Algorithms on graphs
Module 1

Lecture 2
Graph traversals: depth-first search, breadth-first search and their applications. Part 1

Adigeev Mikhail Georgievich

mgadigeev@sfedu.ru

Graph traversals

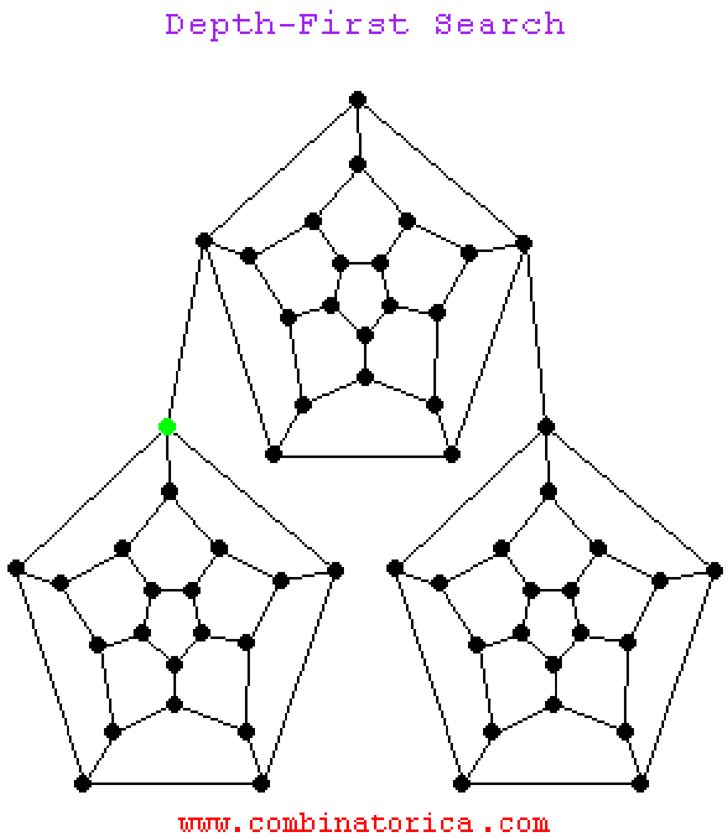
Graph G=(V,E).

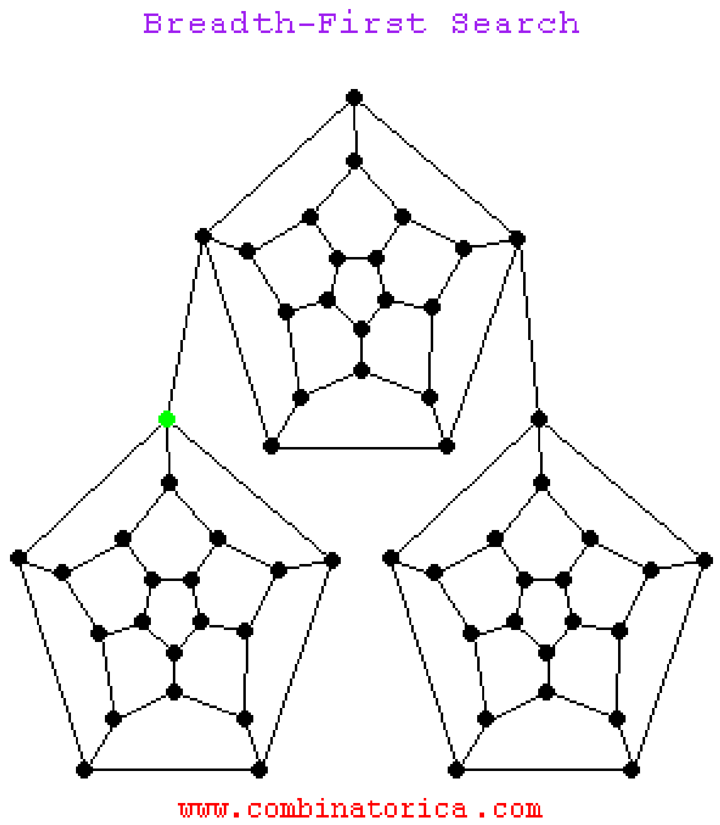
A graph *traversal*: start at a certain vertex and visit other vertices of G in a specific order.

Traversals let us explore the graph and discover its structure.

- Depth-first traversal (DFS)
- Breadth-first traversal (BFS)

Graph traversals





https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

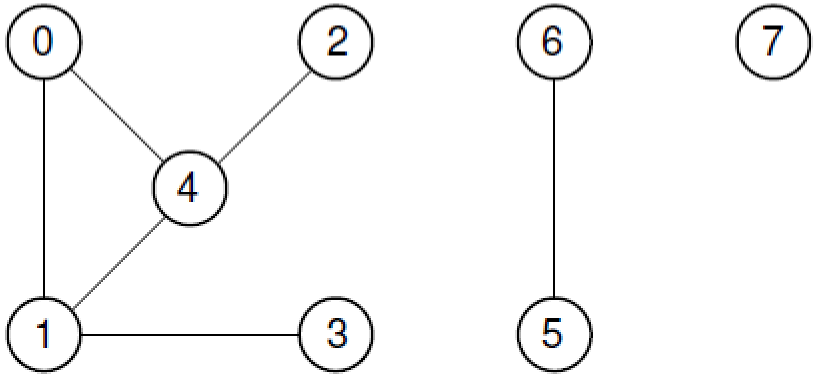
Graph connectivity

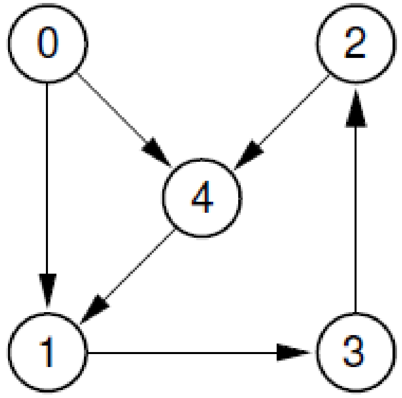
Graph G=(V,E).

A *path* (*walk*) is a sequence of edges $\{e_1, e_2, \ldots, e_l\}$ such that for each i the end-point vertex of e_i is a start-point of e_{i+1}.

Alternative representation: a sequence of vertices $\{v_1, v_2, \ldots, v_{l+1}\}$.

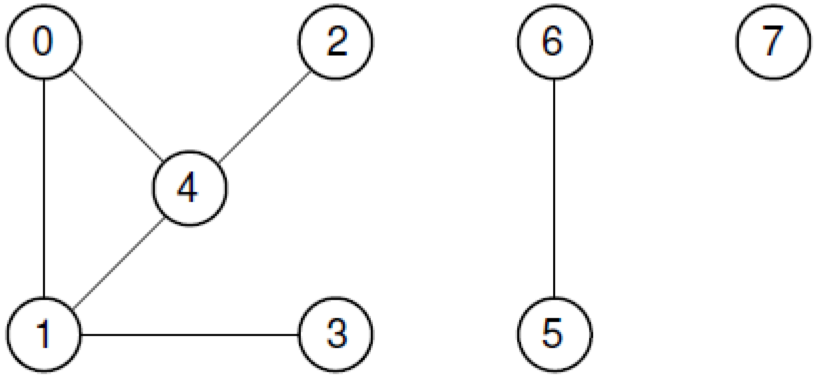
The number of edges = *length* of the path.

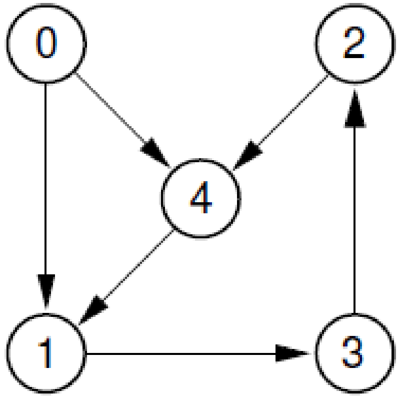




Graph connectivity

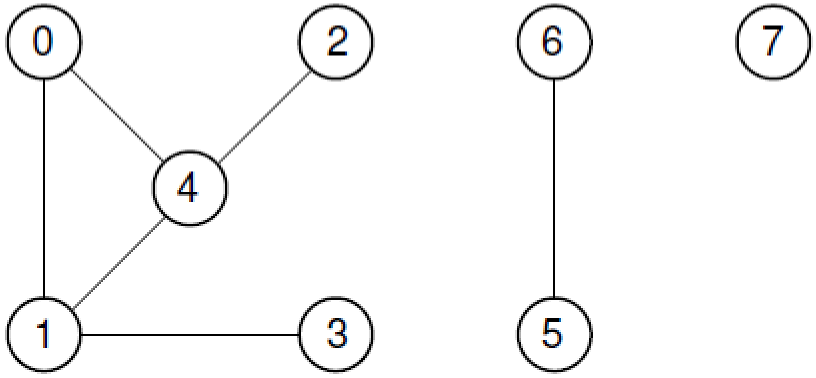
- A path $\{v_1, v_2, \ldots, v_{l+1}\}$ is a *cycle* iff $v_1 = v_{i+1}$.
- A vertex v is *reachable* from the vertex u on G iff there is a path on G from u to v .

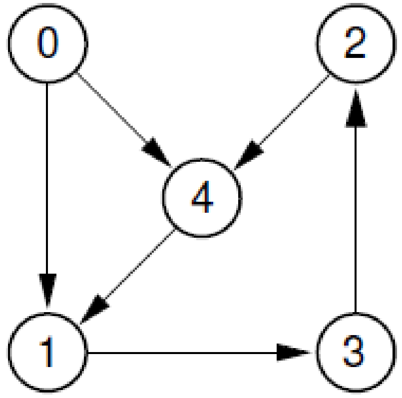




Graph connectivity

- A graph is called *(strongly) connected* iff for each pair of vertices $\{u, v\}$ there is a path between u and v.
- The maximally connected subgraphs of *G* are called *(strong) connected components*.



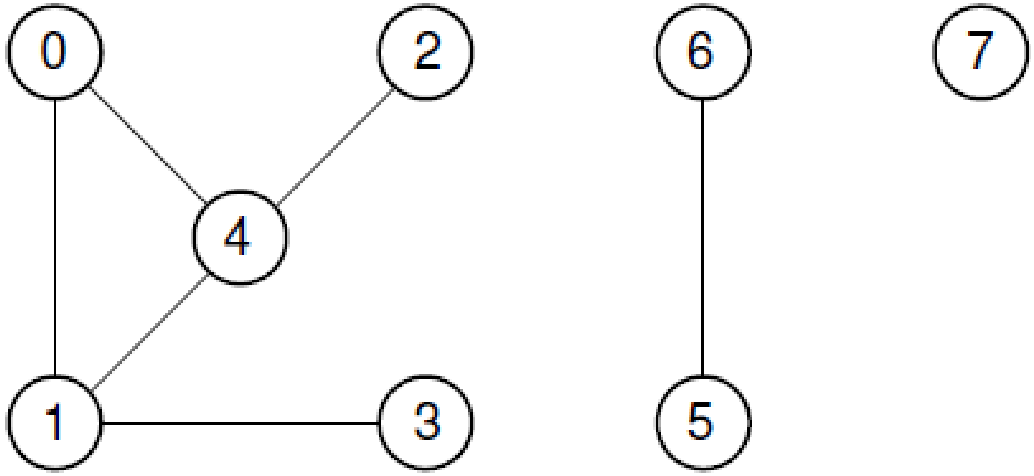


Graph connectivity

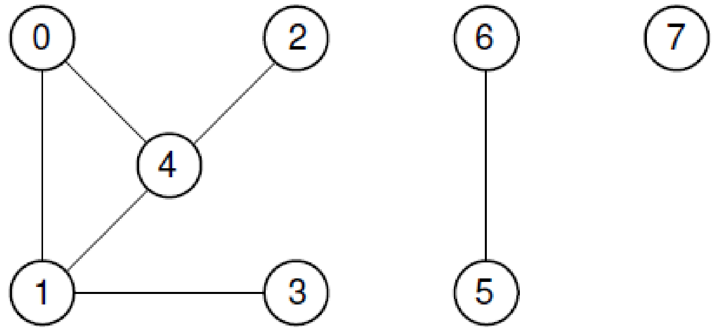
<u>Problem</u>

Given a graph $G(V, E)$, detect all its connected components.

1. $\{0, 1,2,3,4\}$
2. {5,6}
3. {7}



Graph connectivity

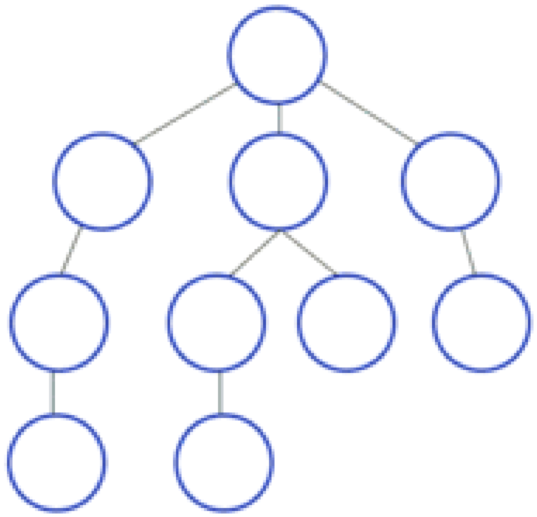


<u>Solution</u>

1. Mark all vertices as ‘unvisited’.
2. While there is an unvisited vertex *s*:
3. Initialize a new component C_k.
4. Start DFS/BFS from *s*.
5. Visiting a vertex, put it into C_k.

DFS: Depth-First Search

Visiting a vertex v, recursively visit (start DFS) each of its unvisited neighbors.



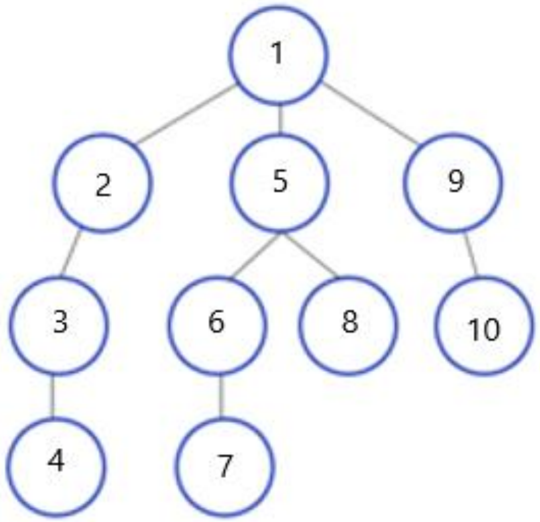
https://en.wikipedia.org/wiki/Depth-first_search

```
DFS(v)
Mark v as 'visited'
For each u in Adj(v):
    if u is unvisited:
        DFS(u)
```

DFS: Depth-First Search

Visiting a vertex v, recursively visit (start DFS) each of its unvisited neighbors.

```
DFS(v)
Mark v as 'visited'
For each u in Adj(v):
    if u is unvisited:
        DFS(u)
```



https://en.wikipedia.org/wiki/Depth-first_search

DFS: Depth-First Search

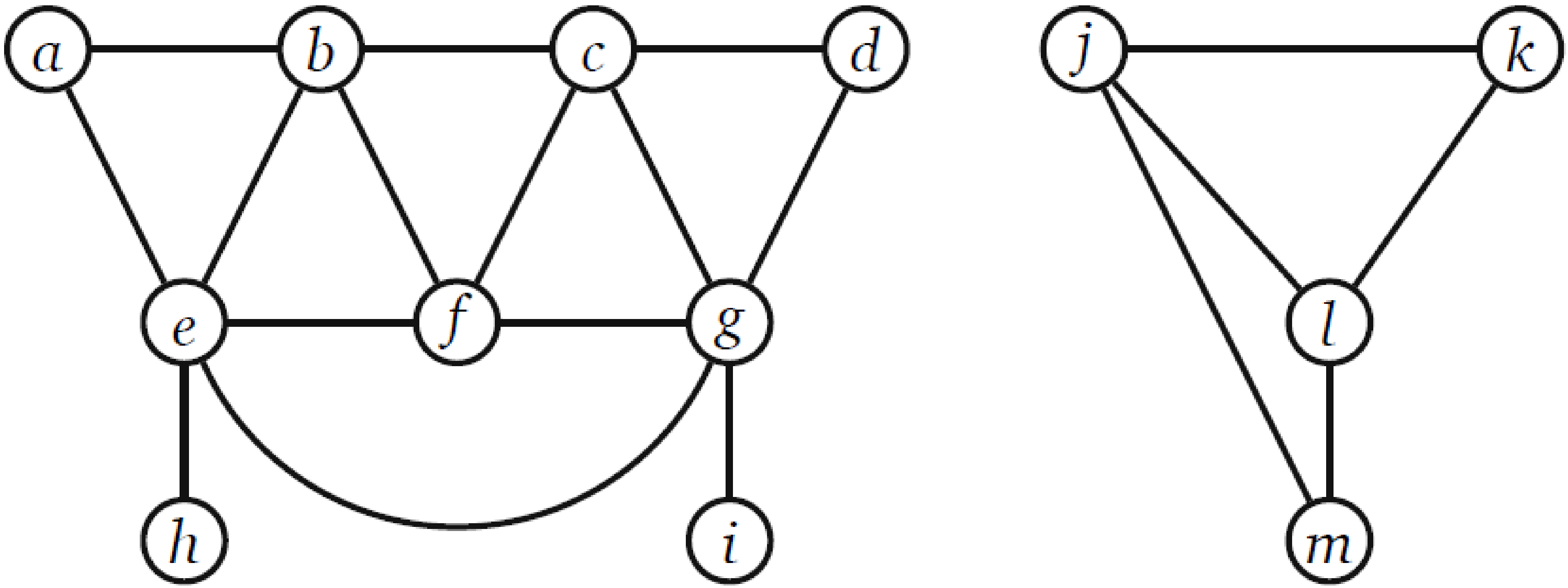
For graph exploration, we often need to perform some processing before / after recursive DFS.

```
DFS(v)
PreVisit(v)
Mark v as 'visited'
For each u in Adj(v):
   if u is unvisited: DFS(u)
PostVisit(v)
```

DFS: explicit stack implementation

```
StackDFS(G)
Select s ∈ V
Push(s)
While (stack is not empty):
    v = Pop()
    if v is unvisited:
          Mark v as 'visited'
          For each u in Adj(v):
                Push(u)
```

DFS: example



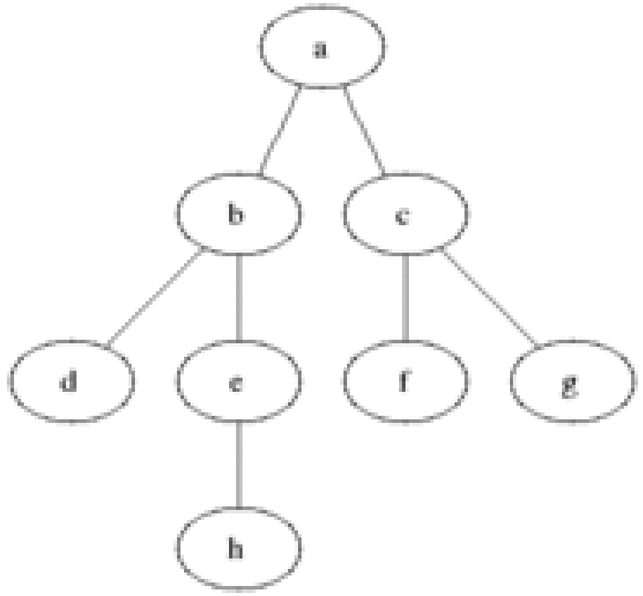
BFS: Breadth-First Search

Visiting a vertex v,

visit each of its unvisited neighbors,

then neighbors of the neighbors,

etc.

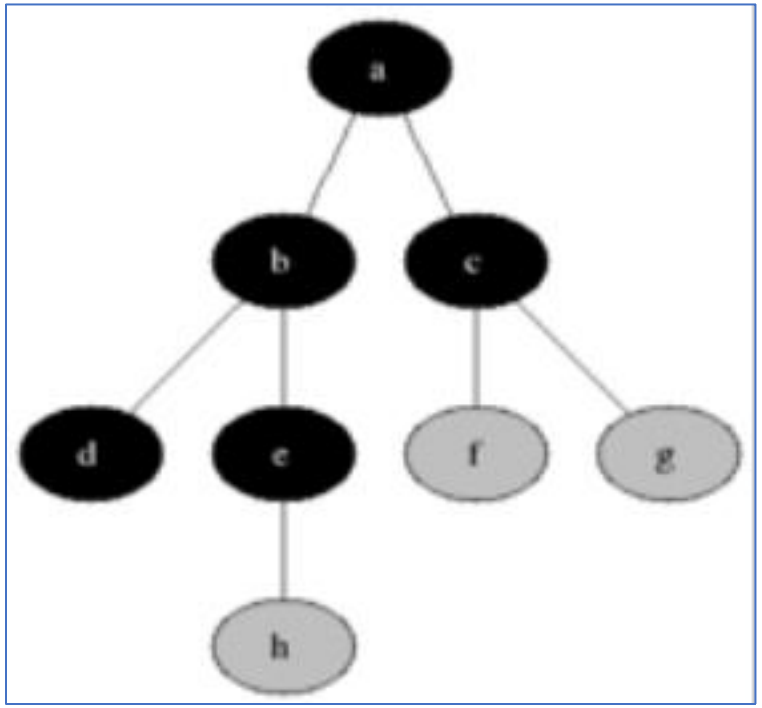


https://en.wikipedia.org/wiki/Breadth-first_search

BFS: Breadth-First Search

For keeping this order of visiting, we need to store neighbor vertices until we get them for processing.

We need a queue.



https://en.wikipedia.org/wiki/Breadth-first_search

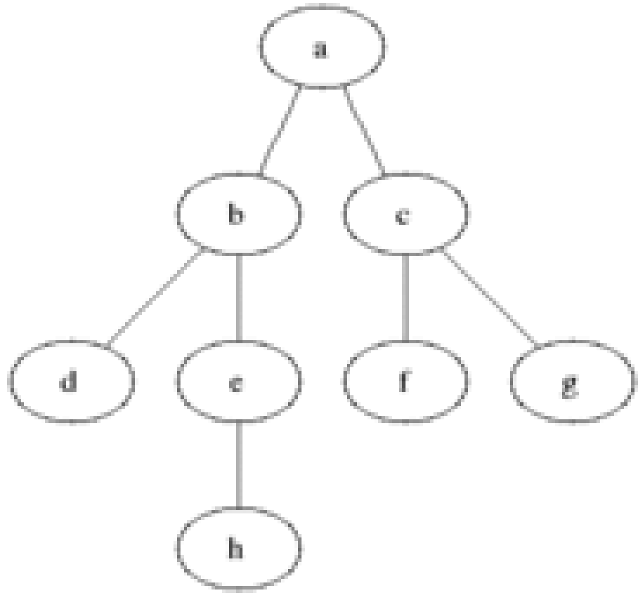
BFS: queue-based implementation

```
BFS(G)
Select s ∈ V
Enqueue(s)
While (Queue is not empty):
    v = Dequeue()
    if v is unvisited:
        Mark v as 'visited'
        For each u in Adj(v):
            Enqueue(u)
```

BFS: applications

1) Detecting connected components.
2) Calculating distances.

Principal idea: visiting a vertex v,
visit each of its unvisited neighbors,
then neighbors of the neighbors,
etc.



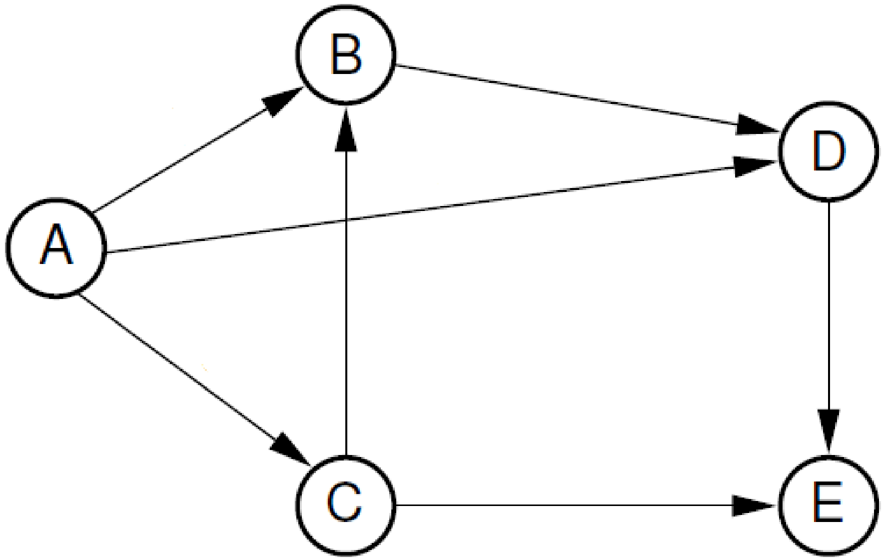
https://en.wikipedia.org/wiki/Breadth-first_search

BFS: applications

Graph G=(V,E).

A *distance* between vertices *u* and *v* is the minimum length of the path between *u* and *v*.

dist(A,E) = 2

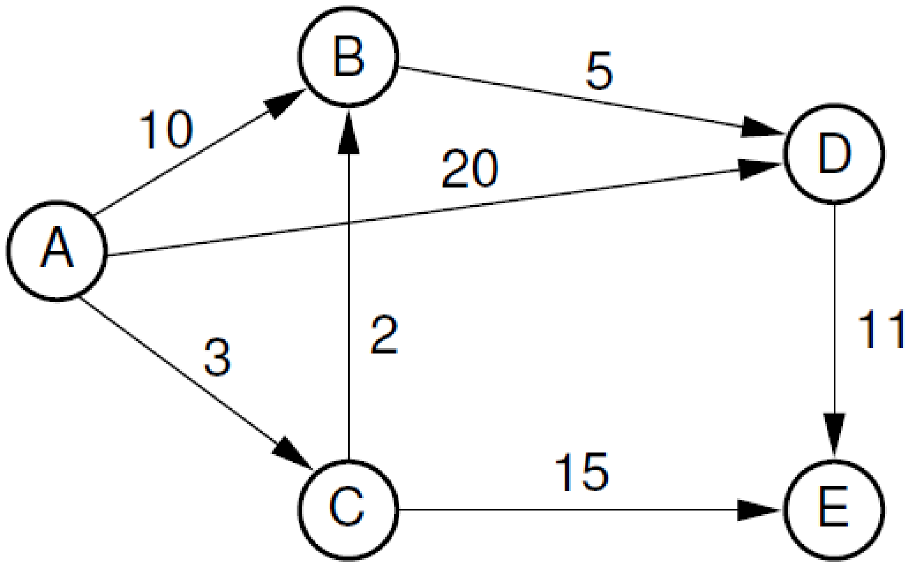


BFS: applications

Weighted graph G=(V,E), $w: E \to R$

A *distance* between vertices *u* and *v* is the minimum weight (=sum of edges' weights) of the path between *u* and *v*.

dist(A,E) = 18



BFS: applications

For unweighted graphs distances from $s \in V$ to all other vertices can be calculated using BFS.

For weighted graphs: Dijkstra's algorithm works like BFS and calculates distances (from $s \in V$ to all other vertices) on a graph.