Graphs (part ½)

Competitive programming

Prof. Rhadamés Carmona

Concepts

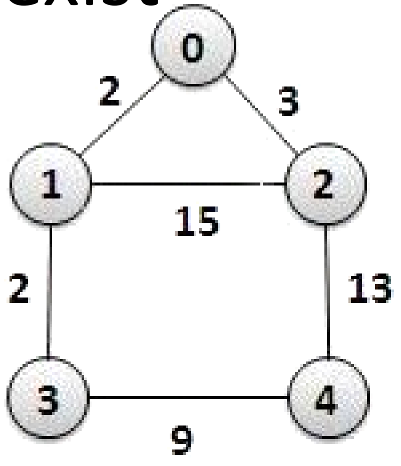
- It is a set of elements called vertices or nodes that are joined by links called edges or arcs.
- Arcs represent relationships between pairs of vertices.
- G = (V, E), V=vertices, E = sides the Edges
- Grade ($v \in V$) = #arcs with v on one extreme
- We can have directed and undirected graphs
- Multigraphs: supports more than one edge (a,b) between 2 vertices

Concepts

- Two vertices are adjacent if there is an edge that joins them
- Two edges are adjacent if they share a common vertex
- An edge can have a weight
- Vertices and edges can be labeled

Representation

- **Adjacency Matrix**: |V|=n. An array A[n][n] is created, where aij is the weight of the edge that joins the vertex i and j, or a "special" value (0, -1, INF, NaN, -INF) if the edge does not exist



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Representation

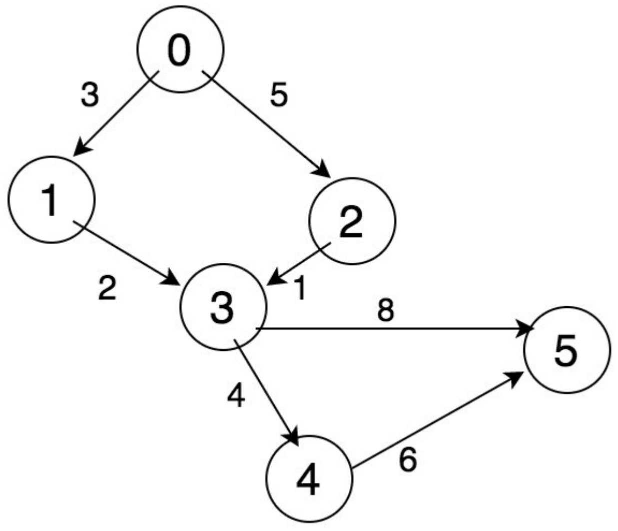
- **Adjacency Matrix:**
- Existence of an arc is O(1)
- Inserting/Updating an Arc is O(1)
- Walking through the adjacency of a vertex is O(n)= O(|V|)
- Deleting or inserting vertex is a problem with this representation
- The top triangular can be stored in case of undirected graph

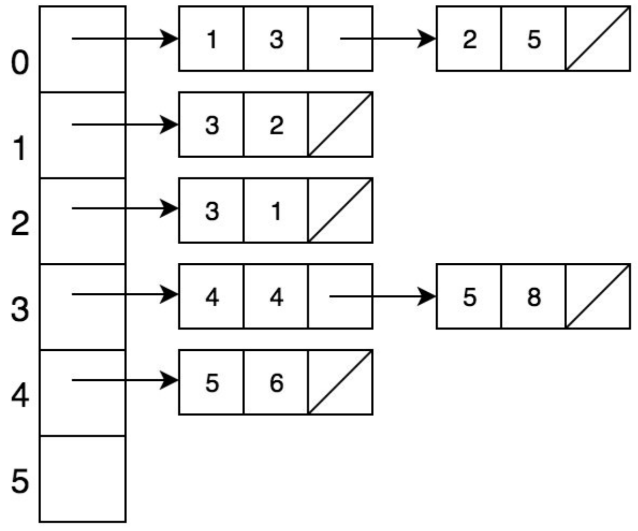
Representation

- **List of arcs:** The arcs are stored in a list of pairs (1,2), (0,3), ...
- Insert Arc is O(1)
- Delete/search arc is O(m) =O(|E|)
- Walking through the adjacency of a vertex is O(E)

Representation

- **Adjacency list:** Vertices can be stored in an array or in a list; for each vertex you have an array or a list of its arcs. Each arc has information about the adjacent vertex (index, pointer, or label), as well as the weight of the arc if applies.





Representation

- **Adjacency list:**
  - Arc Insertion is O(1)
  - Arc search/removal (a,b) are O(|Ady(a)| + | Ady(b)|)
  - Visit to the neighbors of a vertex v is O(|ady(v)|)

DFS

- Deep Search or Depth First Search
- Traverse the nodes with backtracking, discovering other nodes through adjacency. Adjacent nodes are traversed recursively, one by one. Hence the term depth first. The next node is not considered until you have explored all reachable nodes from the current node

DFS

- Typical problems you solve
  - Determine if it is a connected graph
  - Determine connected components
  - Determining the existence of cycles
  - Topological ordering
  - Biconnected graph
  - Strongly connected components
  - Labyrinth generation

DFS

```
typedef vector<vertex> graph;
int count;

void visit(graph  &g, int k) {
      g [k].visited = count++;            // labeled as visited: 0, 1, ...
      for (edge *p = g[k].ady; p!=NULL; p=p->next)
            if (g[p->v].visited == -1)
                  visit(g,p->v);
}

void DFS(graph &g) {
      count = 0;
      for (int i=0; i<g.size(); i++)
             g[i].visited = -1;
      for (int i=0; i<g.size(); i++)
            if (g[i].visited == -1)  // undiscovered
                  visit(g,i);
}
```

```
//also possible like this
vector<vector<edge_info>> g;
```

```
struct vertex
{
            edge *ady;
             int visited = -1;
            …
};
```

```
//also possible like this
class graph: public vector<vertex>
{
            …
};
```

```
struct edge
{
            int v; // vert index
            float weight;
            edge *next;
};
```

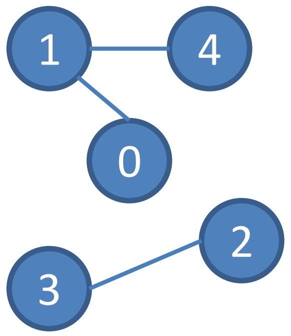
DFS

- Complexity:
  - Using the adjacency list, you walk through the graph visiting each node 1 time, and each arc once: O(n+m)
  - Using the adjacency matrix, O(n^2), because for each node you must visit the n columns in the 2D array

DFS

- **Count connected components:** just count how many calls to visit(i) are made from the DFS function. Each call to visit(i) means that we find an unreachable node from nodes with lower index (j<i). Therefore, g[i] is a pivot to discover a new connected component

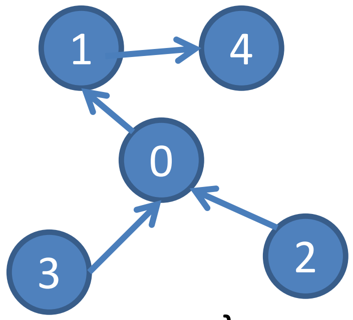
```
int DFS(graph &g) {
        count = 0;
        int components =0;
        for (int i=0; i<g.size(); i++)
                g[i].visited = -1;
        for (int i=0; i<g.size(); i++)
                if (g[i].visited == -1) {
                        components++;
                        visit(g,i);
                }
        return components;
}
```

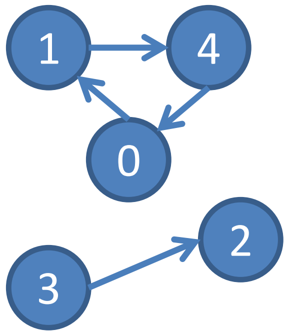


DFS

- **Cycles in directed graphs:** if we find a node within "visit" that has NOT visited all of its children, we find a cycle (there is a path from v to v)

```
void visit(graph &g, int k, bool &loop) {
        g[k].visited = 1;       // visited
        for (edge *p = g[k].ady; p!=NULL && !loop; p=p->next)
                if (g[p->v].visited == -1)
                        visit(g, p->v);
                else if (g[p->v].visited == 1)
                        loop = true;
        g[k].visited = 2;  // all children were visited
}
```



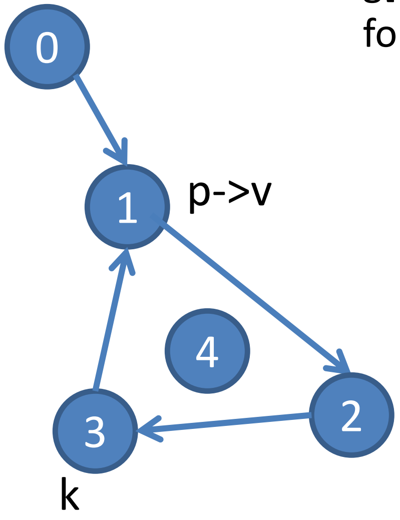


DFS

- **Cycles in directed graphs:** if we want to "get" the cycle, we can update an array or a hash table that tells us for each node, who visited it: table[p->v]=k. By finding the cycle with a p->v node, we already know that k tried to visit p->v. We searched the hash who visited k (k’=table[k]), then who visited k' (k‘’=table[k']) and thus until we find again p->v. This cycle reconstruction is O(n) using an extra vector of n inputs (one input per vertex)

DFS

- **Cycles in directed graphs:** rebuilding the cycle



```
bool visit(graph &g, int k, bool &loop) {
        g[k].visited = 1;          // visited
        for (edge *p = g[k].ady; p!=NULL && !loop; p=p->next)
                if (g[p->v].visited == -1) {
                        visit(g, p->v);  table[p->v] = k;
                }
                else if (g[p->v].visited == 1) {
                        loop = true;  g[p->v].visited = IN_LOOP;
                         g[k].visited = IN_LOOP;
                        while(table[k] != p->v)  {
                                g[table[k]].visited = IN_LOOP;
                                k=table[k];
                        }
                        return;
                }
         if (!loop) g[k].visited = 2;  // all children were visited
}
```

DFS

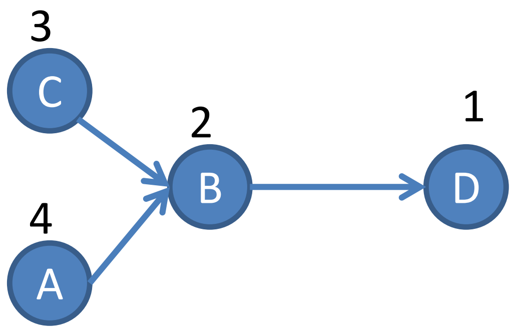
- **Topological sorting:** applies to acyclic and directed graphs (DAGs)
- It consists of sorting the vertices "from left to right" so that the arcs can only go from left to right
- From another point of view, we can re-label the vertices in such a way that in each arc (a,b), a<b is satisfied

DFS

- **Topological sorting:** the utility is the orderly realization of a sequence of tasks, respecting the dependencies between tasks (job scheduling)
- The topological ordering of a DAG is not necessarily unique
- Solution 1: Start from nodes that do not have arcs to it (no dependencies), and list the vertices in post-order. It is the reverse topological ordering

DFS

- We need to know which vertices are “sources”. Then, make a DFS for each "source“ vertex (C and A in the example).
- Notice that the graph must be acyclic!.



Topological ordering: A C B D (visit order: 4 3 2 1)
A "task" is not excecuted before those that "precede" it

DFS

```
std::stack<int> stackObj; int count;

int DFS(graph &g) {
        count = 0; stackObj.clear();
        for (int i=0; i<g.size(); i++)
                g[i].visited = -1;
        for (int i=0; i<g.size(); i++)
                if (g[i].isSource)
                        visit(g,i);
        print(stackObj);
}

void visit(graph &g, int k) {
        for (edge *p = g[k].ady; p!=NULL; p=p->next)
                if (g[p->v].visited == -1)
                        visit(g,p->v);
        stackObj.push_back(k);
         g[k].visited  = count++;
}
```

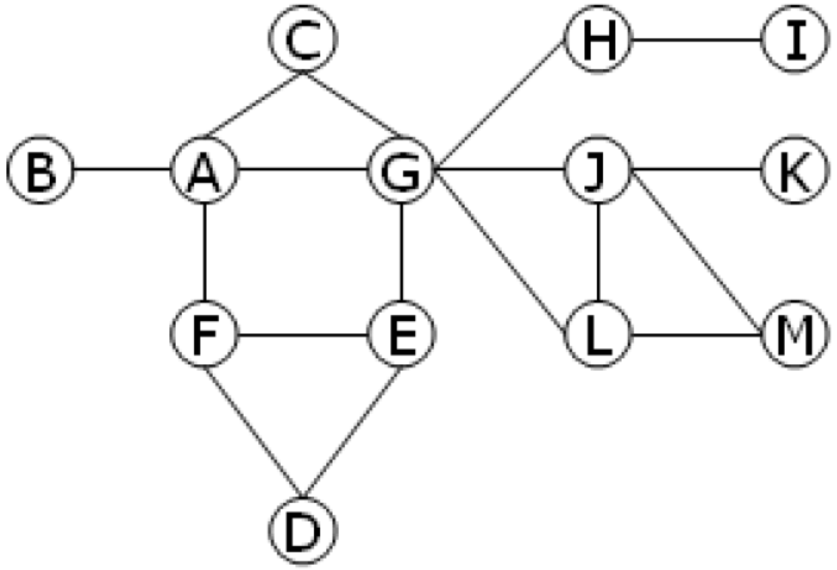
```
struct vertex
{
                edge *ady;
                 int visited = -1;
                 bool isSource;
};
```

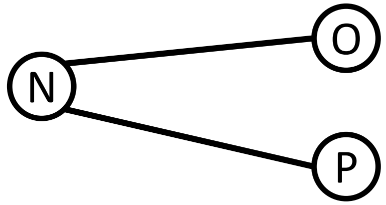
DFS

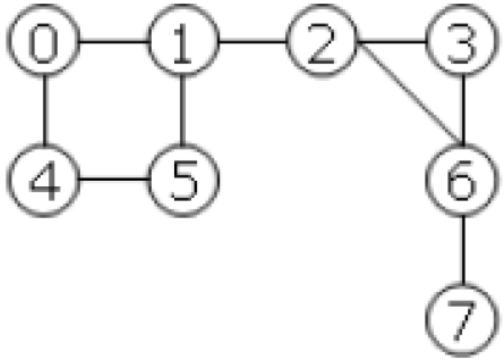
- **Articulation points:** applies to connected graphs. A vertex is a point of articulation if removing it from the graph disconnects the graph (generating 2 or more connected components). The node must be at least grade 2
- **Solution:** For a candidate vertex k, check that a child does not have descendants with an arc to a node labeled "before k". If no such arc exists, then k is a Point of Articulation

DFS

- **Articulation points: example**





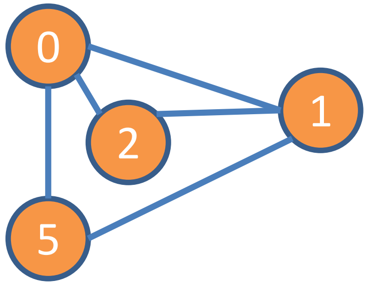


Articulation points: A G H J N 1 2 6

https://www.eecs.wsu.edu/~holder/courses/CptS223/spr08/slides/graphapps.pdf

DFS

Can you avoid pre-calculating the degree of the node, and do the check out of the cycle once you know the degree?
Missing check root node (apart)



- **Articulation points**:

```
int visit(graph &g, int k, vector<int> &art_points) {
        int min=count;
        g[k].visited = count++;
        for (edge *p = g[k].ady; p!=NULL; p=p->next)
                if (g[p->v].visited == -1) {  // no visited
                        int m = visit(g,p->v, art_points);
                        if (m >= g[k].visited && g[k].degree>=2)
                                art_points.push_back(k);
                        if (m<min) min=m; //minimal mark visited
                }
                else if (g[p->v].visited < min)   // visiting before the minimum?
                        min = g[p->v].visited; // update the minimum mark
        return min;
}
```

https://stackoverrun.com/es/q/4310072

DFS

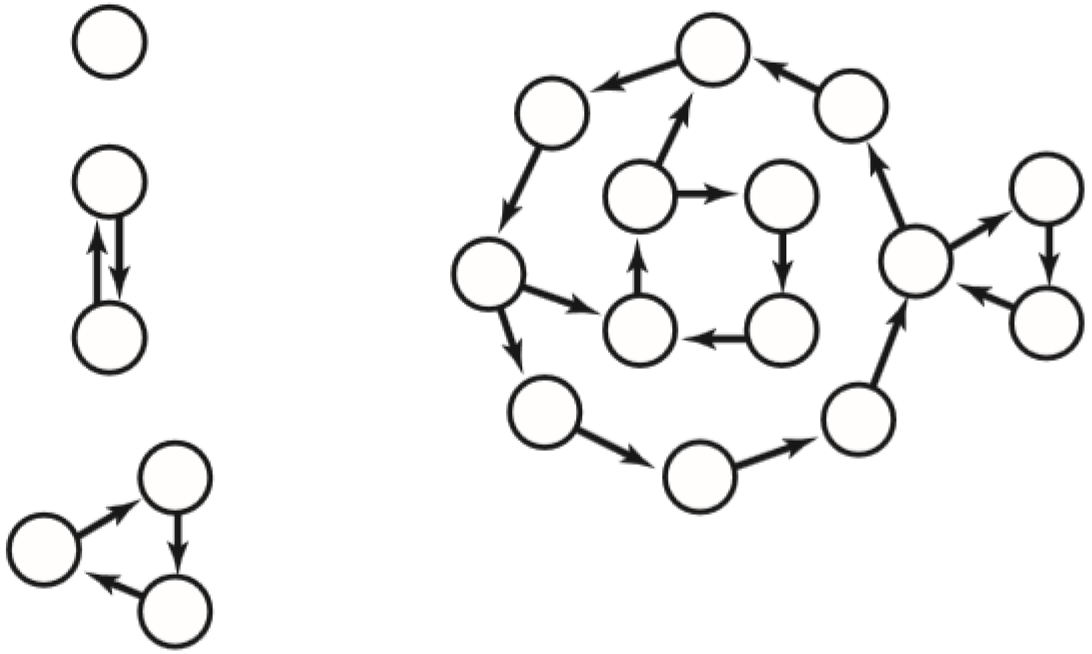
- **Biconnected graph:** occurs when there are at least two paths between each pair of vertices.
- A connected graph is biconnected if it does not have articulation points.
- To determine that a graph is biconnected, we do a DFS, invoking the function “ visit “ only once (all nodes must be reachable to be connected) and there must be no articulation points (art_points.size()=0)

DFS

- **Strongly connected component (SCC):** A SCC in a directed graph is a subgraph where there is a path between each pair of vertices
- There is an algorithm created by Kosaraju (1978) and then published by Sharir (1981) to extract these components in linear order O(n+m)

DFS

- **Strongly connected component (SCC):** Each pair of vertices must belong to a directed cycle.

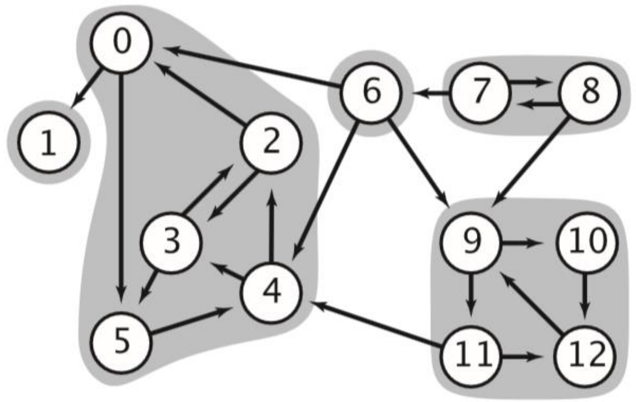


DFS

- **Strongly connected components:** possible applications, textbook authors decide how to group the book's topics; similarly software developers can decide how to partition software into modules; even partition a large set of related web pages, into more manageable sizes

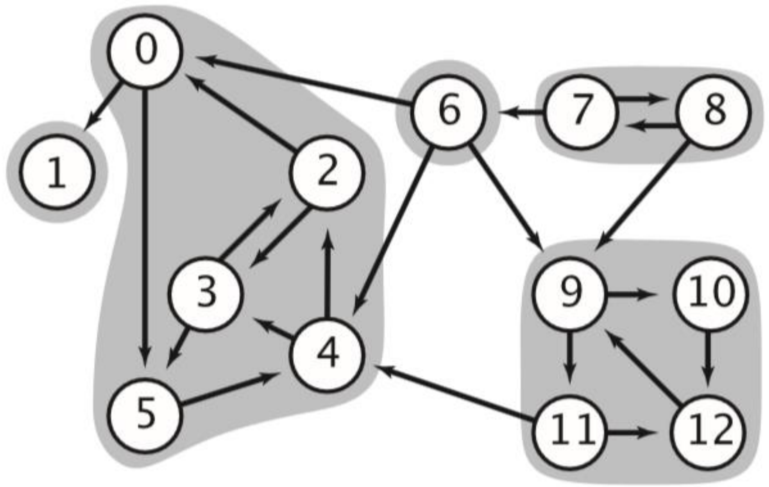
DFS

- **Strongly connected components (SCC):** how determine the SCC's of a directed graph?
- There is an equivalence relationship between the vertices of a SCC (reflective, symmetrical, and transitive)



DFS

- **Strongly related component (SCC):** We could do DFS for each vertex in search of cycle. Then all vertices in a cycle belong to the same equivalence class. Disjointed sets?

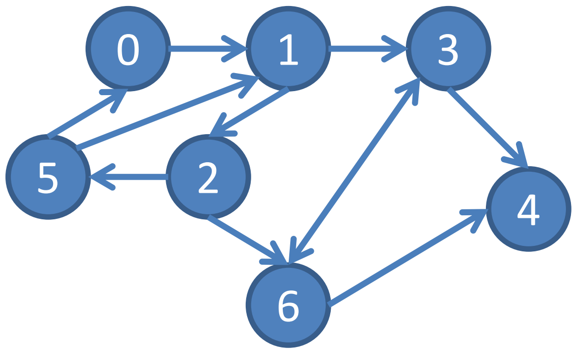


DFS

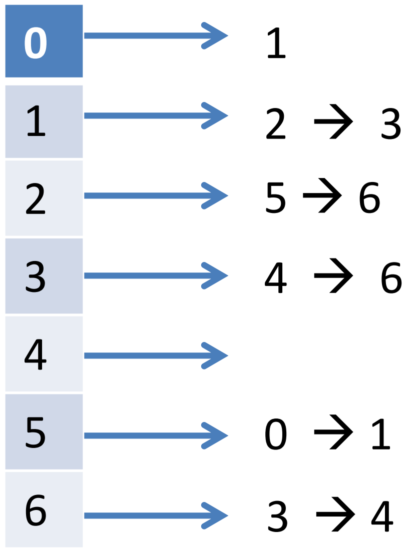
- **Strongly connected component,** Kosaraju O algorithm(n+m)
- Make a conventional DFS (visiting reachable nodes per vertex), introducing the vertices in post-order in a stack
- Get the "reverse" graph by reversing the direction of all arcs. Uncheck all vertices
- Make a DFS for each vertex of the stack. The reachable nodes on each visit call form a SCC

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Make a conventional DFS (visiting reachable nodes per vertex), introducing the vertices in post-order in a stack

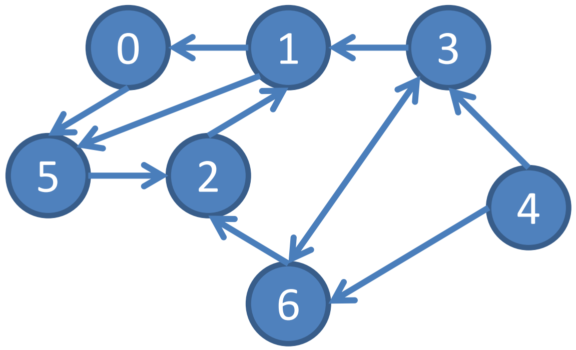


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)



DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Get the "reverse" graph by reversing the direction of all arcs. Uncheck all vertices.

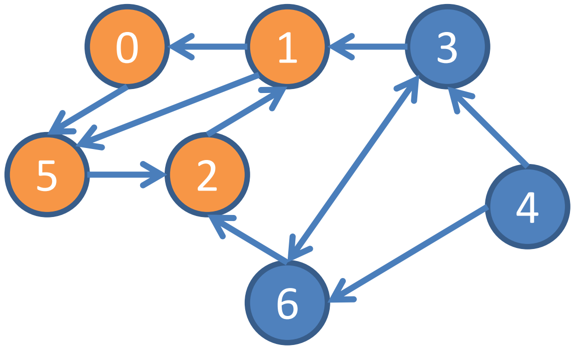


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	→ 5
1	→ 0 → 5
2	→ 1
3	→ 1 → 6
4	→ 3 → 6
5	→ 2
6	→ 2

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Make a DFS for each vertex of the stack. The reachable nodes on each call to visit form a SCC

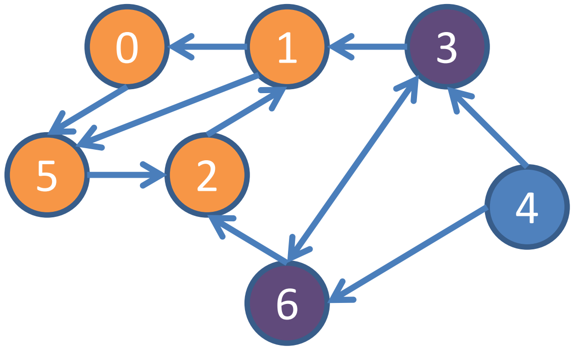


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	5
1	0 → 5
2	1
3	1 → 6
4	3 → 6
5	2
6	2

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Make a DFS for each vertex of the stack. The reachable nodes on each call to visit form a SCC

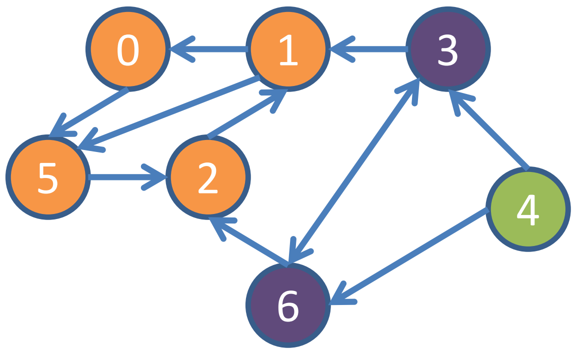


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	$\rightarrow$ 5
1	$\rightarrow$ 0 $\rightarrow$ 5
2	$\rightarrow$ 1
3	$\rightarrow$ 1 $\rightarrow$ 6
4	$\rightarrow$ 3 $\rightarrow$ 6
5	$\rightarrow$ 2
6	$\rightarrow$ 2

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Make a DFS for each vertex of the stack. The reachable nodes on each call to visit form a CFC

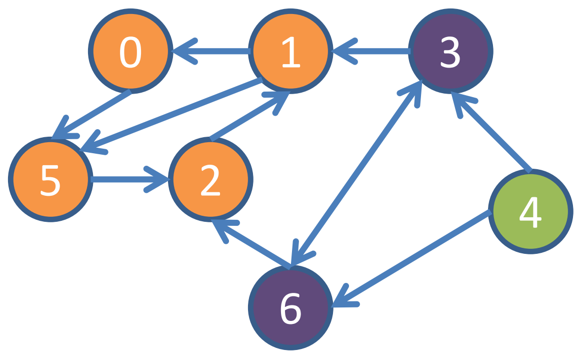


P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	5
1	0 → 5
2	1
3	1 → 6
4	3 → 6
5	2
6	2

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)
- Make a DFS for each vertex of the stack. The reachable nodes on each call to visit form a CFC



P: 5 – 4 – 3 – 6 – 2 – 1 – 0 (top)

0	$\rightarrow$ 5
1	$\rightarrow$ 0 $\rightarrow$ 5
2	$\rightarrow$ 1
3	$\rightarrow$ 1 $\rightarrow$ 6
4	$\rightarrow$ 3 $\rightarrow$ 6
5	$\rightarrow$ 2
6	$\rightarrow$ 2

```
struct vertex
{
            edge *ady;
             int visited = -1;
};
```

DFS

- **Strongly connected components**, Kosaraju O algorithm(n+m)

```
std::stack<int> s; int count;

int DFS1(digraph &g) {
  count = 0;
  for (int i=0; i<g.size(); i++)
            g[i].visited = -1;
  for (int i=0; i<g.size(); i++)
            visit1(g,i);
}
```

```
void visit1(digraph &g, int k) {
   g[k].visited  = 1; edge *p;
  for (p = g[k].ady; p; p=p->next)
     if (g[p->v].visited == -1)
         visit1(g, p->v);
  s.push (k);
}
```

DFS

```
struct vertex
{
                edge *ady;
                 int visited = -1;
};
```

- **Componente fuertemente conexa**, algoritmo de Kosaraju O(n+m)

```
int DFS2(digraph &g) {
  digraph r = g.reverse();
  count = 0;
  for (int i=0; i<r.size(); i++)
              r[i].visited = -1;
  while (! s.empty()) {
    int x = s.top();
    if (r[x]->visited == -1) visit2(r, x);
    s.pop();
    count++;
  }
}
```

```
void visit2(digraph &g, int k) {
  g[k].visited  = count; edge *p;
  for (p = g[k].ady; p; p=p->next)
      if (g[p->v].visited == -1)
         visit2(g, p->v);
}
```

Each node is marked with the number of the SCC

DFS

- **Transitive closure:** The transitive closure of a G digraph is another digraph with the same set of vertices, but with a side from v to w if and only if w is accessible from v in G
- It is usually stored in a matrix, as it is usually dense
- Think of a solution to this problem. There is no algorithm less than n^2 that then allows ExitePath(a,b) queries in O*(1)

DFS

- **Hamiltonian Path:** Given a DAG, design a linear time algorithm to determine if there is a route that visits each vertex once
- Answer: Calculate a topological sorting and check if there is an arc between each pair of consecutive vertices in the topological order

Spoj.com

- Some of them are easy ones

https://www.spoj.com/problems/PT07Y/

https://www.spoj.com/problems/BENEFACT/

https://www.spoj.com/problems/BUGLIFE/

https://www.spoj.com/problems/ALLIZWEL/

https://www.spoj.com/problems/POLQUERY/

Spoj.com

- Biconnected component

https://www.spoj.com/problems/BUZZ/

- SCC

https://www.spoj.com/problems/CAPCITY/
https://www.spoj.com/problems/SUBMERGE/

- Topological sorting

https://www.spoj.com/problems/SCAVHUNT/
https://www.spoj.com/problems/TOPOSORT/
https://www.spoj.com/problems/TOPOSORT2/

Spoj.com

- Hamiltonian Path

https://www.spoj.com/problems/LONGPATH/

https://www.spoj.com/problems/DATINGPA/

https://www.spoj.com/problems/QTNOEL/

- Hamiltonian cycle/circuits (for later – BFS)

https://www.spoj.com/problems/IE5/

https://www.spoj.com/problems/CIRCUITS/