Graphs

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

Today Objectives

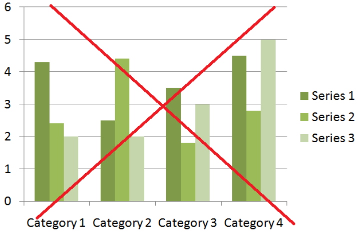
- Introduce Graph data structure
- Study well-known problems then implement examples in C/C++.

So far we have seen linear structures

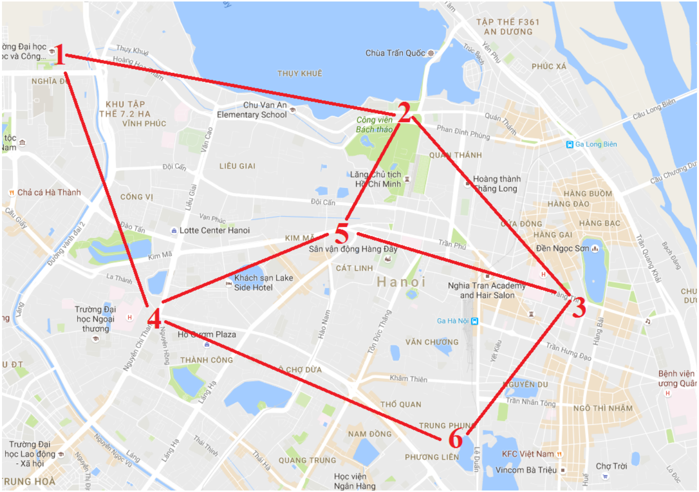
- lists, vectors, arrays, stacks, queues

Previously, Trees are already introduced. Today, we study non-linear structure: Graphs

- probably the most fundamental structure in computing
- hierarchical structure

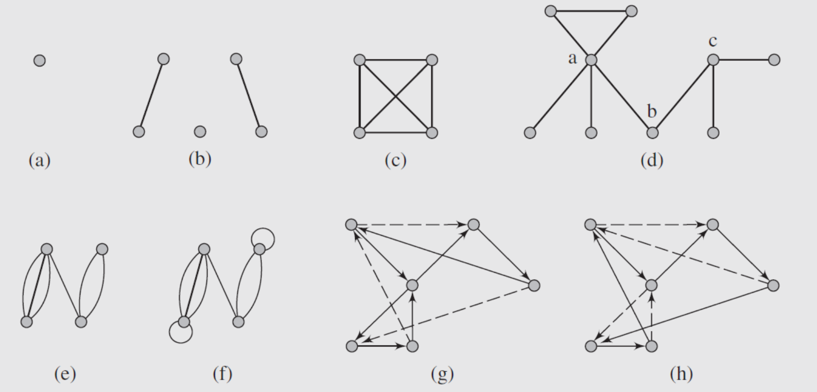


We don't study these kinds of graphs.



Shortest path problems

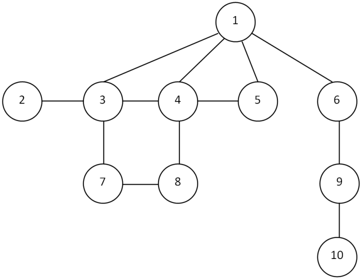
Graphs

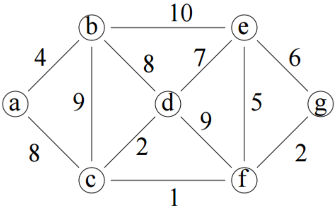


Definition

A **graph** G is a representation of a set of objects V and of edges E $(G = (V, E))$

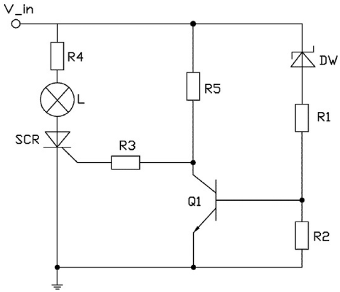
- V is a set of nodes (objects) called **vertices** (vertex in singular).
- $E = \{(u, v)\}, u, v \in V$ is a collection of **edges**, pairs of vertices.

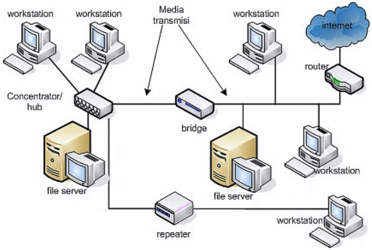




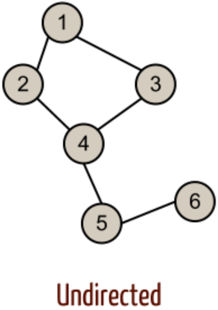
- Transportation Networks
  - Bus networks: Bus stations (vertices) and roads (edges)
  - Flight networks: Airports (vertices) and directions (edges)

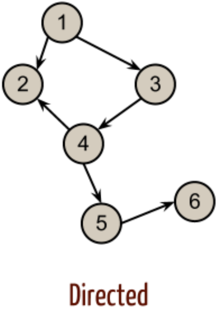
- Computer Networks, Internet: Computers and capables
- Social Networks: Users and relationships
- Electronic Circuits: Components and lines





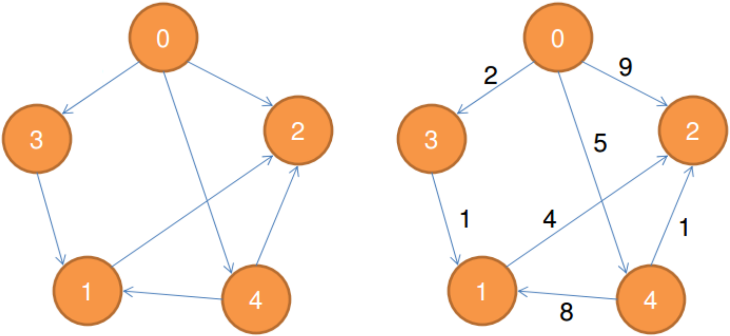
- Undirected and directed graphs
  - Undirected graphs: the pairs of vertices are **unordered** or bidirectional.
  - Direct graphs: the edges have a direction associated with them. A pair of vertices (u, v) is ordered where u is the source and v is the destination.





Unweighted and weighted graphs

- A weight is a numerical value, assigned as a label to a vertex or edge of a graph. A weighted graph is a graph whose vertices or edges have been assigned weights;
- A unweighted graph does not have any weight.

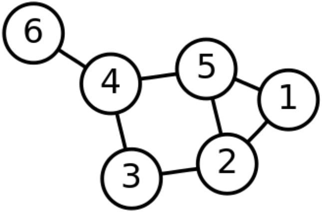


Adjacency matrix

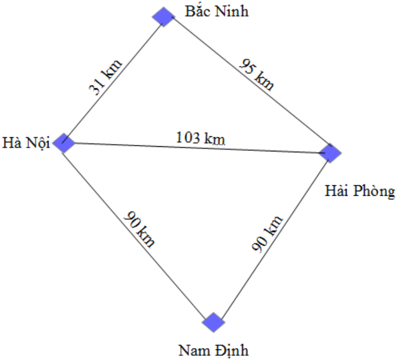
An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

$$A = \begin{cases} a_{ij} & \text{if}(u, v) \in E \\ 0 & \text{if}(u, v) \notin E \end{cases}$$

$a_{ij} = 1$ if the graph is unweighted



$$A = \begin{array}{c c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$



$$A = \begin{array}{c c} & \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} & \begin{pmatrix} 0 & 31 & 103 & 90 \\ 31 & 0 & 95 & 0 \\ 103 & 95 & 0 & 90 \\ 90 & 0 & 90 & 0 \end{pmatrix} \end{array}$$

v_1 = Ha Noi, v_2 = Bac Ninh, v_3 = Hai Phong, v_4 = Nam inh.

Graph ADT

Application

Graph ADT can be as following:

- init(): initialize an empty graph
- addVertex(v): add new vertices in a graph
- addEdge(u,v): add a new edge between a pair of vertices
- isEmpty(): verify whether a graph is empty
- isLinked(u,v): return true if there is an edge between this pair of vertices; otherwise return false.
- remove(v): remove a vertex from a graph
- search(u,v): search a path from a source vertex to the destination vertex

Graph ADT

Common different approaches to implement a Graph ADT

- Static array: arrays can be simply used to manipulate collections of elements.
- Dynamic array: using **malloc()** is capable of representing a list to avoid the fixed-size list
- Linked list: A very flexible mechanism for dynamic memory management is provided by pointers.

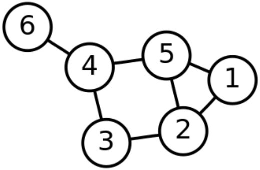
Static Array-based Graph may be used to build the adjacency matrix. However, there is a limit in graph sizes.

```
const int N = 100;
int G[N][N];
```

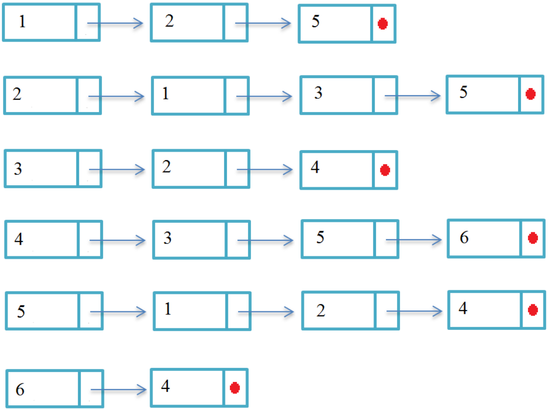
Linked List Graph can be implemented as following

```
struct Node{
    int vertex;
    Node * next;
};
const int N = 100;
typedef Node * Graph[N];
```

Graph ADT



$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$



Graph Traversal

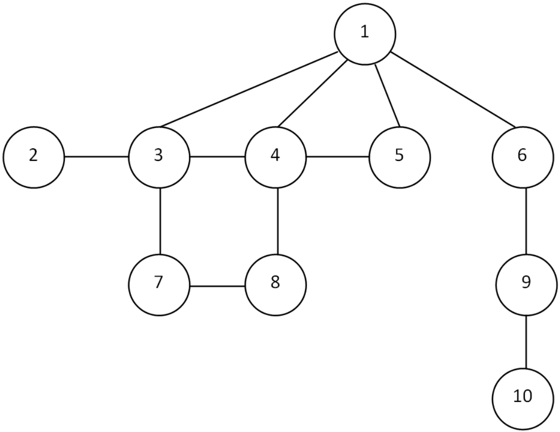
Graph traversal (also known as graph search) refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

- Searching algorithms: BFS (Breadth-First Search), DFS (Depth First Search), etc.
- Shortest Path: Minimum Spanning Tree, Greedy Algorithms.

Breadth First Search

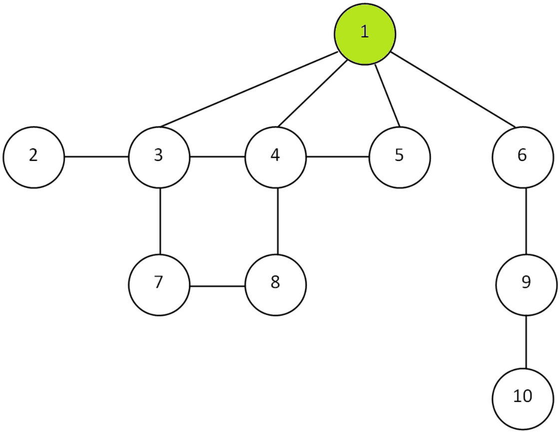
1. From a vertex $v \in G$, find all the adjacent vertices u with v and u is not yet visited.
2. Visit all these vertices u and find all their adjacent vertices. .
3. This process repeats until all the vertices of G are visited.

Breadth First Search



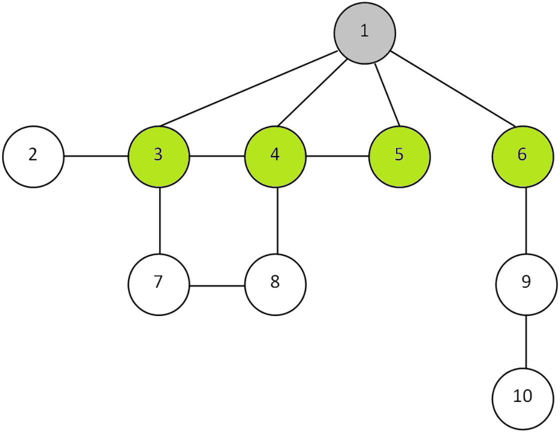
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



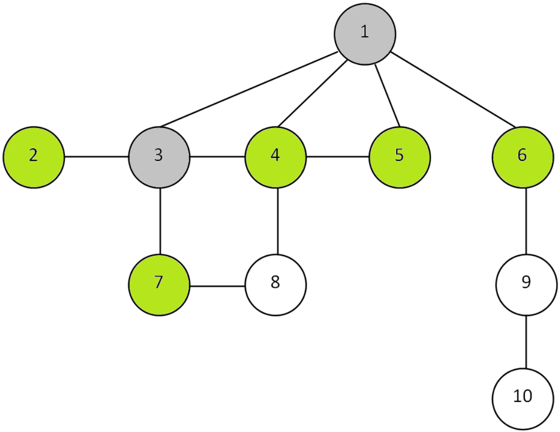
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



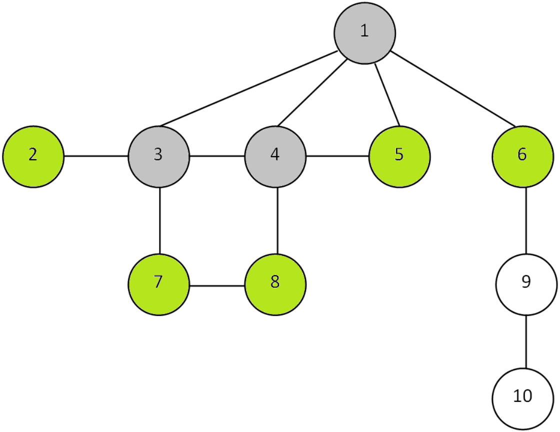
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search

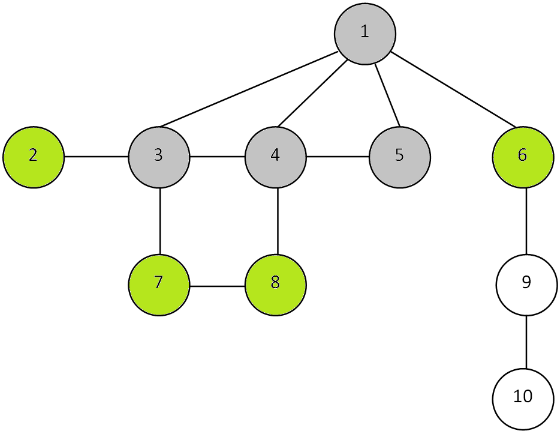


Green nodes are to be visited in a queue and gray nodes are visited.

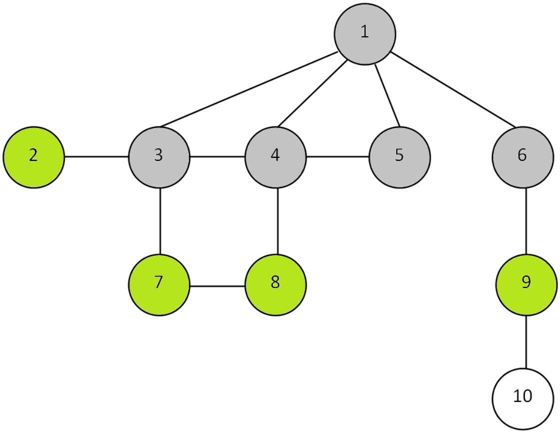
Breadth First Search



Green nodes are to be visited in a queue and gray nodes are visited.

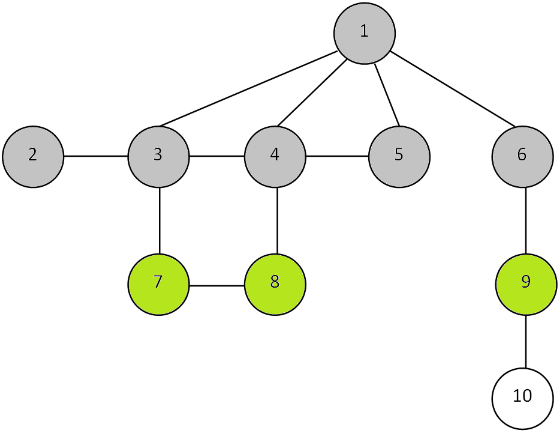


Green nodes are to be visited in a queue and gray nodes are visited.



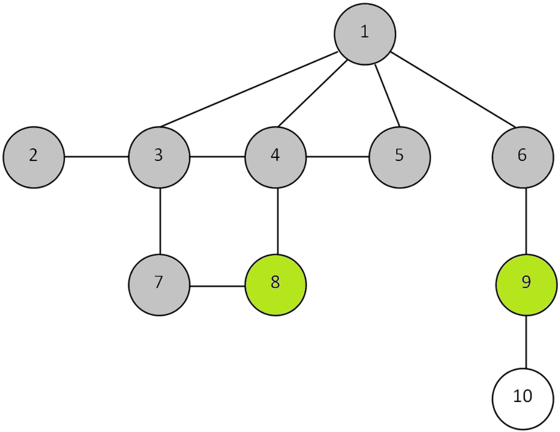
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



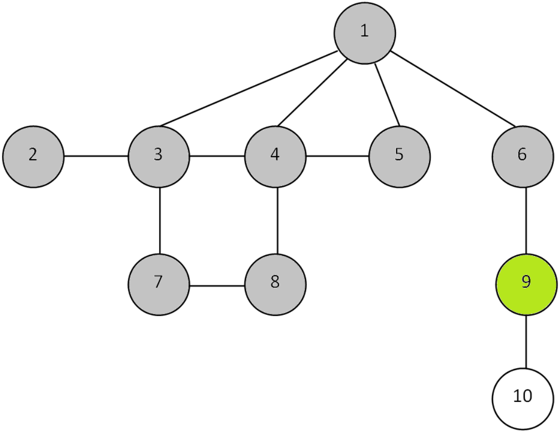
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



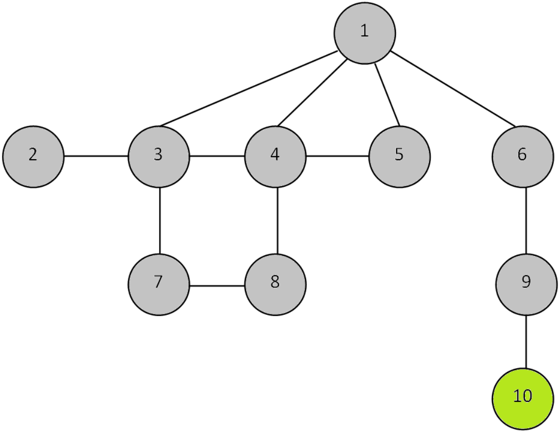
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



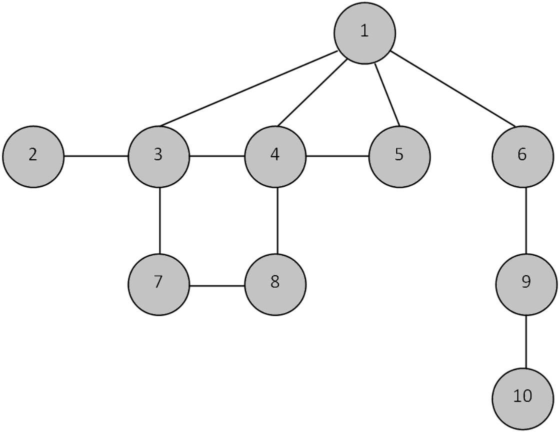
Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



Green nodes are to be visited in a queue and gray nodes are visited.

Breadth First Search



Green nodes are to be visited in a queue and gray nodes are visited.

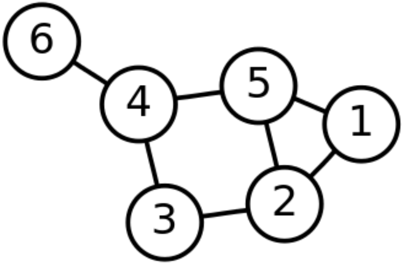
Breadth First Search

Algorithm 1 Breadth First Search

```
Input: v is not yet visited
Initialize an empty queue
Q.enqueue(v)
change v to visited
while Q.empty()=false do
    w ← Q.dequeue()
    if w is not visited then
        for ∀u are the adjacent vertices of w do
            change u to visited
            Q.enqueue(u)
        end for
    end if
end while
```

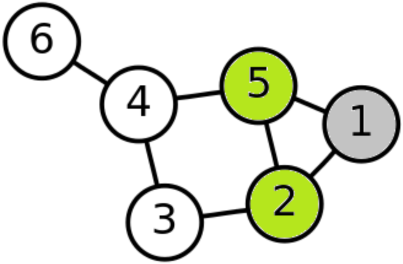
```
void BFS(int v){
    list<int> queue;
    queue.push_back(v);
    visited[v] = 1;
    while(!queue.empty()){
        int w = queue.front();
        queue.pop_front();
        for(int j=0;j<N;j++)
            if(!visited[j] && G[w][j]==1){
                visited[j] = 1;
                queue.push_back(j);
            }
    }
}
```

1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

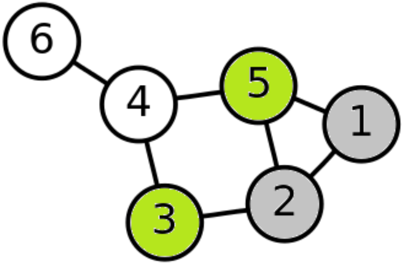
1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

Depth First Search

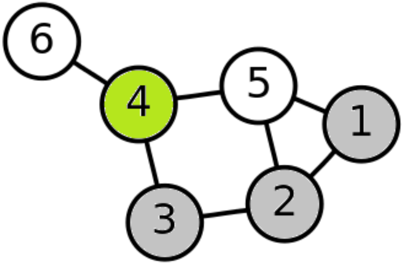
1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

Depth First Search

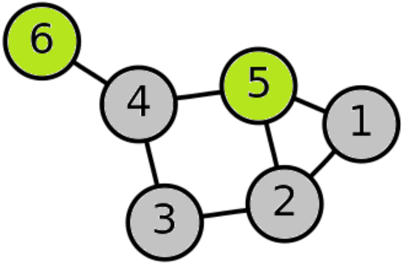
1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

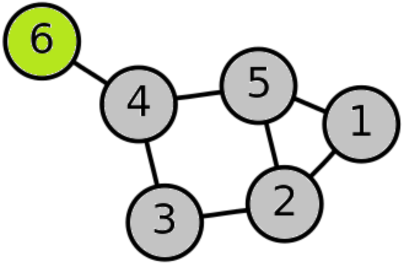
Depth First Search

1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



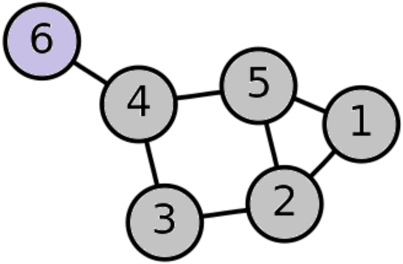
Green nodes are to be visited in a queue and gray nodes are visited.

1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

1. From a vertex $v \in G$, visit an adjacent vertex u of v and u is not visited.
2. This process repeats if there is unvisited adjacent vertex. The process stops when all adjacent vertices are visited.
3. Find all the remain vertices of G which are not visited and repeat the two previous steps.



Green nodes are to be visited in a queue and gray nodes are visited.

Depth First Search

Input: A vertex v is not visited

```
for ∀u are adjacent vertices of v do
    if u is not visited then
        change u to visited
        DFS(u)
    end if
end for
```

Depth First Search

Code C/C++

```
void DFS(int i){
    int j;
    printf("\n%d",i+1);
    visited[i]=1;
    for(j=0;j<n;j++)
        if(!visited[j]&&G[i][j]==1)
            DFS(j);
}
```

Shortest Path Algorithms

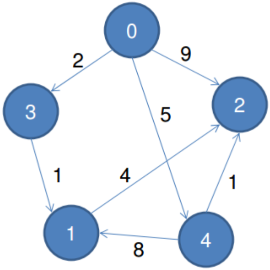
Find shortest paths in graphs

- Unweighted graph: BFS or DFS can be applied
- Weighted graph:
  - find a shortest path from a source to all other vertices: single-source shortest path
  - find a shortest path between pairs of vertices: all-pairs shortest path

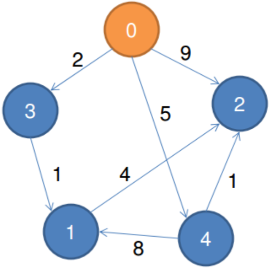
Dijkstra's Algorithm

Dijkstra's Algorithm is a single-source shortest path approach:

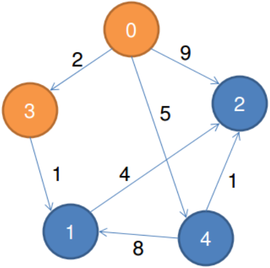
- put the starting vertex (or the source) into S.
- find the vertext s which has the total distance to all vertices from S (the total distance is accumulated from the source), then put s into S.
- This process repeats untill all the vertices are found in S.



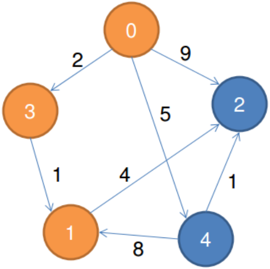
Dijkstra's Algorithm



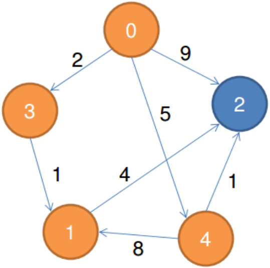
- $S = \{0\}$
- $D[1] = w_{01} = \infty$
- $D[2] = w_{02} = 9$
- $D[3] = w_{03} = 2$
- $D[4] = w_{04} = 5$



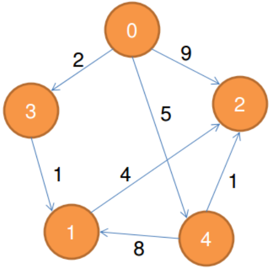
- $S = \{0, 3\}$
- $D[1] = min(\infty, D[3]) = 3$
- $D[2] = min(9, D[3] + \infty) = 9$
- $D[3] = 2$
- $D[4] = min(5, D[3] + \infty) = 5$



- $S = \{0, 3, 1\}$
- $D[1] = 3$
- $D[2] = \min(9, D[1] + 4) = 7$
- $D[3] = 2$
- $D[4] = \min(5, D[1] + \infty) = 5$



- $S = \{0, 3, 1, 4\}$
- $D[1] = 3$
- $D[2] = \min(7, D[4] + 1) = 6$
- $D[3] = 2$
- $D[4] = 5$



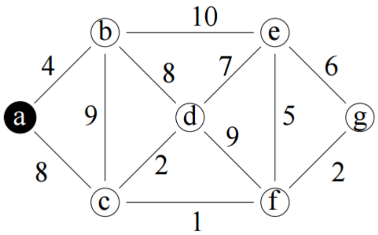
- $S = \{0, 3, 1, 4, 2\}$

This is the shortest path in this graph given by the Dijkstra's Algorithm

Dijkstra's Algorithm

Require: A vertex $u \in V$

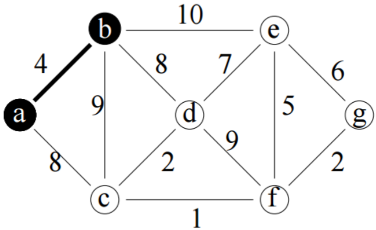
create vextex set Q
while Q is not empty **do**
    remove u from Q
    for all neighbor v of u **do**
        $alt \leftarrow D[u] + w(v, u)$
        if $alt < D[v]$ **then**
            $D[v] \leftarrow alt$
            $prev[v] \leftarrow u$
        end if
    end for
end while



- $S = \{a\}$
- $V \setminus S = \{b, c, d, e, f, g\}$
- $d_{min} = d(a, b) = 4$

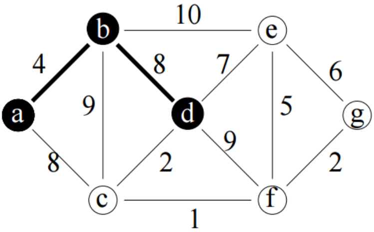
$\rightarrow S = S \cup b$

Prim's Algorithm



- $S = \{a, b\}$
- $V \setminus S = \{c, d, e, f, g\}$
- $d_{min} = d(b, d) = d(a, c) = 8$

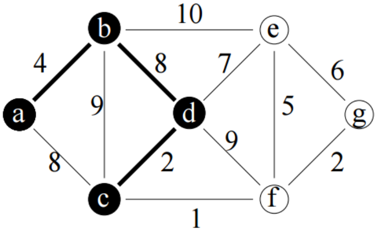
$\rightarrow S = S \cup d$ (hoc c)



- $S = \{a, b, d\}$
- $V \setminus S = \{c, e, f, g\}$
- $d_{min} = d(d, c) = 2$

$\rightarrow S = S \cup c$

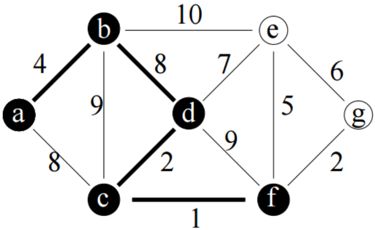
Prim's Algorithm



- $S = \{a, b, d, c\}$
- $V \setminus S = \{e, f, g\}$
- $d_{min} = d(c, f) = 1$

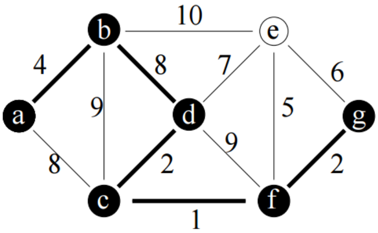
$\rightarrow S = S \cup f$

Prim's Algorithm



- $S = \{a, b, d, c, f\}$
- $V \setminus S = \{e, g\}$
- $d_{min} = d(f, g) = 2$

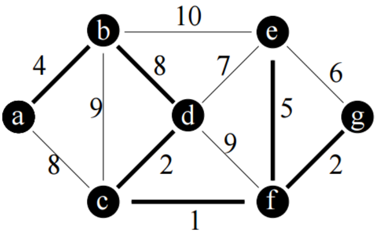
$\rightarrow S = S \cup g$



- $S = \{a, b, d, c, f, g\}$
- $V \setminus S = \{e\}$
- $d_{min} = d(f, e) = 5$

$\rightarrow S = S \cup e$

Prim's Algorithm



- $S = \{a, b, d, c, f, g, e\}$
- $E_S = \{(a, b), (b, d), (d, c)$
  $, (c, f), (f, g), (g, e)\}$
- MST (Minimum Spanning Tree) $= (S, E_S)$