I111E Algorithms & Data Structures 10. Graph Algorithms (1): Graph Representations, Breadth-First Search and Depth-First Search

School of Information Science

Ryuhei Uehara & Giovanni Viglietta

uehara@jaist.ac.jp & johnny@jaist.ac.jp

2019-11-20

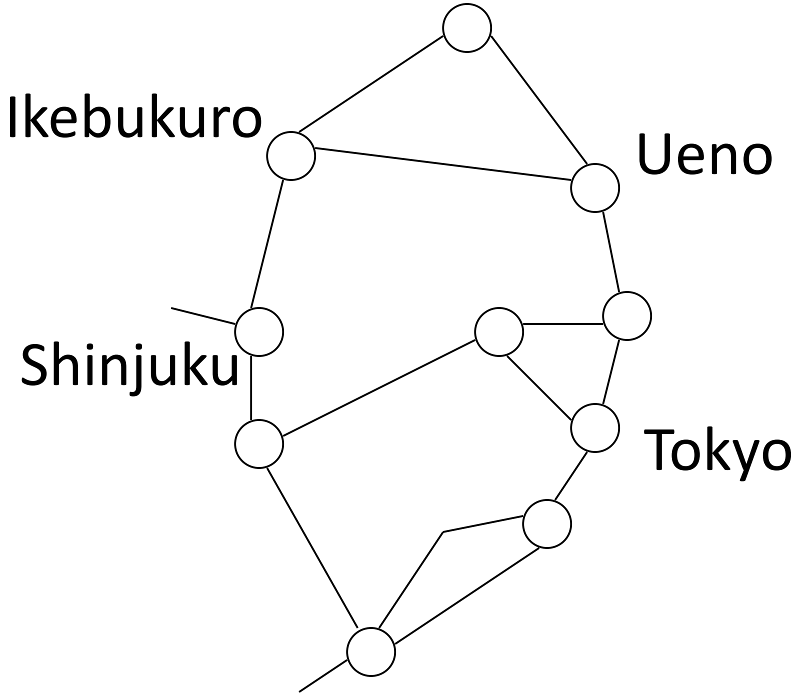
All materials are available at

http://www.jaist.ac.jp/~uehara/couse/2019/i111e

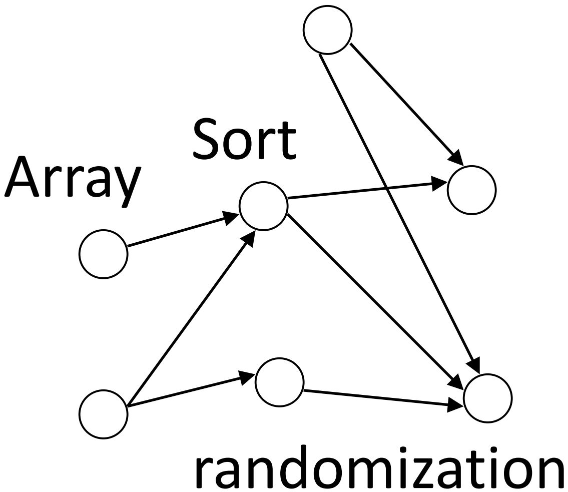
Graph

- “Vertices” (nodes) are joined by edges (arcs)
  - Directed graph: each edge has direction
  - Undirected graph: each edge has no direction

Example: railway in Tokyo

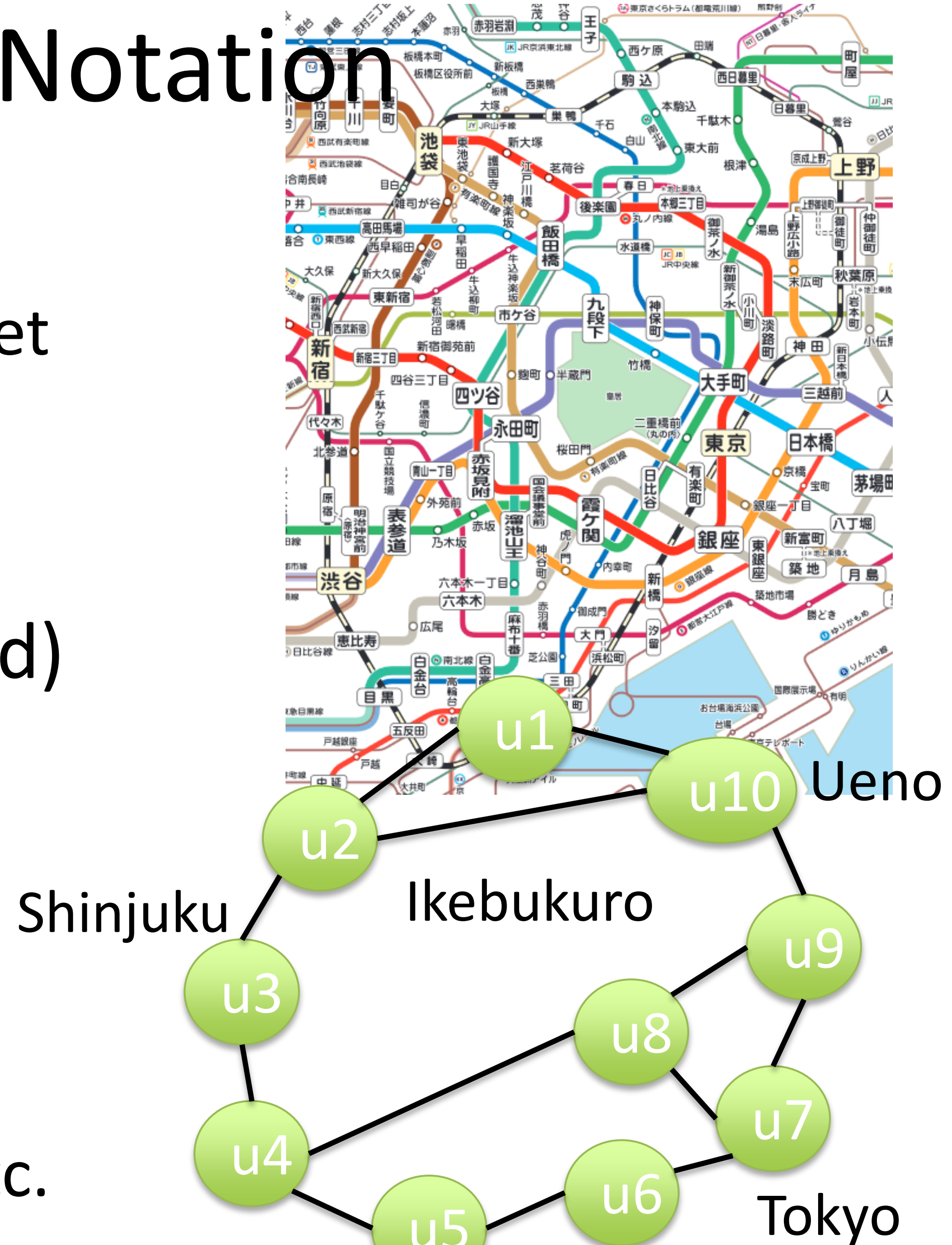


Example:
relationship between topics



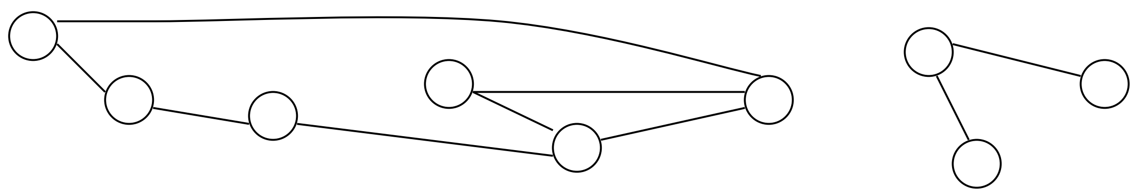
Graph: Notation

- Graph $G = (V, E)$
  - V: vertex set, E: edge set
- Vertices: $u, v, \ldots \in V$
- Edges: $e = \{u, v\} \in E$
  (undirected)
  $a = (u, v) \in E$
  (directed)
- Weighted variants;
  - $w(u), w(e)$
  - Distance, cost, time, etc.

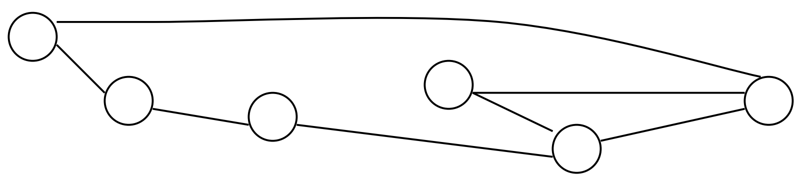


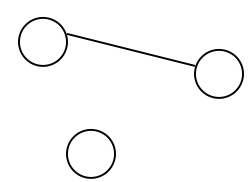
Graph: basic notions/notations (1/2)

- Path: sequence of vertices joined by edges
  - Simple path: it never visit the same vertex again



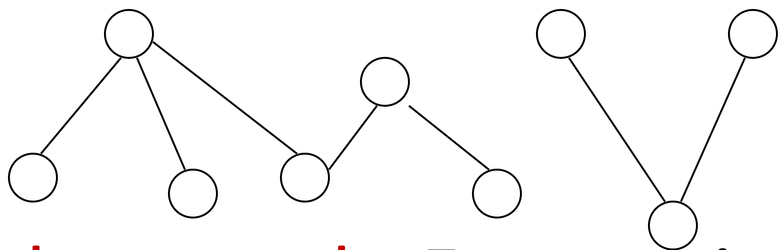
- Cycle, closed path: path from *v* to *v*
- Connected graph: Every pair of vertices is joined by a path



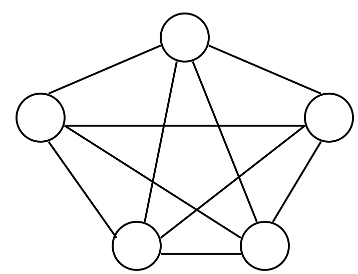


Graph: basic notions/notations (2/2)

- Forest: Graph with no cycle (acyclic)
- Tree: Connected acyclic graph



- Complete graph: Every pair of vertices is connected by an edge, denoted by K_n, where n is the number of vertices.
  – Example: K_5



Computational complexity of graph problems

- The number n of vertices, the number m of edges;
  - Undirected graph: $m \leqq n(n\text{-}1)/2$
  - Directed graph: $m \leqq n(n\text{-}1)$
    - $m \in O(n^2)$

- Every tree has m=n-1 edges, so $m \in O(n)$.

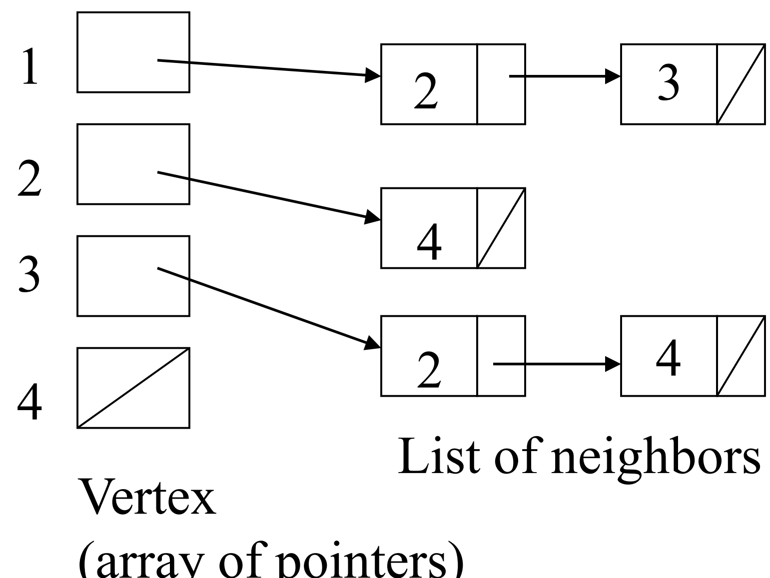
- Computational complexity of graph algorithm is described by equations of n and m.

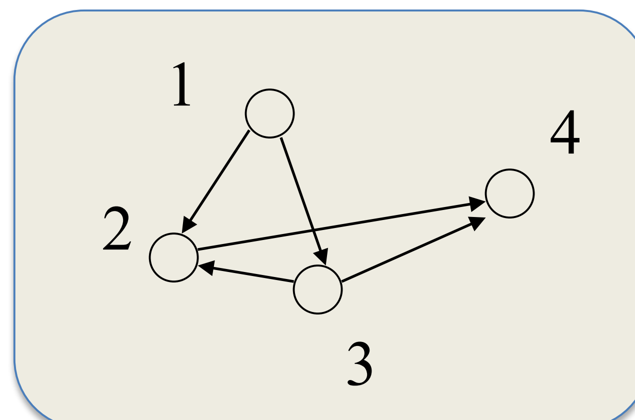
Representative representations of a graph in computer

- Adjacency matrix

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

- Adjacency list

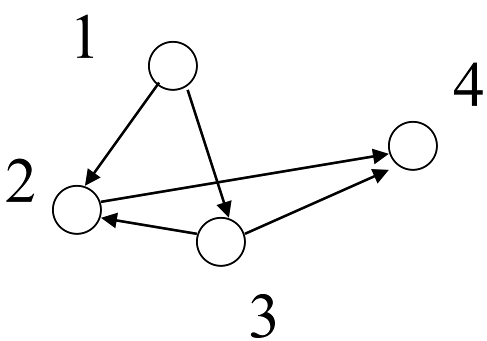




Representation of a graph:
matrix representation (adjacency matrix)

- $(u, v) \in E \Rightarrow M[u, v] = 1$
- $(u, v) \notin E \Rightarrow M[u, v] = 0$

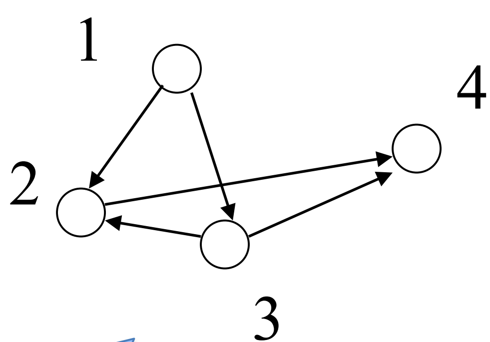
It is easy to extend edge-weighted graph.

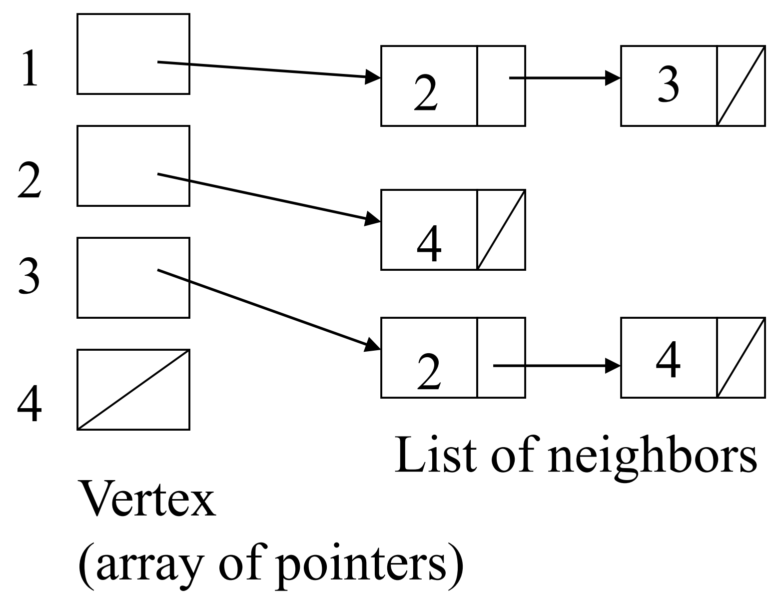


$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Representation of a graph: list representation (adjacency list)

- $(u, v) \in E \Leftrightarrow v \in L(u)$
  - $L(u)$ is the list of neighbors of u





It is easy to extend vertex-weighted graph.

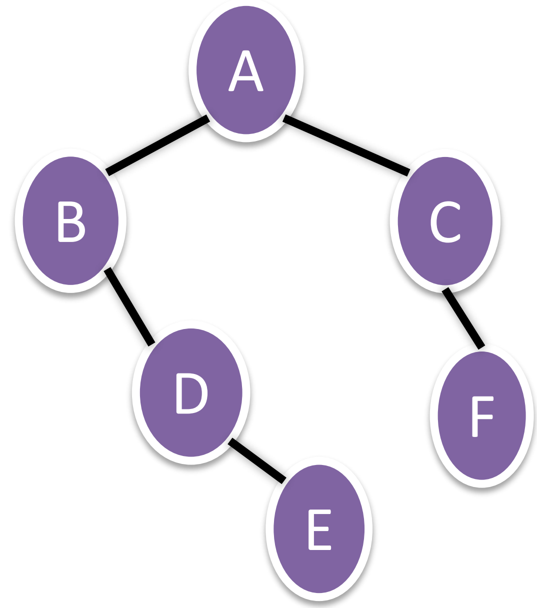
Adj. matrix v.s. Adj. list

- Space complexity
  - Adjacency matrix: $\Theta(n^2)$
  - Adjacency list: $\Theta(m \log n)$
- Time complexity of checking if $(u, v) \in E$?
  - Adjacency matrix: $\Theta(1)$
  - Adjacency list : $\Theta(n)$

Q. How about update graph?
(e.g., add/remove vertex/edge)

Search in Graph

Search in Graph



- How can we check all vertices in a graph systematically, and solve some problem?
  - e.g., Do you have a path from A to D?
- Two major (efficient) algorithms:
  - Breadth-First Search: A -> B -> C -> D -> F -> E
    it starts from a vertex v, and visit all (reachable) vertices from the vertices closer to v.
  - Depth-First Search: A -> B -> D -> E -> C -> F
    it starts from a vertex v, and visit every reachable vertex from the current vertex, and back to the last vertex which has unvisited neighbor.

BFS (Breadth-First Search)

- For a graph G=(V,E) and any start point s∈V, all reachable vertices from s will be visited from s in order of distance from s.
- Outline of method: color all vertices by white, gray, or black as follows;
  - White: Unvisited vertex
  - Gray: It is visited, but it has unvisited neighbors
  - Black: It is already visited, and all neighbors are also visited

  - Search is completed when all vertices got black
  - Color of each vertex is changed as white→gray→black

BFS (Breadth-First Search): Program code

```
BFS(V,E,s){
  for v∈V do toWhite(v); endfor
  toGray(s);
  Q={s};
  while( Q!={} ){
    u=pop(Q); // Q → Q' where Q={u}UQ'
    for v∈{v∈V|(v,u)∈E}
      if isWhite(v) then
        toGray(v); push(Q,v);
      endif
    endfor
    toBlack(u);
  }
}
```

Queue is the best structure!

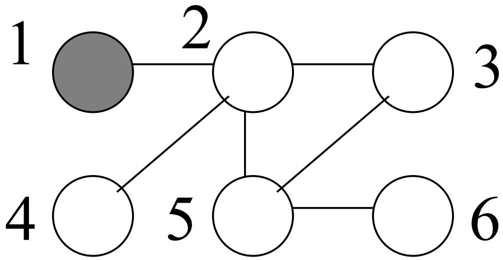
Take u from left (the first gray node)

If neighbor v of u is white,

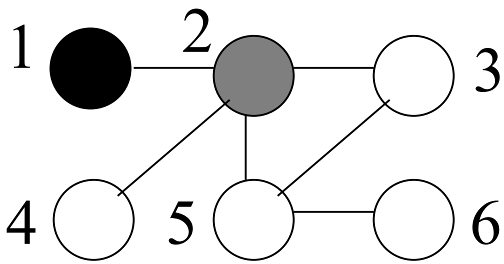
Push v to the right of Q (processed lastly)

Make u black when it has no unvisited neighbors

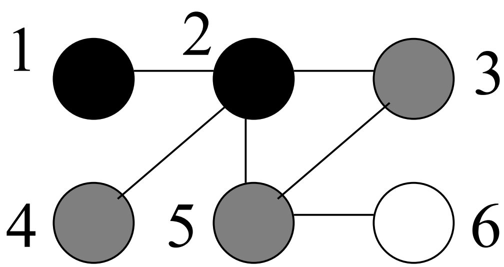
BFS (Breadth-First Search): Example



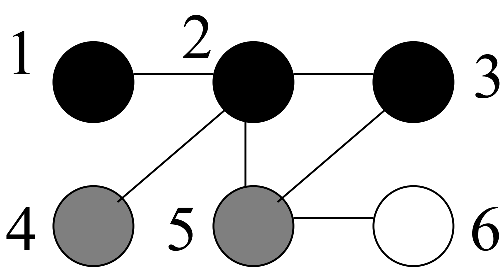
Q={1}



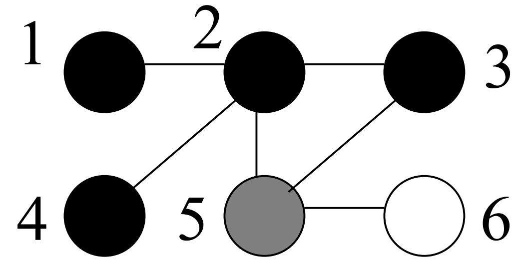
u=1,
visit 2
Q={2}
black 1



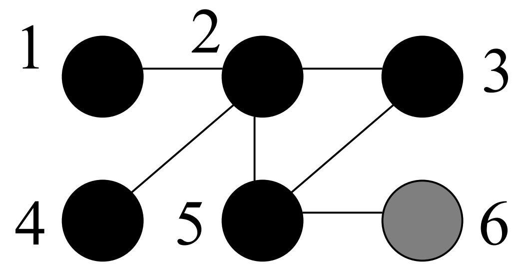
u=2,
visit 3,4,5
Q={3,4,5}
black 2



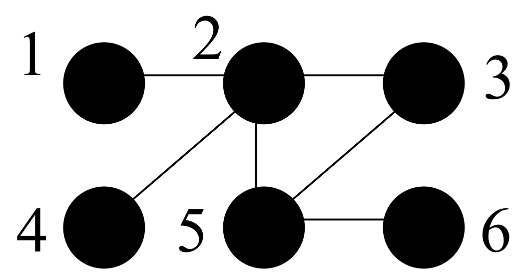
u=3,
visit null
Q={4,5}
black 3



u=4,
visit null
Q={5}
black 4



u=5,
visit 6
Q={6}
black 5



u=6,
visit null
Q={}
black 6

BFS: Time complexity

Time complexity is not easy from program…

Consider from the viewpoints of vertices & edges

- Each vertex never gets white again after initialization.
- Each vertex gets into Q and gets out from Q at most once
- Each edge is checked at most once
  - when one endpoint vertex is taken from Q and its neighbors are checked along edges
- $\therefore O(|V| + |E|)$

```
BFS(V,E,s){
  for v∈V do
    toWhite(v);
  endfor
  toGray(s);
  Q={s};
  while( Q!={} ){
    u=pop(Q);
    for v∈{v∈V|(v,u)∈E}
      if isWhite(v) then
        toGray(v);
        push(Q,v);
      endif
    endfor
    toBlack(u);
}}
```

It's not easy to do efficiently in adj. matrix

Application of BFS: Shortest path problem on graph

Definition of “distance”

- Start vertex v has distance 0
- Except start vertex, each vertex u has distance d+1, where d is the distance of parent of u.

• On BFS, modify that each gray vertex receives its “distance” from black neighbor, then you get (shortest) distance from v to it.

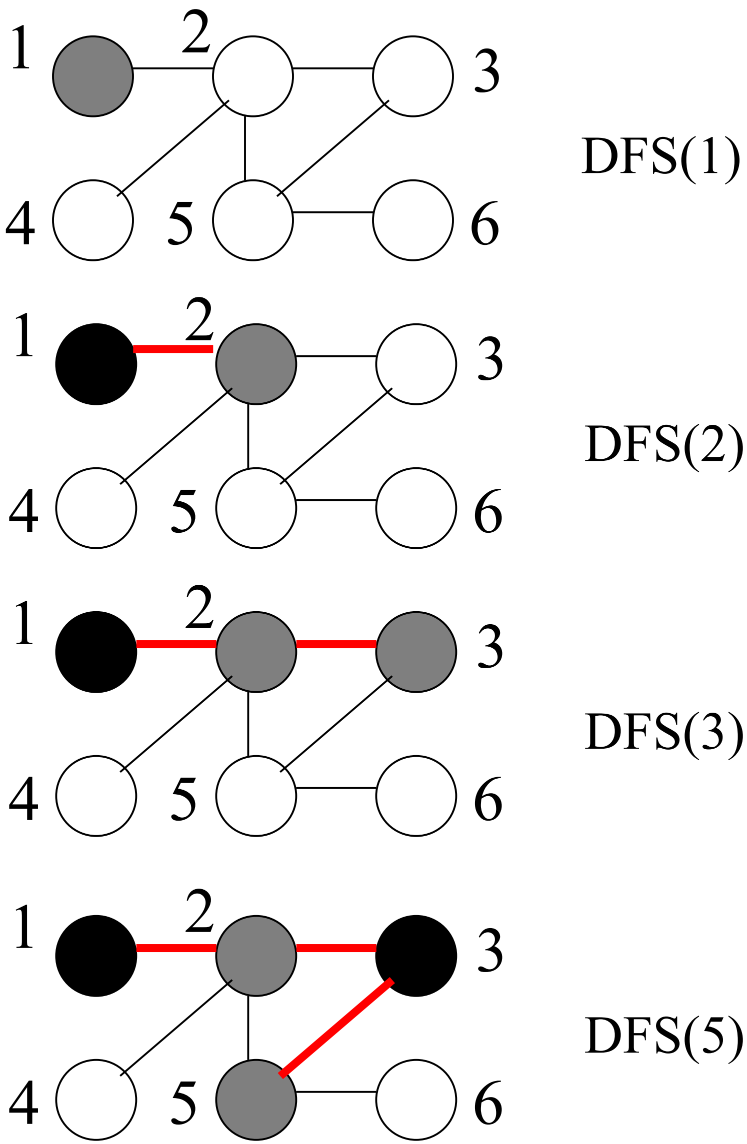
DFS (Depth-First Search)

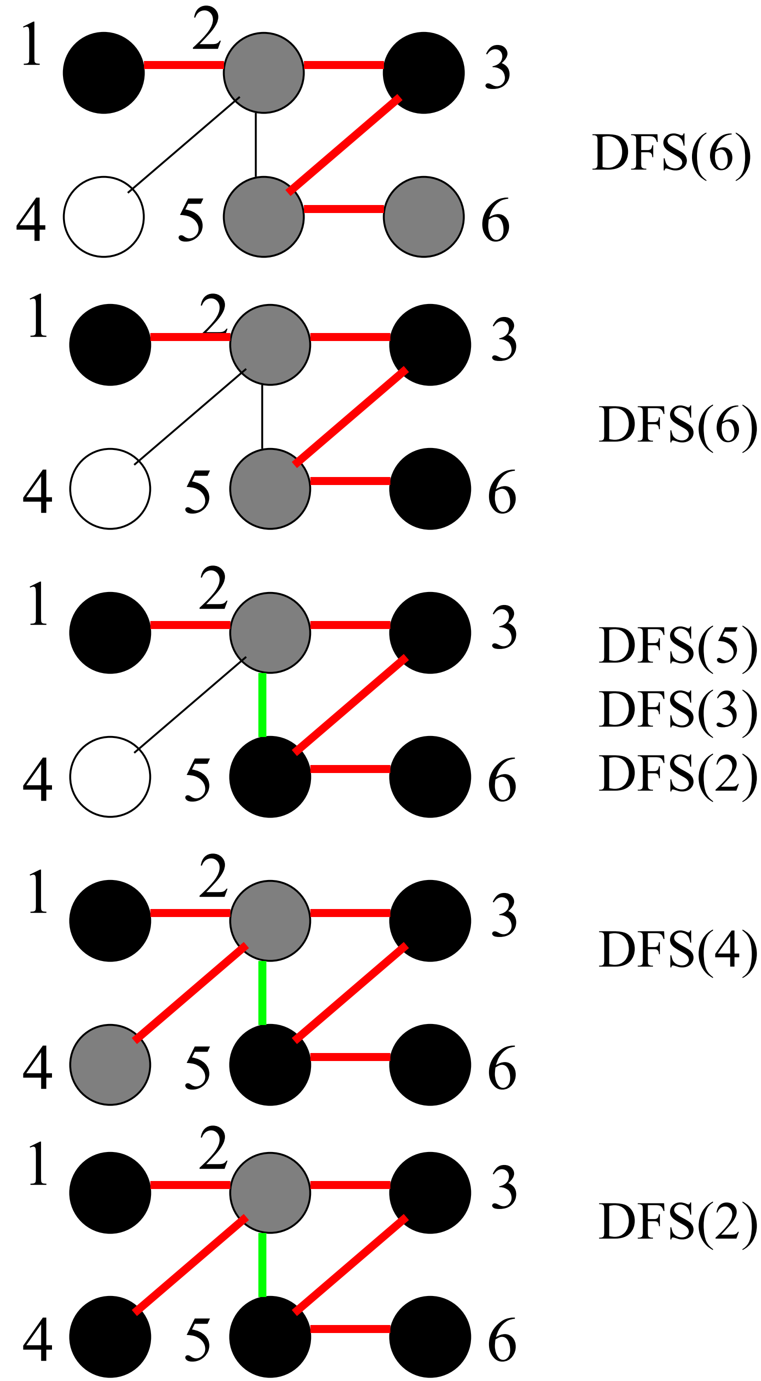
- For a graph G=(V,E) and start point s∈V, it follows reachable vertices from s until it reaches a vertex that has no unvisited neighbor, and returns to the last vertex that has unvisited neighbors.

```
dfs(V, E, s) {
    visit(s) // make gray
    for  (s, w)∈E  do
        if  notVisited(w)  then
            dfs(V, E, w)
    toBlack(u)
}
```

Program code is relatively simple, and vertices are put into a stack when dfs makes a recursive call.

DFS: Example





DFS non-recursive version

: We can use stack explicitly to search a tree

```
DFS(V,E,s){
  for v∈V do toWhite(v); endfor
  toGray(s);
  S={s};
  while( S!={} ){
    u=pop(S);
      for v∈{v∈V|(u,v)∈E}
      if isnotBlack(v) then
        toGray(v); push(S,v);
      endif
    endfor
    toBlack(u);
  }
}
```

stack of gray nodes

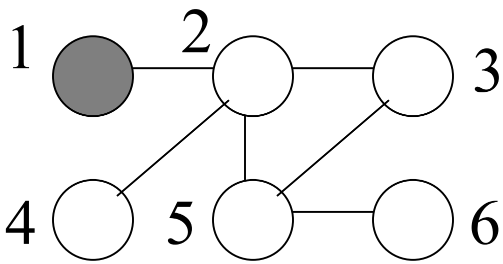
Pop u from top (last node in gray)

If neighbor v of u is not black

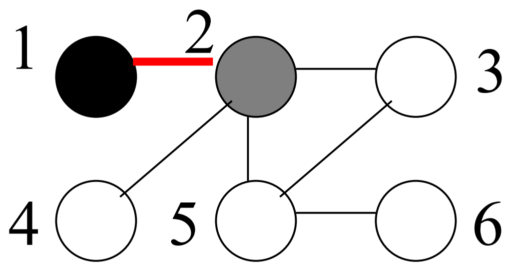
Push v into S on top (which will be processed at first)

Make u black when it has no unvisited neighbors

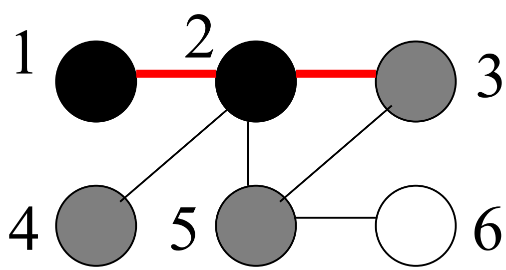
Example (non-rec. ver.)



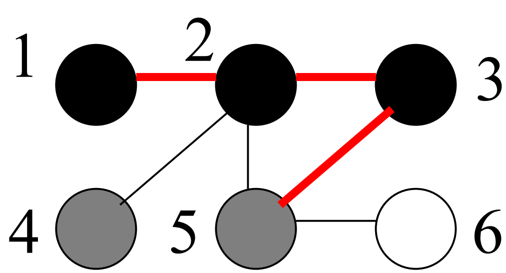
S={1}



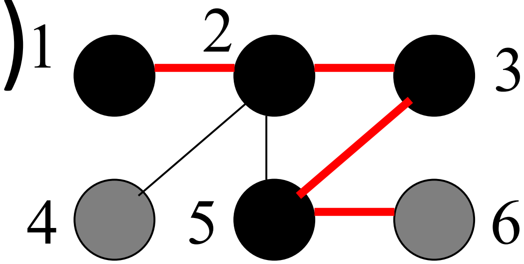
u=1
visit 2
S={2}
black 1



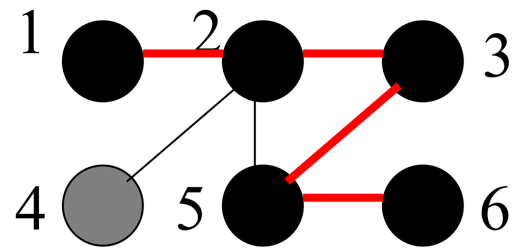
u=2
visit 5,4,3
S={5,4,3}
black 2



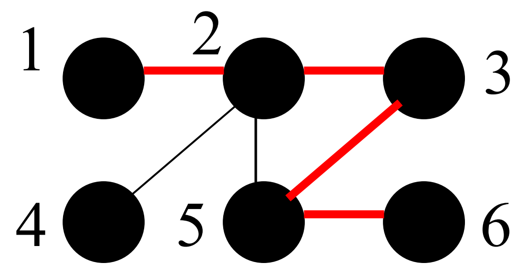
u=3
visit 5
S={5,4,5}
black 3



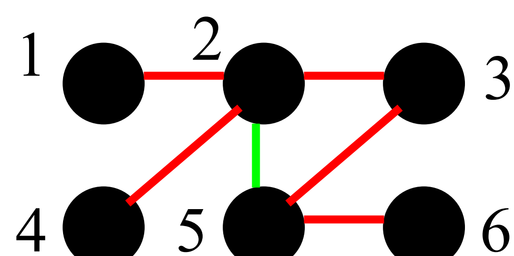
u=5
visit 6
S={5,4,6}
black 5



u=6
visit null
S={5,4}
black 6



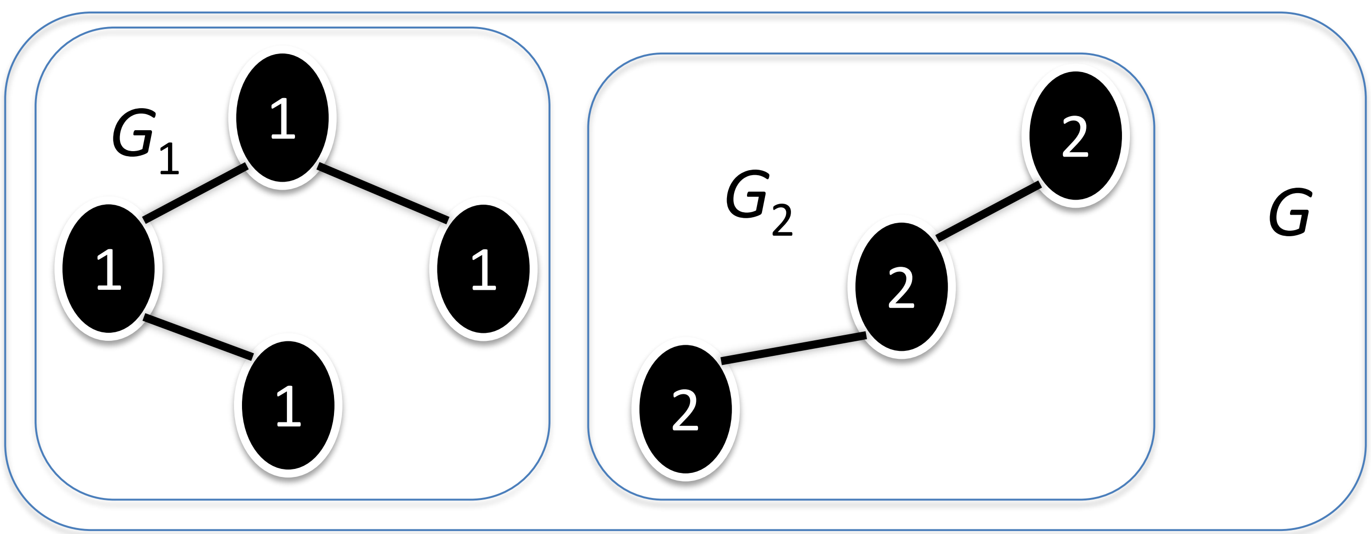
u=4
visit null
S={5}
black 4



u=5
visit null
S={}

Application of DFS: Find connected components in a graph

- For a given (disconnected) graph $G = (V, E)$, divide it into connected graphs $G_1 = (V_1, E_1), \ldots, G_c = (V_c, E_c)$.
  - We will give a numbering array cn[] such that $\forall u,v \in V,\ u \in V_i \land v \in V_j \land i \neq j \Rightarrow cn[u] \neq cn[v]$



Application of DFS:
Find connected components of a graph

```
cc(V,E,cn){ //cn[|V|]
  for v∈V do
      cn[v] = 0; /*initialize*/
  endfor
  k = 1;
  for v∈V do
    if cn[v]==0 then
      dfs(V,E,v,k,cn);
      k=k+1;
    endif
  endfor
}
```

```
dfs(V,E,v,k,cn){
  cn[v]=k;
  for u∈{u|(v,u)∈E} do
    if cn[u]==0 then
      dfs(V,E,u,k,cn);
    endif
  endfor
}
```

BFS v.s. DFS on a graph

- Two major (efficient) algorithms:
  - Breadth-First Search:

    It is easy to implement by "queue"
  - Depth-First Search:

    It is easy to implement by "stack"
  - Both algorithms are easy to implement to run in O(|V|+|E|) time if you use reasonable data representation and data structure. (This time complexity is optimal since you have to check all input data.)
  - Depending on applications, we choose better algorithm.