

Graph theory

Glossary

Graph Representation

Adjacency matrix:

Adjacency list

Runtimes

Algorithms

Depth-First Search (DFS)

Pseudocode

Runtime

Edge classification (post and pre numbers)

Breadth-First Search (BFS)

Pseudocode

Runtime

Find shortest path in DAG (Directed Acyclic Graph)

Pseudocode

Runtime

Dijkstra

Pseudocode

Runtime

Bellman-Ford

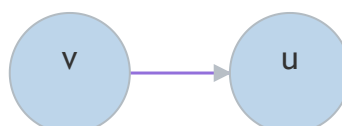
Pseudocode

Runtime

Graph theory

Glossary

- **Graph $G (V, E)$:**
 - **V :** vertices set
 - **E :** edges set
- **Degree:** number of vertices
- **Walk:** series of connected vertices
- **Path:** walk without repeated vertices
- **Closed walk:** walk where $v_0 = v_n$
- **Cycle:** closed walk without repeating vertices
- **Euler path:** visit each edge exactly once
- **Hamilton path:** visit each vertex exactly once
- **Directed graph:** edges are ordered pairs
- **Ancestor:** v , **Successor:** u in



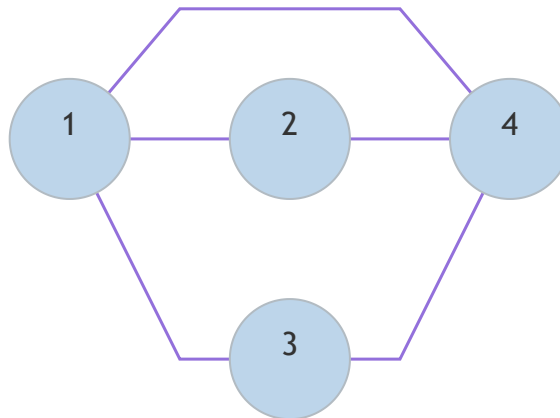
- **deg_{in}(v)**: number of incoming edges into v
- **deg_{out}(v)**: number of outgoing edges into v

Graph Representation

Adjacency matrix:

matrix where $A_{uv} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$

Graph:



Matrix:

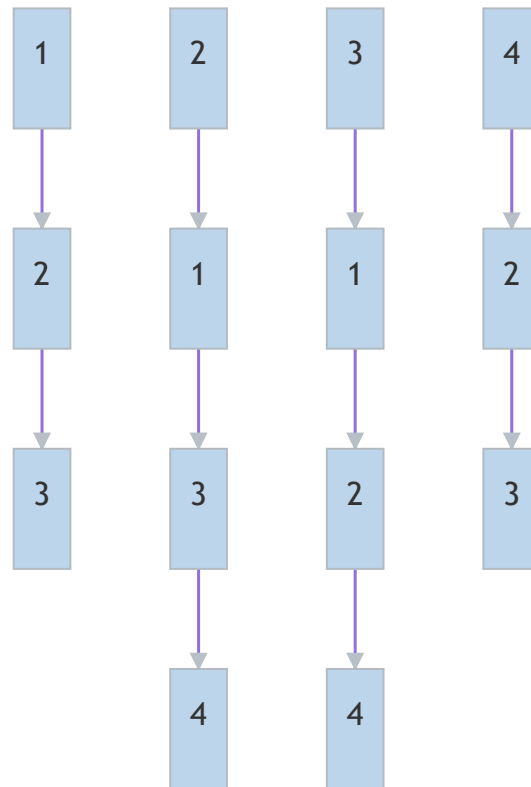
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency list

Array of linked lists, where Adj[u] contains a list containing all the neighbors of u.

Graph: Same as above

List:



Runtimes

	Matrix	List
Find all neighbors of v	$\mathcal{O}(n)$	$\mathcal{O}(\deg_{out}(v))$
Find $v \in V$ without neighbors	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Check if $(v, u) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(1 + \min(\deg_{out}(v), \deg_{out}(u)))$
Insert edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove edge v	$\mathcal{O}(1)$	$\mathcal{O}(\deg_{out}(v))$
Check whether an Eulerian path exists or not	$\mathcal{O}(V * E)$	$\mathcal{O}(V + E)$

Algorithms

Depth-First Search (DFS)

Used mainly to check whether a Graph can be topological sorted or not (\Leftrightarrow has a cycle). A **topological sorting** of a graph it's a sequence of all its nodes with the property that a node u comes after a node v **if and only if** either a walk from v to u exists or u cannot be reached starting from v .

Pseudocode

```
1 DFS(G):  
2   for (v in V not marked):  
3     DFS-Visit(v)
```

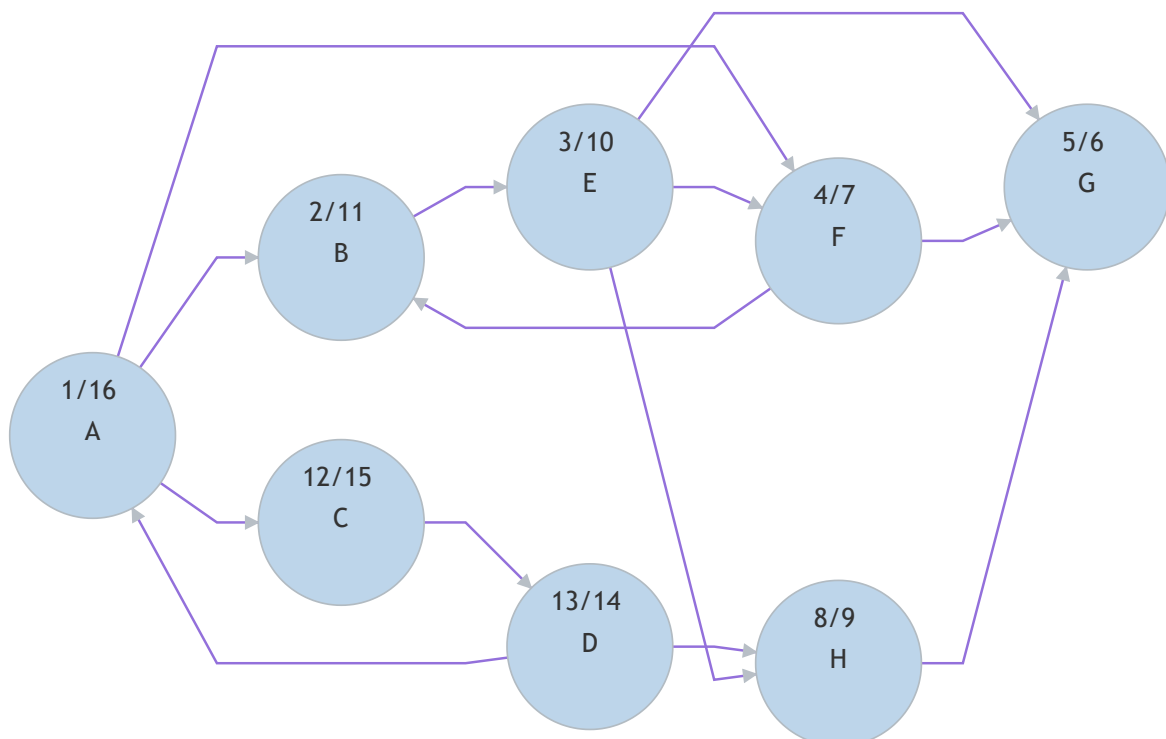
```
1 DFS-Visit(v):  
2   t = 0  
3   pre[v] = t++  
4   marked[v] = true  
5   for ((u, v) in E not marked)  
6     DFS_Visit(u)  
7   post[u] = t++
```

Runtime

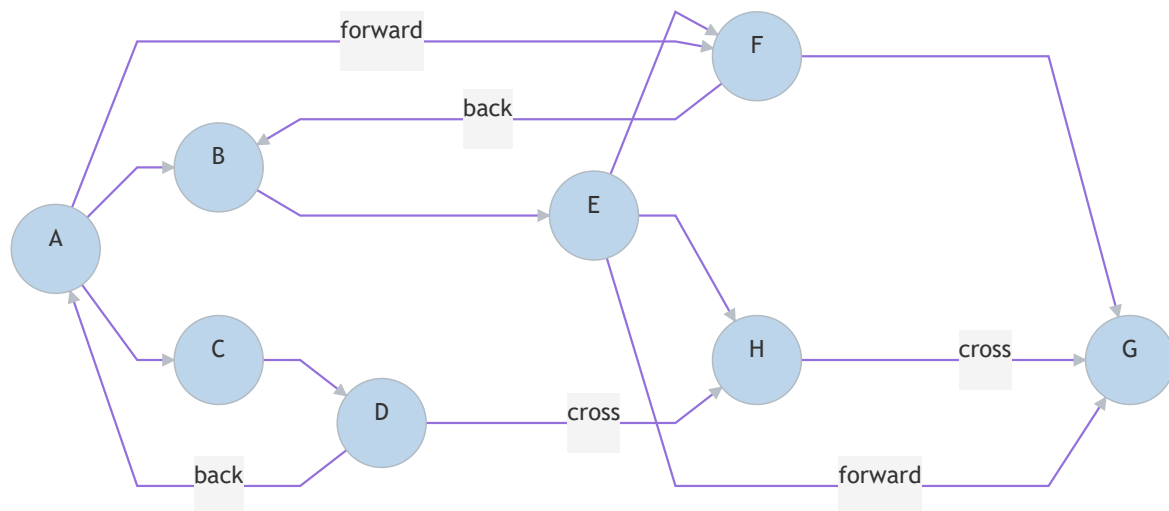
Operations	$T(n) \in \Theta(E + V)$
Memory	$T(n) \in \Theta(V)$

Edge classification (post and pre numbers)

Example: DFS(A) got called



This graph generate the following tree (rotated of 90 degree to save space):



Pre and post number	Name of the edge $(v, u) \in E$
$pre(u) < pre(v)$ and $post(u) < post(v)$	Not possible
$pre(u) < pre(v)$ and $post(u) > post(v)$	forward or simply no name, since it is part of the tree
$pre(u) < pre(v)$ and $post(u) < post(v)$ but $(u, v) \notin E$	forward edge
$pre(u) > pre(v)$ and $post(u) > post(v)$	back edge
$pre(u) > pre(v)$ and $post(u) < post(v)$	cross edge
$pre(u) < pre(v)$ and $post(u) < post(v)$	Not possible

Remark: \nexists back edge $\Leftrightarrow \nexists$ closed walk (cycle)

Breadth-First Search (BFS)

Instead of searching through the depth of a graph, one can also go first through all the successor of the root with the BFS algorithm.

Pseudocode

```

1  BFS(G) :
2      for (v in V not marked) :
3          DFS-Visit(v)

```

```

1 DFS-Visit(v):
2     Q = new Queue()
3     active[v] = true //used to check whether a vertex is in the queue or not
4     enqueue(v, Q)
5     while (!isEmpty(Q)):
6         w = dequeue(Q)
7         visited[w] = true
8         for ((w, x) in E):
9             if(!active[x] && !visited[x]):
10                 active[x] = true
11                 enqueue(x, Q)

```

Runtime

Operations	$T(n) \in \Theta(E + V)$
Memory	$T(n) \in \Theta(V)$

Find shortest path in DAG (Directed Acyclic Graph)

We can compute a recurrence following the topological sorting of the graph.

Pseudocode

```

1 ShortestPath(V):
2     d[s] = 0, d[v] = inf
3     for (v in V \ {s}, following topological sorting):
4         for (u, v, s.t. (u, v) in E):
5             d[v] = min(d[u] + c(u,v))

```

Runtime

$T(n) \in \mathcal{O}(|E| * |V|)$ if adjacency list is given

Dijkstra

Used to find the shortest (cheapest) path between two nodes in a graph.

Remark: The graph must **not** have negative weights

Pseudocode

```

1 DijkstraG, s):
2     for (v in V):
3         distance[v] = infinity
4         parent[v] = null
5     distance[s] = 0
6     Q = new Queue()
7     insert(s, 0, Q) // insert s into the queue Q, with priority 0 (min)
8     while(!isEmpty(Q)):
9         v* = Q.extractMin() // extract from Q the node with minimum distance
10        for ((v*, v) in E):
11            if (parent[v] == null):
12                distance[v] = distance[v*] + w(v*, v)
13                parent[v] = v*
14            else if (distance[v*] + w(v*, v) < distance[v]):

```

```

15     distance[v] = distance[v*] + w(v*, v)
16     parent[v] = v*
17     decreaseKey(v, distance[v], Q)

```

Runtime

$T(n) \in \mathcal{O}(|E| * |V| * \log(|V|))$ if implemented with a Heap.

If implemented with a **Fibonacci-Heap**: $T(n) \in \mathcal{O}(|E| + |V| * \log(|V|))$

Bellman-Ford

Used for graph with general weight (**positive and negative!**)

Pseudocode

```

1  BellmanFord(G, s):
2      for (v in V):
3          distance[v] = infinity
4          parent[v] = null
5      distance[s] = 0
6      for (i = 1, 2, ..., |V| - 1):
7          for ((u, v) in E):
8              if (distance[v] > distance[u] + w(u, v)):
9                  distance[v] = distance[u] + w(u, v)
10                 parent[v] = u
11      for ((u, v) in E):
12          if (distance[u] + w(u, v) < distance[v]):
13              return "negative cycle!"

```

Runtime

$T(n) \in \mathcal{O}(|E| * |V|)$