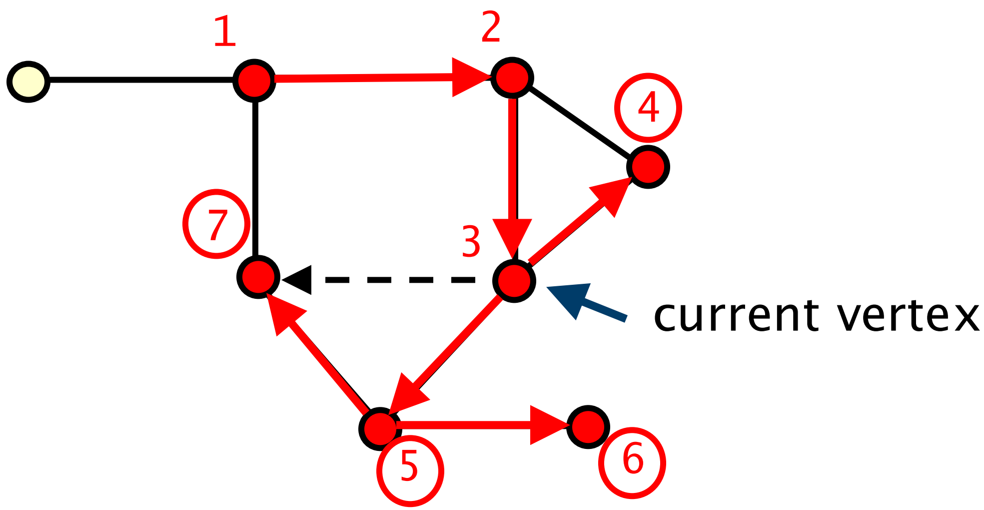
Depth first traversal – Example

Undirected graph G



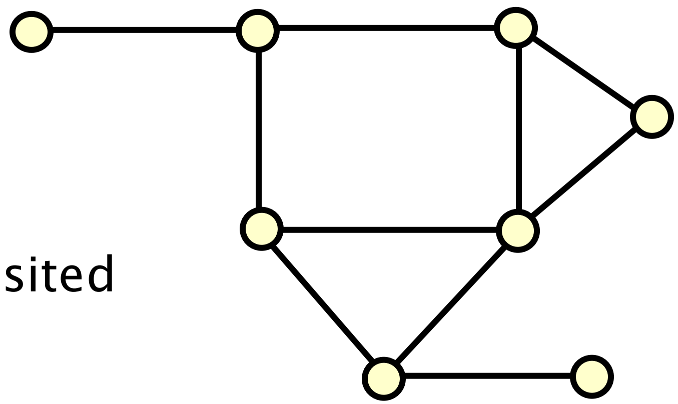
● denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



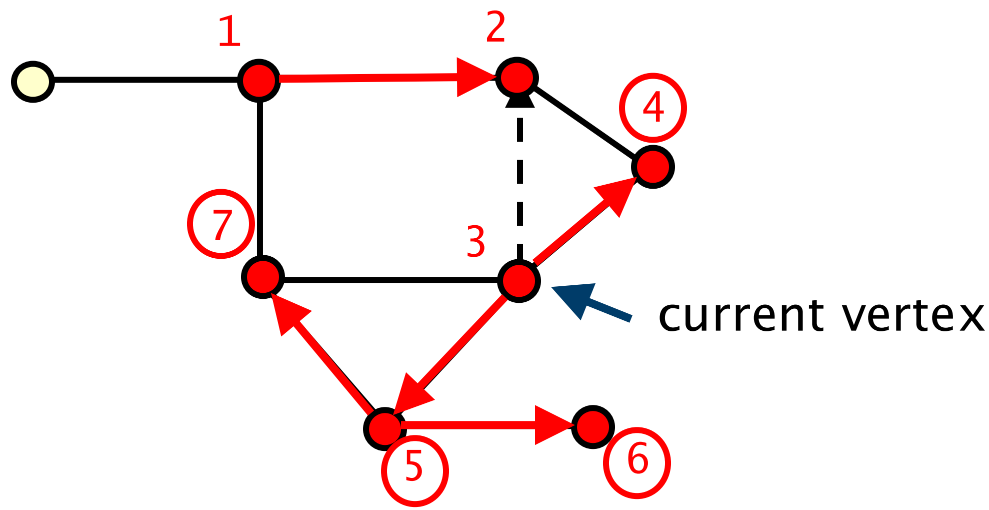
Depth first traversal – Example

Undirected graph G



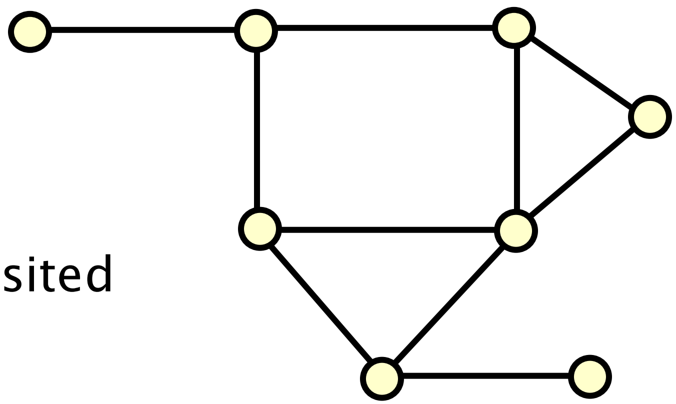
denotes vertex has been visited

i means all adjacent vertices of i have been considered



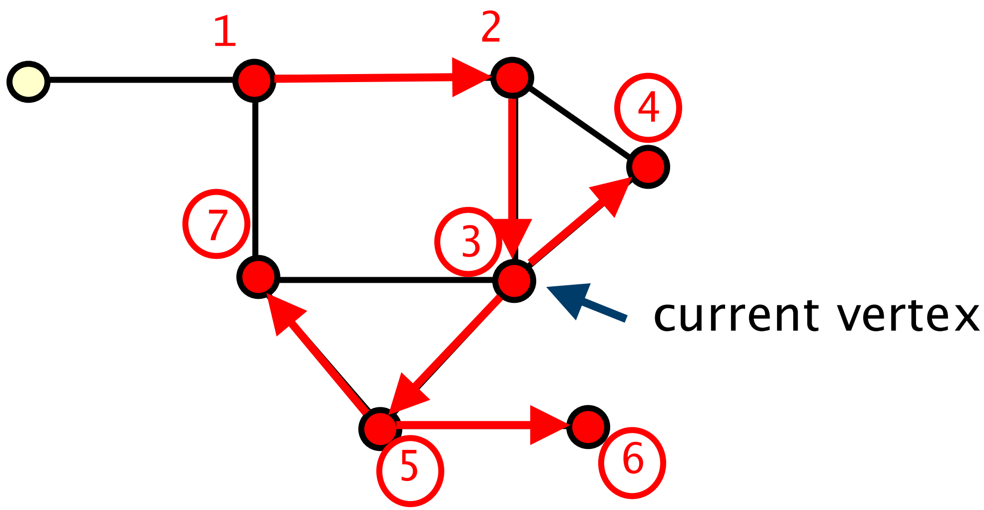
Depth first traversal – Example

Undirected graph G



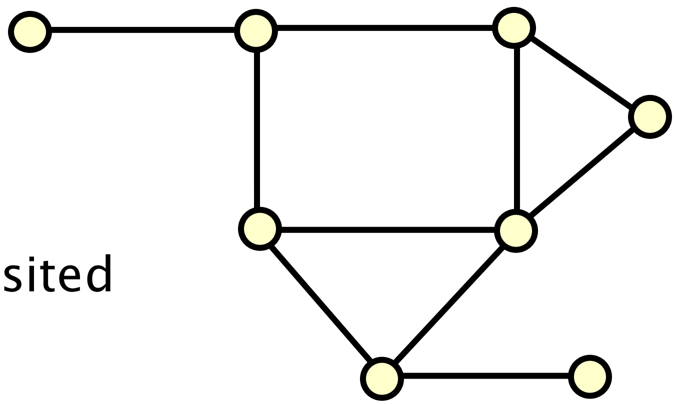
denotes vertex has been visited

(i) means all adjacent vertices of i have been considered



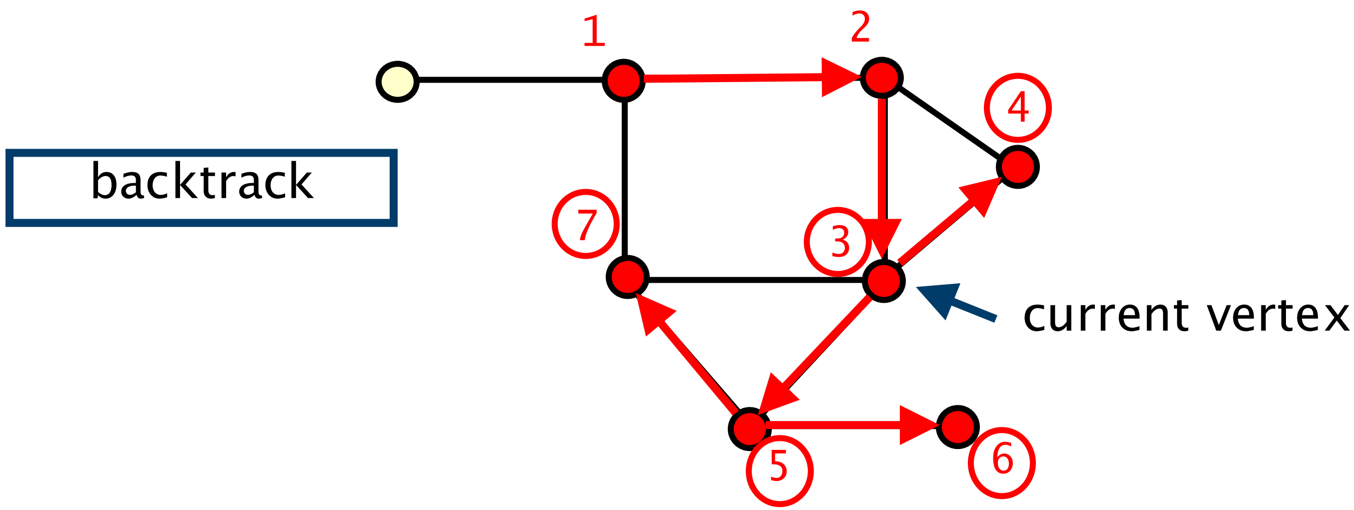
Depth first traversal – Example

Undirected graph G



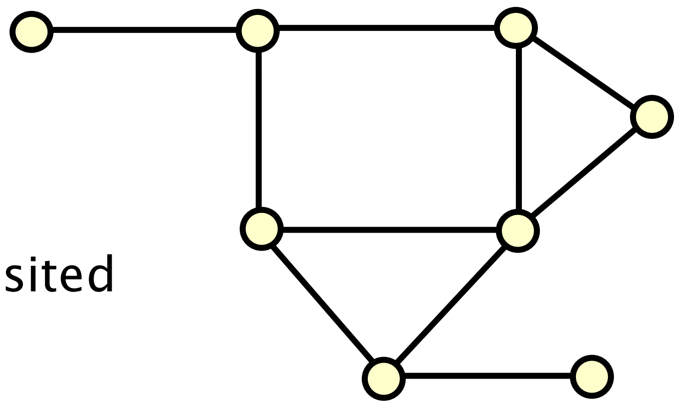
denotes vertex has been visited

i means all adjacent vertices of i have been considered



Depth first traversal – Example

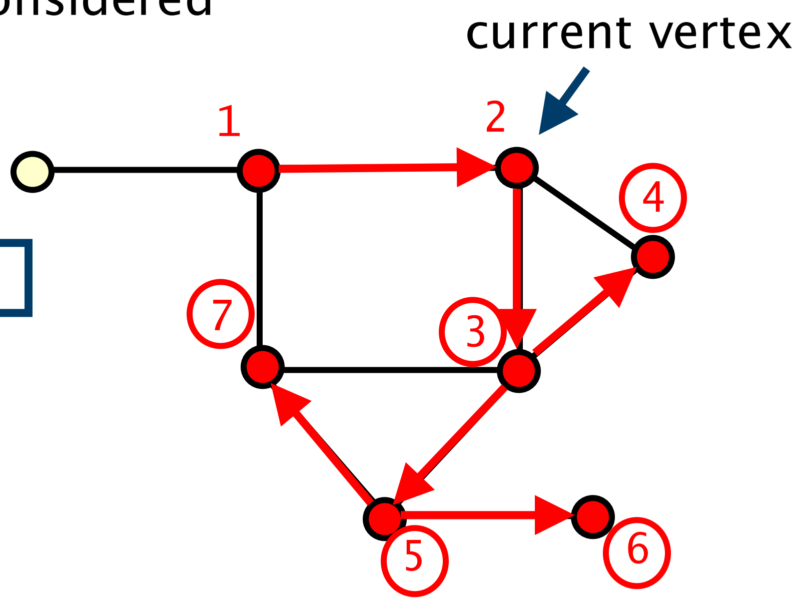
Undirected graph G



denotes vertex has been visited

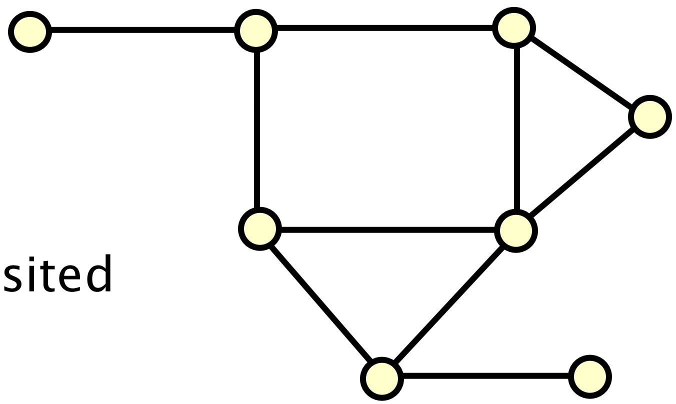
i means all adjacent vertices of i have been considered

backtrack



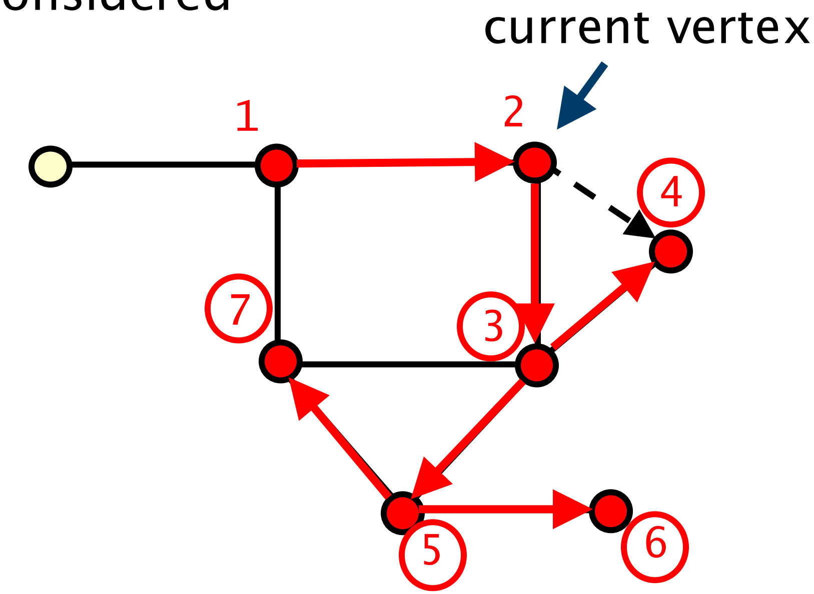
Depth first traversal – Example

Undirected graph G



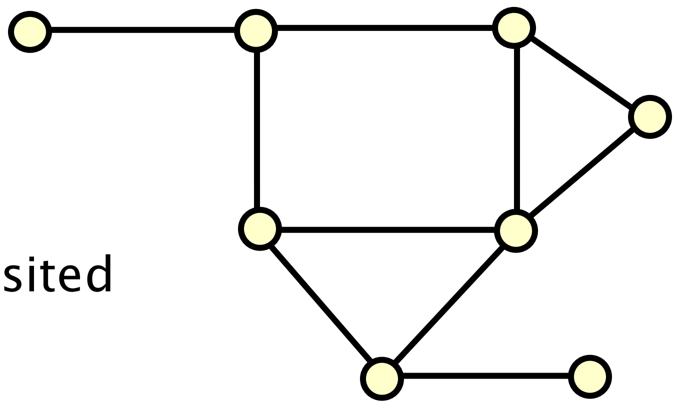
denotes vertex has been visited

means all adjacent vertices of i have been considered



Depth first traversal – Example

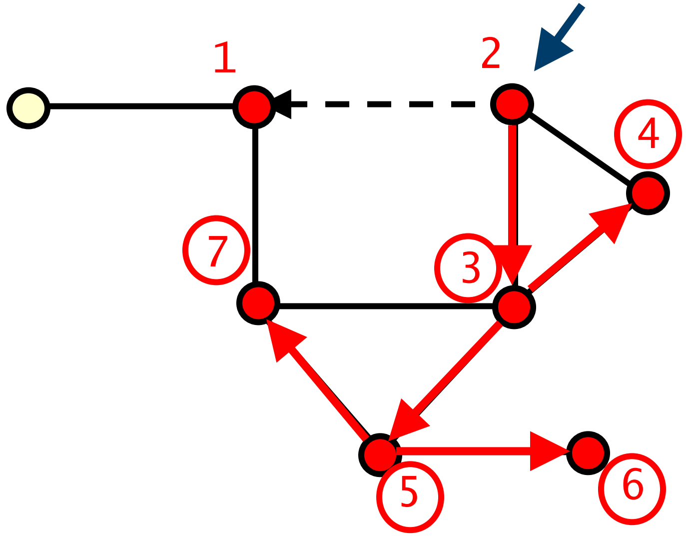
Undirected graph G



denotes vertex has been visited

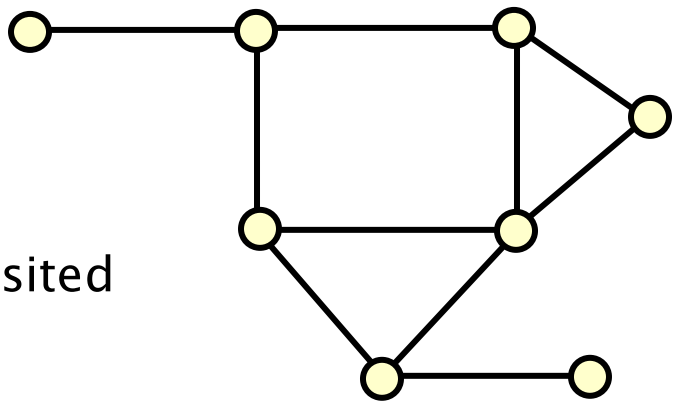
i means all adjacent vertices of i have been considered

current vertex



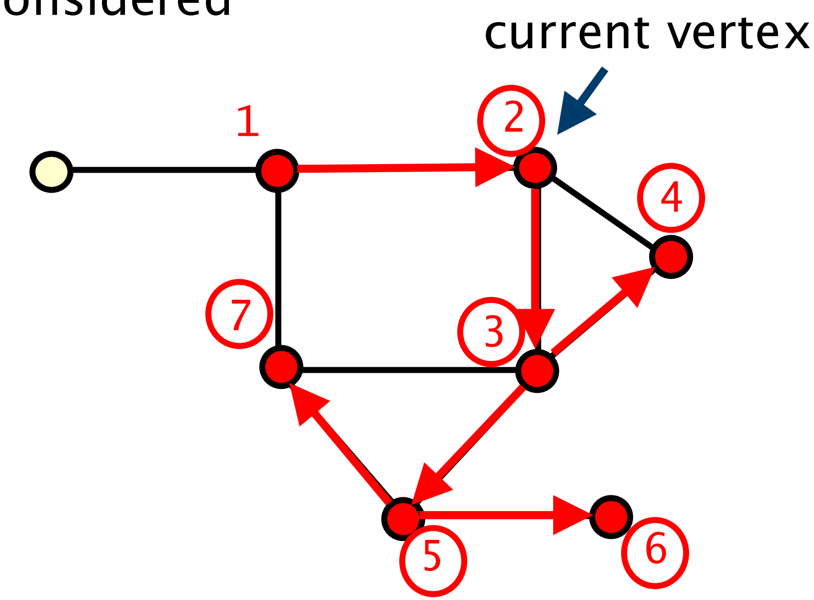
Depth first traversal – Example

Undirected graph G



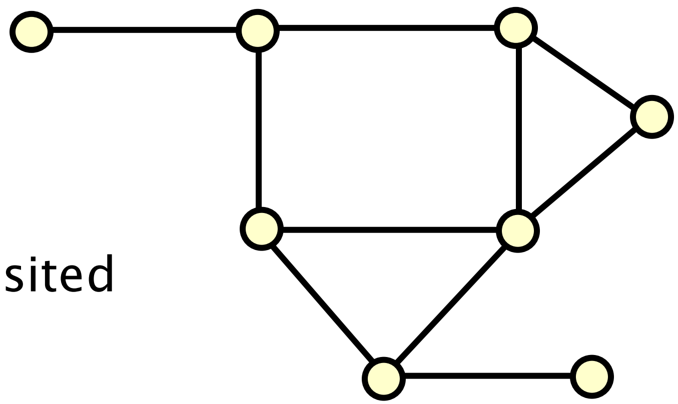
denotes vertex has been visited

i means all adjacent vertices of i have been considered



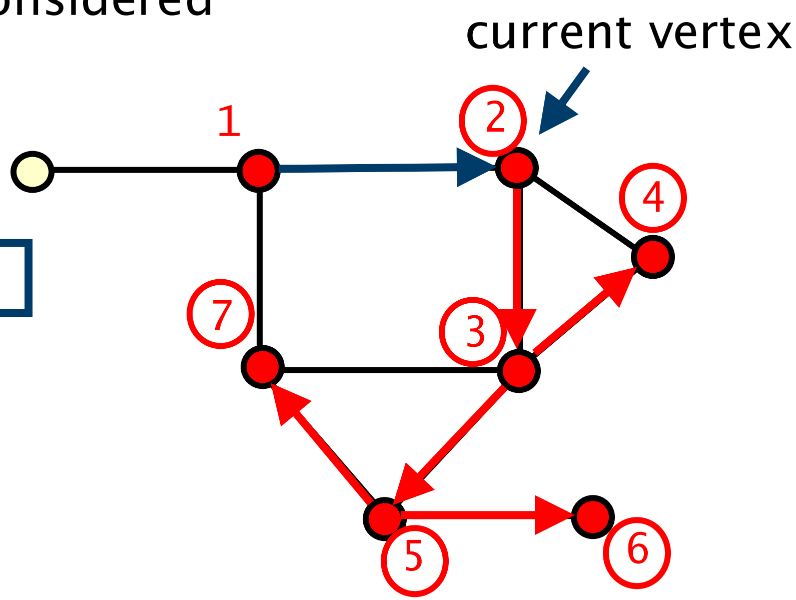
Depth first traversal – Example

Undirected graph G



● denotes vertex has been visited

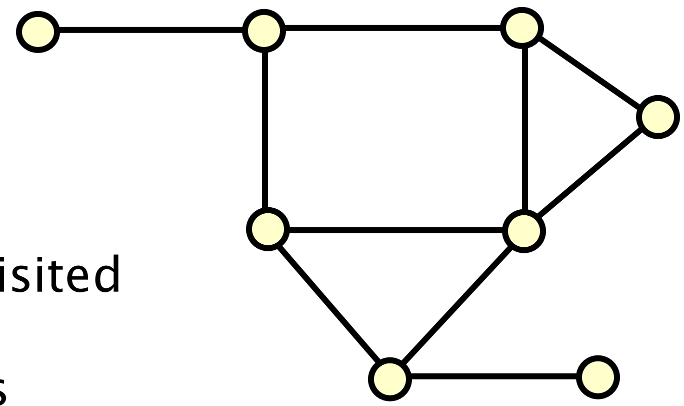
(i) means all adjacent vertices of i have been considered



backtrack

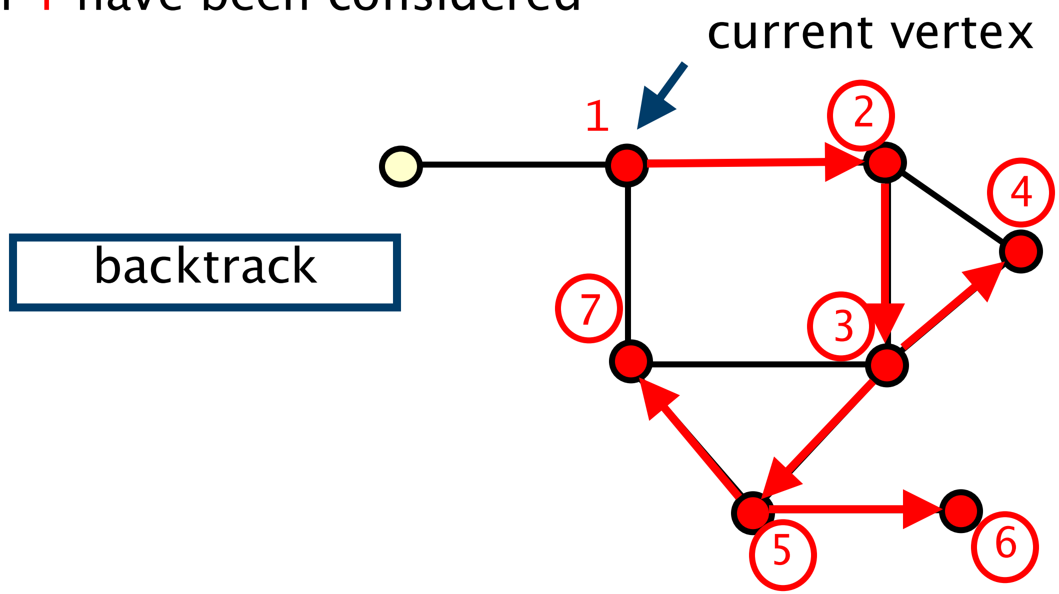
Depth first traversal – Example

Undirected graph G



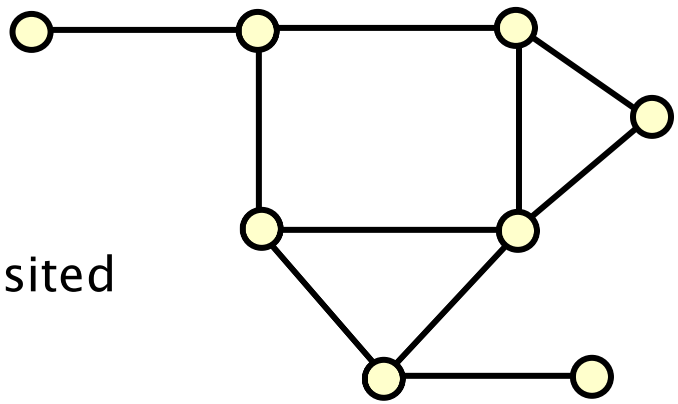
denotes vertex has been visited

means all adjacent vertices of i have been considered



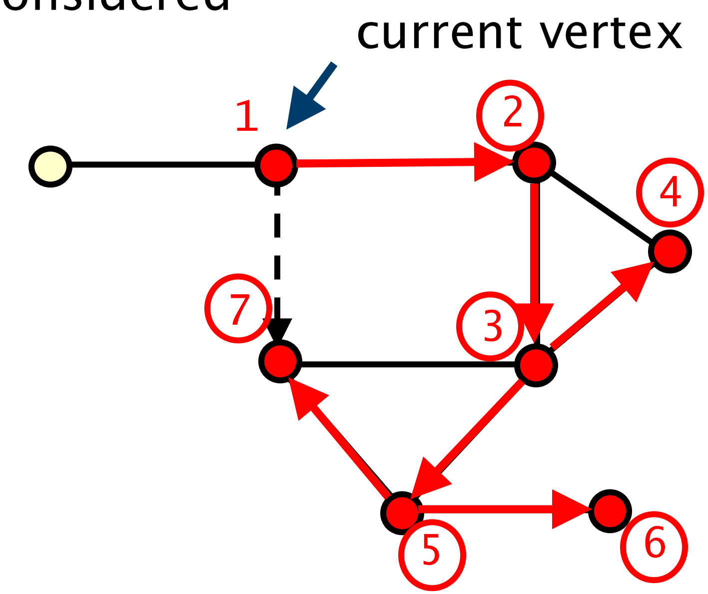
Depth first traversal – Example

Undirected graph G



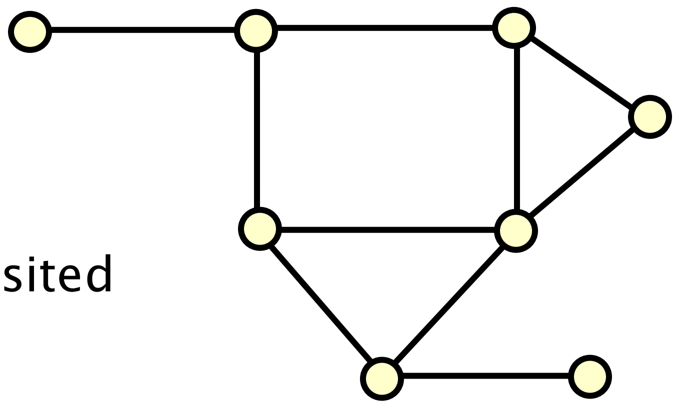
denotes vertex has been visited

i means all adjacent vertices of i have been considered



Depth first traversal – Example

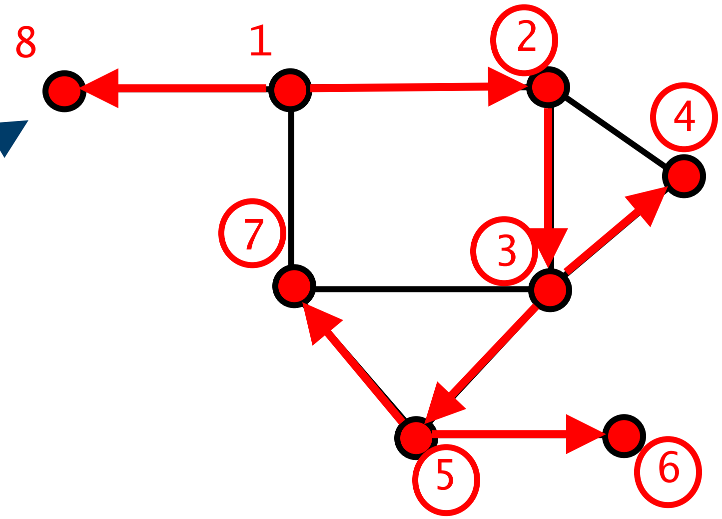
Undirected graph G



denotes vertex has been visited

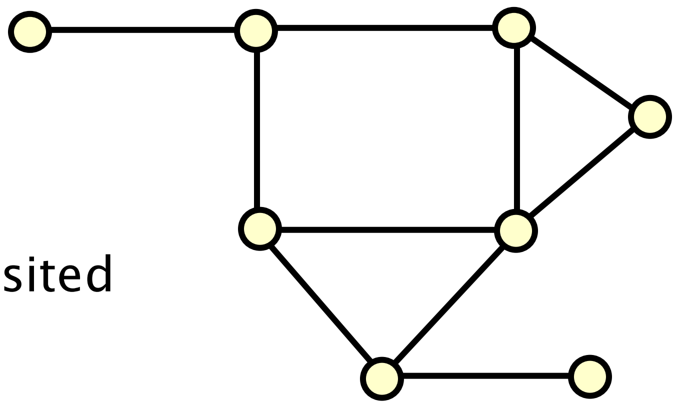
(i) means all adjacent vertices of i have been considered

current vertex



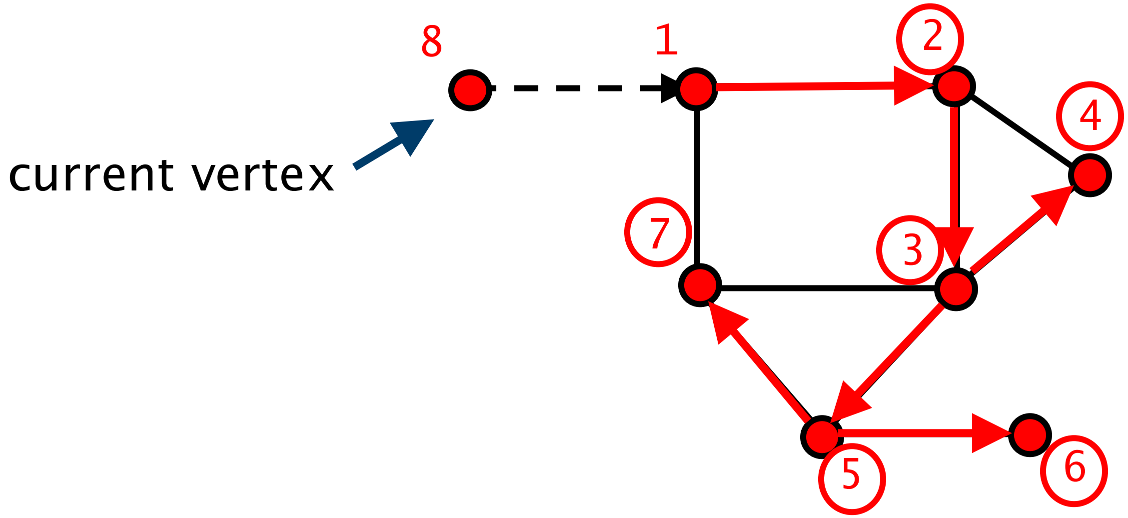
Depth first traversal – Example

Undirected graph G



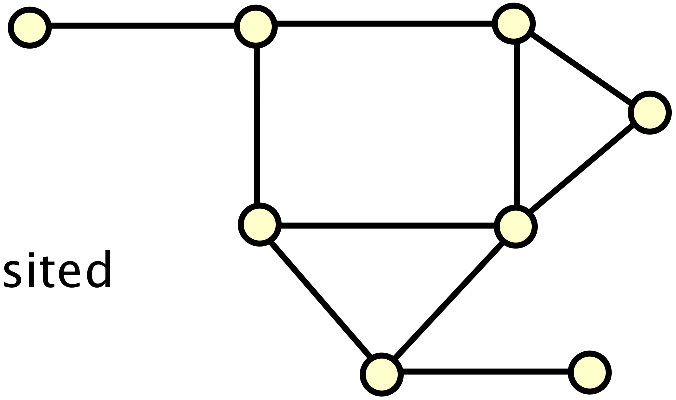
denotes vertex has been visited

means all adjacent vertices of i have been considered



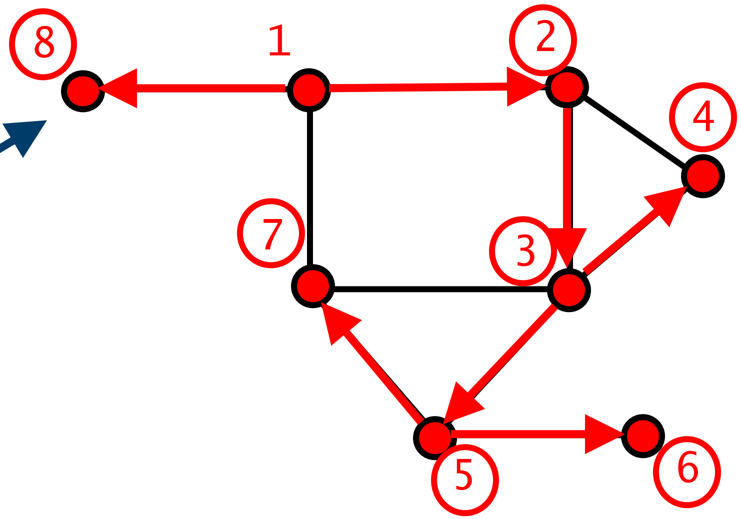
Depth first traversal – Example

Undirected graph G



denotes vertex has been visited

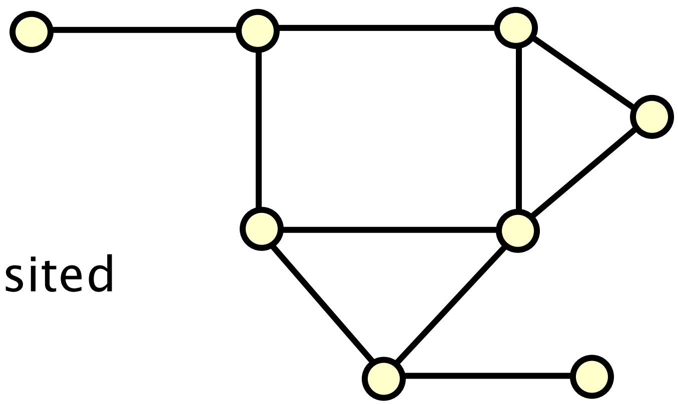
i means all adjacent vertices of i have been considered



current vertex

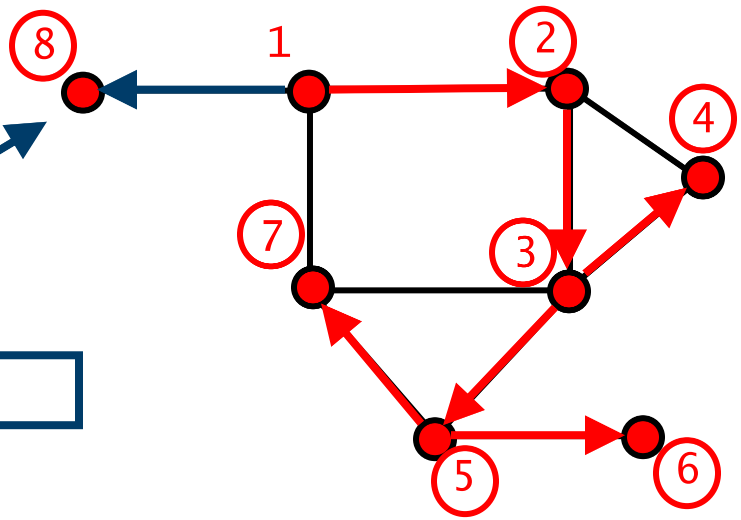
Depth first traversal – Example

Undirected graph G



denotes vertex has been visited

i means all adjacent vertices of i have been considered

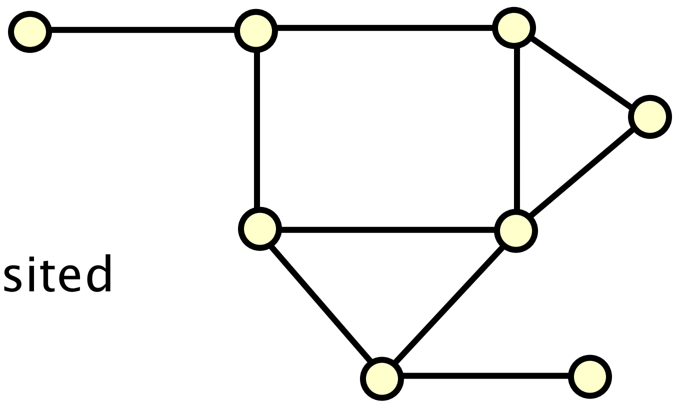


current vertex

backtrack

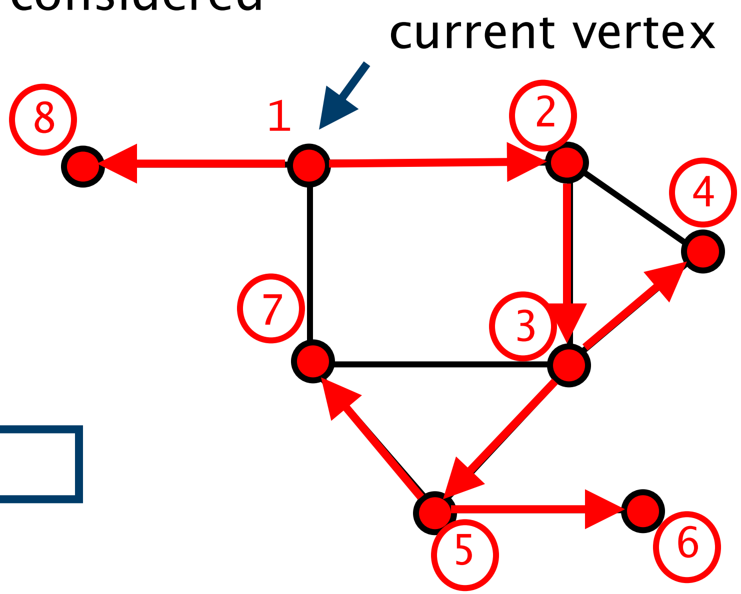
Depth first traversal – Example

Undirected graph G



denotes vertex has been visited

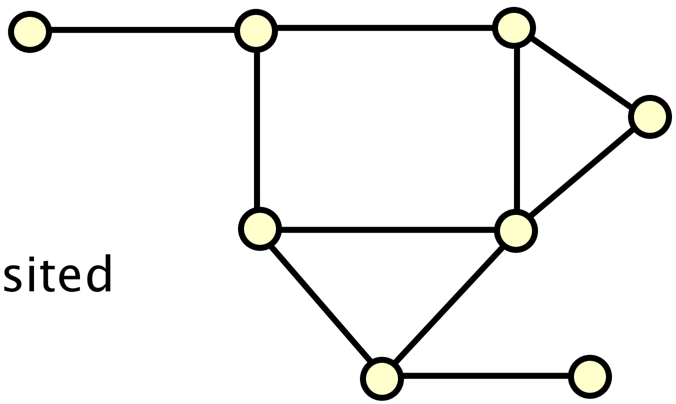
i means all adjacent vertices of i have been considered



backtrack

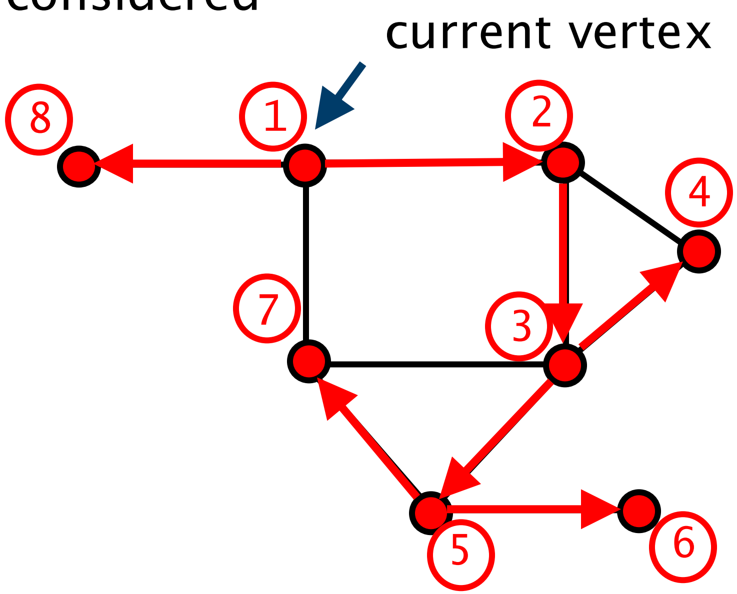
Depth first traversal – Example

Undirected graph G



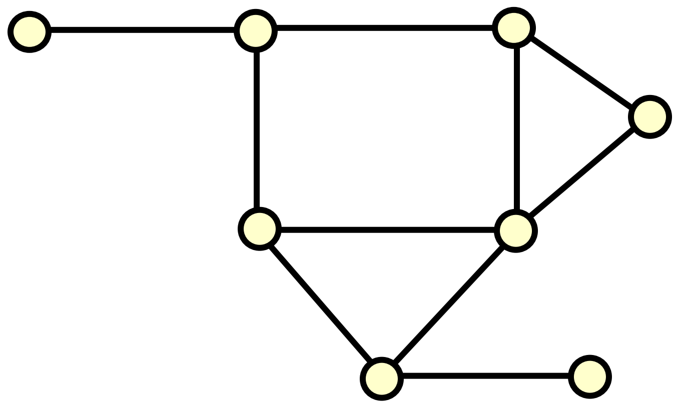
● denotes vertex has been visited

(i) means all adjacent vertices of i have been considered

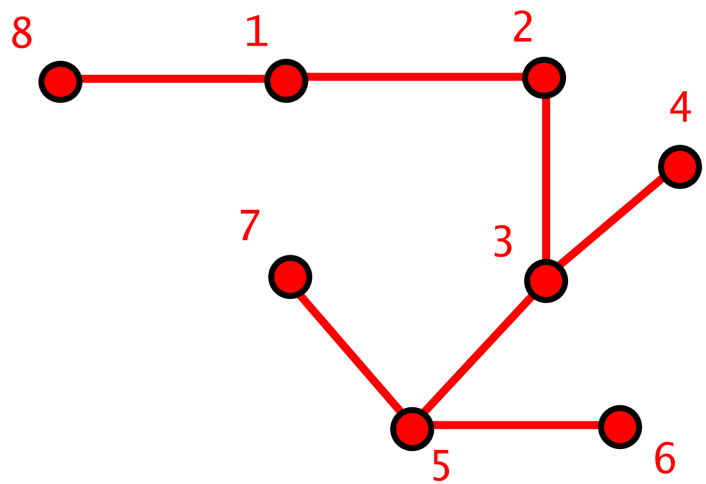


Depth first traversal – Example

Undirected graph **G**



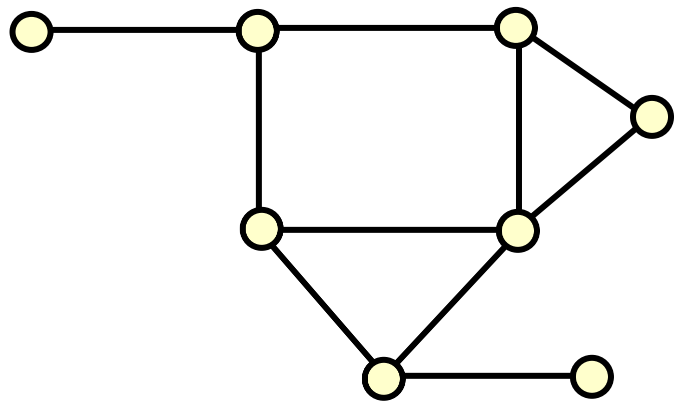
Depth first spanning tree of **G**



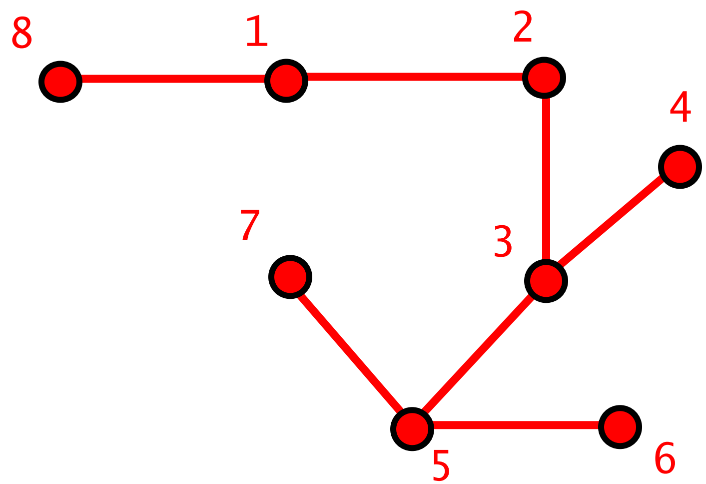
Is it unique?

Depth first traversal – Example

Undirected graph **G**



A depth first spanning tree of **G**



What if from 1 we went to 7 first?

Recall adjacency list implementation

Class: adjacency node

- represents an element of an adjacency list
- includes the index of the corresponding vertex

Class: vertex

- represents a single vertex of the graph
- includes linked list of adjacency nodes representing the adjacent vertices

Class: graph

- an array of vertices

Implementation – DFS – Add to vertex class

```
private boolean visited; // has vertex been visited in a traversal?

private int pred; // index of the predecessor vertex in a traversal

public boolean getVisited(){ // was this vertex visited?
  return visited;
}
public void setVisited(boolean b){ // on 1st encounter, set as true
  visited = b;
}
public int getPred(){ // for when we're backtracking
  return pred;
}
public void setPred(int i){ // when we find new vertex during search
  pred = i;
}
```

Implementation – DFS – Add to graph class

```
/** visit vertex v, with predecessor index p, during a dfs */
private void visit(Vertex v, int p){
  v.setVisited(true); // update as now visited
  v.setPred(p);  // set predecessor (indicates edge used to find vertex)
  LinkedList<AdjListNode> L = v.getAdjList(); // get adjacency list

  for (AdjListNode node : L){ // go through all adjacent vertices
    int i = node.getIndex(); // find index of current vertex in list
    if (!vertices[i].getVisited()) // if vertex has not been visited
      visit(vertices[i], v.getIndex()); // continue dfs search from it
      // setting the predecessor vertex index to the index of v
  }
}
/** carry out a depth first search/traversal of the graph */
public void dfs(){
  for (Vertex v : vertices) v.setVisited(false); // initialise
  for (Vertex v : vertices) if (!v.getVisited()) visit(v,-1);
  // if vertex is not yet visited, then start dfs on vertex w/ predecessor
  // -1 is used to indicate v was not found through an edge of the graph
}
```

Analysis – Depth first search

Each vertex is visited, and each element in the adjacency lists is processed, so overall **$O(n+m)$**

- where n is the number of vertices and m the number of edges

Can be adapted to the adjacency matrix representation

- but now $O(n^2)$ since look at every entry of the adjacency matrix

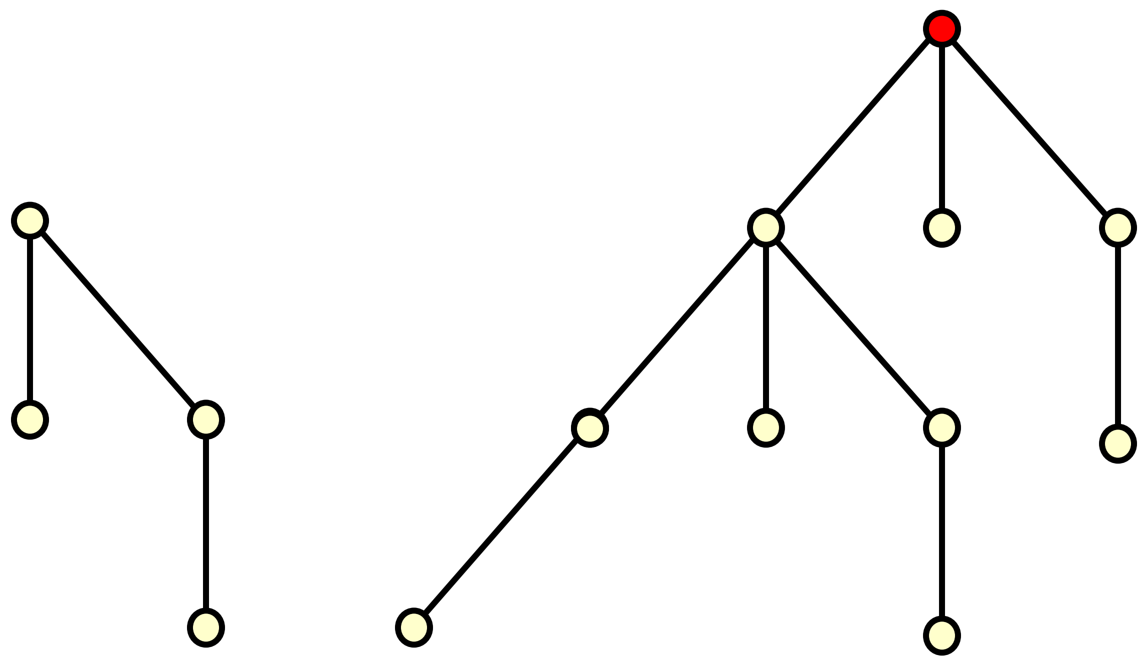
Some applications

- to determine if a given graph is connected
- to identify the connected components of a graph
- to determine if a given graph contains a cycle (see tutorial 5)
- to determine if a given graph is bipartite (see tutorial 5)
-

Recall – Depth first search/traversal (DFS)

From starting vertex

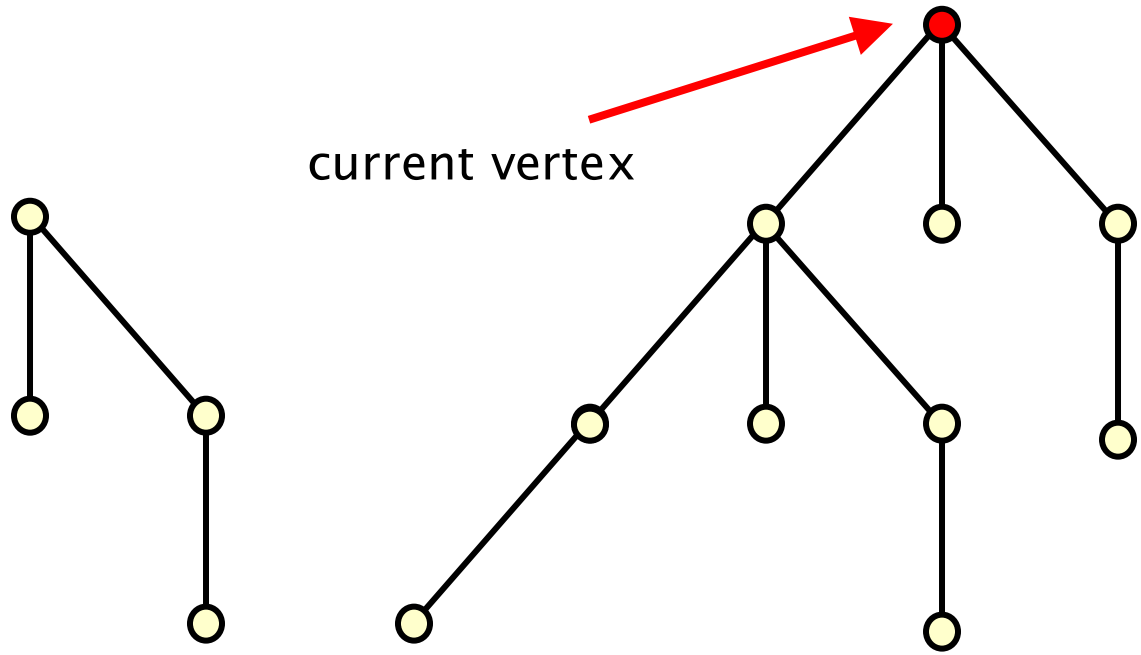
- follow a path of unvisited vertices until path can be extended no further



Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex

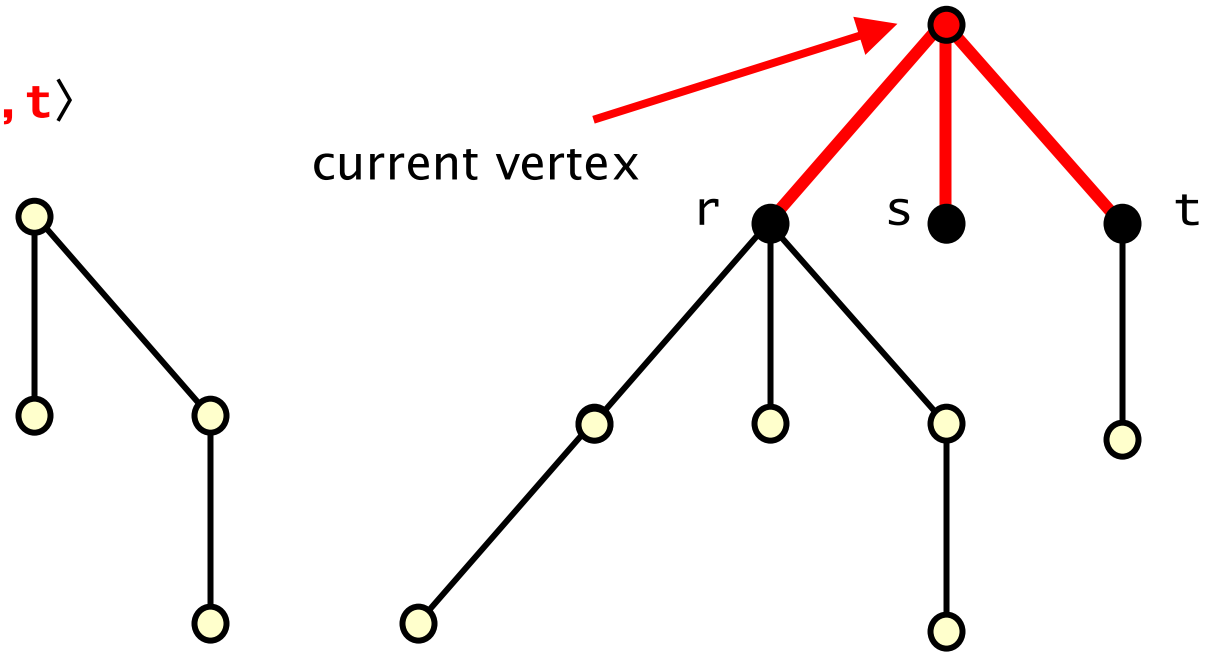


Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue (FIFO)

queue = ⟨**r,s,t**⟩

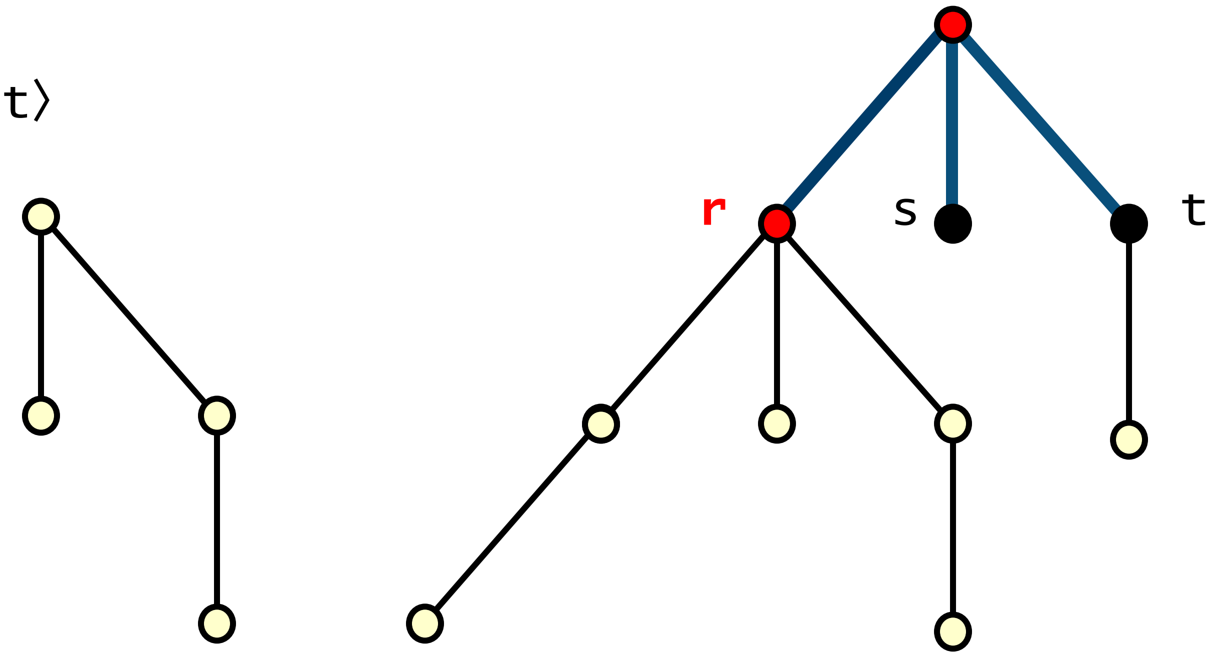


Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue (FIFO)

queue = $\langle s, t \rangle$

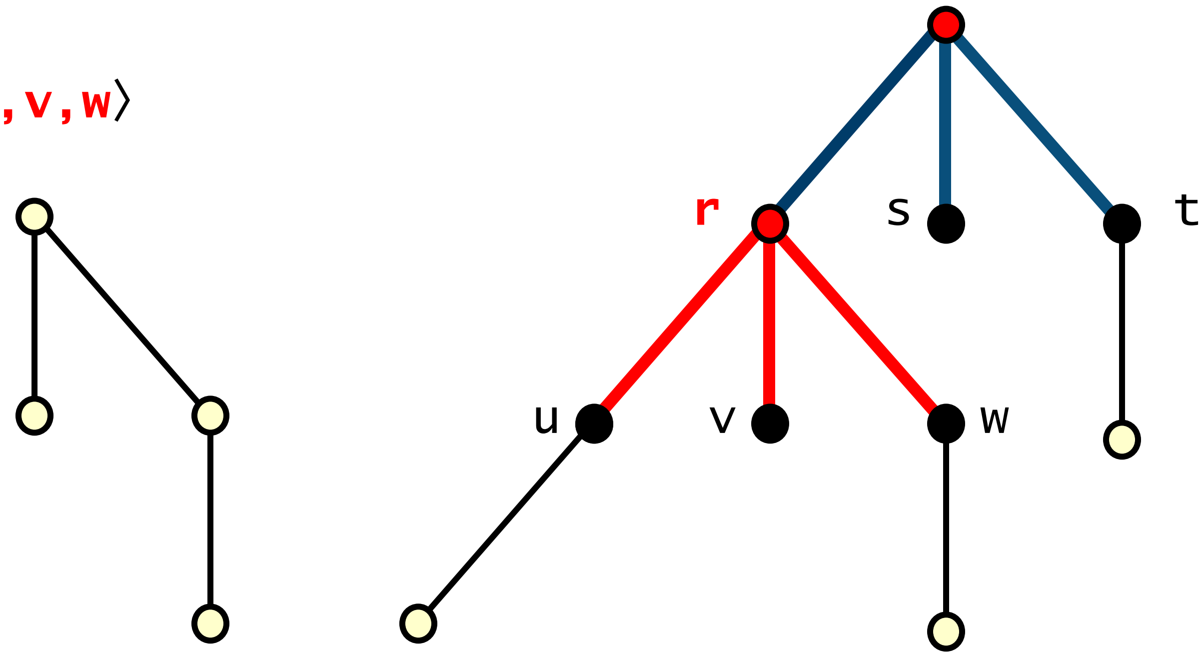


Breadth first search/traversal (BFS)

Search **fans out** as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue

queue = ⟨s,t,**u,v,w**⟩

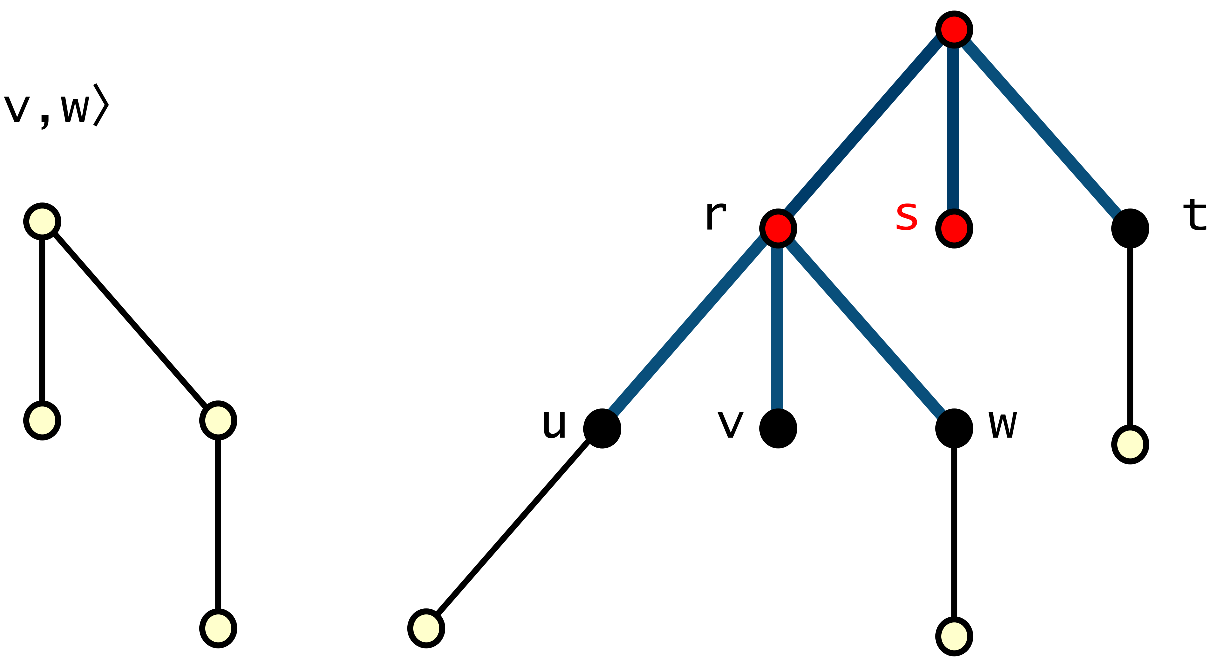


Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue

queue = $\langle t, u, v, w \rangle$

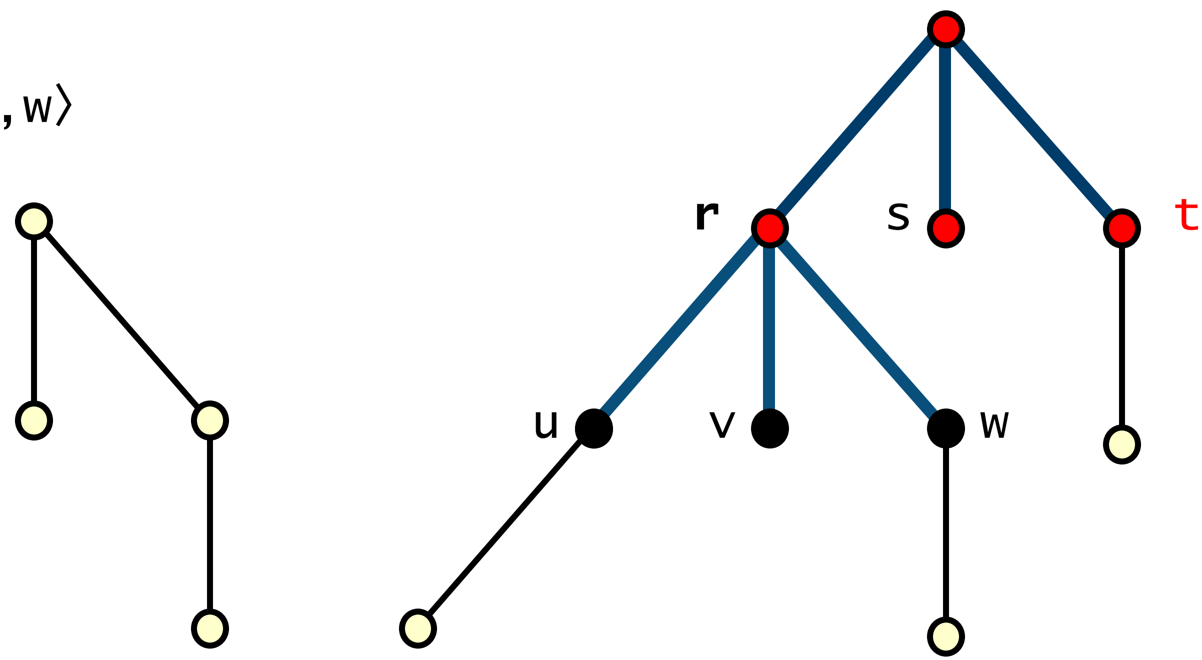


Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue

queue = $\langle u, v, w \rangle$

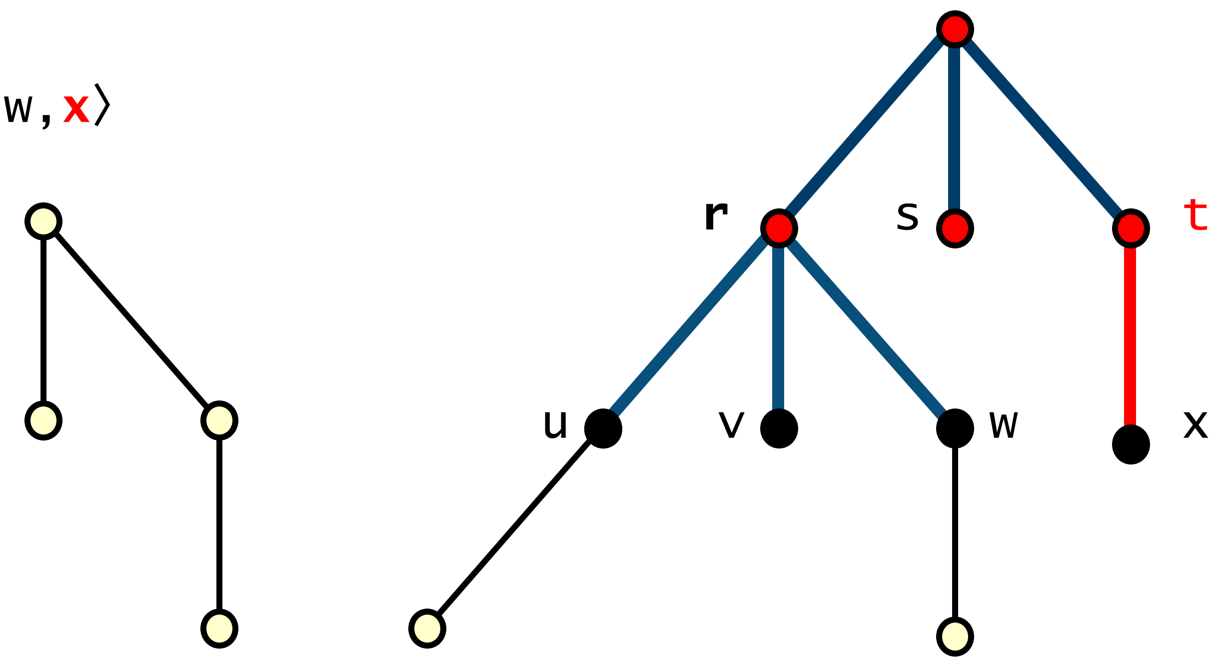


Breadth first search/traversal (BFS)

Search **fans out** as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue

queue = $\langle$u,v,w,**x**$\rangle$



Breadth first search/traversal (BFS)

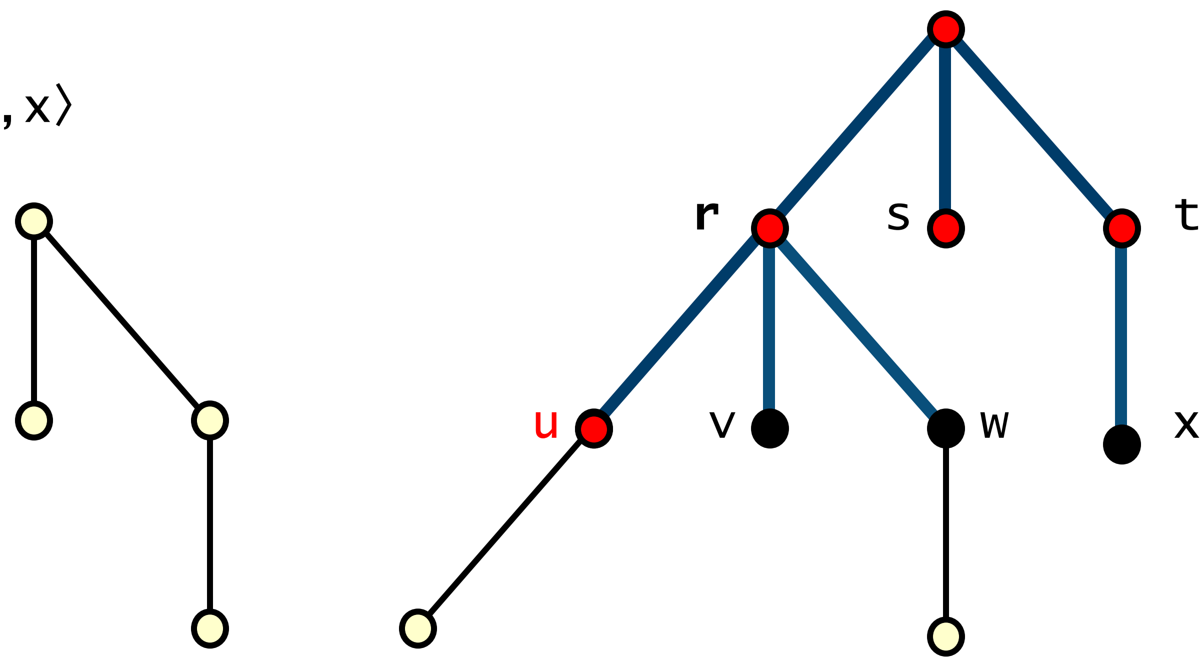
Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices

  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited

  therefore visited vertices are added/removed from a queue

queue = ⟨v,w,x⟩

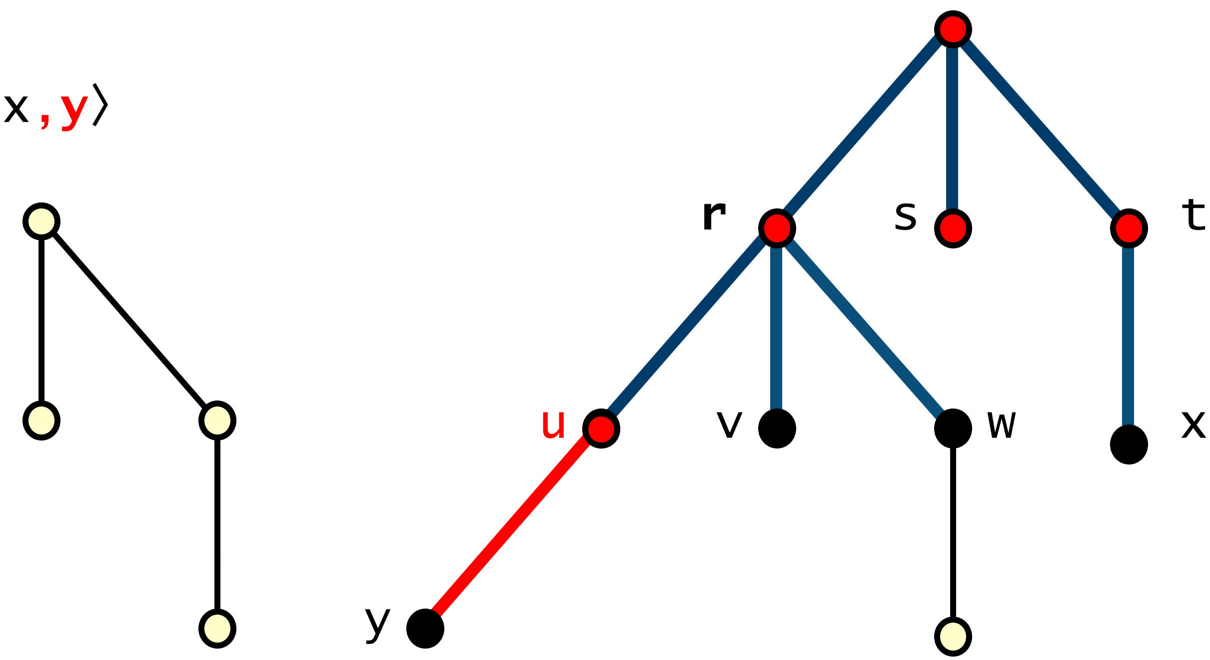


Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

– from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
– vertices are processed in the order in which they are visited
  therefore visited vertices are added/removed from a queue

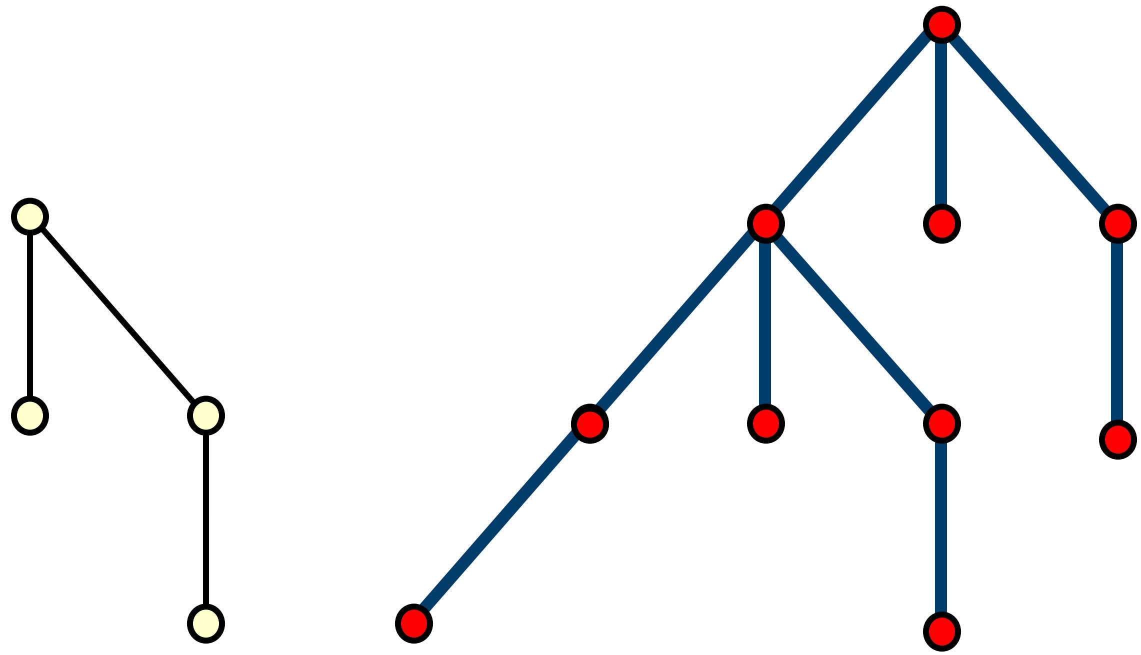
queue = ⟨v,w,x,y⟩



Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

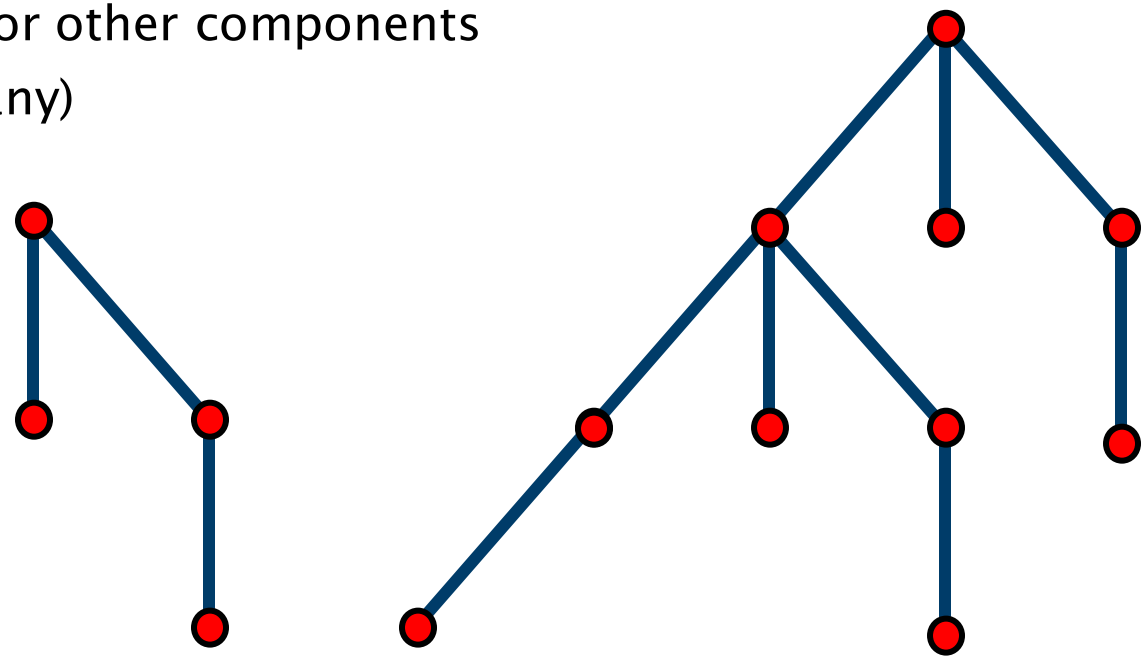
- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
- continue until all vertices in current component have been processed



Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
- continue until all vertices in current component have been processed
- then repeat for other components
  (if there are any)



Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

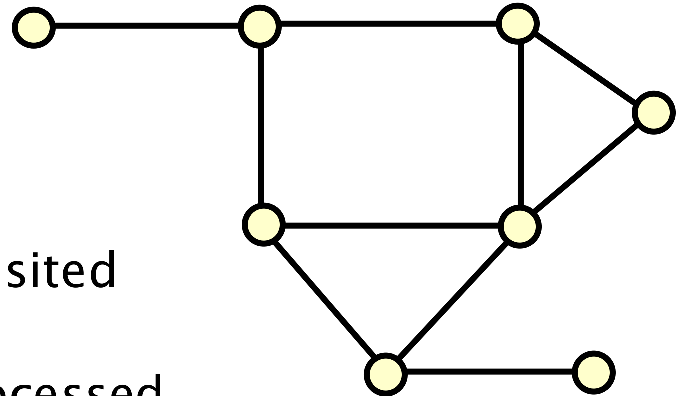
- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
- continue until all vertices in current component have been processed
- then repeat for other components
  (if there are any)

Again the edges traversed form a spanning tree (or forest)

- a breadth-first spanning tree (forest)
- spanning tree of a graph is tree composed of all the vertices and some
  (or perhaps all) of the edges of the graph

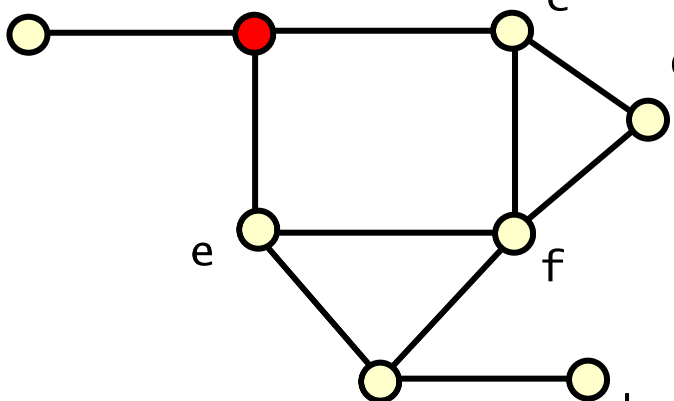
Breadth first traversal – Example

Undirected graph G



● denotes vertex has been visited

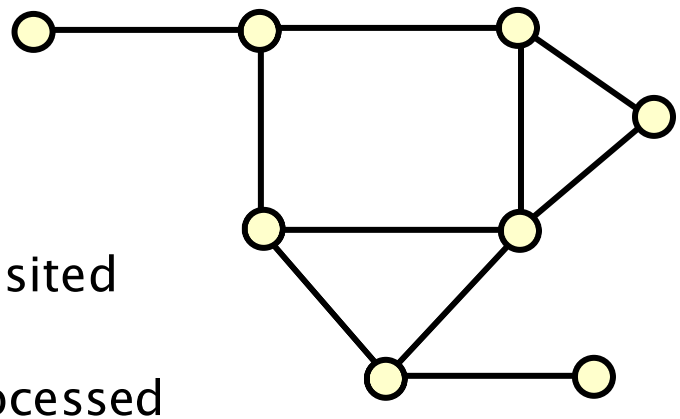
v means vertex has been processed



queue = ⟨a⟩

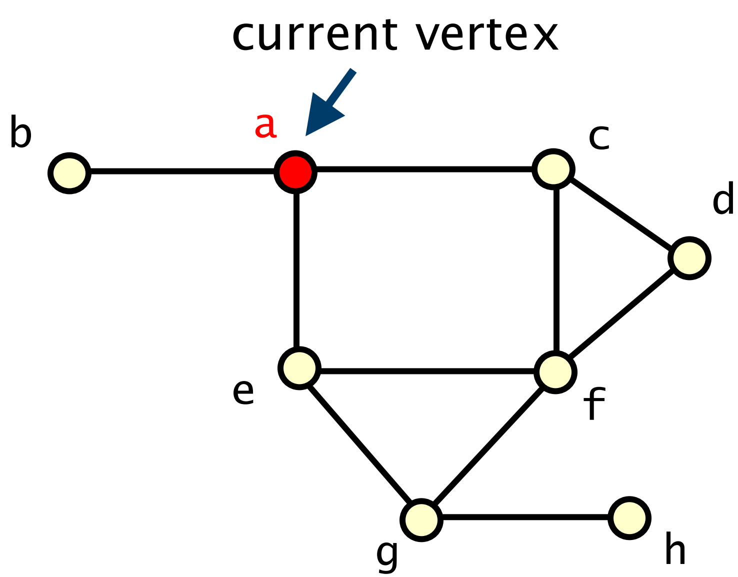
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

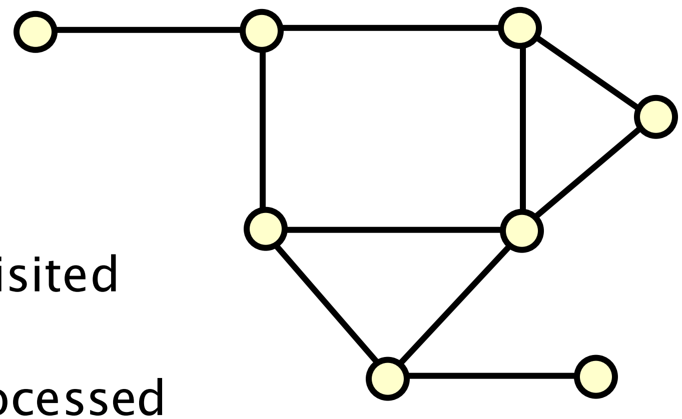
v means vertex has been processed



queue = $\langle\rangle$

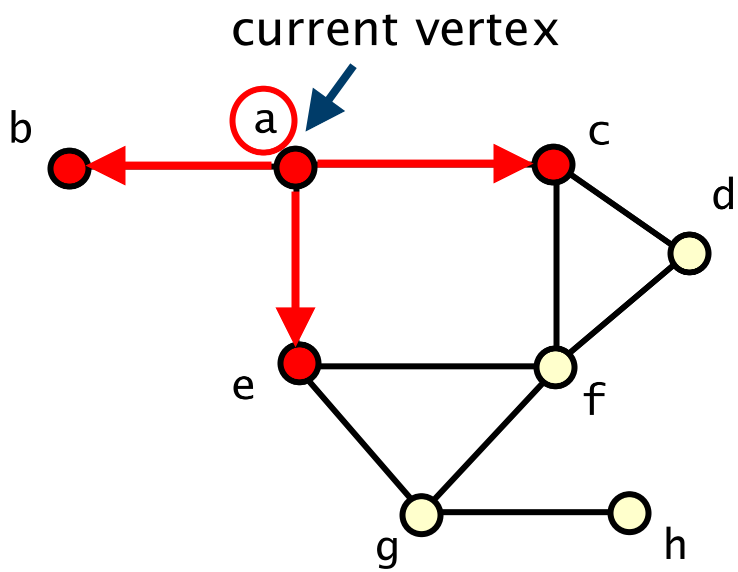
Breadth first traversal – Example

Undirected graph G



● denotes vertex has been visited

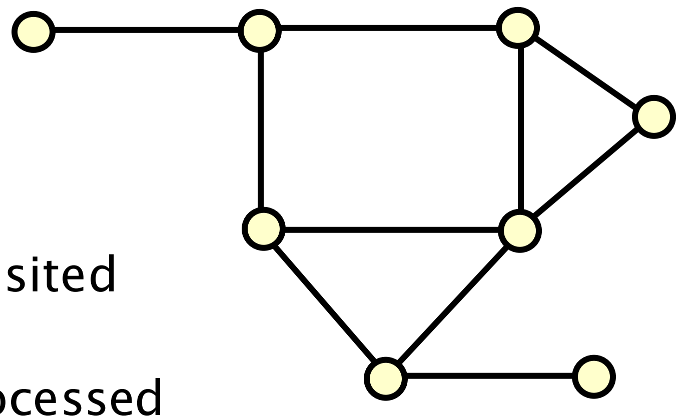
(v) means vertex has been processed



queue = ⟨b, c, e⟩

Breadth first traversal – Example

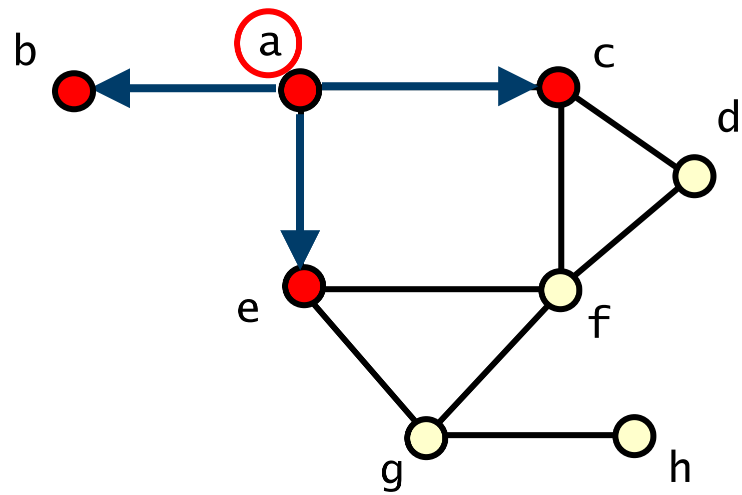
Undirected graph G



denotes vertex has been visited

v means vertex has been processed

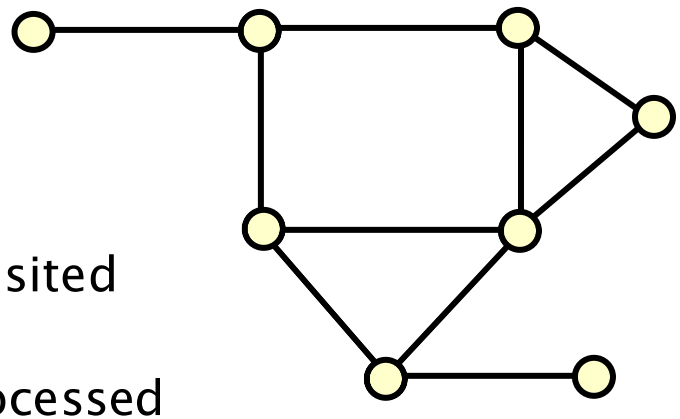
current vertex



queue = ⟨c, e⟩

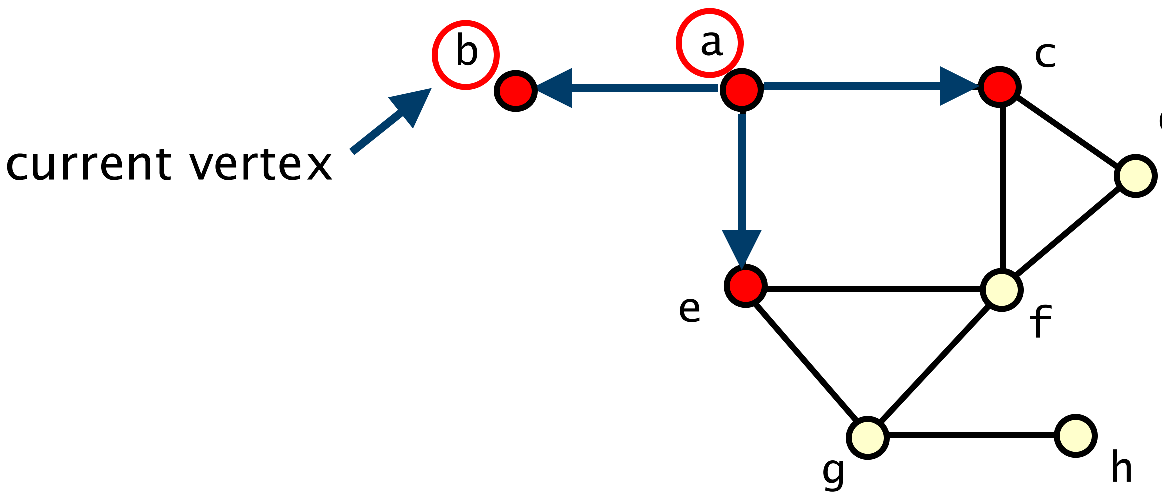
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

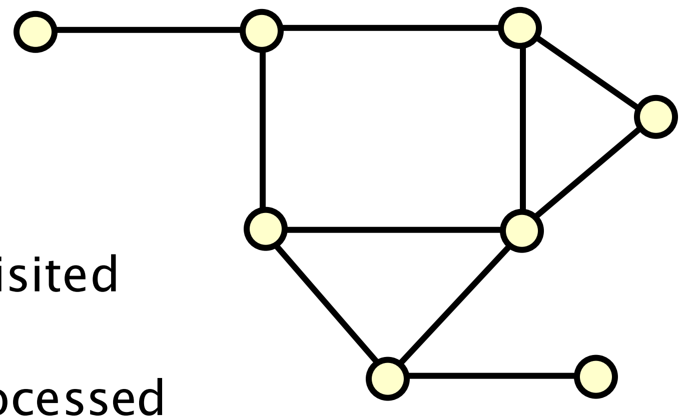
v means vertex has been processed



queue = $\langle c, e \rangle$

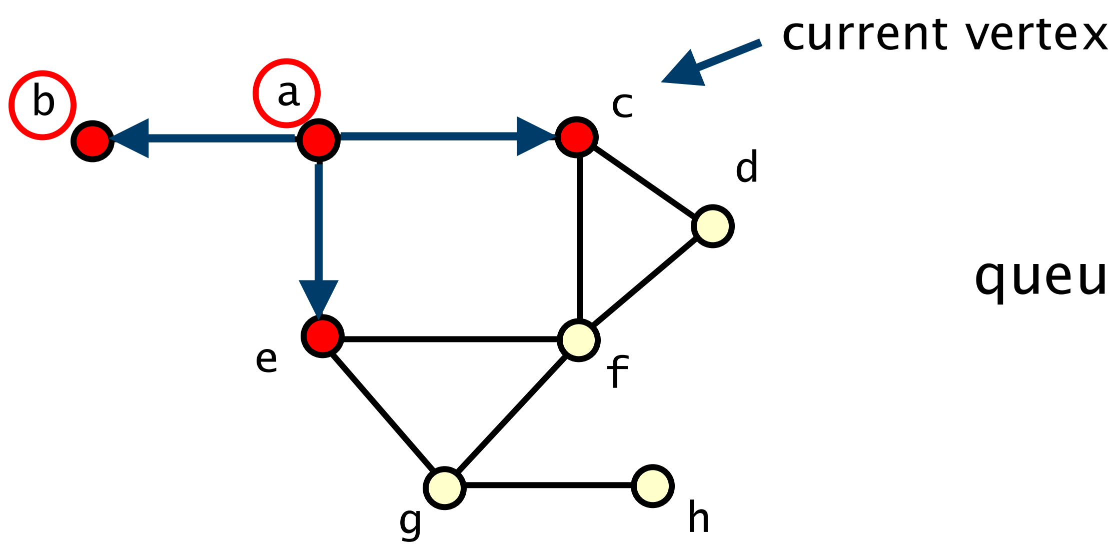
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

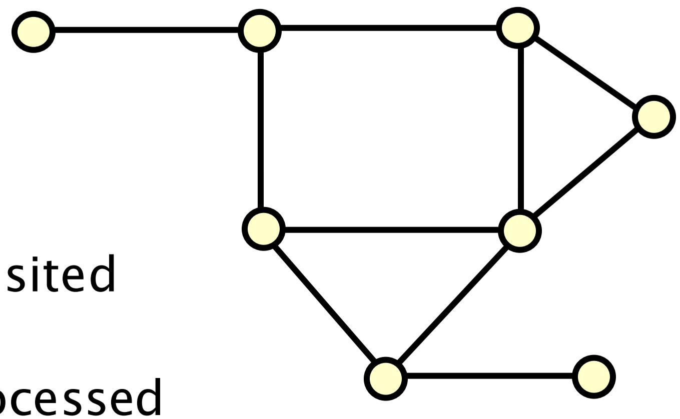
v means vertex has been processed



queue = ⟨e⟩

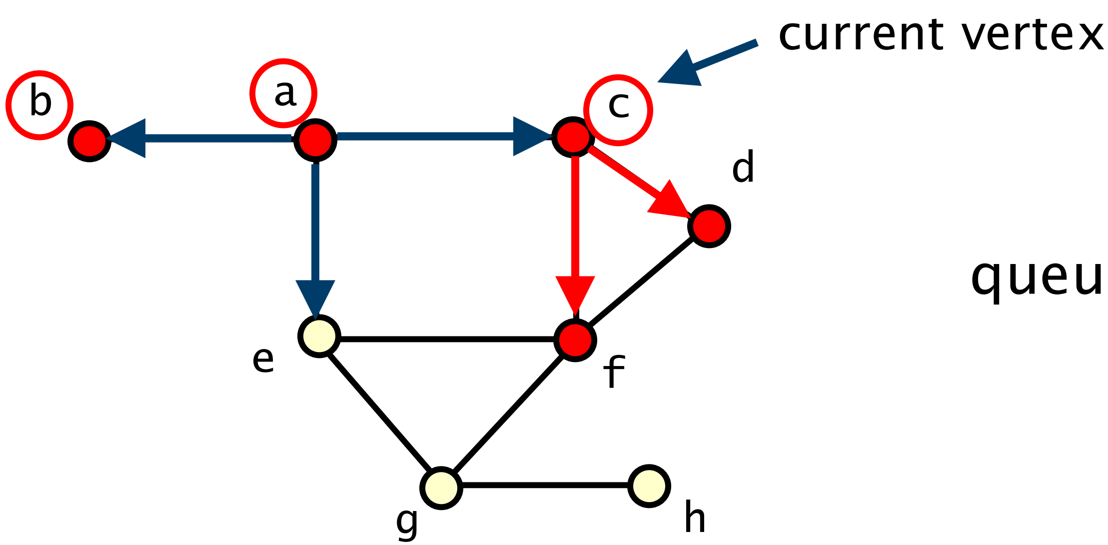
Breadth first traversal – Example

Undirected graph G



● denotes vertex has been visited

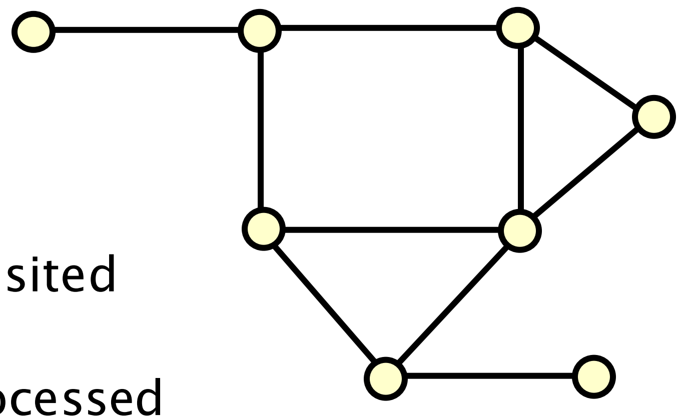
(v) means vertex has been processed



$$\text{queue} = \langle e, d, f \rangle$$

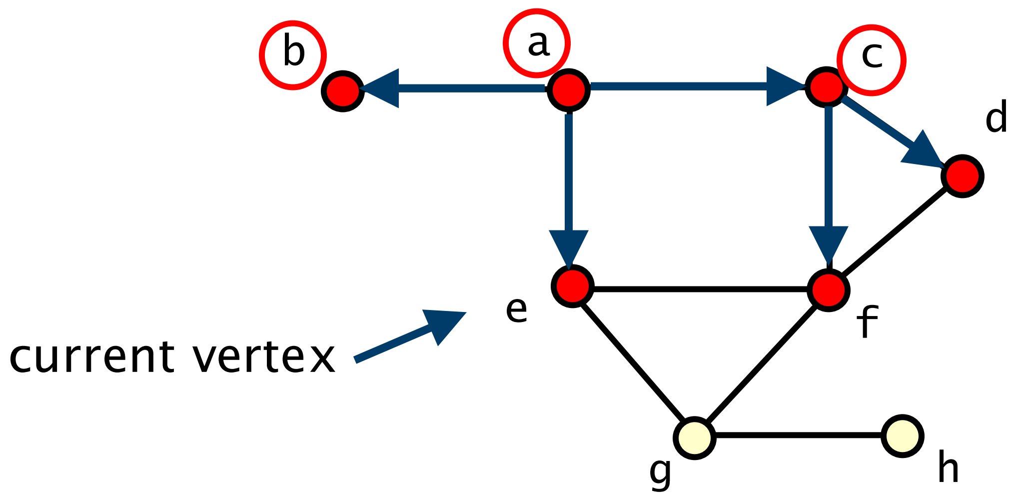
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

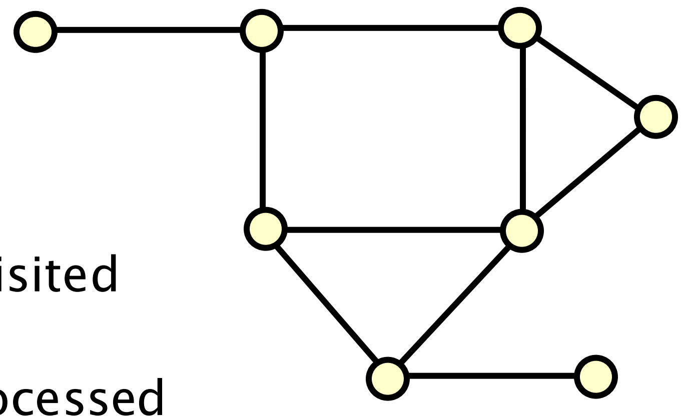
v means vertex has been processed



queue = ⟨d, f⟩

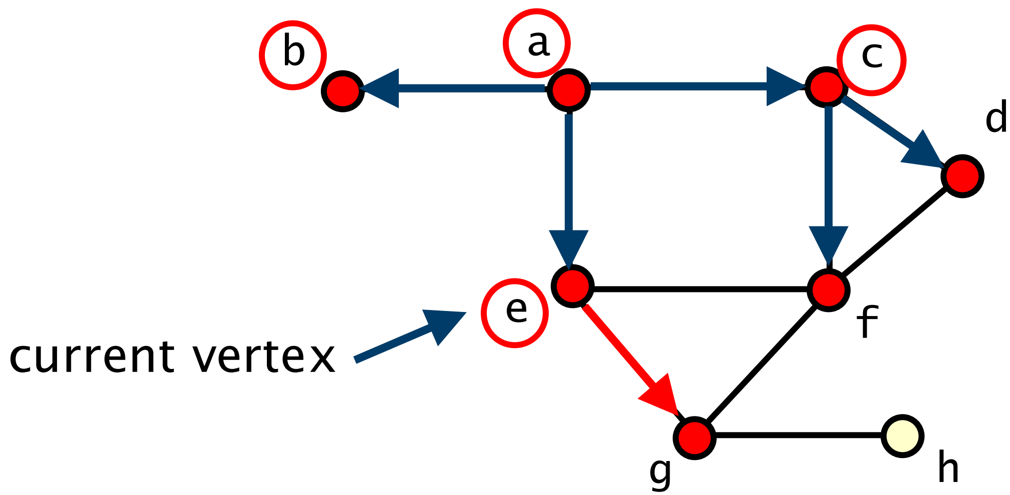
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

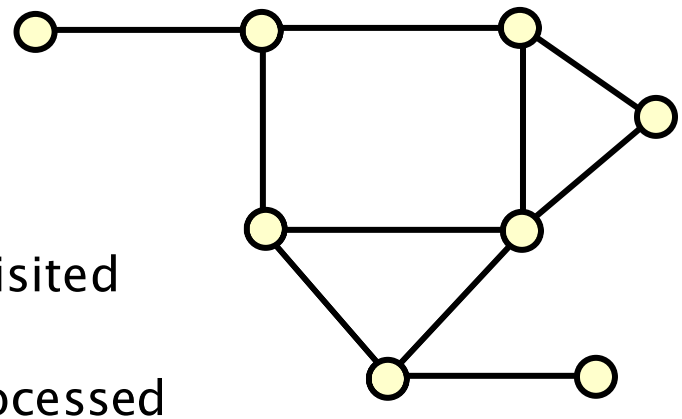
v means vertex has been processed



$$\text{queue} = \langle d, f, g \rangle$$

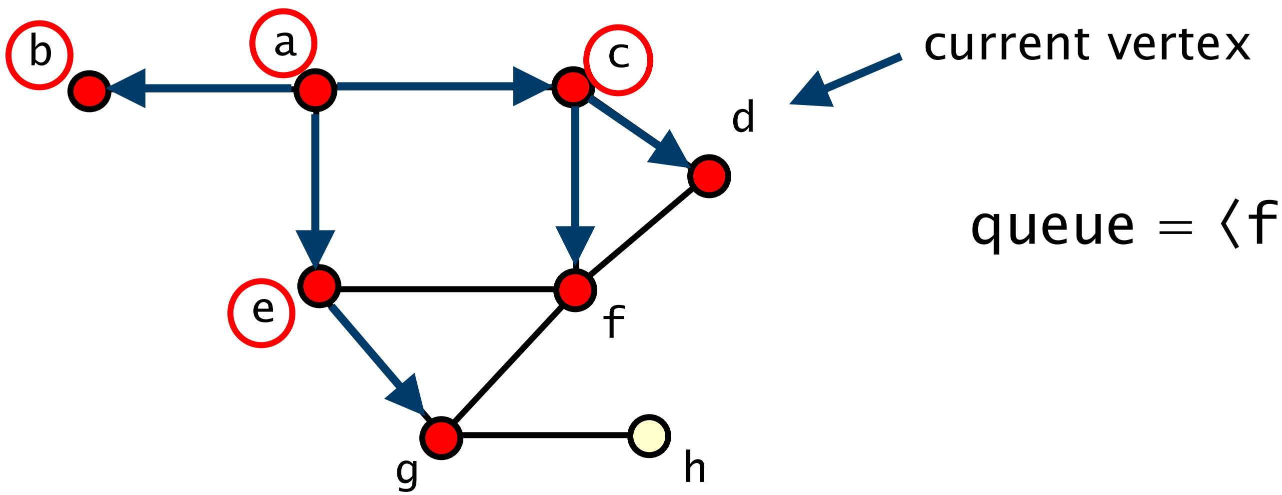
Breadth first traversal – Example

Undirected graph **G**



● denotes vertex has been visited

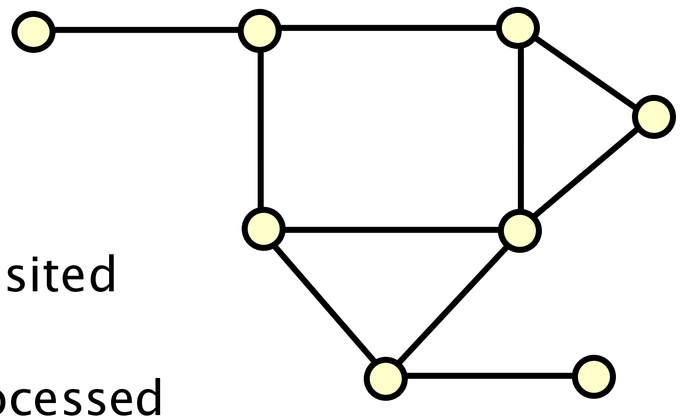
ⓥ means vertex has been processed



queue = ⟨f,g⟩

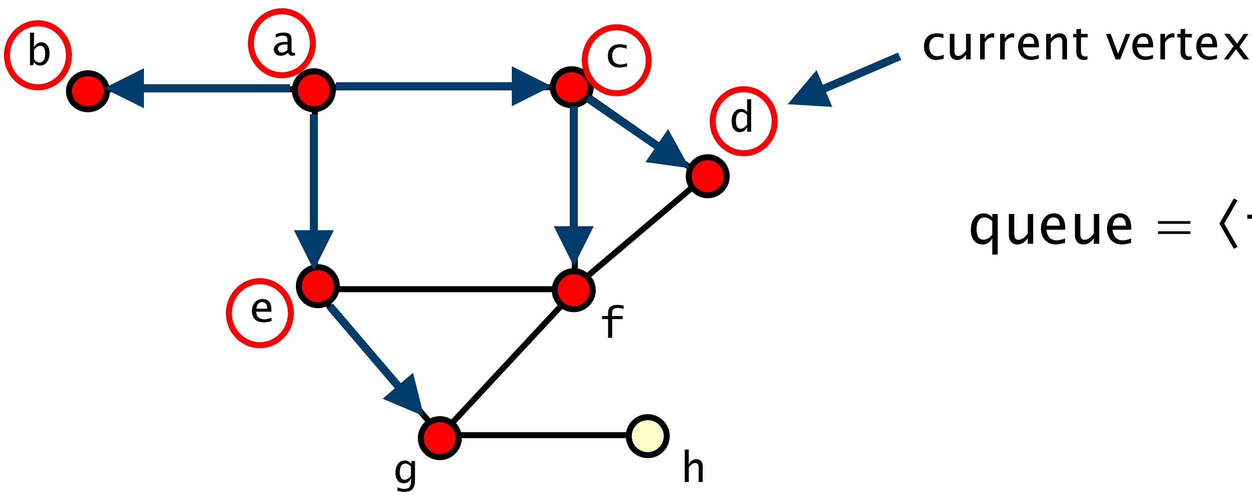
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

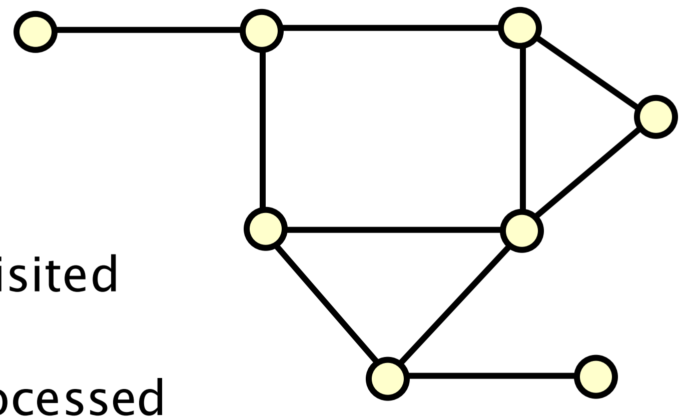
v means vertex has been processed



queue = $\langle f, g \rangle$

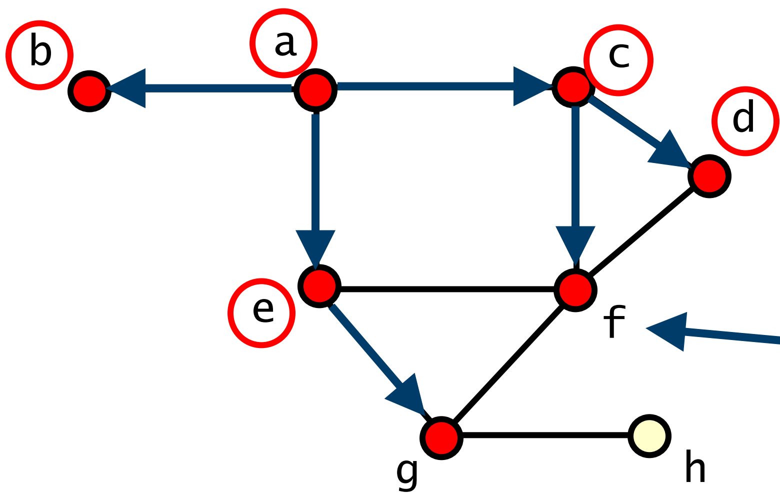
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

v means vertex has been processed

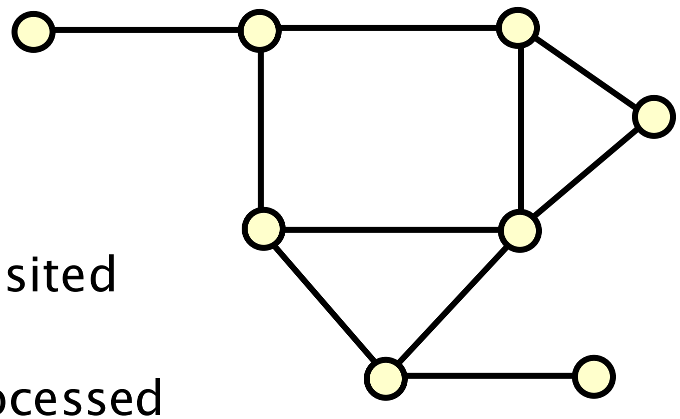


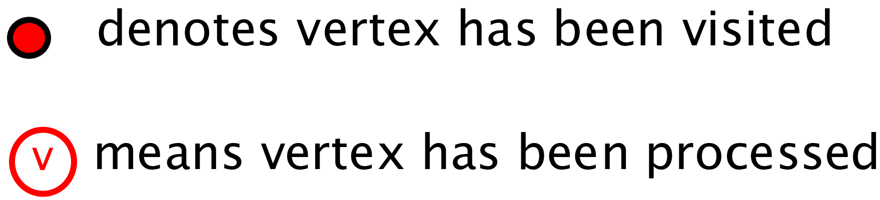
queue = ⟨g⟩

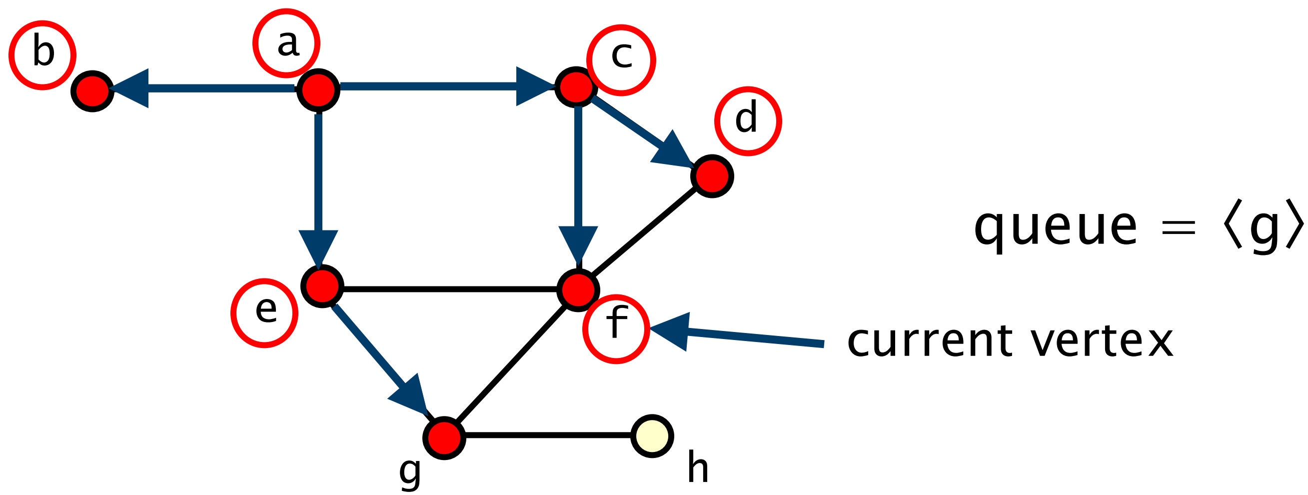
current vertex

Breadth first traversal – Example

Undirected graph G

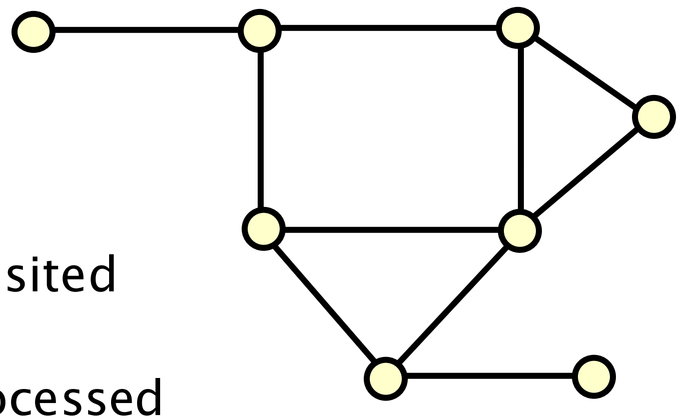






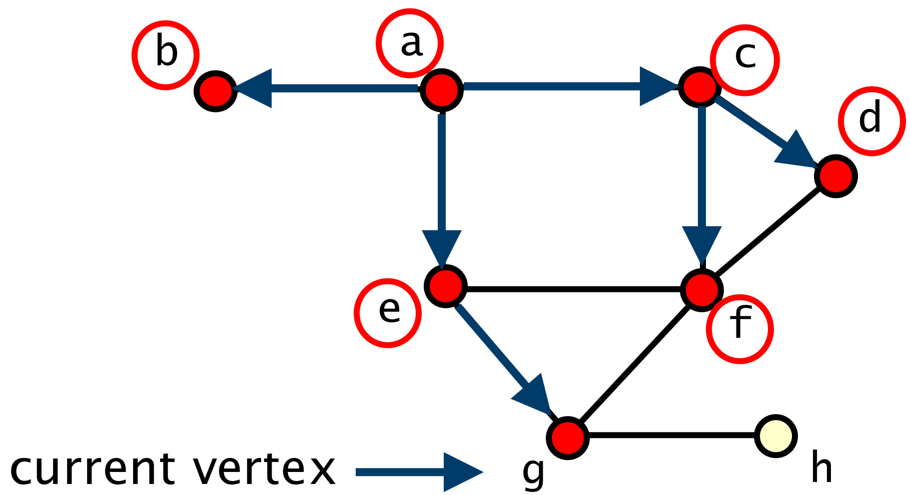
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

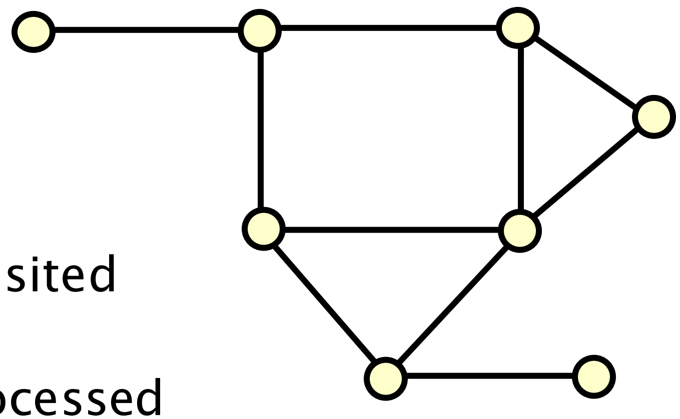
v means vertex has been processed



queue $= \langle \rangle$

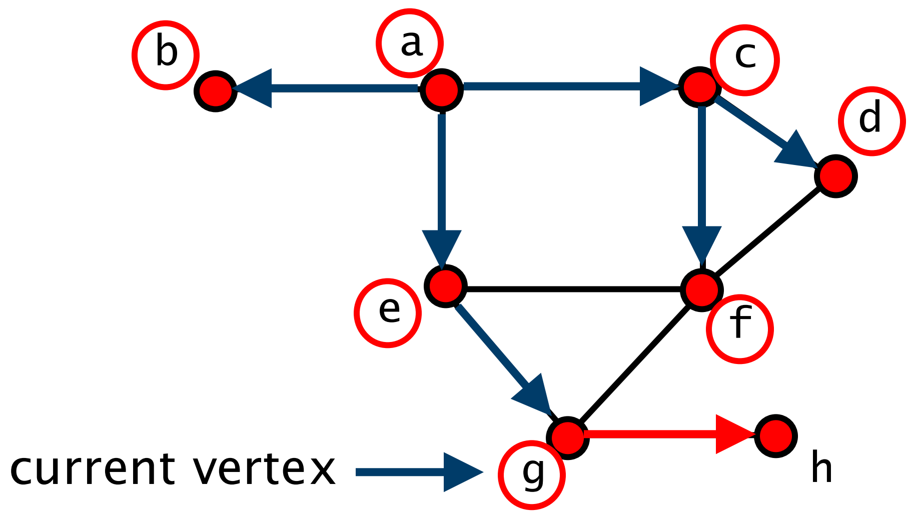
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

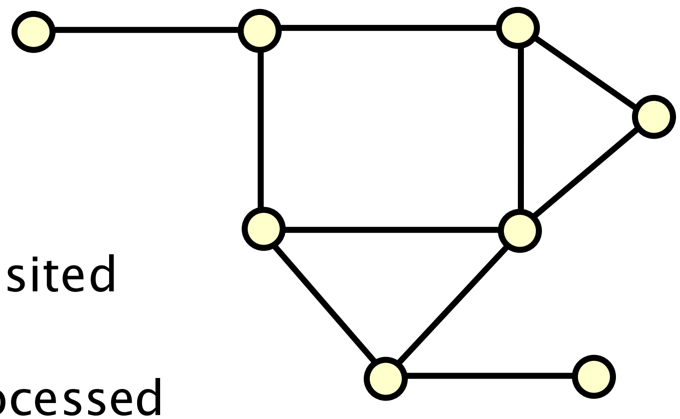
v means vertex has been processed



queue = ⟨h⟩

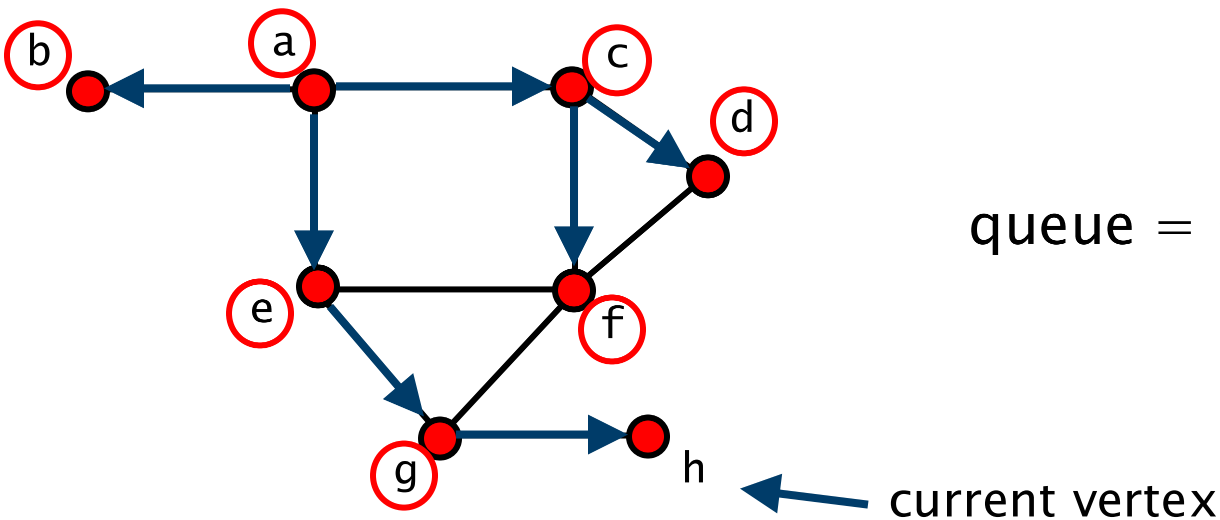
Breadth first traversal – Example

Undirected graph G



denotes vertex has been visited

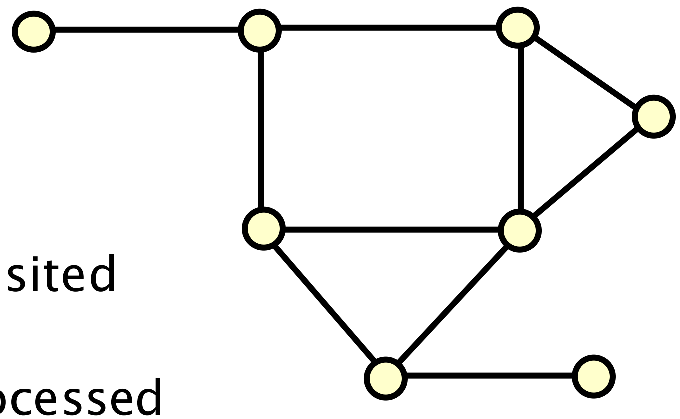
(v) means vertex has been processed



queue = $\langle\rangle$

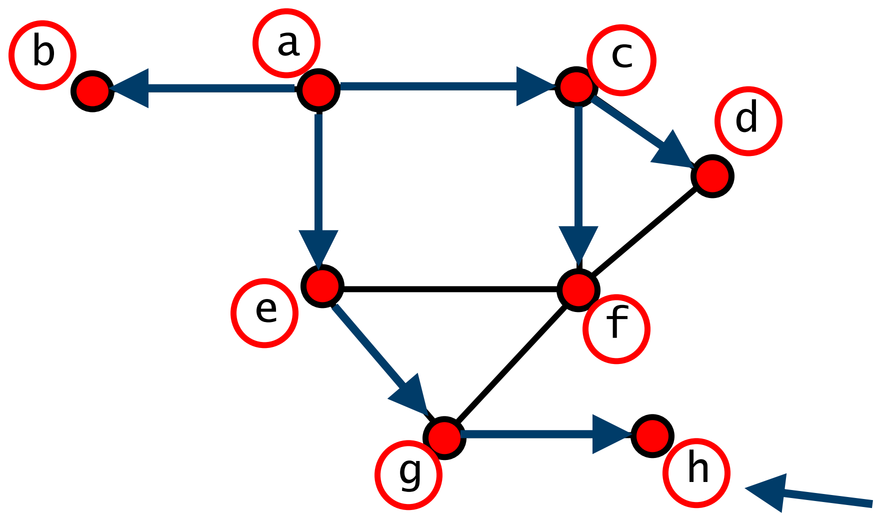
Breadth first traversal – Example

Undirected graph G



● denotes vertex has been visited

ⓥ means vertex has been processed

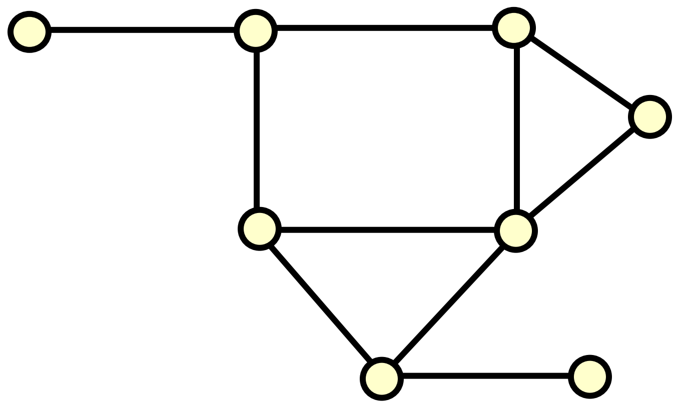


queue = $\langle\,\rangle$

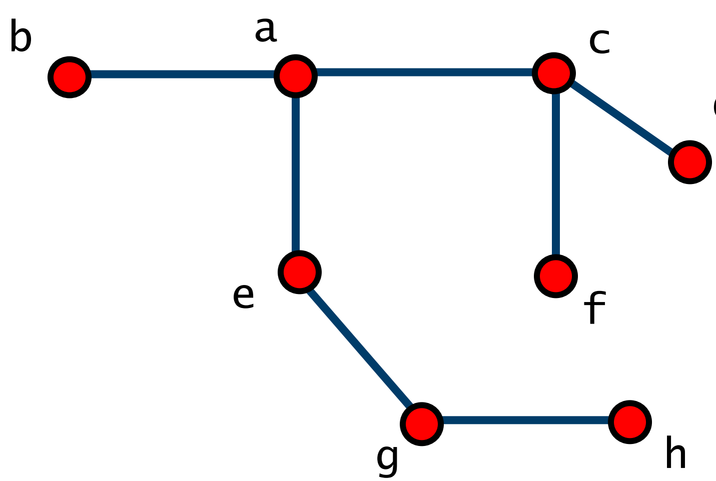
current vertex

Breadth first traversal – Example

Undirected graph **G**



A breadth first spanning tree of **G**



Recall adjacency list implementation

Class: adjacency node

- represents an element of an adjacency list
- includes a vertex index (the vertex the element corresponds to)

Class: vertex

- represents a single vertex of the graph
- includes linked list of adjacency nodes representing the adjacent vertices

Class: graph

- an array of vertices

Implementation – Breadth first search

```
for (Vertex v : vertices) v.setVisited(false); // initialise
LinkedList<Vertex> queue = new LinkedList<Vertex>(); // set up queue
for (Vertex v : vertices) { // go through vertices in the graph
  if (!v.getVisited()) { // vertex not visited (start search)
    v.setVisited(true); // now visited
    v.setPredecessor(-1); // v initial/starting vertex
    queue.add(v); // ready to be processed (add to queue)
    while (!queue.isEmpty()) { // something to process
      Vertex u = queue.remove(); // get next vertex from queue
      LinkedList<AdjListNode> list = u.getAdjList(); // get adj list for u
      for (AdjListNode node : list) { // go through adj list of u
        Vertex w = vertices[node.getVertexIndex()]; // next vertex in list
        if (!w.getVisited()) { // not previous found
          w.setVisited(true); // now visited
          w.setPredecessor(u.getIndex()); // set predecessor of w to be u
          queue.add(w); // add to queue
        }
      }
    }
  }
}
```

Breadth first search – complexity

Each vertex is visited and queued exactly once

Each adjacency list is traversed once (when it's processed)

So overall $O(n+m)$

- n is the number of vertices and m number of edges

We can adapt to adjacency matrix representation

- complexity $O(n^2)$ as for DFS
- have to access every element of the matrix

Breadth first search – application

Computing the distance between two vertices in a graph

- let v and w be to vertices in the graph
- the distance is the number of edges in the shortest path from v to w

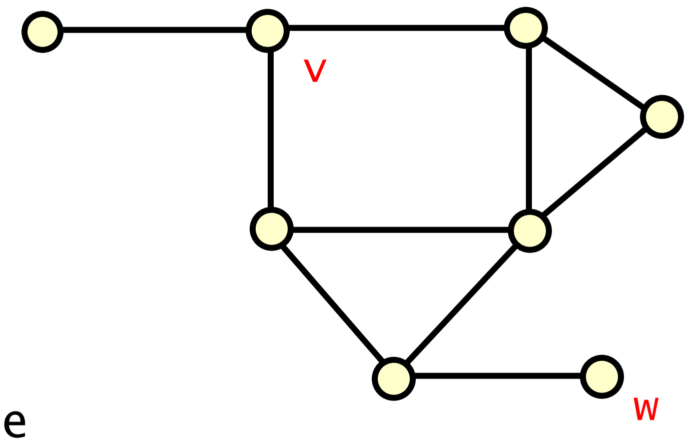
Algorithm

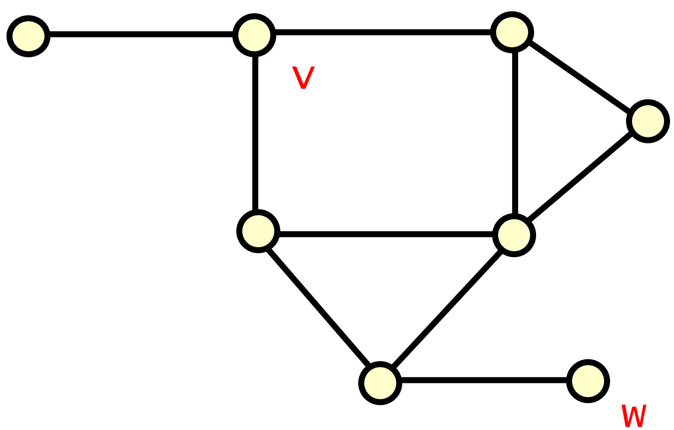
- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time, assign its distance to be 1 + the distance to its predecessor in the BF spanning tree
- stop when w is reached

Distance between two vertices – Example

Distance between **v** and **w**

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time
  assign its distance to be 1+ the distance
  to its predecessor in the BF spanning tree

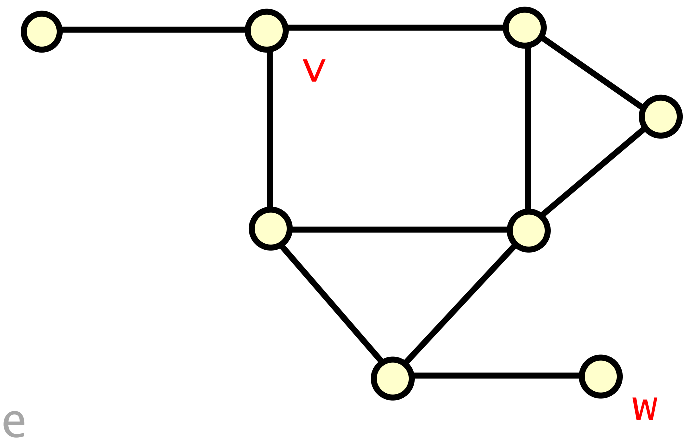


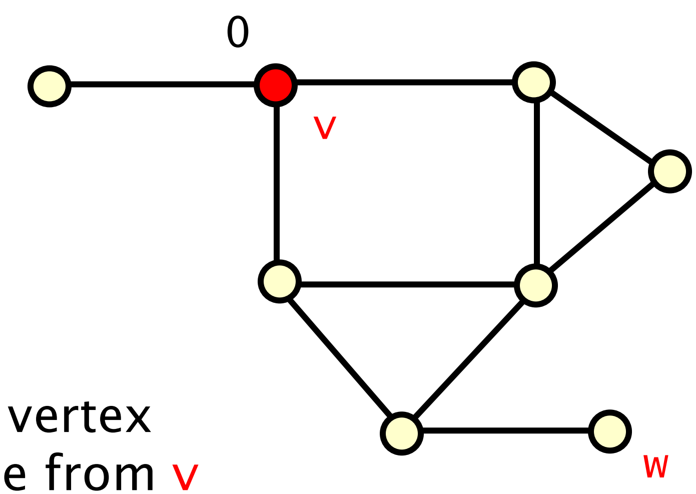


Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time assign its distance to be 1+ the distance to its predecessor in the BF spanning tree



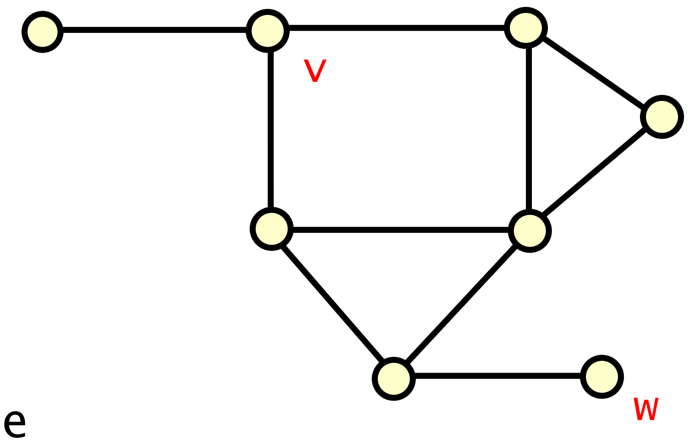


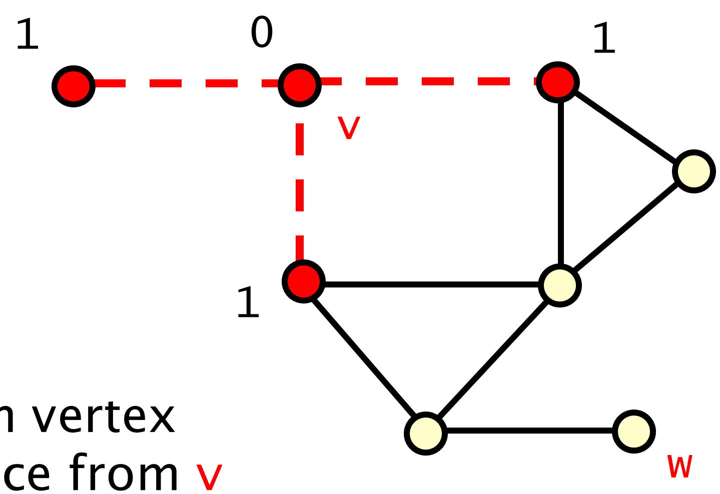
number beside each vertex indicates the distance from v

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time assign its distance to be 1+ the distance to its predecessor in the BF spanning tree



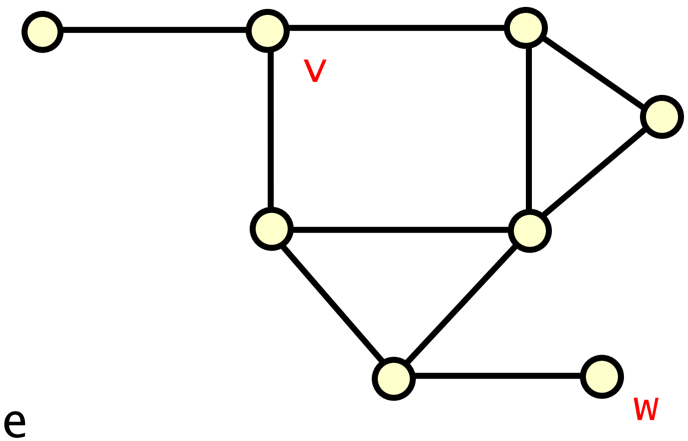


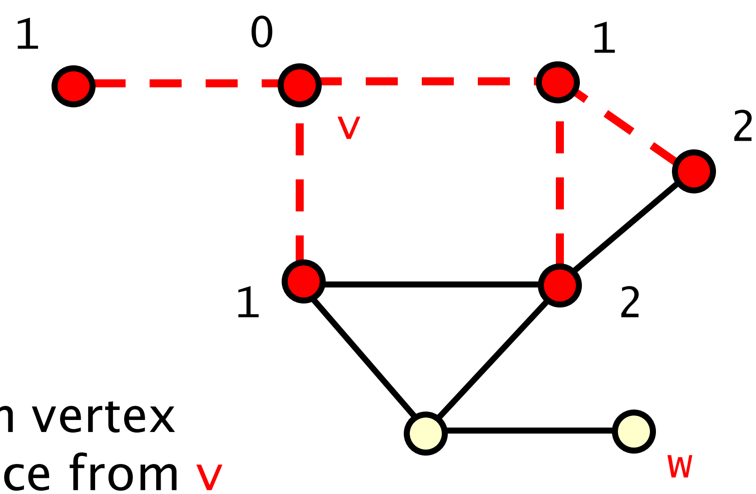
number beside each vertex
indicates the distance from v

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time assign its distance to be 1+ the distance to its predecessor in the BF spanning tree



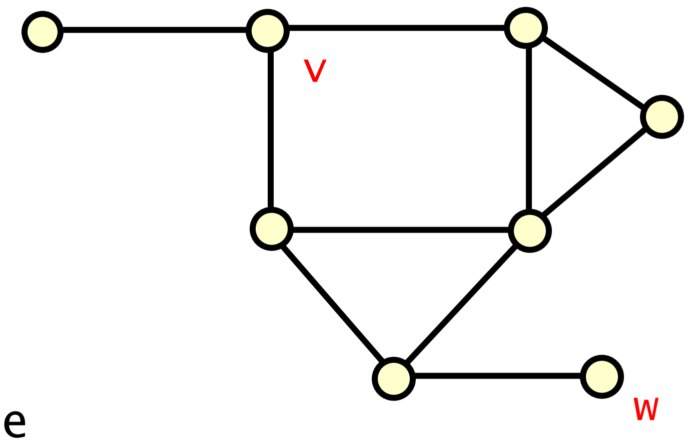


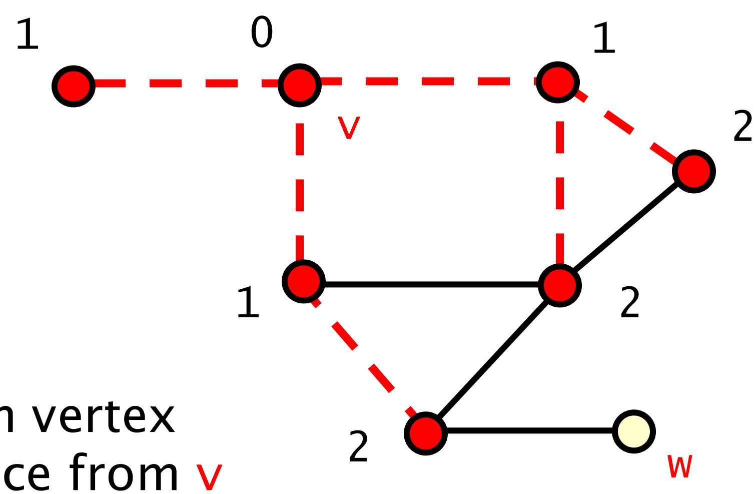
number beside each vertex
indicates the distance from v

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time
  assign its distance to be 1+ the distance
  to its predecessor in the BF spanning tree



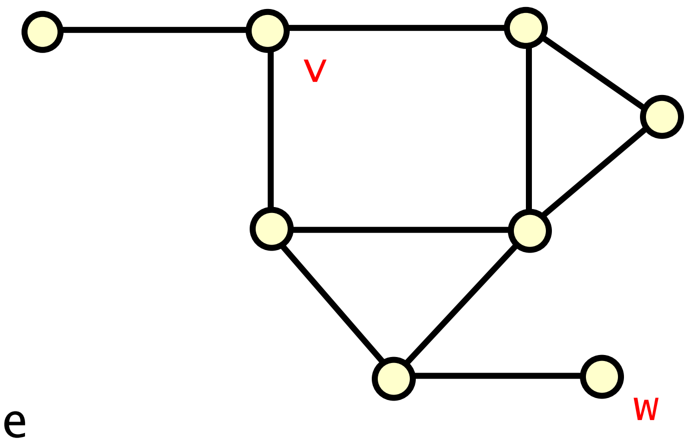


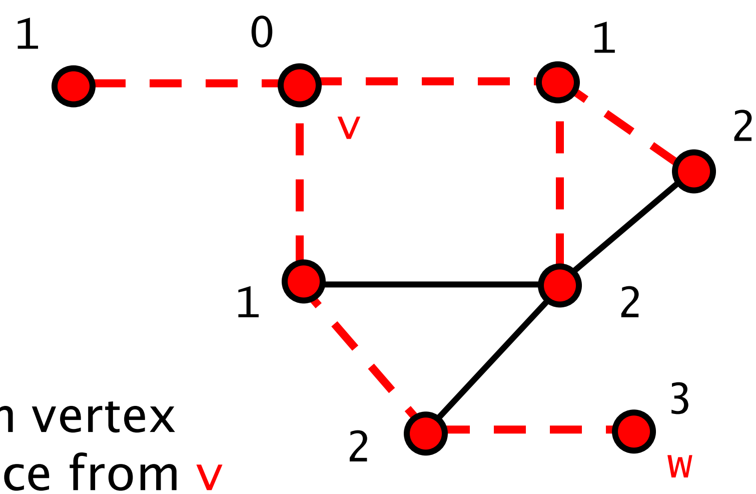
number beside each vertex
indicates the distance from v

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time assign its distance to be 1+ the distance to its predecessor in the BF spanning tree



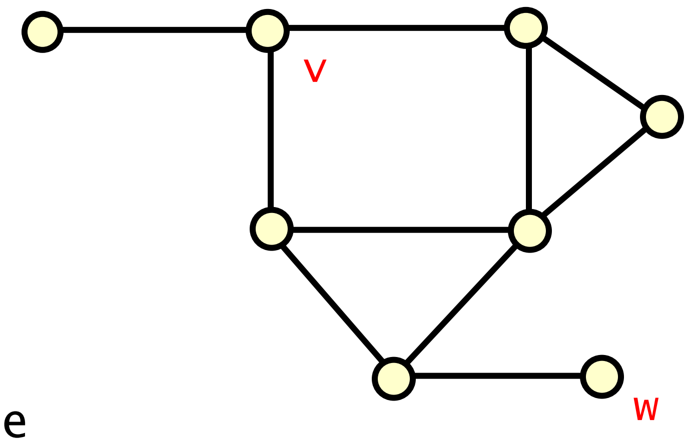


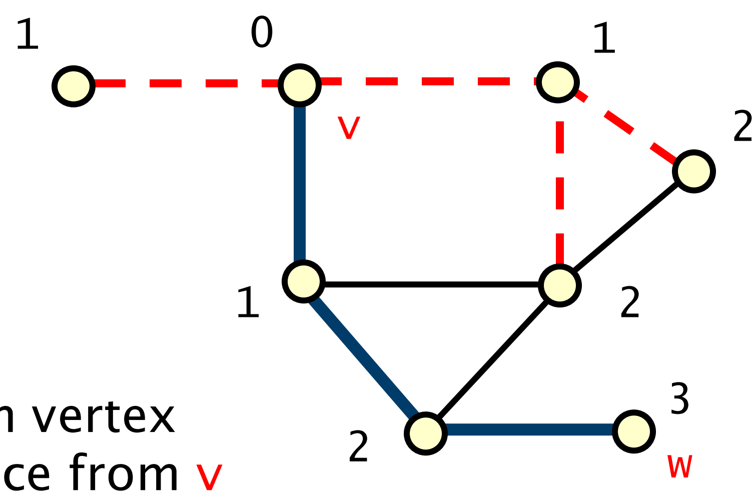
number beside each vertex indicates the distance from v

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time assign its distance to be 1+ the distance to its predecessor in the BF spanning tree





number beside each vertex indicates the distance from v

Next lecture

Graph basics – recap

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)