Elementary Graph Algorithms

- Design and analysis of algorithms
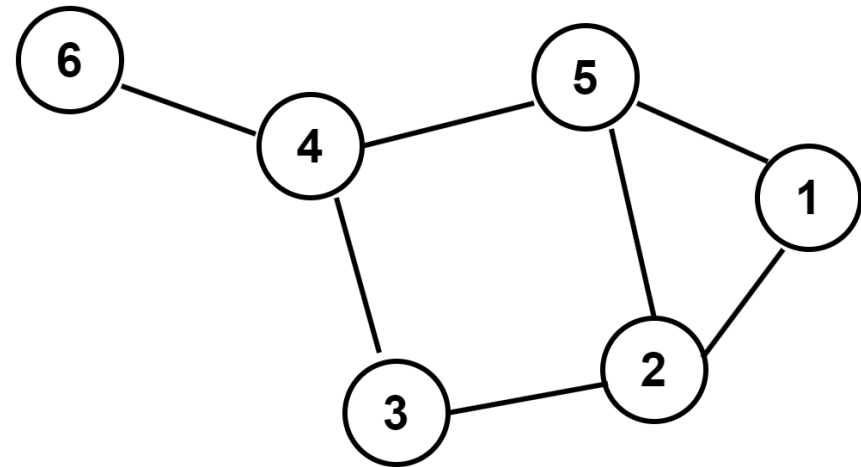
Introduction

➢ Graph and set theory

➢ Graph and computer science

Graph and set theory

➤ In set theory, a graph G is defined as a set of vertices V and a relation E describing how the vertices are related to each other.

➤ A graph is denoted as: $G = (V, E)$

➤ A graph may be directed or undirected.

➤ A graph may also be weighted, i.e. the edges of the graph are assigned values or weights.

An undirected graph $G = (V, E)$ where $V = \{6, 4, 5, 3, 1, 2\}$ and $E = \{(6, 4), (4, 5), (4, 3), (5, 1), (5, 2), (3, 2), (2, 1)\}$

Graph and computer science

➢ In computer science, graph is an abstract data type which is an implementation of a graph in set theory.

➢ One of the ways of implementing graphs is by using linked lists.

➢ The circular nodes are the vertices and the square nodes are the vertices they are connected to.

➢ So the path from a circular node to any of the square node in the adjacent list is an edge.
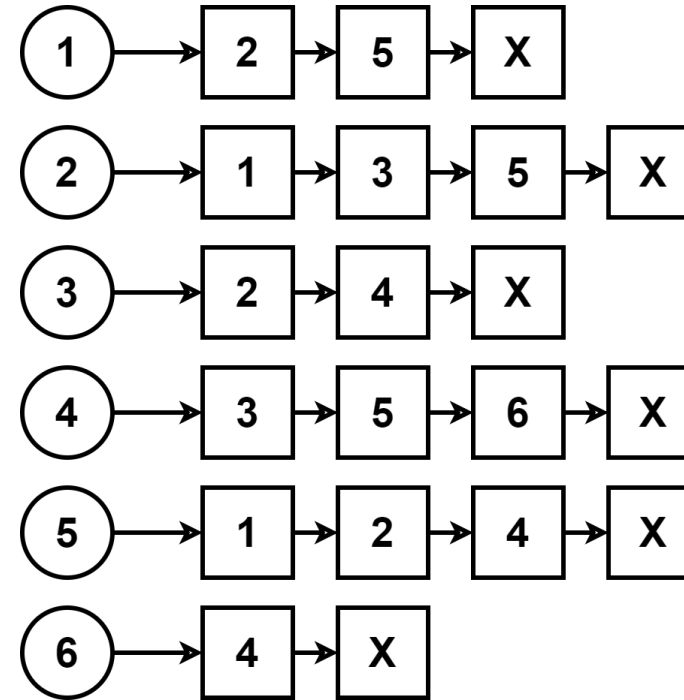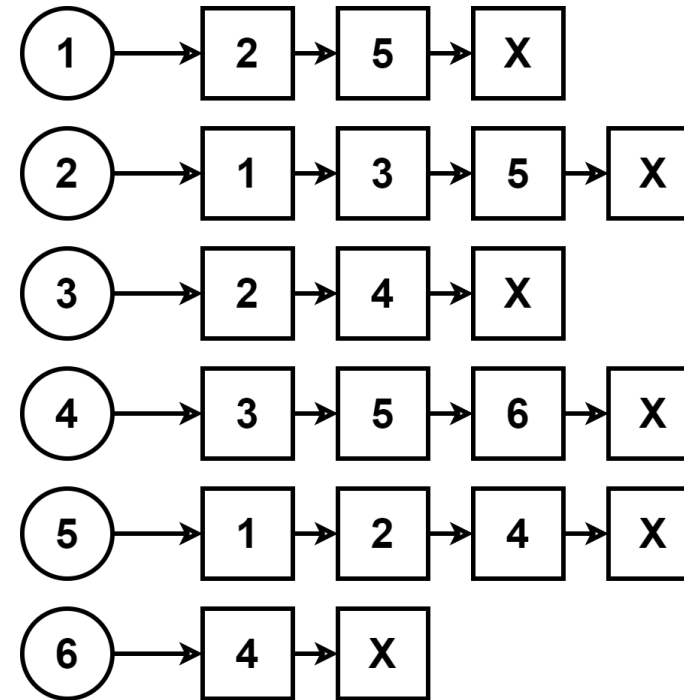

Implementation using linked lists

Representations of graph

➢ Adjacency list

➢ Adjacency list – time complexity

➢ Adjacency matrix

➢ Adjacency matrix – time complexity

Adjacency list

➢ In this method, linked lists are used to store the adjacent vertices.

➢ A graph consists of a list of vertices. Each vertex in the list, in turn, has a pointer to a list of vertices directly connected to it.

➢ So every vertex in the graph has an edge to each vertex in its adjacency list.

➢ To represent a weighted graph, an additional attribute of weight can be added to each vertex in the adjacency list.

Implementation using adjacency list

Adjacency list – time complexity

Operation	Time complexity
Add vertex	$O(1)$
Add edge	$O(1)$
Remove vertex	$O(|E|)$
Remove edge	$O(|V|)$
Adjacency	$O(|V|)$

Adjacency matrix

➢ In this method, a matrix is used to represent if two vertices are connected.

➢ A graph contains a list of vertices and a $n \times n$ matrix such that n is the number of vertices in the graph.

➢ If there is an edge between the xth and y^{th} vertex, then $M_{xy} = 1$. Else $M_{xy} = 0$.

➢ To represent a weighted graph, weights can be stored at the row and column corresponding to the edge in the adjacency matrix.

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

Implementation using adjacency matrix

Adjacency matrix – time complexity

| Operation	Time complexity
Add vertex	$O(|V|^2)$
Add edge	$O(1)$
Remove vertex	$O(|V|^2)$
Remove edge	$O(1)$
Adjacency	$O(1)$

Breadth first search

- ➢ What is BFS?
- ➢ Attributes
- ➢ Algorithm
- ➢ Analysis
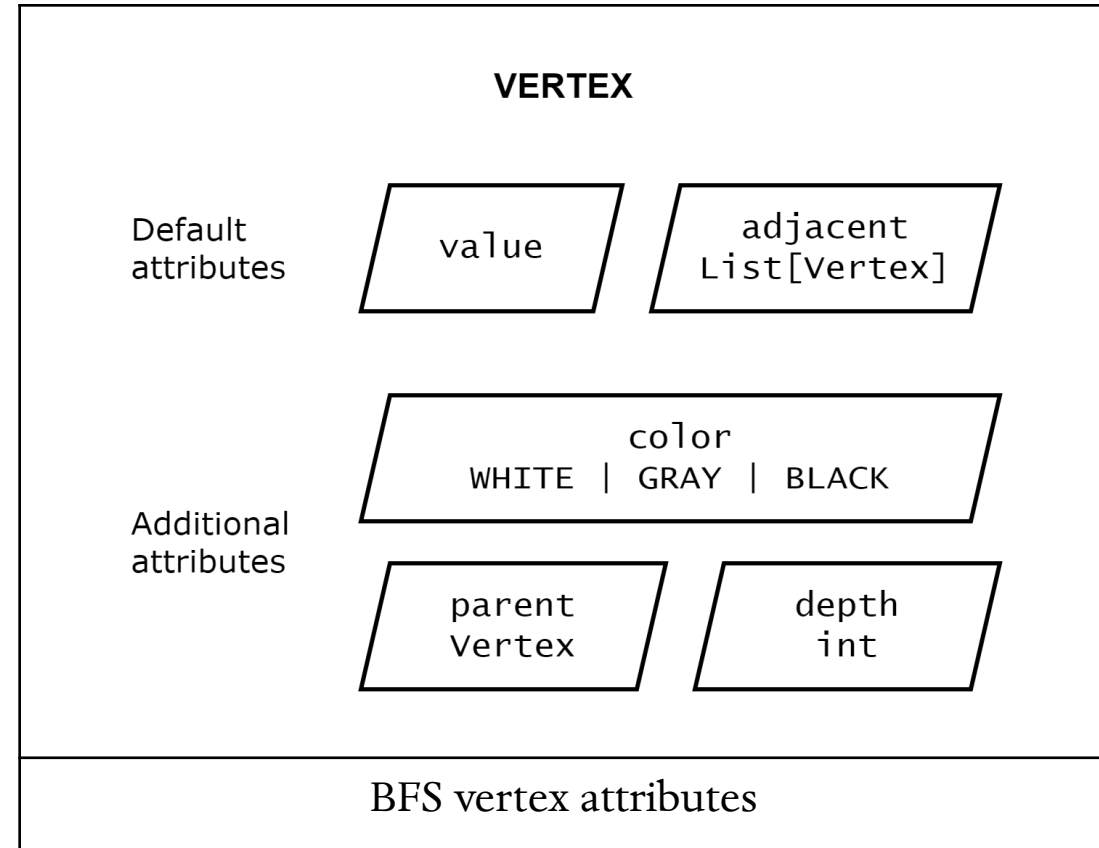- ➢ Properties
- ➢ Applications

What is BFS?

➢ During breadth first search, a *source* vertex is selected from where traversal will start.

➢ The algorithm systematically traverses all the adjacent vertices of *source*.

➢ Then for all the discovered vertices, it then discovers the next set of adjacent vertices.

➢ This goes on until all the vertices connected to *source* have been discovered.
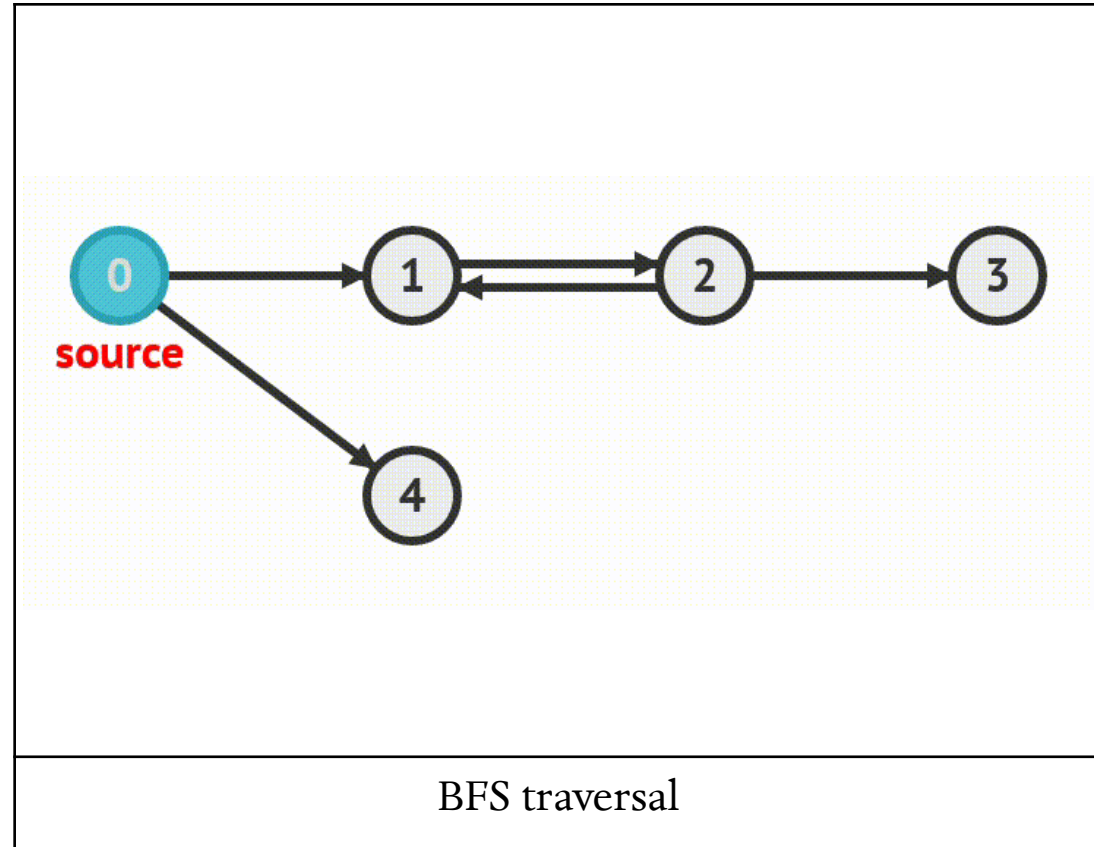
BFS traversal

Attributes

➢ For BFS traversal, some additional attributes are added to each vertex.

➢ *color*: *WHITE* means that the vertex is yet to be reached, *GRAY* means that the vertex has been reached but not yet traversed, and *BLACK* means that the vertex has been traversed.

➢ *depth*: Distance of the vertex from the source vertex.

➢ *parent*: Parent of the vertex in the breadth first tree.

VERTEX

Default attributes
- value
- adjacent List[Vertex]

Additional attributes
- color WHITE | GRAY | BLACK
- parent Vertex
- depth int

BFS vertex attributes

Algorithm

```
function BFS(graph, source):

    # whiten all vertices

    for v in graph.vertices:

        v.color = WHITE

        v.parent = NULL

    # traverse source

    source.color = GRAY

    source.depth = 0

    source.parent = NULL

    enqueue(source)
```

```
# keep dequeuing

while not is_queue_empty():

    curr = dequeue()

    # traverse nodes adjacent to source

    for v in curr.adjacent:

        if v.color == WHITE:

            v.color = GRAY

            v.depth = curr.depth + 1

            v.parent = curr

            enqueue(v)
```

Analysis

➢ Whitening all vertices takes $O(V)$ time.

➢ Traversing source vertex takes $O(1)$ time.

➢ Since a vertex is enqueued only once, the queue can be dequeued V times.

➢ The sum of all adjacent vertices of every vertex is E.

➢ Time complexity of BFS is $O(V + E)$.

Properties

➤ Breadth first search creates a breadth first tree where the parent of v is $v.parent$

➤ $source$ is the root of the tree and $source.parent$ is $NULL$.

➤ For any vertex v, $v.depth$ is the shortest distance from $source$ to v.

Shortest path from source
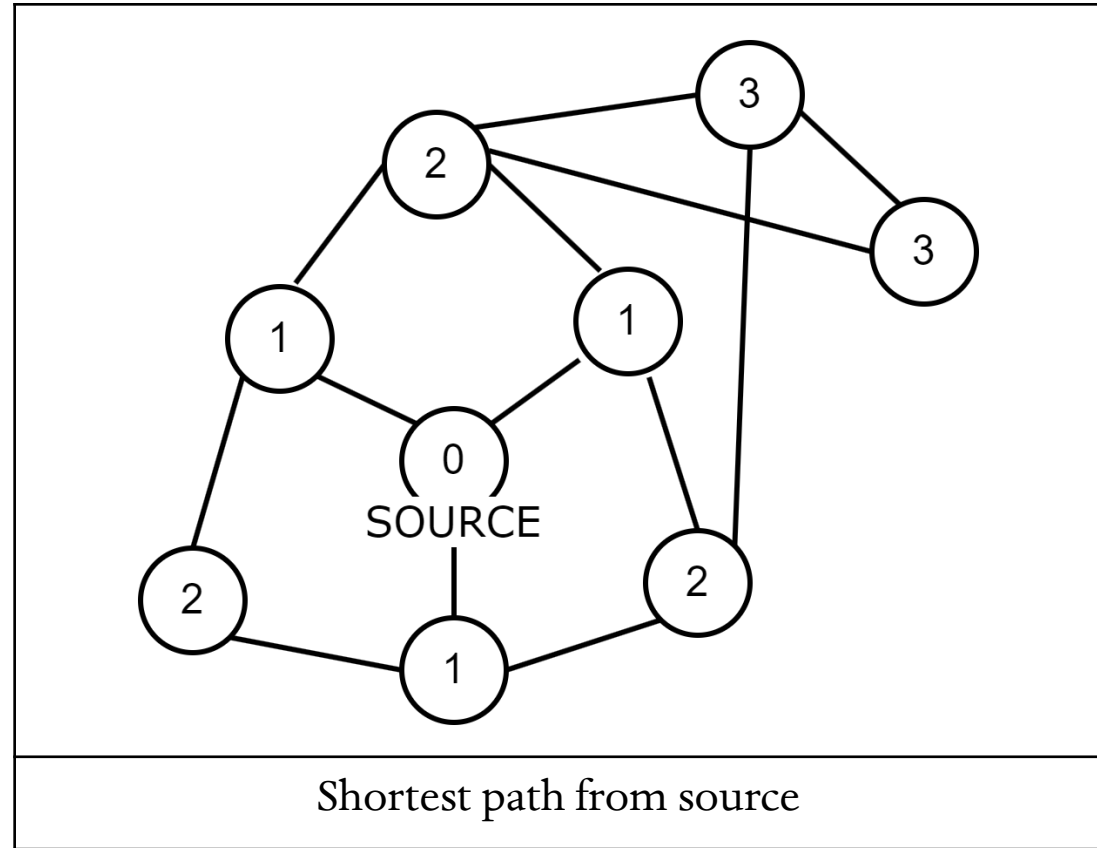
Applications

➤ Prim's algorithm uses BFS procedure to find minimal spanning tree in weighted graphs.

➤ Dijkstra's shortest path algorithm also uses BFS procedure for path finding.

➤ Alternatively, it can also be used to find solution to mazes.

➤ BFS traversal is also used to analyze relationships and connections, especially in social networking.


Path finding
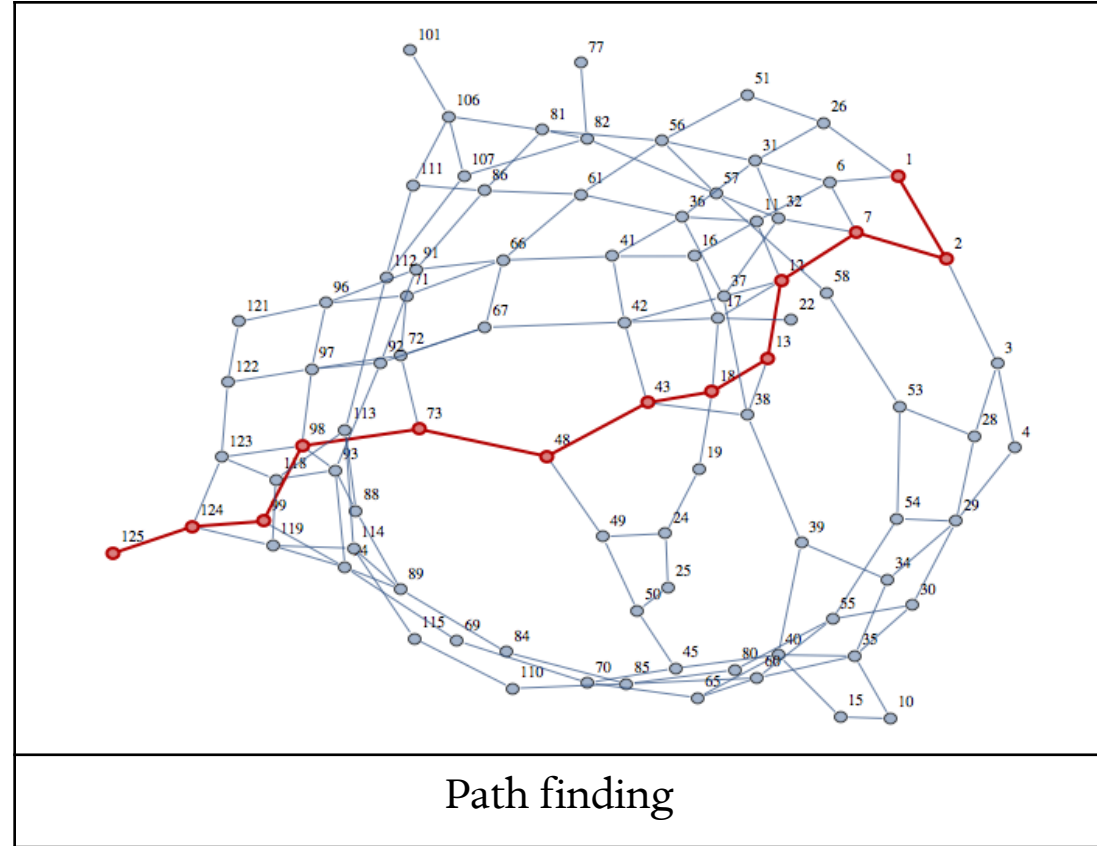
Depth first search

- What is DFS?
- Attributes
- Algorithm
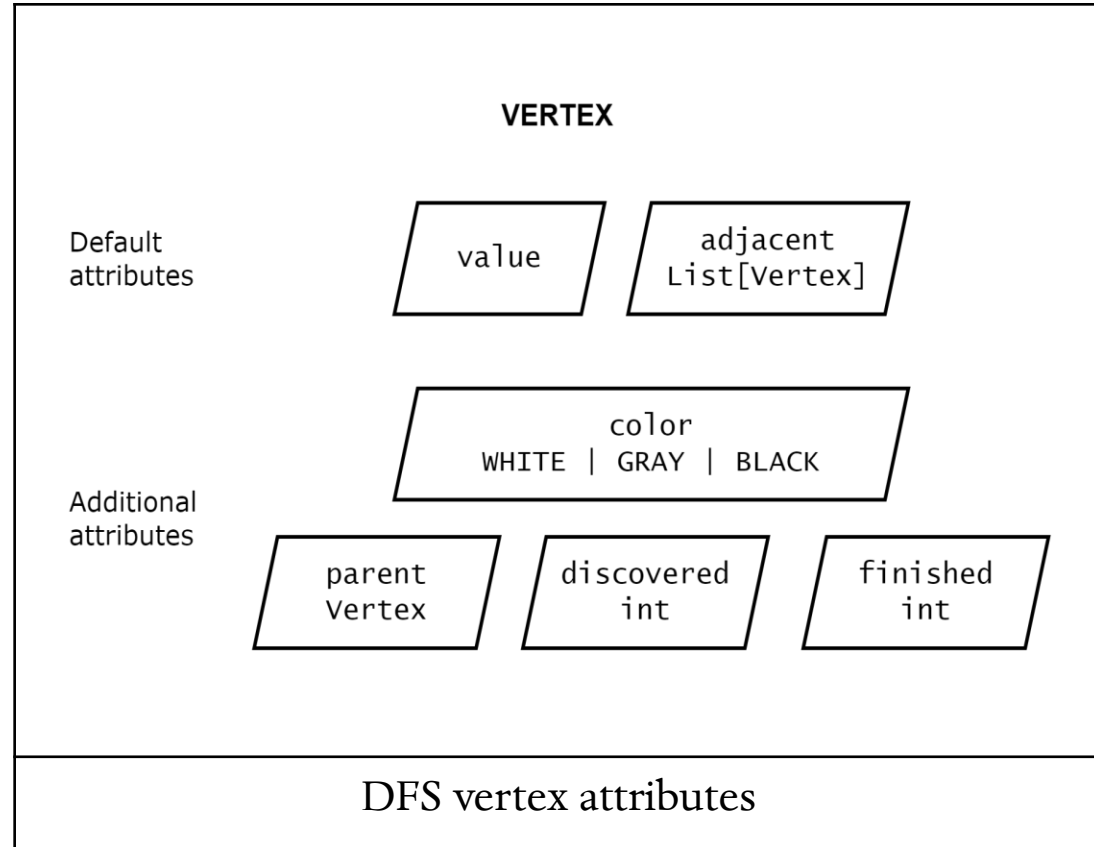- Analysis
- Properties
- Applications

What is DFS?

➢ The aim of depth first search is to traverse *deeper* vertices first whenever possible.

➢ The algorithm starts with a vertex, keeps discovering *deeper* vertices until there are no more undiscovered vertices.

➢ Then the same process is repeated for every other undiscovered vertices.
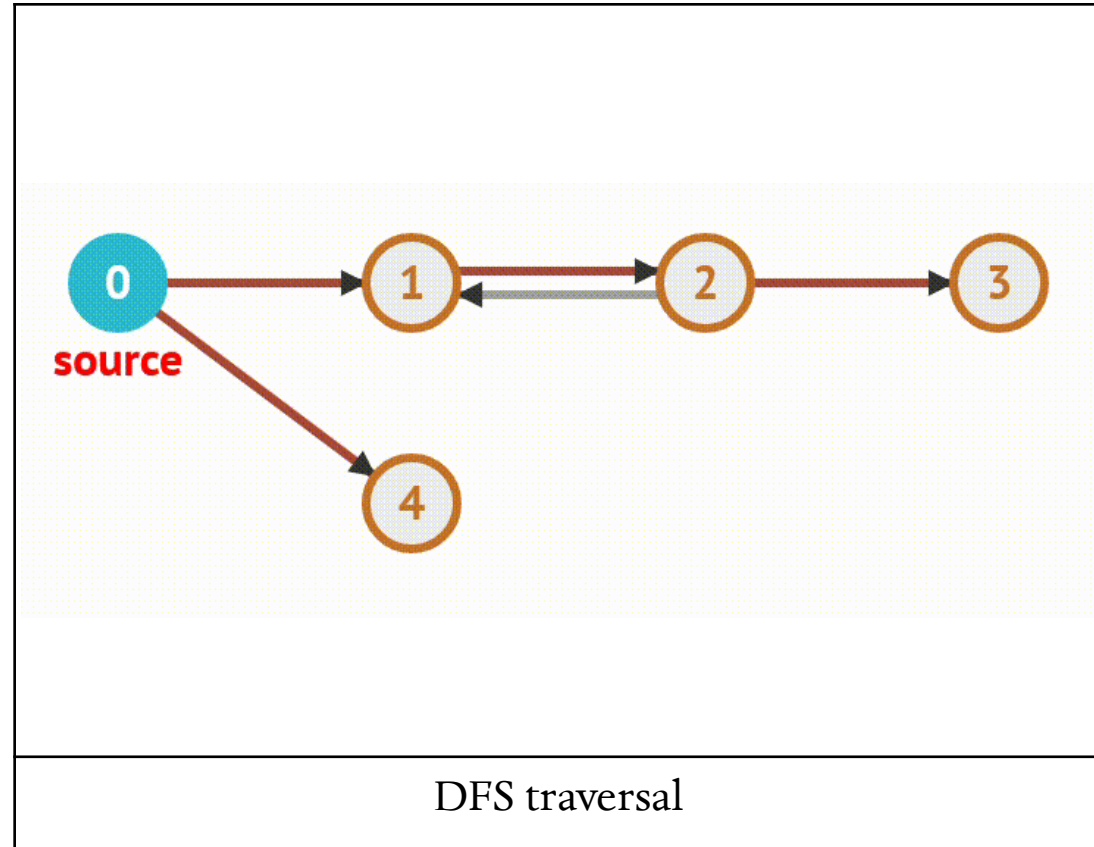
DFS traversal

Attributes

➤ DFS traversal also defines some more attributes for vertices.

➤ *color*: *WHITE*, *GRAY* or *BLACK*.

➤ *discovered*: Timestamp when vertex was discovered.

➤ *finished*: Timestamp when the traversal of the vertex finished.

➤ *parent*: Parent of the vertex in the breadth first tree.

VERTEX

Default attributes

| value | adjacent List[Vertex] |

Additional attributes

| color WHITE \| GRAY \| BLACK |

| parent Vertex | discovered int | finished int |

DFS vertex attributes

Algorithm

```
function DFS(graph):

    # whiten all vertices

    for v in graph.vertices:

        v.color = WHITE

        v.parent = NULL

    # reset timestamp

    timestamp = 0

    # visit every vertex

    for v in graph.vertices:

        if v.color == WHITE:

            DFS_VISIT(graph, v)
```

```
function DFS_VISIT(vertex):

    timestamp += 1

    vertex.discovered = timestamp

    vertex.color = GRAY

    for v in vertex.adjacent:

        if v.color == WHITE:

            v.parent = vertex

            DFS_VISIT(v)

    timestamp += 1

    vertex.finished = timestamp

    vertex.color = BLACK
```

Analysis

➢ Whitening all vertices takes $O(V)$ time.

➢ Since a vertex is pushed only once, the stack can be popped V times, i.e. DFS_VISIT will be called V times.

➢ The sum of all adjacent vertices of every vertex is E, i.e. during DFS_VISIT, adjacent vertices will be checked E times.

➢ Time complexity of DFS is $O(V + E)$.

Properties

➢ Depth first search creates multiple trees or a depth first forest where the parent of v is $v.parent$

➢ Each disconnected vertex has a corresponding tree, the root of the trees are vertices whose $v.parent$ is $NULL$.

➢ DFS discovery and finish times have parenthesis structure.

Parenthesis structure
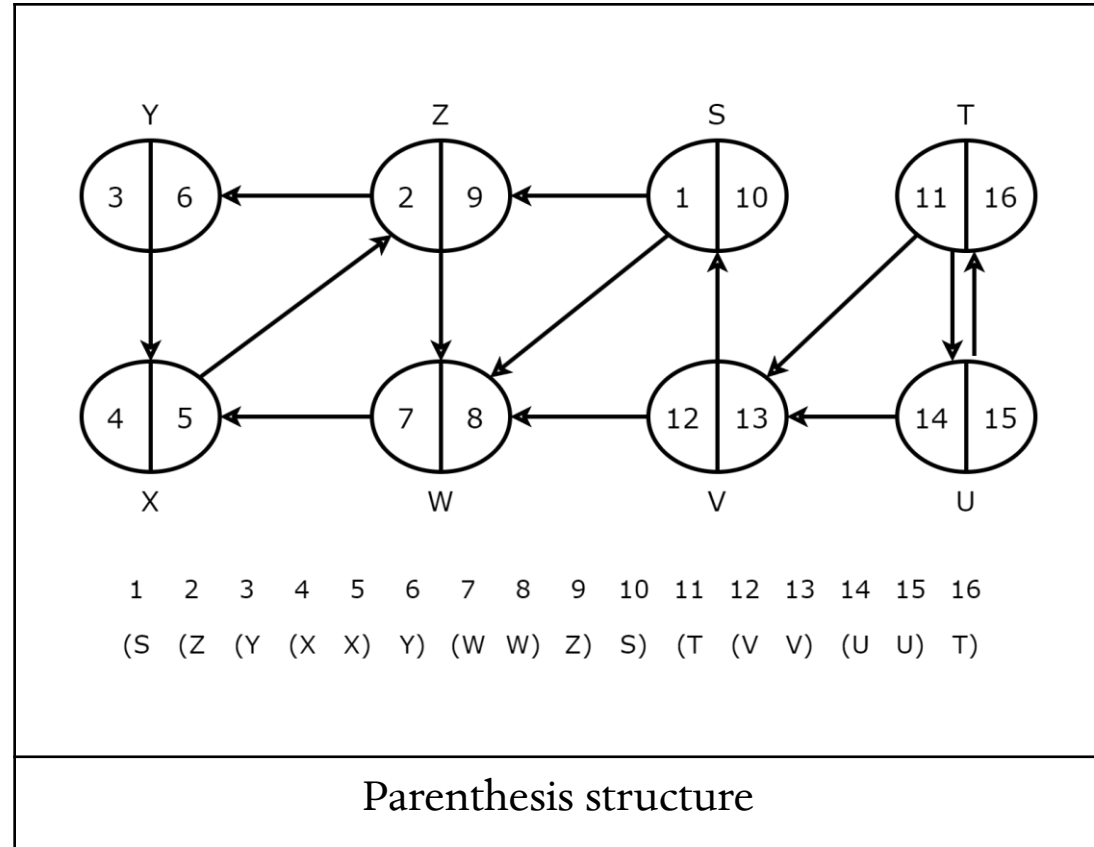
Applications

➢ DFS provides key information about the structure of the graph, e.g. if the graph is cyclic in nature.

➢ DFS can also be used to identify connected components in a graph.

➢ DFS is used in topological sort and topological sort is used to resolve dependencies in complex environments.

➢ It can also be used to solve mazes.

Maze solving
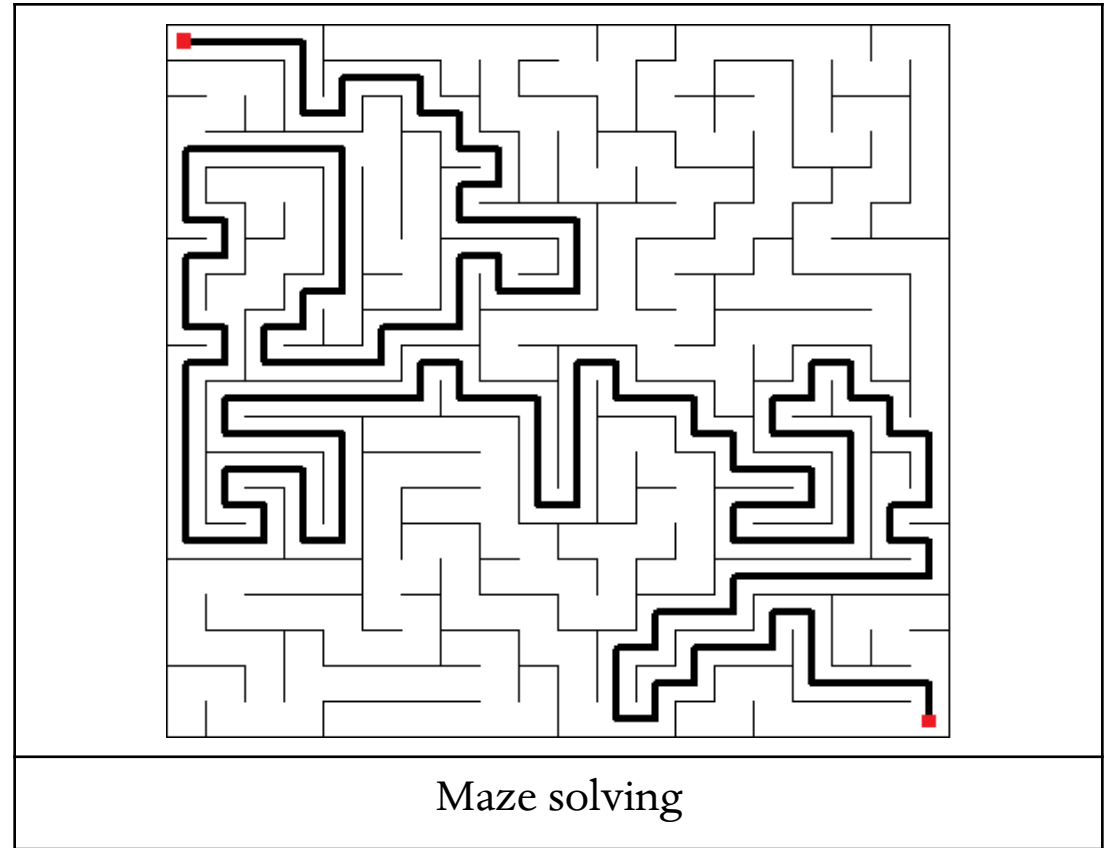
Topological sort

➤ What is topological sort?

➤ Algorithm

➤ Analysis

➤ Applications

What is topological sort?

➤ Topological sort is a way to order the vertices of a directed acyclic graph such that for any edge $a \rightarrow b$, a always appears before b in the ordering.

➤ A directed acyclic graph or DAG is a graph with directed edges and no loops.

➤ Topological sort is only possible for DAGs because no linear ordering is possible for graphs that contain a loop.
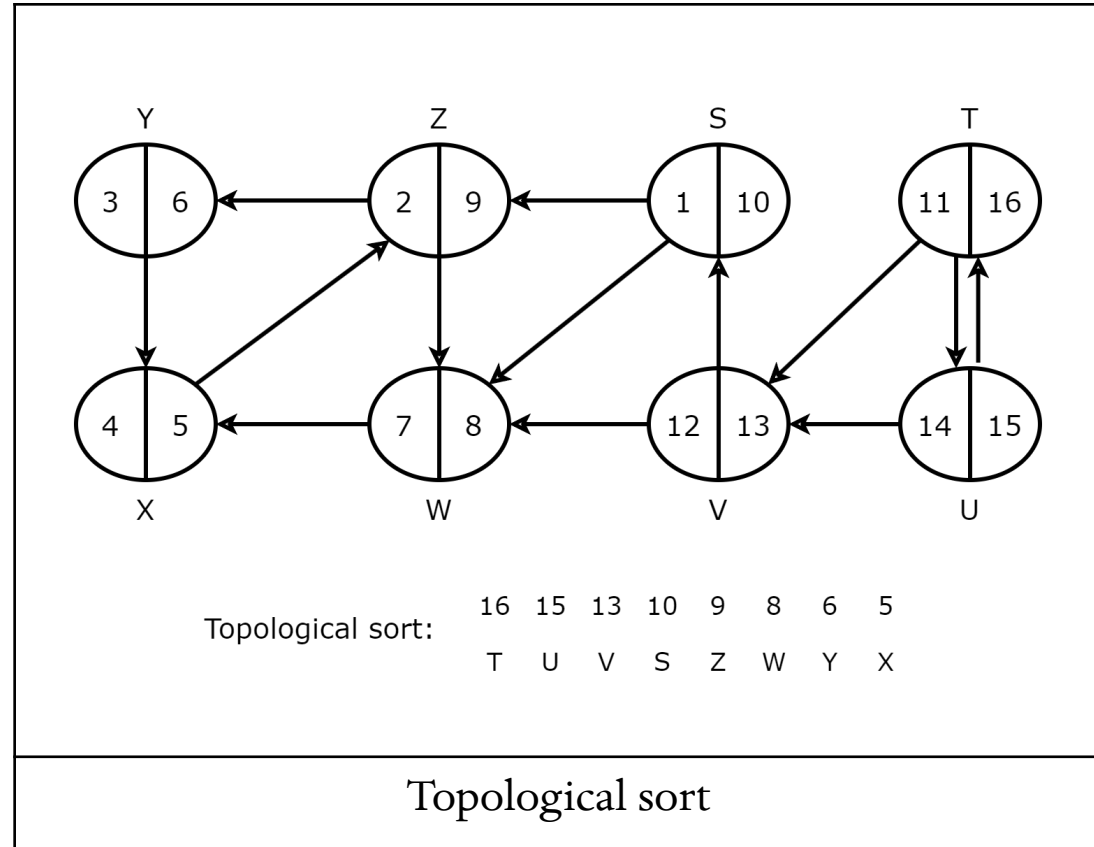
Topological sort

Algorithm

```
function TOPOLOGICAL_SORT(graph):

    stack = []

    timestamp = 0

    # whiten all vertices

    for v in graph.vertices:

        v.color = WHITE

    # visit every vertex

    for v in graph.vertices:

        if v.color == WHITE:

            TOPOLOGICAL_VISIT(v, stack)
```

```
function TOPOLOGICAL_VISIT(vertex, stack):

    # increase timestamp

    timestamp += 1

    vertex.color = GRAY

    for v in vertex.adjacent:

        if v.color == WHITE:

            DFS_VISIT(v)

    vertex.finished = timestamp

    stack.push_front(vertex)

    vertex.color = BLACK
```

Analysis

➢ Topological sort either uses a slightly modified depth first search algorithm or just uses DFS algorithm and then sorts the vertices in decreasing order of finish timestamps.

➢ Thus the time complexity of topological sort is the same as that of DFS.

➢ Time complexity of topological sort: $O(V + E)$

Applications

➢ Software compilation: Compilers use topological sort to find the order in which to compile source files according to their dependencies.

➢ Task scheduling: Similarly, CPU schedulers also use this sorting technique to find the order of tasks according to their dependencies.

➢ Project planning: It can be also useful while planning a complex projects.

Scheduling
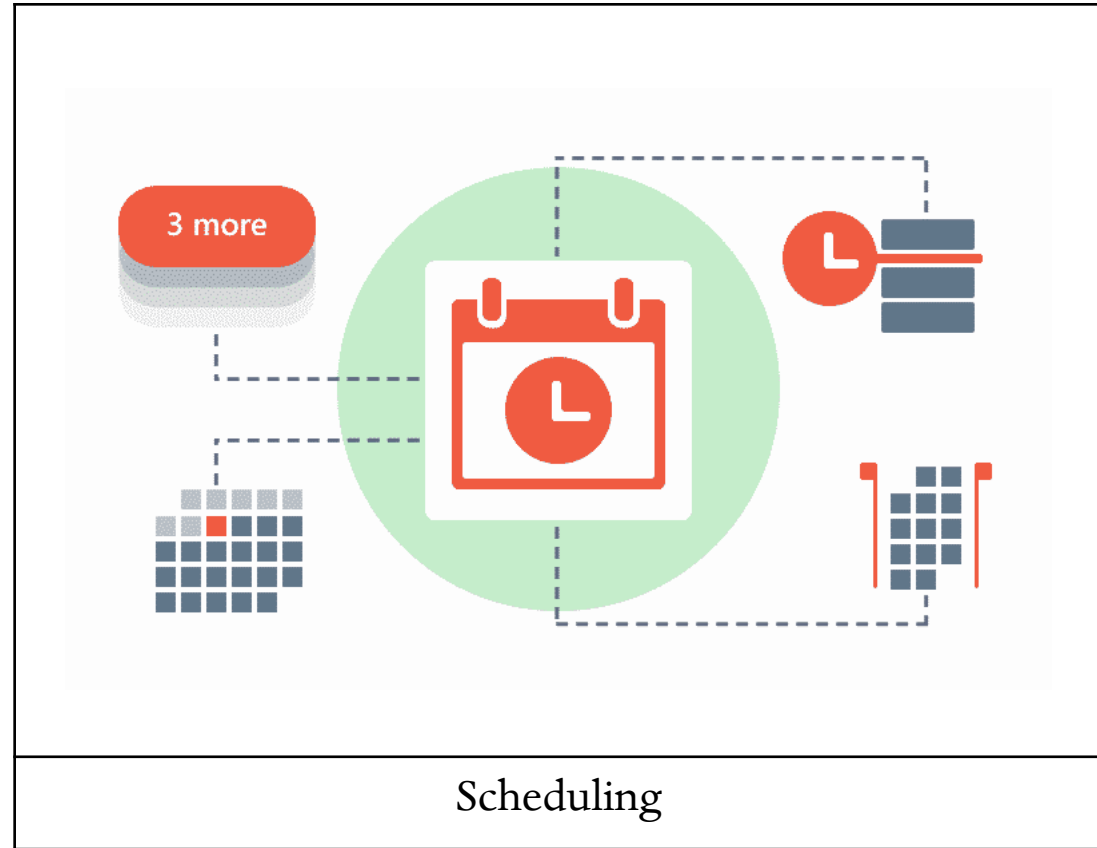
Strongly connected components

➤ What is a strongly connected component?

➤ Algorithm

➤ Analysis

What is strongly connected component?

➢ A strongly connected component or SCC of a directed graph is a maximal set of vertices where each vertex is connected to every other vertex.

➢ SCCs can be used to divide a complex graph into multiple sub-graphs which can be then worked upon individually.

➢ After the individual operations are complete, they can be then merged together.

Kosaraju's algorithm
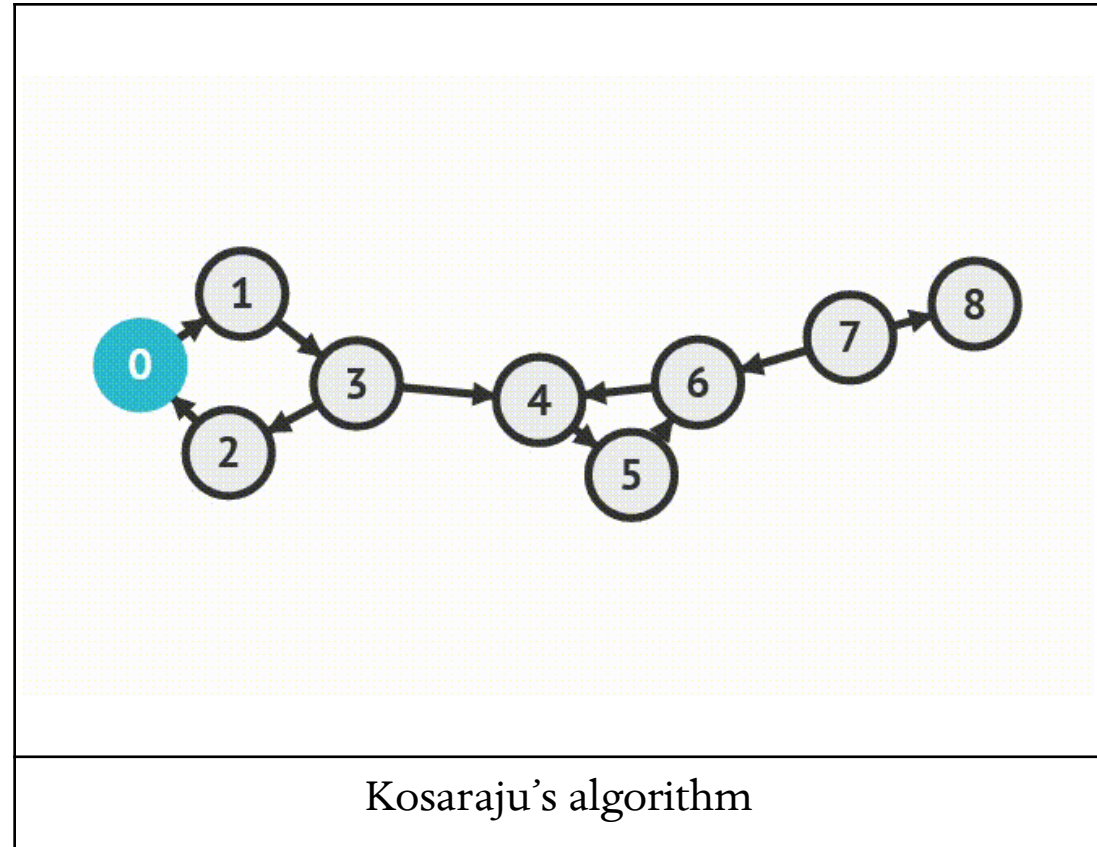
Algorithm

```
function STRONGLY_CONNECTED_COMPONENTS(graph):

    DFS(graph)

    TOPOLOGICAL_SORT(graph)

    transpose = TRANSPOSE(graph)

    # each tree in the forest is a strongly

    # connected component

    DFS(transpose)

    forest = transpose.forest

    return TRANSPOSE(forest)
```

```
function TRANSPOSE(graph):

    transpose = Graph()

    # reverse every edge direction

    for v in graph.vertices:

        for u in v.adjacent:

            graph.vertex[u] = v

    return transpose
```

Analysis

➢ Time complexity of DFS is $O(V + E)$

➢ During transpose, outer for loop runs V times and inner for loop cumulatively runs E times. Thus time complexity of transpose is also $O(V + E)$

➢ DFS on transpose graph is again $O(V + E)$

➢ Time complexity of finding strongly connected components is $O(V + E)$

Conclusion

➢ Applications of graph: I
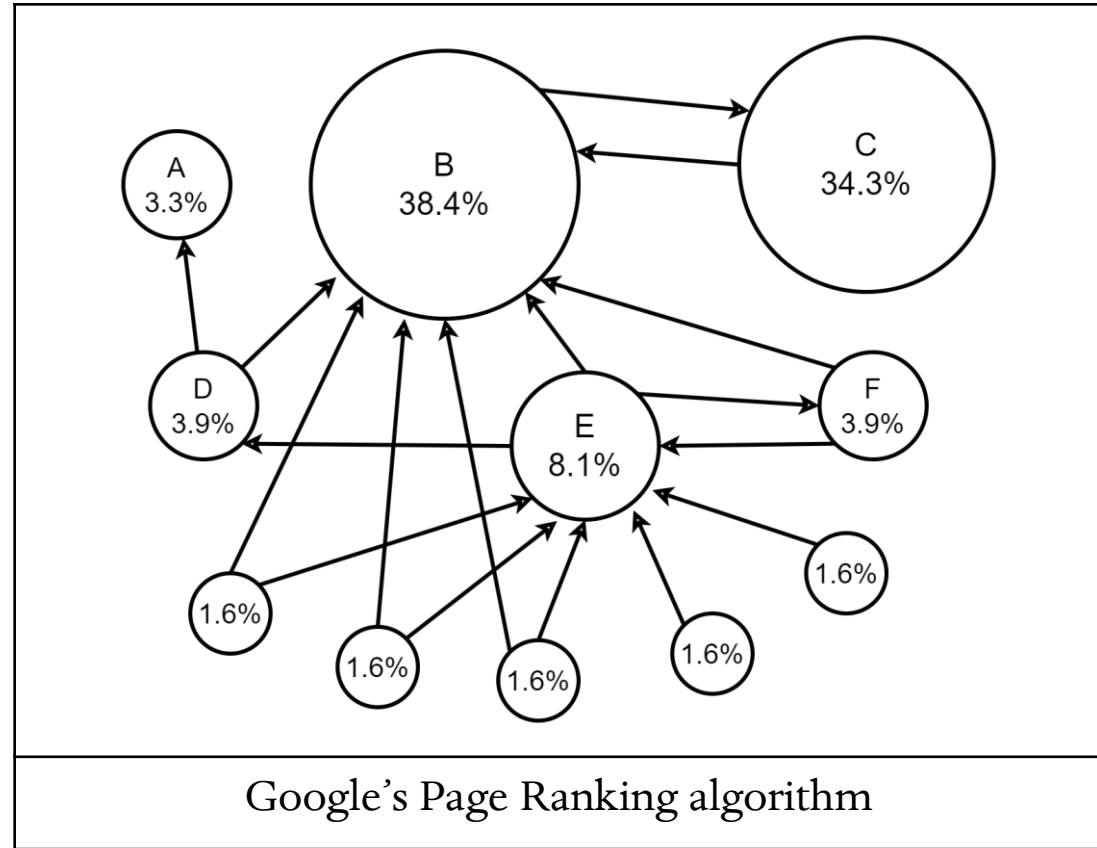
➢ Applications of graph: II

Applications of graph: I

➢ Maps: Locations in a map are vertices and the paths connecting them are edges.

➢ Social media: All the users are vertices and if two users are friends there is an edge between them.

➢ Resource allocation graphs: In operating systems, processes and resources are vertices and an edge represents if an user is using a resource.

Resource allocation graph
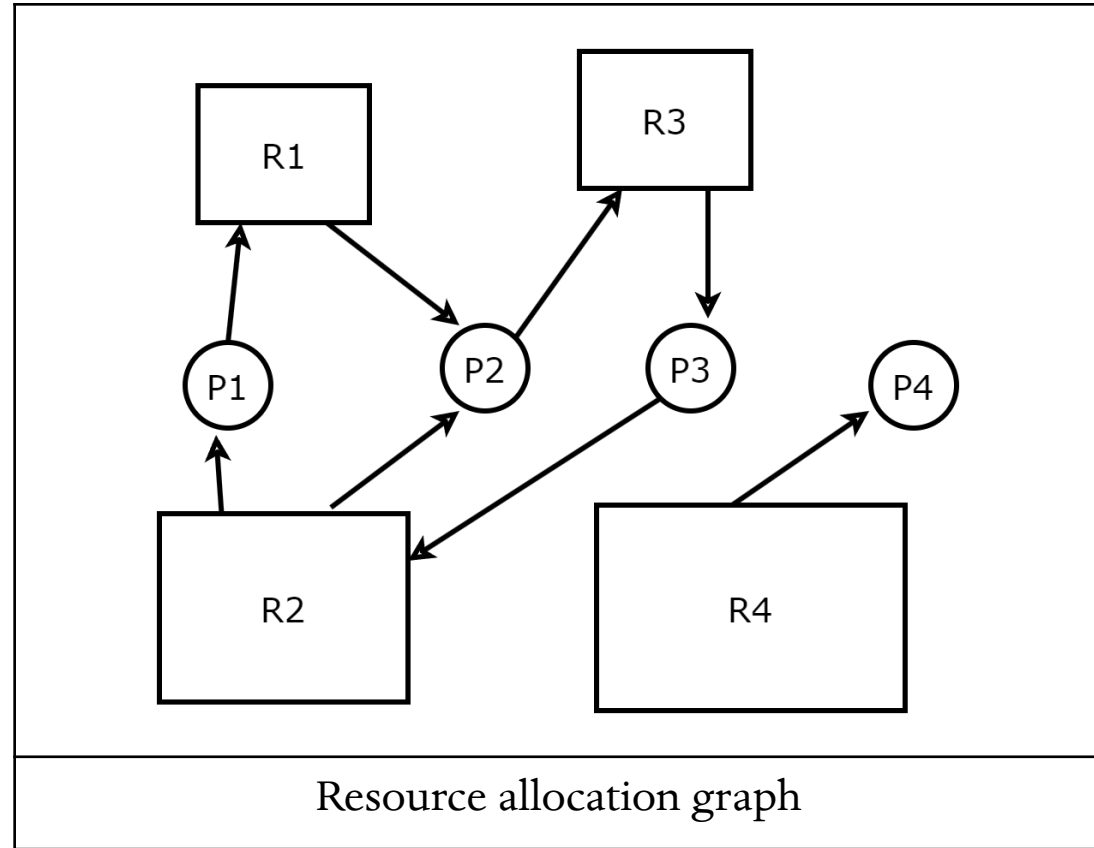
Applications of graph: II

➢ Web: All the pages are vertices and if two pages are linked an edge is created between the two pages.

➢ Google uses graph algorithms to find the popularity of webpages. The popularity of a page is decided on the basis of the number and popularity of the pages which have links to it.

➢ Graphs are also used to solve many real life problems, like path finding and scheduling.

Google's Page Ranking algorithm

Bibliography

➢ Introduction to Algorithms – Cormen

➢ https://visualgo.net

➢ https://www.iacr.org

➢ https://en.wikipedia.org

➢ https://codinginfinite.com

➢ https://www.researchgate.net

➢ https://www.geeksforgeeks.org

➢ https://courses.cs.washington.edu