Lecture 9

Graph

Md. Asif Bin Khaled

Graph

In mathematics, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called link or line).

Running Time of Graph Algorithms

When we characterize the running time of a graph algorithm on a given graph G = (V, E), we usually measure the size of the input in terms of the number of vertices |V| and the number of edges |E| of the graph. That is, we describe the size of the input with two parameters, not just one. We adopt a common notational convention for these parameters. Inside asymptotic notation (such as O-notation or Θ-notation), and only inside such notation, the symbol V denotes |V| and the symbol E denotes |E|. For example, we might say, “the algorithm runs in time O(VE),” meaning that the algorithm runs in time O(|V||E|). This convention makes the running-time formulas easier to read, without risk of ambiguity.

Directed & Undirected Graph

1. **Directed Graph**: A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network.
2. **Undirected Graph**: An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional.

Graph Representation

We can choose between two standard ways to represent a graph G = (V, E) as a collection of

1. **Adjacency List:** Because the adjacency-list representation provides a compact way to represent sparse graphs, those for which |E| is much less than $|V|^2$, it is usually the method of choice.
2. **Adjacency Matrix:** We may prefer an adjacency-matrix representation, however, when the graph is dense |E| is close to $|V|^2$ or when we need to be able to tell quickly if there is an edge connecting two given vertices.

Either way applies to both directed and undirected graphs.

Adjacency List Continued

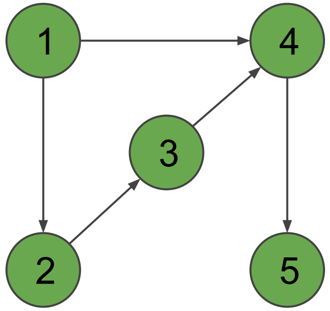
The adjacency-list representation of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V . For each u ∈ V , the adjacency list Adj[u] contains all the vertices such that there is an edge (u, v) ∈ E. That is, Adj[u] consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array Adj as an attribute of the graph, just as we treat the edge set E. If G is a directed graph, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u, v) is represented by having appear in Adj[u].

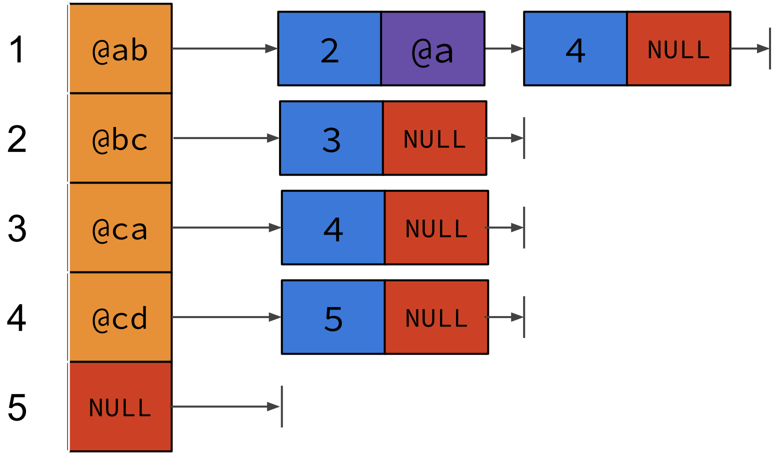
Adjacency List Continued

Moreover, if G is an undirected graph, the sum of the lengths of all the adjacency lists is 2|E|, since if (u, v) is an undirected edge, then u appears in v’s adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is Θ(V + E).

Adjacency List Continued

```
V = {1, 2, 3, 4, 5}
E = {(1, 2), (1, 4), (2, 3), (3, 4), (4, 5)}
```



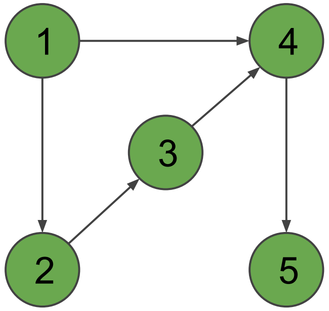


Adjacency Matrix

For the adjacency-matrix representation of a graph G = (V, E) we assume that the vertices are numbered 1, 2, ..., |V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V| x |V| matrix A = (a_{ij}) such that if (i, j) ∊ E then a_{ij} = 1 otherwise a_{ij} = 0. Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose, $A = A^T$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half. Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph.

Adjacency Matrix Continued

```
V = {1, 2, 3, 4, 5}
E = {(1, 2), (1, 4), (2, 3), (3, 4), (4, 5)}
```



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Breadth First Search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim’s minimum-spanning tree algorithm and Dijkstra’s single-source shortest-paths algorithm use ideas similar to those in breadth-first search.

Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a “breadth-first tree” with root

Breadth First Search Continued

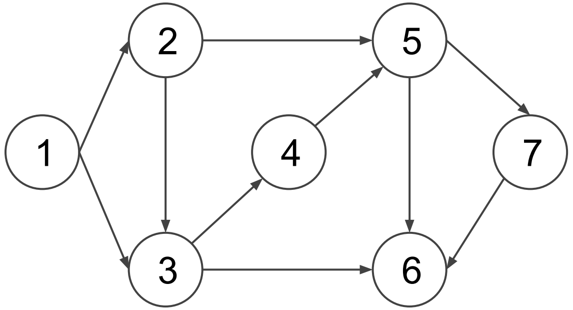
s that contains all reachable vertices. For any vertex reachable from s, the simple path in the breadth-first tree from s to v corresponds to a “shortest path” from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth First Search Pseudocode

```
BFS(G, s):
    for each vertex u ∈ G.V - {s}:
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    ENQUEUE(Q, s)
```

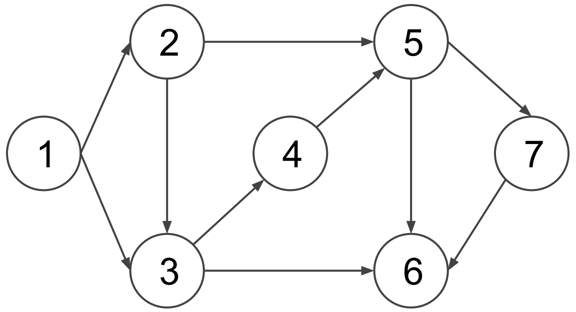
```
while Q != ∅:
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]:
        if v.color == WHITE:
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Simulation of Breadth First Search



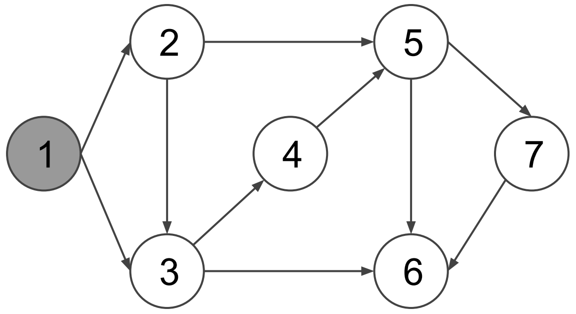
Vertex	Parent	Distance	Visited
1	-	-	-
2	-	-	-
3	-	-	-
4	-	-	-
5	-	-	-
6	-	-	-
7	-	-	-

Simulation of Breadth First Search Continued.



Vertex	Parent	Distance	Visited
1	-	-	-
2	NIL	∞	WHITE
3	NIL	∞	WHITE
4	NIL	∞	WHITE
5	NIL	∞	WHITE
6	NIL	∞	WHITE
7	NIL	∞	WHITE

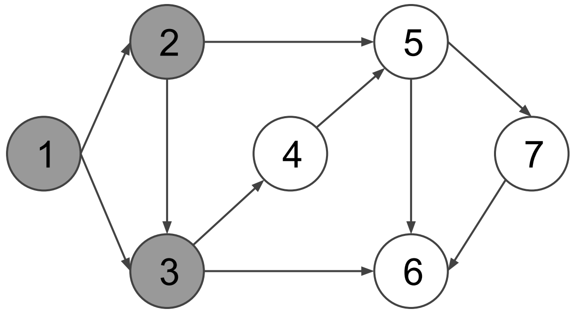
Simulation of Breadth First Search Continued



Queue: 1

Vertex	Parent	Distance	Visited
1	NIL	0	GRAY
2	NIL	∞	WHITE
3	NIL	∞	WHITE
4	NIL	∞	WHITE
5	NIL	∞	WHITE
6	NIL	∞	WHITE
7	NIL	∞	WHITE

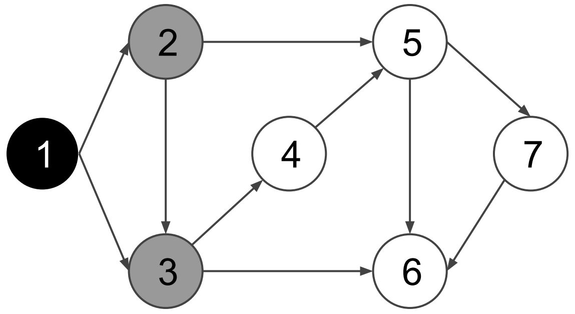
Simulation of Breadth First Search Continued.



Queue: 2 3

Vertex	Parent	Distance	Visited
1	NIL	0	GRAY
2	1	1	GRAY
3	1	1	GRAY
4	NIL	∞	WHITE
5	NIL	∞	WHITE
6	NIL	∞	WHITE
7	NIL	∞	WHITE

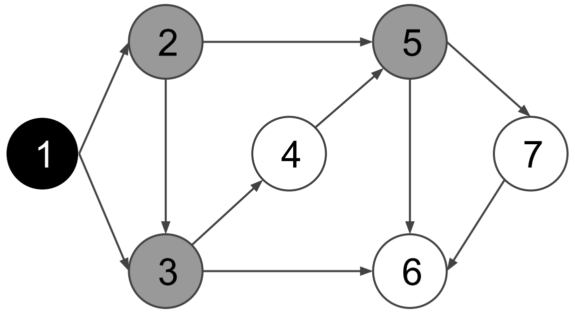
Simulation of Breadth First Search Continued.



Queue: 2 3

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	GRAY
3	1	1	GRAY
4	NIL	∞	WHITE
5	NIL	∞	WHITE
6	NIL	∞	WHITE
7	NIL	∞	WHITE

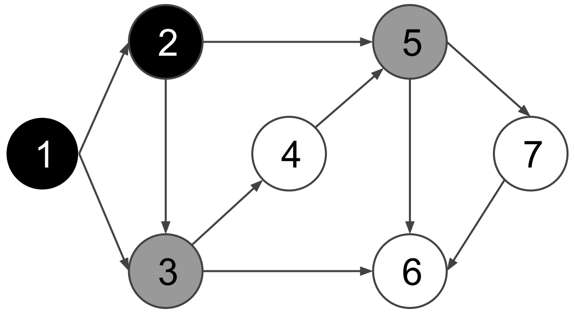
Simulation of Breadth First Search Continued.



Queue: 3 5

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	GRAY
3	1	1	GRAY
4	NIL	∞	WHITE
5	2	2	GRAY
6	NIL	∞	WHITE
7	NIL	∞	WHITE

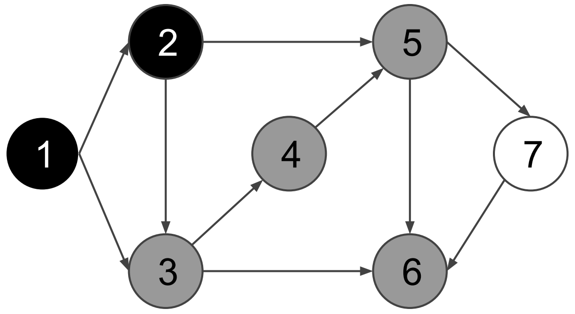
Simulation of Breadth First Search Continued.



Queue: 3 5

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	GRAY
4	NIL	∞	WHITE
5	2	2	GRAY
6	NIL	∞	WHITE
7	NIL	∞	WHITE

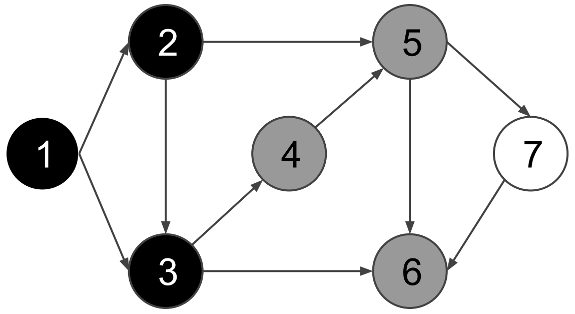
Simulation of Breadth First Search Continued.



Queue: 5 4 6

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	GRAY
4	3	2	GRAY
5	2	2	GRAY
6	3	2	GRAY
7	NIL	∞	WHITE

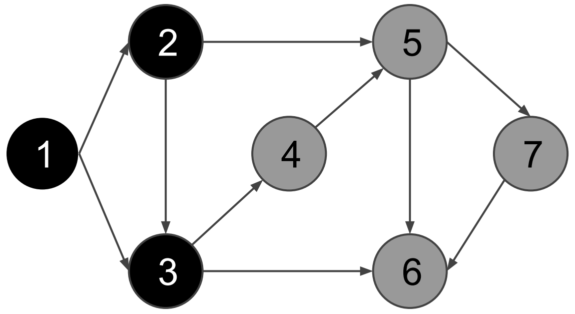
Simulation of Breadth First Search Continued.



Queue: 5 4 6

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	GRAY
4	3	2	GRAY
5	2	2	GRAY
6	3	2	GRAY
7	NIL	∞	WHITE

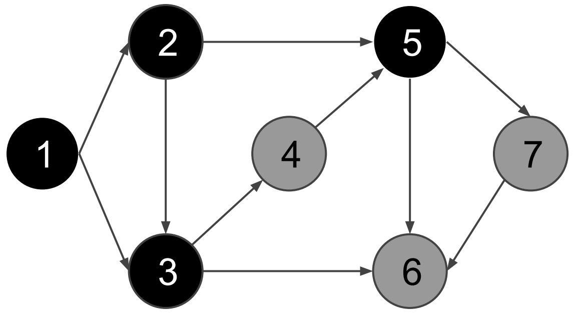
Simulation of Breadth First Search Continued.



Queue: 4 6 7

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	BLACK
4	3	2	GRAY
5	2	2	GRAY
6	3	2	GRAY
7	5	3	GRAY

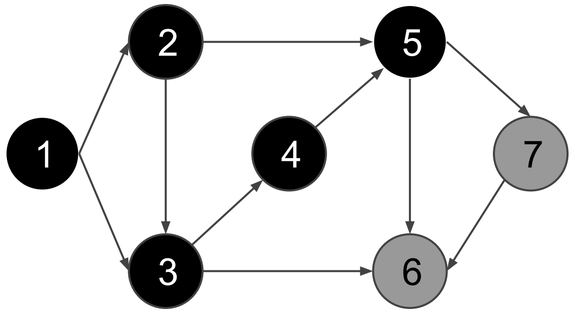
Simulation of Breadth First Search Continued.



Queue: 4 6 7

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	BLACK
4	3	2	GRAY
5	2	2	BLACK
6	3	2	GRAY
7	5	3	GRAY

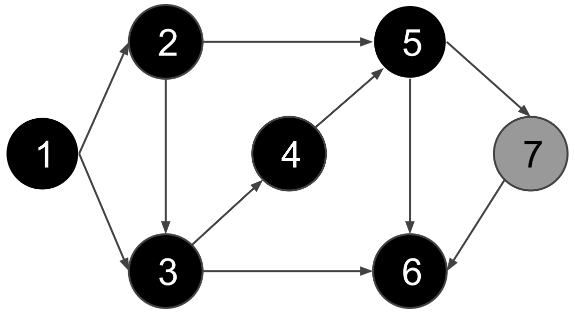
Simulation of Breadth First Search Continued.



Queue: 6 7

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	BLACK
4	3	2	BLACK
5	2	2	BLACK
6	3	2	GRAY
7	5	3	GRAY

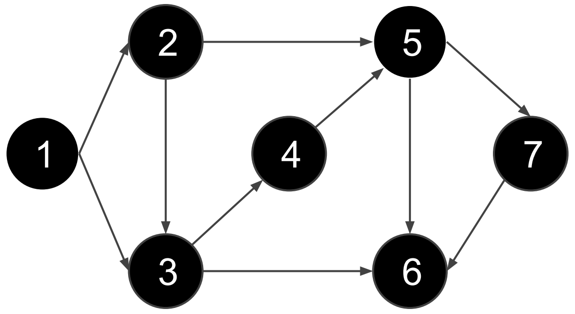
Simulation of Breadth First Search Continued.



Queue: 7

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	BLACK
4	3	2	BLACK
5	2	2	BLACK
6	3	2	BLACK
7	5	3	GRAY

Simulation of Breadth First Search Continued.



Queue:

Vertex	Parent	Distance	Visited
1	NIL	0	BLACK
2	1	1	BLACK
3	1	1	BLACK
4	3	2	BLACK
5	2	2	BLACK
6	3	2	BLACK
7	5	3	BLACK

Breadth-First Trees

The procedure breadth-first search builds a breadth-first tree as it searches the graph. The tree corresponds to the π attributes. More formally, for a graph G = (V, E) with source s, we define the predecessor subgraph of G as,

$G_\pi = (V_\pi, E_\pi)$, where

$V_\pi = \{v \in V: v.\pi \neq \text{NIL}\} \cup \{s\}$ and $E_\pi = \{(v.\pi, v): v \in V_\pi - \{s\}\}$

The predecessor subgraph G_π is a breadth-first tree if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G.

Complexity Analysis of Breadth-First

After initialization, breadth-first search never whitens a vertex, and thus it ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is Θ(E), the total time spent in scanning adjacency lists is O(E). The overhead for initialization is O(V), and thus the total running time of the BFS procedure is O(V + E). Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Depth First Search

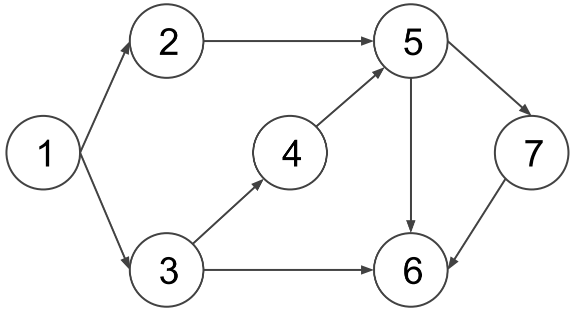
The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Depth First Search Pseudocode

```
DFS(G):
    for each vertex u ∊ G.V:
        u.color = WHITE
        u.π = NIL
    time = 0
    for each vertex u ∊ G.V:
        if u.color == WHITE:
            DFS_Visit(G, u)
```

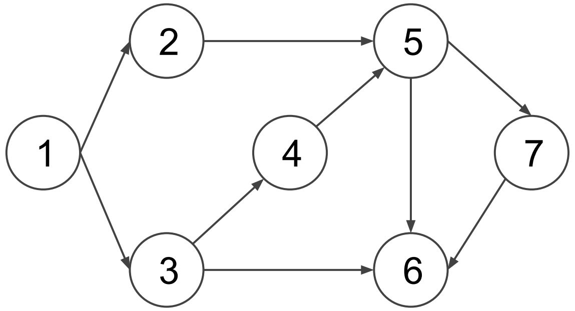
```
DFS_Visit(G, u):
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v ∊ G.Adj[u]:
        if v.color == WHITE:
            v.π = u
            DFS_Visit(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

Simulation of Depth First Search



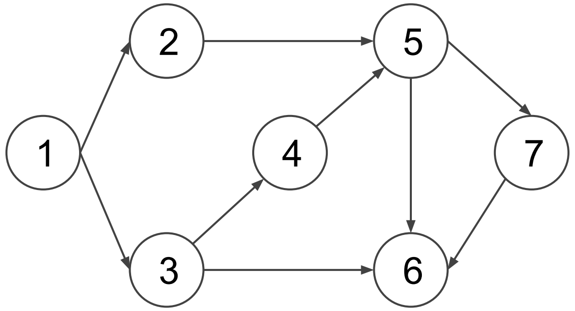
Vertex	Parent	Visited	Starting Time	Finishing Time
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-

Simulation of Depth First Search


$$\begin{tikzcd}
 & 2 \arrow[rr] & & 5 \arrow[dr] \arrow[dd] & \\
1 \arrow[ur] \arrow[dr] & & 4 \arrow[ur] & & 7 \arrow[dl] \\
 & 3 \arrow[ur] \arrow[rr] & & 6 &
\end{tikzcd}$$

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	WHITE	-	-
2	NIL	WHITE	-	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	NIL	WHITE	-	-
6	NIL	WHITE	-	-
7	NIL	WHITE	-	-

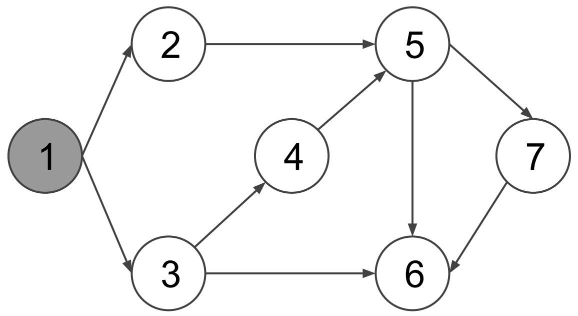
Simulation of Depth First Search



time = 0

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	WHITE	-	-
2	NIL	WHITE	-	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	NIL	WHITE	-	-
6	NIL	WHITE	-	-
7	NIL	WHITE	-	-

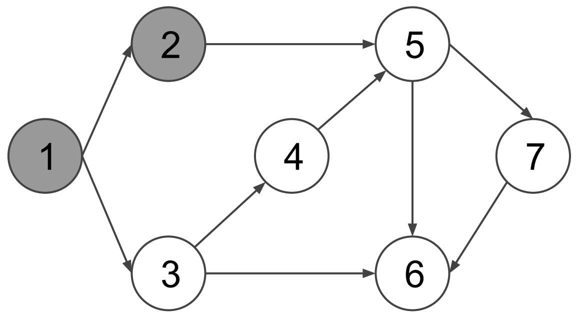
Simulation of Depth First Search



time = 1

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	NIL	WHITE	-	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	NIL	WHITE	-	-
6	NIL	WHITE	-	-
7	NIL	WHITE	-	-

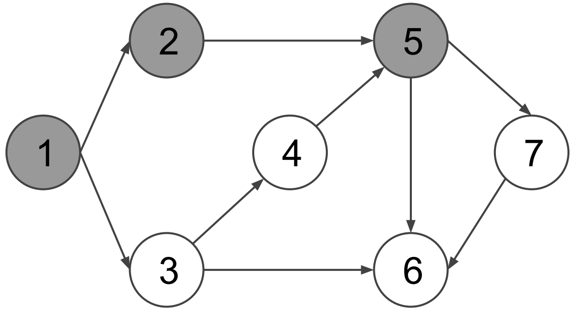
Simulation of Depth First Search



time = 2

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	NIL	WHITE	-	-
6	NIL	WHITE	-	-
7	NIL	WHITE	-	-

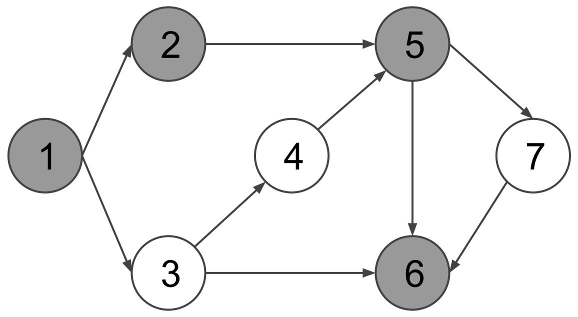
Simulation of Depth First Search



time = 3

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	GRAY	3	-
6	NIL	WHITE	-	-
7	NIL	WHITE	-	-

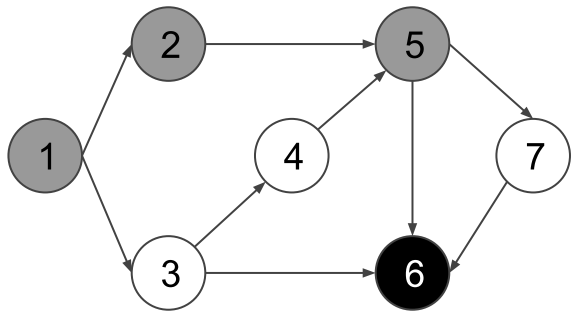
Simulation of Depth First Search



time = 4

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	GRAY	3	-
6	5	GRAY	4	-
7	NIL	WHITE	-	-

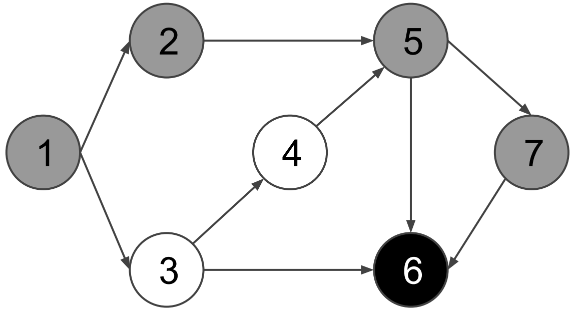
Simulation of Depth First Search



time = 5

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	GRAY	3	-
6	5	BLACK	4	5
7	NIL	WHITE	-	-

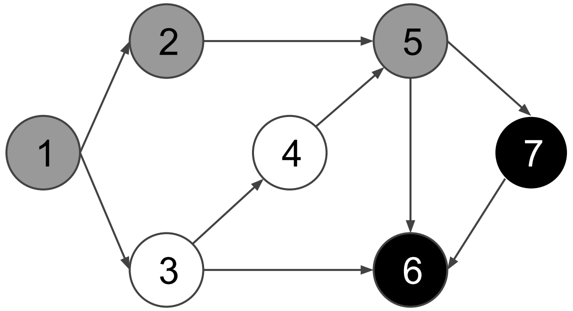
Simulation of Depth First Search



time = 6

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	GRAY	3	-
6	5	BLACK	4	5
7	5	GRAY	6	-

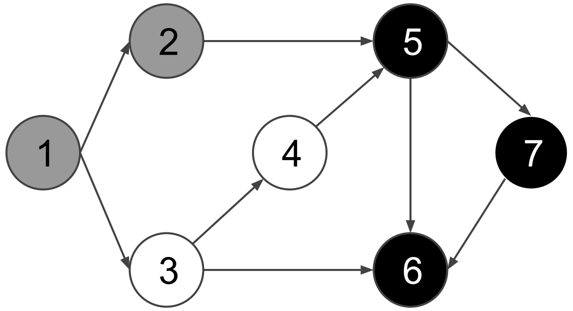
Simulation of Depth First Search



time = 7

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	GRAY	3	-
6	5	BLACK	4	5
7	5	BLACK	6	7

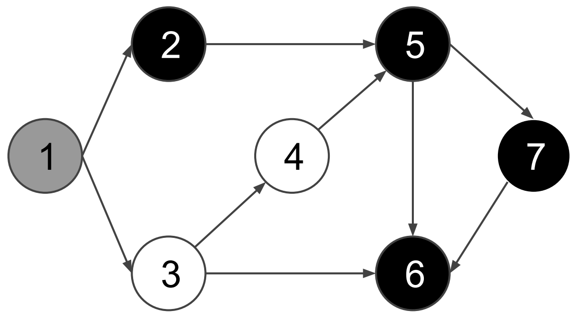
Simulation of Depth First Search



time = 8

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	GRAY	2	-
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

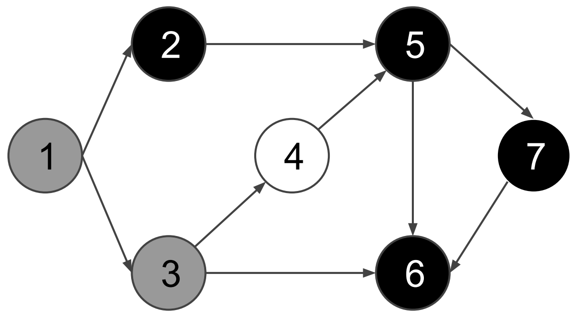
Simulation of Depth First Search



time = 9

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	BLACK	2	9
3	NIL	WHITE	-	-
4	NIL	WHITE	-	-
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

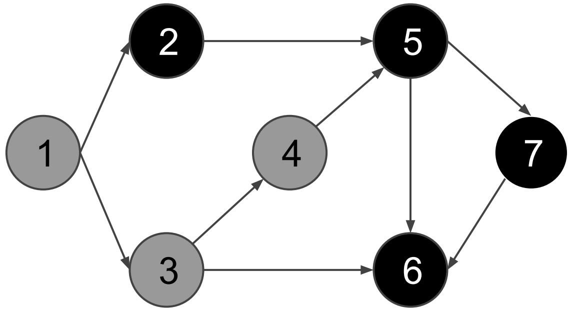
Simulation of Depth First Search



time = 10

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	BLACK	2	9
3	1	GRAY	10	-
4	NIL	WHITE	-	-
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

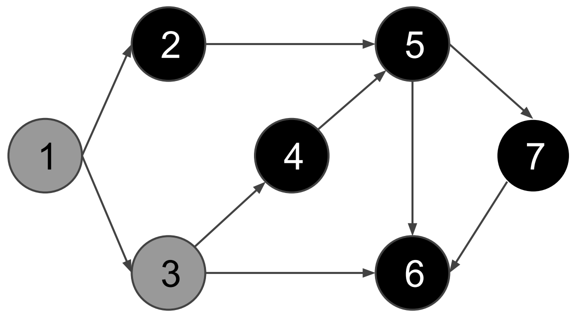
Simulation of Depth First Search



time = 11

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	BLACK	2	9
3	1	GRAY	10	-
4	3	GRAY	11	-
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

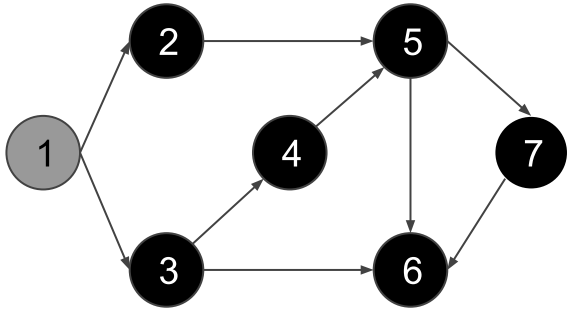
Simulation of Depth First Search



time = 12

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	BLACK	2	9
3	1	GRAY	10	-
4	3	BLACK	11	12
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

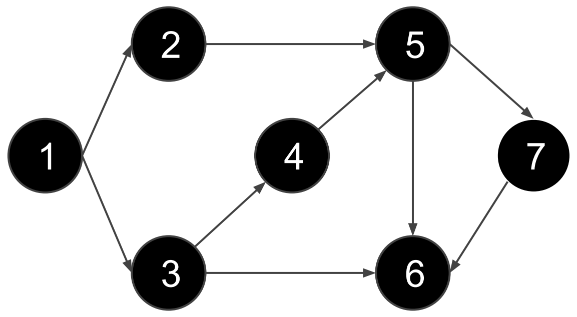
Simulation of Depth First Search



time = 13

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	GRAY	1	-
2	1	BLACK	2	9
3	1	BLACK	10	13
4	3	BLACK	11	12
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

Simulation of Depth First Search



time = 14

Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	BLACK	1	14
2	1	BLACK	2	9
3	1	BLACK	10	13
4	3	BLACK	11	12
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

Complexity Analysis of Depth-First Search

Initially looping through the nodes takes $\Theta(V)$ time, exclusive of the time to execute the calls to DFS_Visit. As we did for breadth-first search, we use aggregate analysis. The procedure DFS_Visit is called exactly once for each vertex $v \in V$, since the vertex u on which DFS_Visit is invoked must be white and the first thing DFS_Visit does is paint vertex u gray. During an execution of DFS_Visit(G, v), the loop executes |Adj[v]| times. Hence, the total cost of executing DFS_Visit is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.

Depth-First Trees

As in breadth-first search, whenever depth-first search discovers a vertex during a scan of the adjacency list of an already discovered vertex u, it records this event by setting ’s predecessor attribute v.π to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Therefore, we define the predecessor subgraph of a depth-first search slightly differently from that of a breadth-first search.

Depth-First Trees Continued

We let,

$G_\pi = (V, E_\pi)$, where

$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E_π are tree edges.

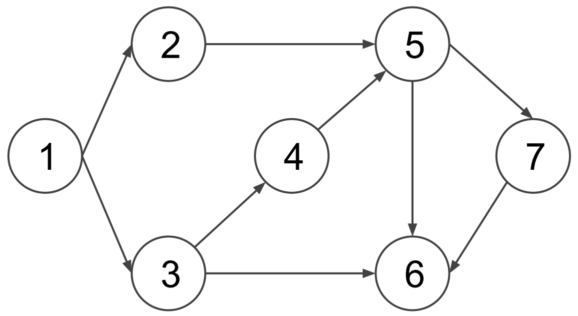
Topological Sort

We can use depth-first search to perform a topological sort of a directed acyclic graph, or a “dag” as it is sometimes called. A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.) We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of “sorting”. Many applications use directed acyclic graphs to indicate precedences among events.

Topological Sort Continued

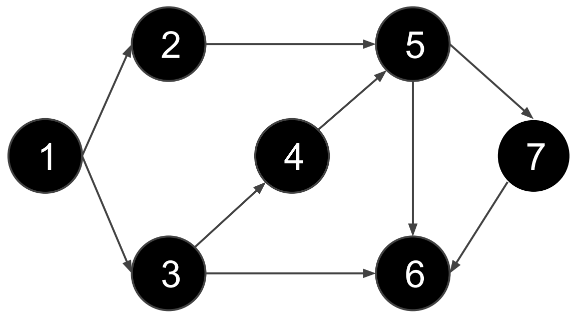
```
Topological_Sort(G):
    call DFS(G) to compute finishing times v.f for each vertex v
    as each vertex is finished, insert it onto the front of a linked list
    return the linked list of vertices
```

Simulation of Topological Sort



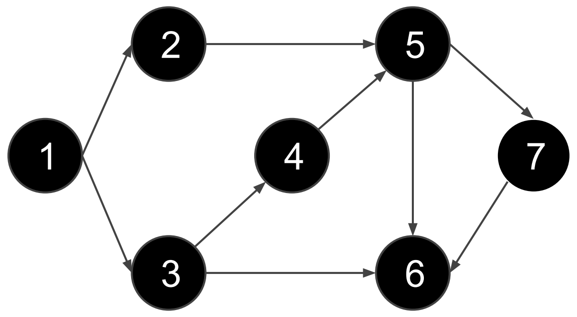
Vertex	Parent	Visited	Starting Time	Finishing Time
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-

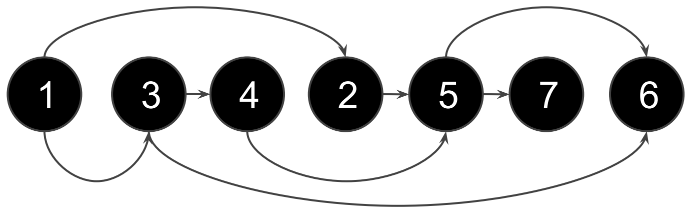
Simulation of Topological Sort Continued



Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	BLACK	1	14
2	1	BLACK	2	9
3	1	BLACK	10	13
4	3	BLACK	11	12
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

Simulation of Topological Sort Continued





Vertex	Parent	Visited	Starting Time	Finishing Time
1	NIL	BLACK	1	14
2	1	BLACK	2	9
3	1	BLACK	10	13
4	3	BLACK	11	12
5	2	BLACK	3	8
6	5	BLACK	4	5
7	5	BLACK	6	7

Topological Sort Running Time

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.