Introduction to Breadth First Search & Depth First Search

Explanations, Comparisons, Implementations

Adrija Bhar
Shreya Chatterjee
Indian Statistical Institute, New Delhi

2024-04-03

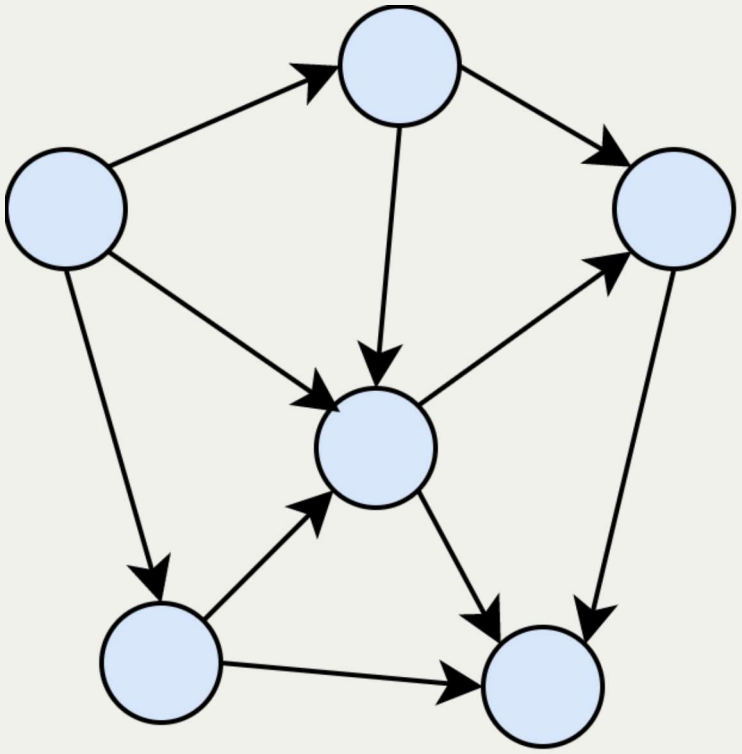
What is a Graph?

- Graphs are like **maps**.
- They have points, called **nodes**, and lines connecting these points, called **edges**.
- One can use graphs to show how things are connected in various areas, like
  - Social Networks (friends on social media)
  - Transportation Systems (Stops on Metro)
  - Computer Networks
- They help us understand relationships between different entities.

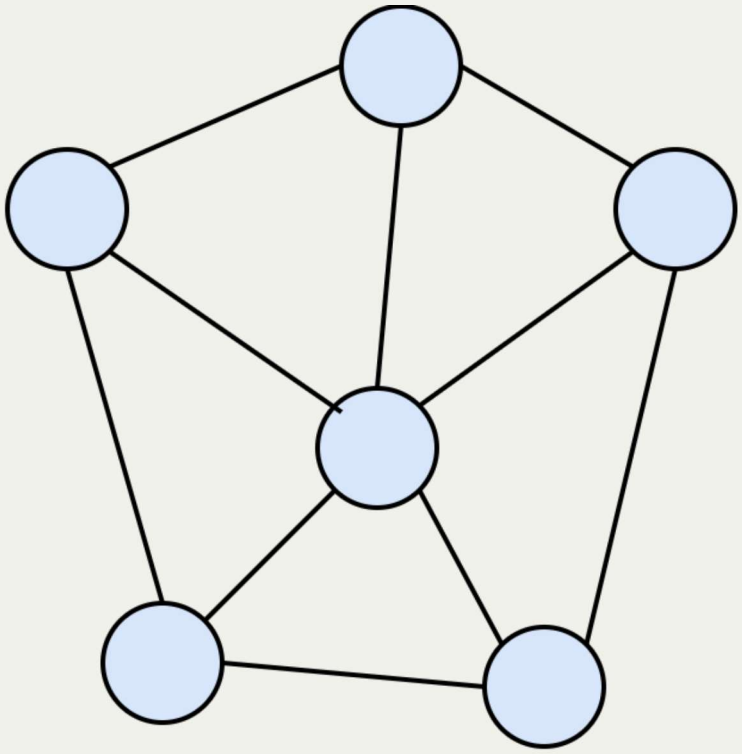
Types of Graphs

1.) Undirected and Directed

- In an undirected graph, edges have no orientation, it means one can go both ways along the edges.
- Conversely, directed graphs have directed edges, It's like driving on a one-way road where one can only drive a car in one direction, either forward or backward, not both.



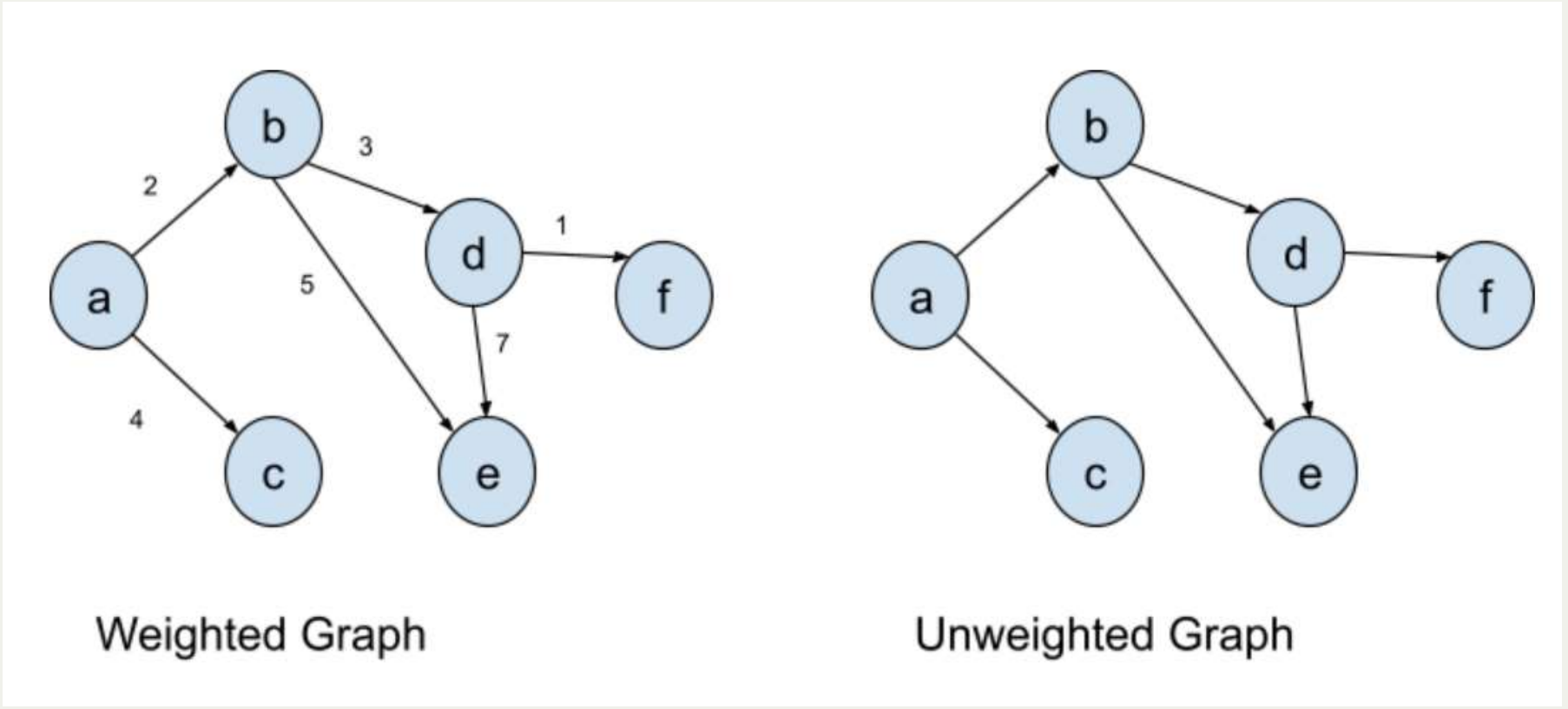
Directed Graph



Undirected Graph

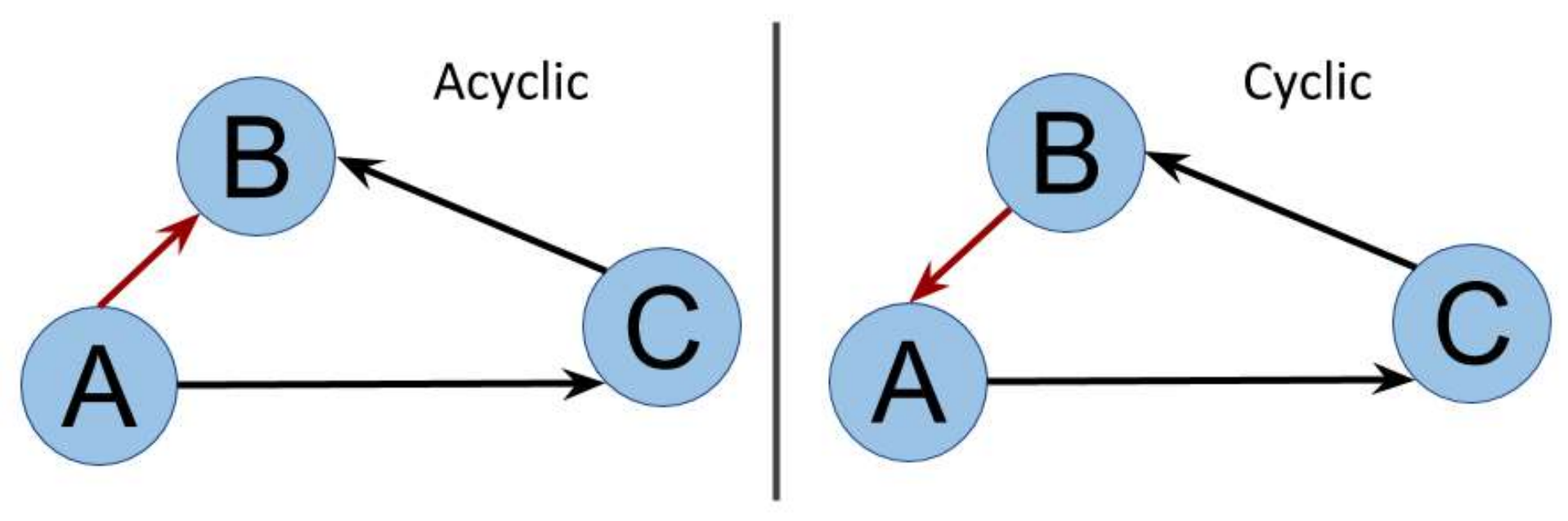
2.) Weighted and Unweighted

- In a weighted graph edges carry a numerical value, sometimes representing some cost or distance associated with travelling along the edge.
- Unweighted graphs, do not assign any values to edges. It's like all routes are considered equal, without any specific cost or distance associated with them.



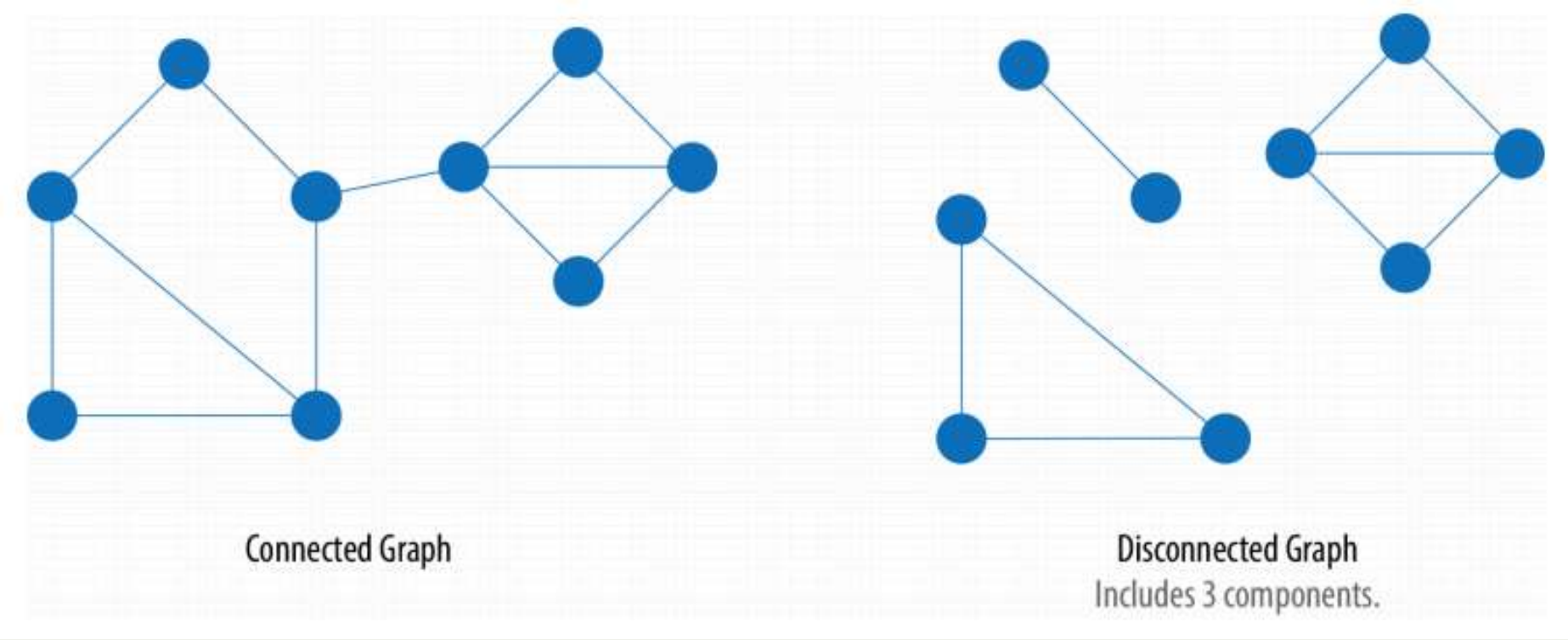
3.) Cyclic and Acyclic

- If a graph has a way to go from one point and come back to the same point, it's called cyclic. It's like walking in circles, ending up where you started.
- But if there's no way to do that in a graph, it's called acyclic. It's like walking in one direction and not coming back to where you began.



4.) Connected and Disconnected

- In a connected graph, you can travel from any point to any other point by following the edges. It's like having a network where everything is linked together, and you can reach any destination from any starting point.
- But in a disconnected graph, there are more then one separate components that aren't directly connected. It's like having islands of points that can't be reached from each other.



Different Methods for Representing Graphs

1.) Adjacency Matrix

The adjacency matrix is like a grid with N rows and N columns, where N is the total number of vertices in the graph. Each cell in the grid shows whether there is a connection between two vertices. If there's a connection, the cell has a 1; if not, it has a 0. It helps us quickly see which vertices are connected to each other in the graph.

2.) Adjacency List

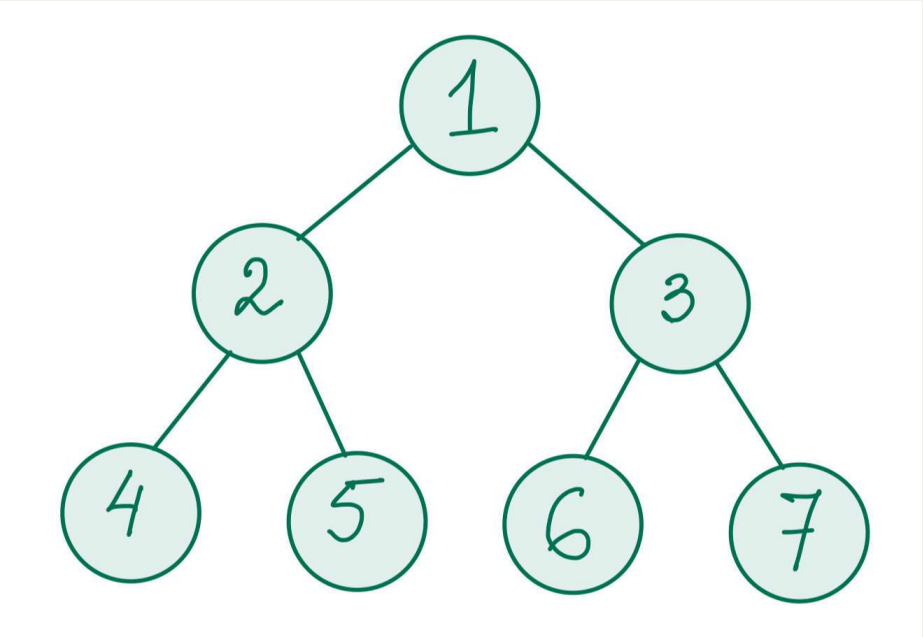
In the adjacency list representation, every vertex in the graph has a list that shows its neighboring vertices. This method works well for graphs where there aren't many connections between vertices, which we call sparse graphs. It's efficient because it only stores the connections that actually exist, rather than keeping track of every possible connection.

3.) Edge List

An edge list is a straightforward way to represent a graph. **It's like having a list that shows all the connections between vertices.** Each entry in the list tells us which two vertices are connected by an edge. It's a simple method to see all the relationships in the graph.

Representation we have used!

Adjacency Matrix:



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

Graph Traversal Algorithms:

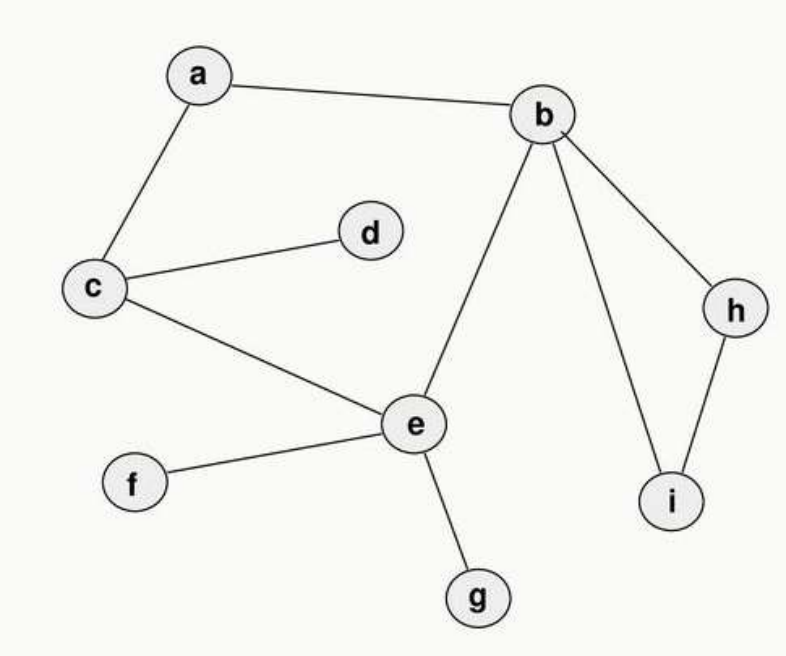
- Graph traversal algorithms are important methods we use to move around and explore graphs.
- They help us solve different kinds of problems involving graphs, like finding the shortest way between two points, figuring out if there are loops in the graph, and seeing how everything is connected.
- These algorithms are like the maps and routes we use to get around in a city—they guide us through the different parts of the graph so we can find what we're looking for.

Graph Traversal Algorithms:

While traversing a graph, the order in which the nodes are visited may be important, and may depend upon the particular algorithm.

The two common traversals:

- **Breadth First Search (BFS)**
- **Depth First Search (DFS)**



Breadth First Search (BFS)

The Breadth-First Search (BFS) algorithm is well-known for its capability to traverse all the vertices of a graph in a breadth-oriented manner.

BFS works with graphs, which are made up of points (vertices) and the lines connecting them (edges).

The main job of BFS is to go through all the points in a graph, starting from one specific point. **It begins by looking at all the points directly connected to the starting point. Then, it moves on to the points connected to those, and so on.** This way, it ensures that all points at the same distance from the starting point are visited before moving to points farther away.

BFS looks at all the points at the same distance from the starting point before moving on to points that are farther away. It's like going through a maze and looking at nearby paths before moving farther away.

This makes BFS really helpful when we want to find the shortest path between two points or understand how everything in the graph is connected.

The BFS algorithm works as follows:

1. Start by putting root node of a graph at the back of a queue.
2. Pick out the top item from the queue and add it to the visited list.
3. Create a list of adjacent nodes of that node.
4. The adjacent nodes that are not yet visited are added at the back of the queue.
5. Repeat the steps 2, 3 and 4, until the queue is empty.

Implementing Breadth-First Search in Code

Here we implement BFS to obtain the sequence of visited nodes of a connected, undirected graph, considering any of the nodes as root node.

▶ Code

Explanation of the code for BFS:

1. The **"bfs"** function takes 4 arguments:

- The adjacency matrix of the graph or tree, named as **"graph"**,
- The starting node, named as **"vertex"**
- The vector denoting the nodes that are visited, named as **"visited"**
- The vector of visited nodes in chronological order, named as **"seq_visit"**.

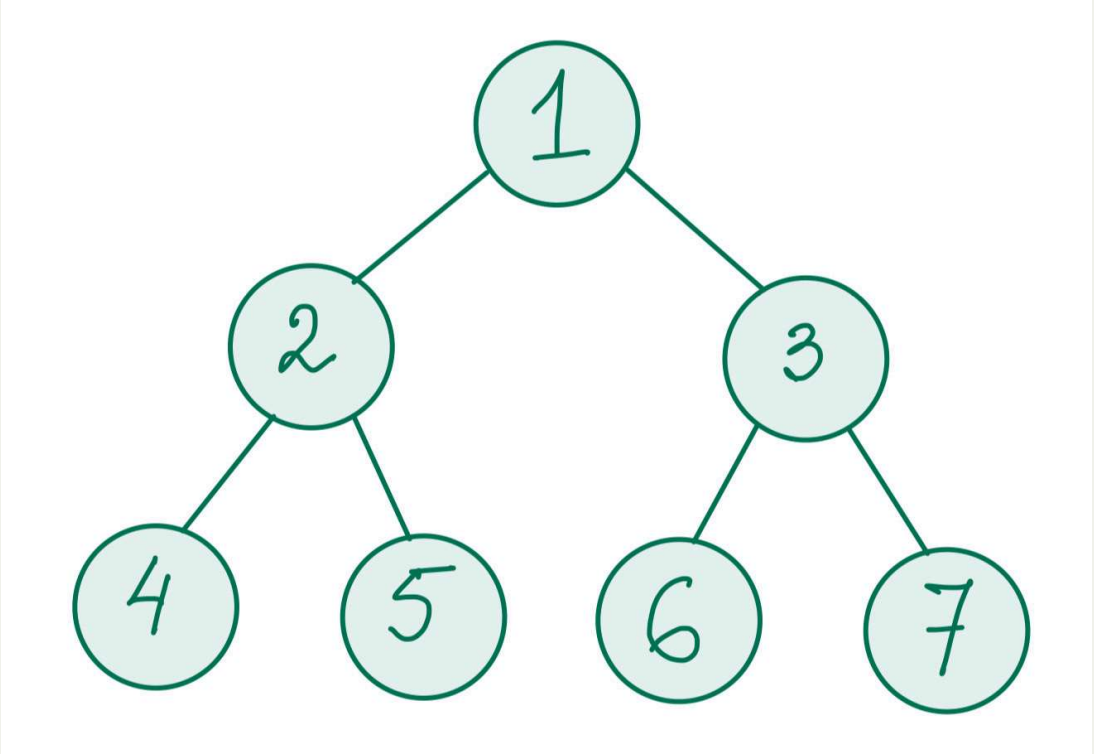
2. Initially when the function is called, the inputs are the adjacency matrix, the root node, a zero vector of length same as number of nodes and again the root node indicating that it is going to be visited at first.
3. If the starting node is the p-th node, then p-th node is added to the **"queue"** and it is visited, i.e., 1 is stored in the p-th position of **"visited"** vector.
4. The first node from the **"queue"** is popped and stored as **"node"**.
5. In the adjacency matrix, if i-th node is a neighbour of node (i.e., if (node,i)-th element is 1) and i-th node is not visited, then it is now visited, added to the vector **"seq_visit"** and also added at the back of the **"queue"**
6. The steps 4 and 5 is repeated, until the **"queue"** is empty.
7. Finally, **"seq_visit"** gives the sequence of nodes visited in chronological order.

Implementing Breadth-First Search in Code - 1

Representing Graph in R

▶ Code

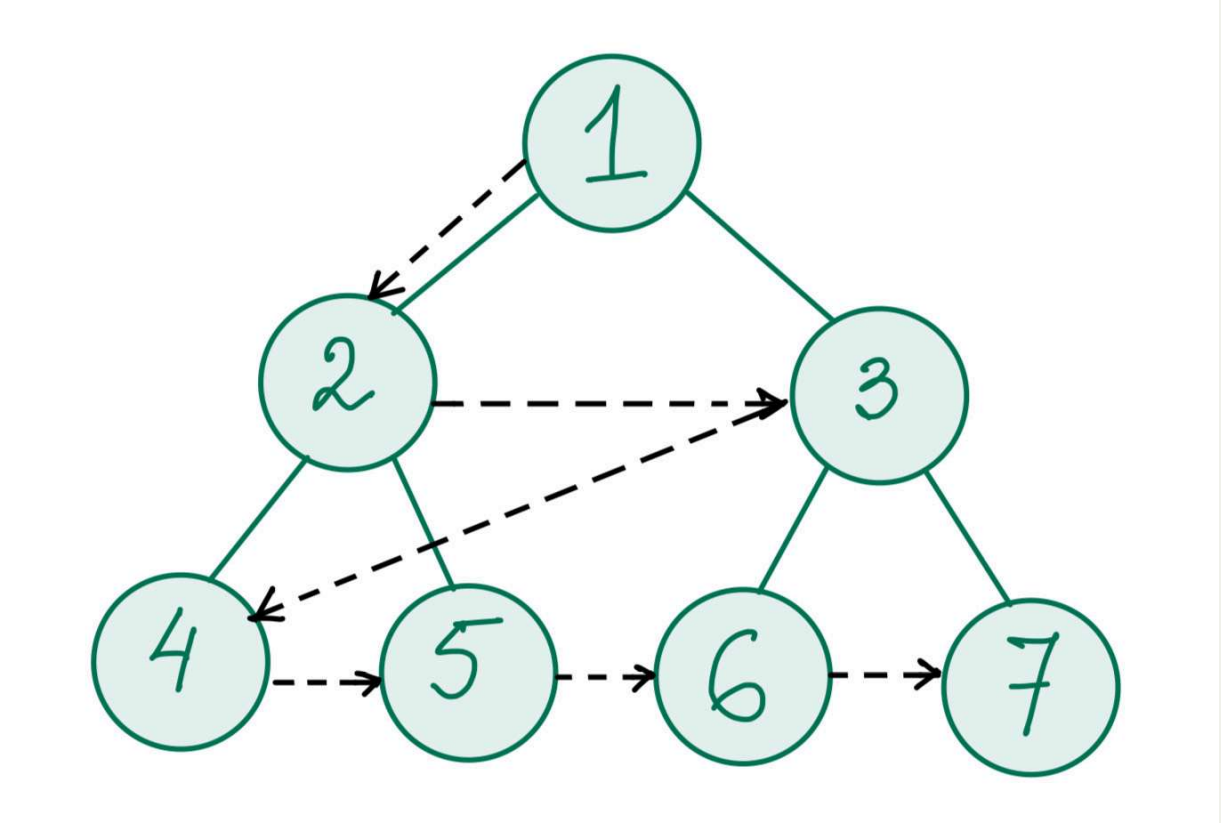
```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    1    1    0    0    0    0
[2,]    1    0    0    1    1    0    0
[3,]    1    0    0    0    0    1    1
[4,]    0    1    0    0    0    0    0
[5,]    0    1    0    0    0    0    0
[6,]    0    0    1    0    0    0    0
[7,]    0    0    1    0    0    0    0
```



Output after Applying bfs() function

▶ Code

```
[1] 1 2 3 4 5 6 7
```

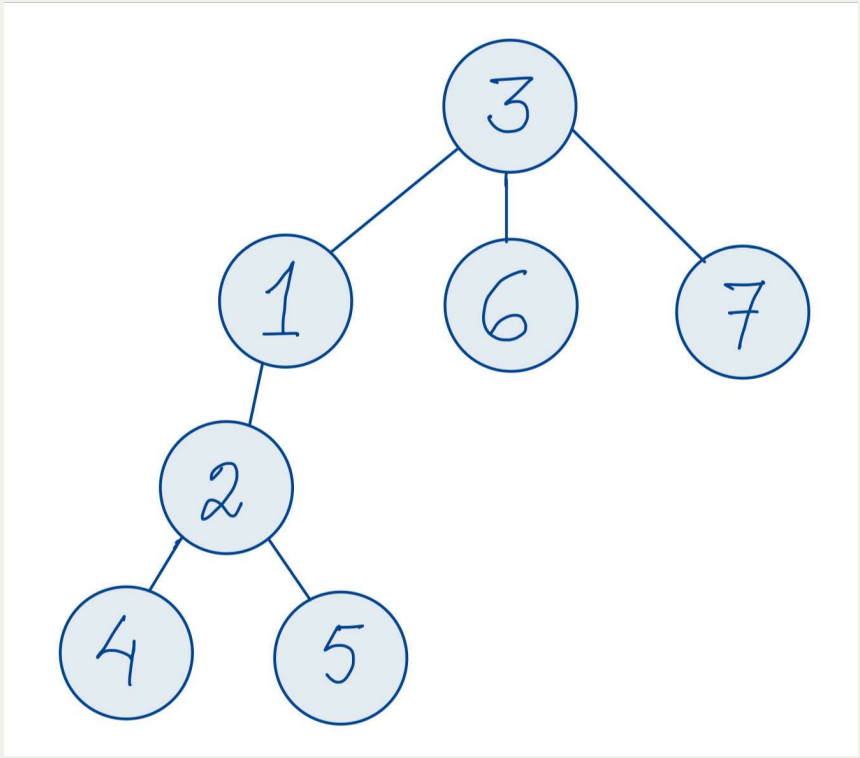


Implementing Breadth-First Search in Code - 2

Representing Graph in R

▶ Code

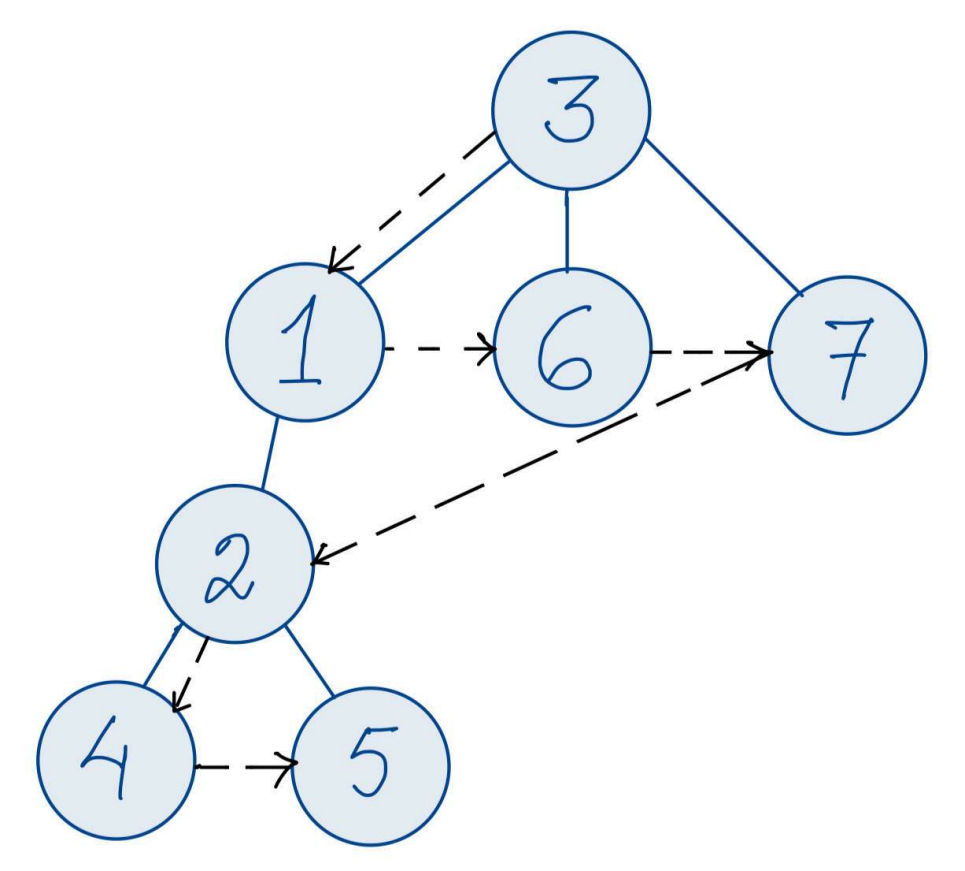
```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    1    1    0    0    0    0
[2,]    1    0    0    1    1    0    0
[3,]    1    0    0    0    0    1    1
[4,]    0    1    0    0    0    0    0
[5,]    0    1    0    0    0    0    0
[6,]    0    0    1    0    0    0    0
[7,]    0    0    1    0    0    0    0
```



Output after Applying bfs() function

▶ Code

```
[1] 3 1 6 7 2 4 5
```



What is Time Complexity ?

- **Time complexity measures how the runtime of an algorithm grows as the size of the input increases.** It's not about the actual time taken to run the algorithm on a specific machine, but about how the runtime changes with the size of the problem.
- An algorithm will always take a finite amount of time to complete, but its time complexity tells us how this time scales with the size of the input. **To estimate time complexity, we consider the cost of each basic operation and how many times these operations are executed.**
- Imagine you're baking cookies, and the recipe tells you to mix flour and sugar together. No matter how many cookies you're making, mixing these ingredients will always take the same amount of time. That's like an algorithm with constant time complexity, denoted as O(1). It means the time it takes doesn't change with the size of the input.
- In our baking example, adding two scalar numbers is similar to mixing flour and sugar—it always takes the same amount of time, no matter how big the numbers are. So, the time complexity of adding two scalar variables is constant, represented as T(n) = O(1).

Time Complexity of BFS:

In this technique, we explore neighboring vertices by inserting them into a queue if they haven't been visited yet. We do this by examining the edges of each vertex. **Once we visit a vertex, we mark it as visited. So, each vertex is visited only once, and we check all the edges of each vertex.**

The time complexity of BFS is O(V+E) because:

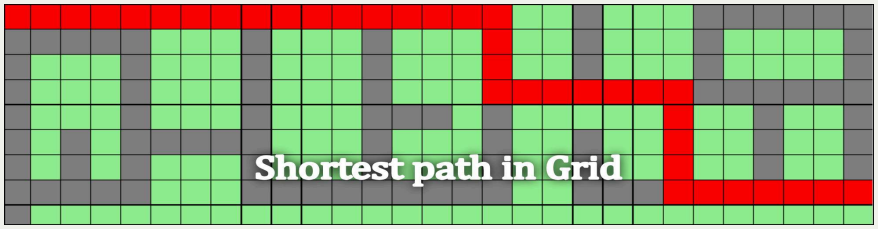
- It visits each vertex once, which adds V to the complexity.
- It checks each edge once when moving from one vertex to its neighboring vertices, contributing E to the complexity.

Since we're only going through the edges and vertices of the graph once, the time complexity for both BFS is linear, denoted as O(V+E).

Applications of BFS

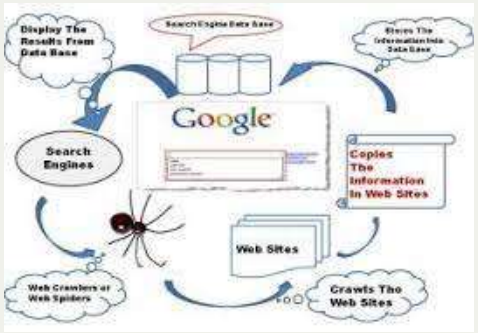
1) Shortest Path Find:

- **BFS can be used in finding the shortest way from one place to another on a map.** Imagine we have a city map where each point is a location and the lines are roads connecting them. If we want to find the quickest route from your home (let's say point A) to a friend's house (point G), we can use BFS.
- Here's how it works: We start at your home (point A) and look at all the nearby locations (nodes). Then, we move to the next set of locations, which are a bit farther away, and so on, gradually expanding your search outward.
- For example, if we're at point A (your home) and we know that point B is our neighbor because there's a road directly connecting them, we'll check point B first. Then, we'll look at all the places connected to point B, like point C and point D. We keep doing this until you reach your friend's house (point G).
- BFS helps us find this shortest path by keeping track of the distance from our home to each location and using a queue to manage which places to check next. This way, we can navigate through the city efficiently and find the quickest way to our friend's house.



2) Web Crawling:

- **Web crawling is like systematically browsing through web pages to gather information**. It's similar to how you might search through a library to find books on a specific topic.
- For example, let's say you're building a website about different types of flowers. You want to gather information from various websites to include on your site. You could use web crawling to do this.
- Starting from a main flower website, you could use BFS to explore all the links on each page. This way, you'd make sure to visit every linked page and gather information about different types of flowers.
- For instance, if you're on a page about roses and it has links to pages about tulips, daisies, and sunflowers, **BFS would help you visit each of those linked pages systematically.**
- This approach is often used by search engines like Google to index web pages and build their databases, so when you search for something, you get relevant results from all over the web.



3) Social Network Analysis:

- BFS is not only useful for web crawling but also for analyzing social networks. Imagine you're exploring connections between people on a social media platform like Facebook or Twitter.
- For example, let's say you want to find the shortest path between two friends on Facebook. You could use BFS to do this. You'd start from one friend and explore their immediate friends first. Then, you'd move on to the friends of those friends, and so on, until you reach the other friend.
- By doing this, BFS helps you find the shortest path between the two friends, showing how they're connected through mutual friends. This can be useful for understanding relationships, identifying key individuals, or even finding communities within the social network.
- So, BFS is not just about exploring web pages; it's also a powerful tool for uncovering relationships and connections between people in social networks.

Depth First Search (DFS):

DFS (Depth First Search) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. **It starts at a selected node (often called the 'root') and explores as far as possible along each branch before backtracking.** This process continues until all nodes have been visited. DFS can be implemented using recursion or by using a stack data structure. It is commonly used in solving problems such as maze traversal, topological sorting, and finding connected components in graphs.

Understanding the DFS algorithm:

1. Start by putting root node of a graph on top of a stack.
2. Pick out the top item from the stack and add it to the visited list.
3. Create a list of adjacent nodes of that node.
4. The adjacent nodes that are not yet visited are added to the top of the stack.
5. Repeat the steps 2, 3 and 4, until the stack is empty.

Implementing Depth-First Search in Code:

▶ Code

Explanation of the code:

1. The **"dfs"** function takes 4 arguments:

- The adjacency matrix of the graph or tree, named as **"graph"**,
- The starting node, named as **"vertex"**
- The vector denoting the nodes that are visited, named as **"visited"**
- The vector of visited nodes in chronological order, named as **"seq_visit"**.

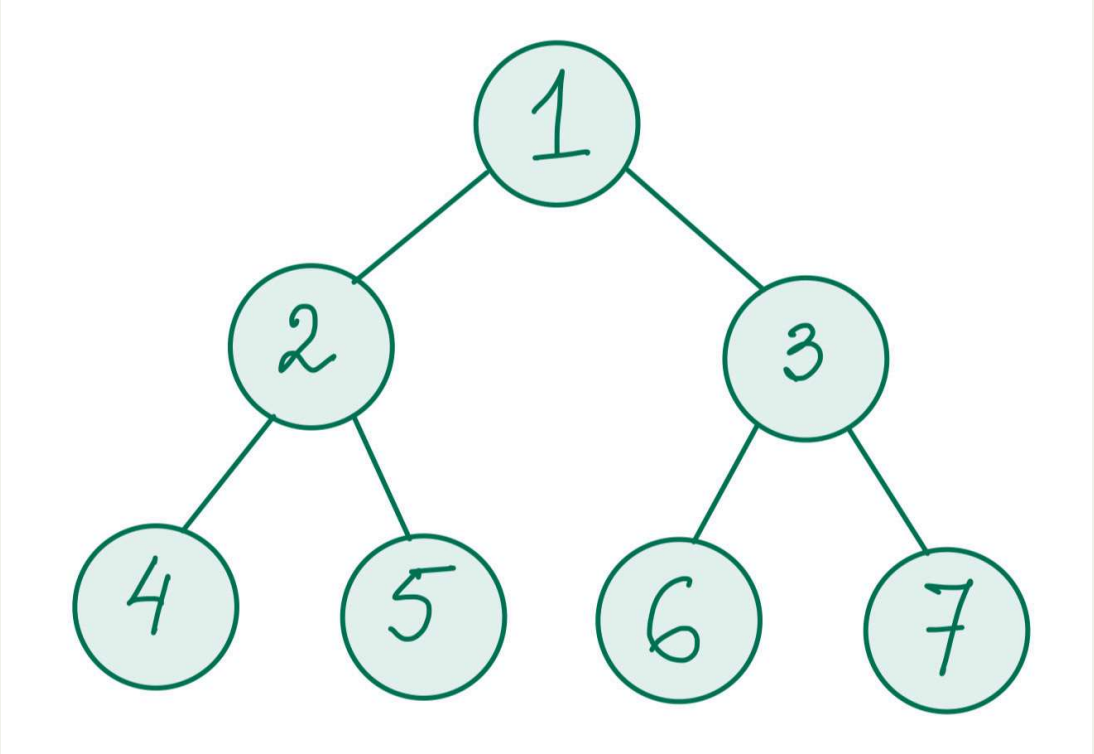
2. Initially when the function is called, the inputs are the adjacency matrix, the root node, a zero vector of length same as number of nodes and again the root node indicating that it is going to be visited at first.
3. If the starting node is the p-th node, then p-th node is visited, i.e., 1 is stored in the p-th position of **"visited"** vector.
4. In the adjacency matrix, if (p,i) th element is 1, then i-th node is a neighbour of p-th node. All the neighbour nodes are stored in the vector **"n".**
5. One node from **"n",** that is not yet visited, is now visited, added to the vector **"seq_visit"** and its child nodes are visited by recursively calling **"dfs"** function.
6. At the end of recursion, again another unvisited node from **"n"** is visited and so on.
7. Finally, **"seq_visit"** gives the sequence of nodes visited in chronological order.

Implementing Depth-First Search Algorithm in Code - 1

Representing Graph in R

▶ Code

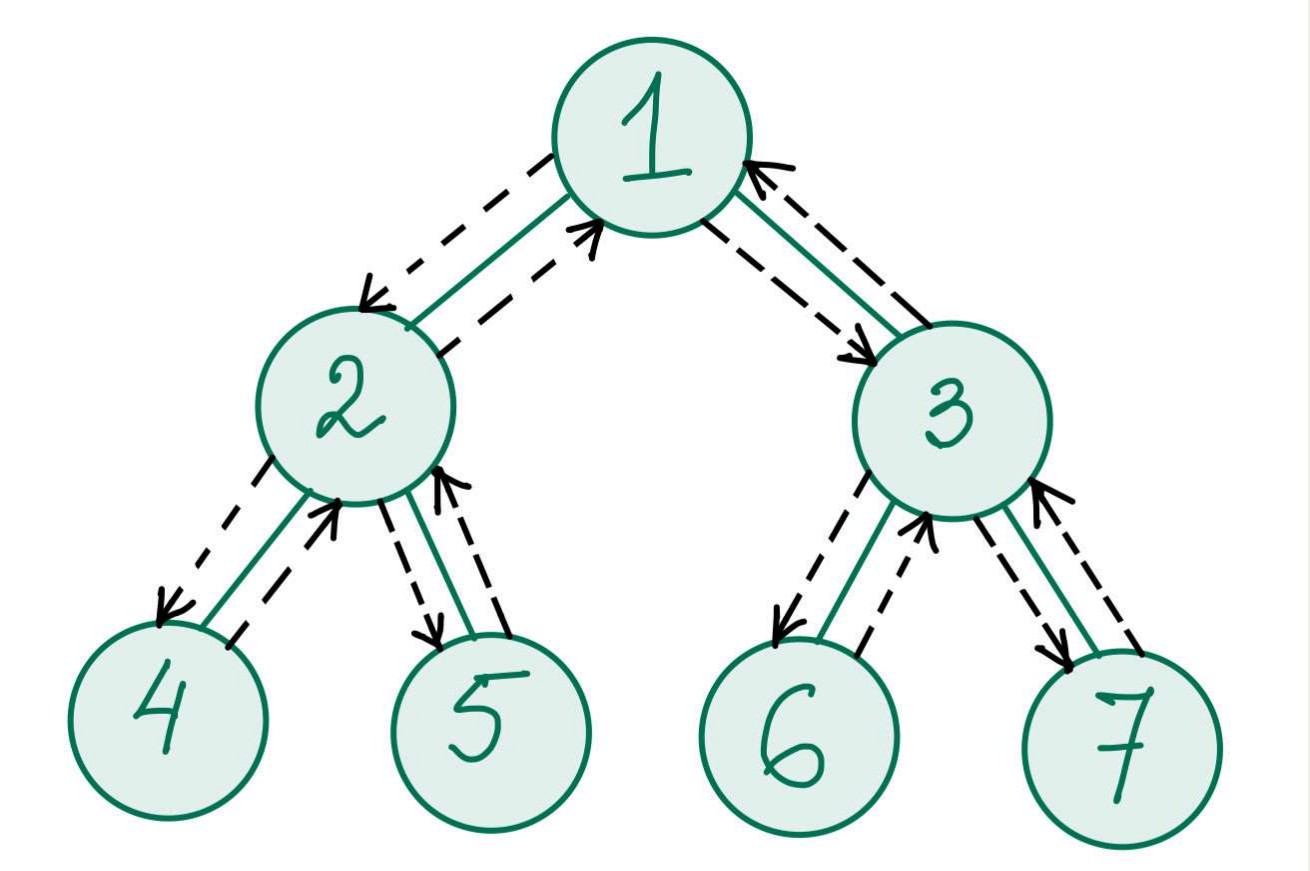
```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    1    1    0    0    0    0
[2,]    1    0    0    1    1    0    0
[3,]    1    0    0    0    0    1    1
[4,]    0    1    0    0    0    0    0
[5,]    0    1    0    0    0    0    0
[6,]    0    0    1    0    0    0    0
[7,]    0    0    1    0    0    0    0
```



Output after Applying dfs() function

▶ Code

```
[1] 1 2 4 5 3 6 7
```

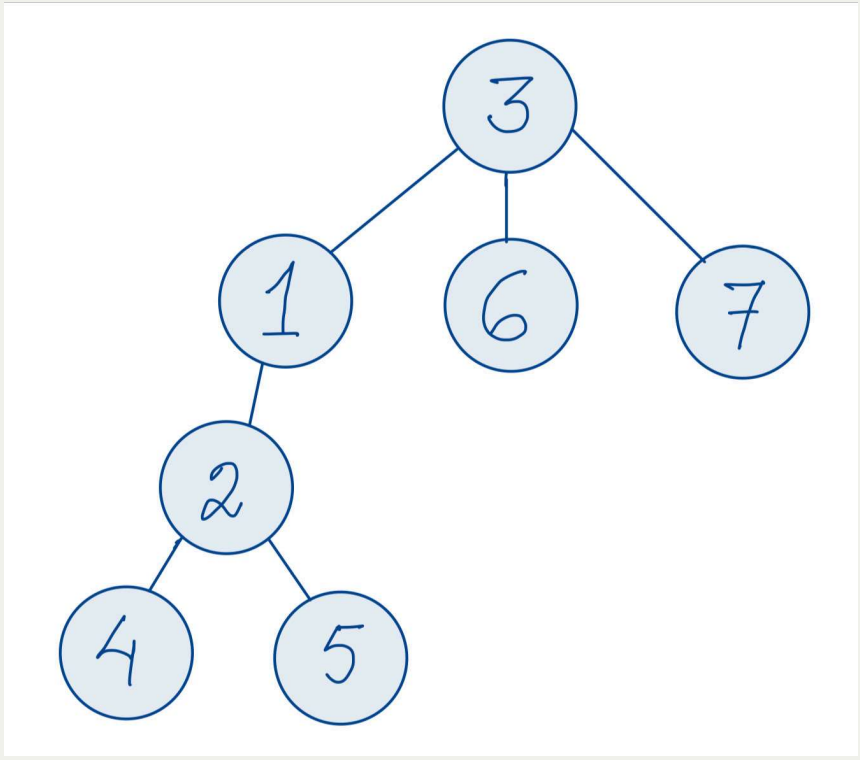


Implementing Depth-First Search Algorithm in Code - 2

Representing Graph in R

▶ Code

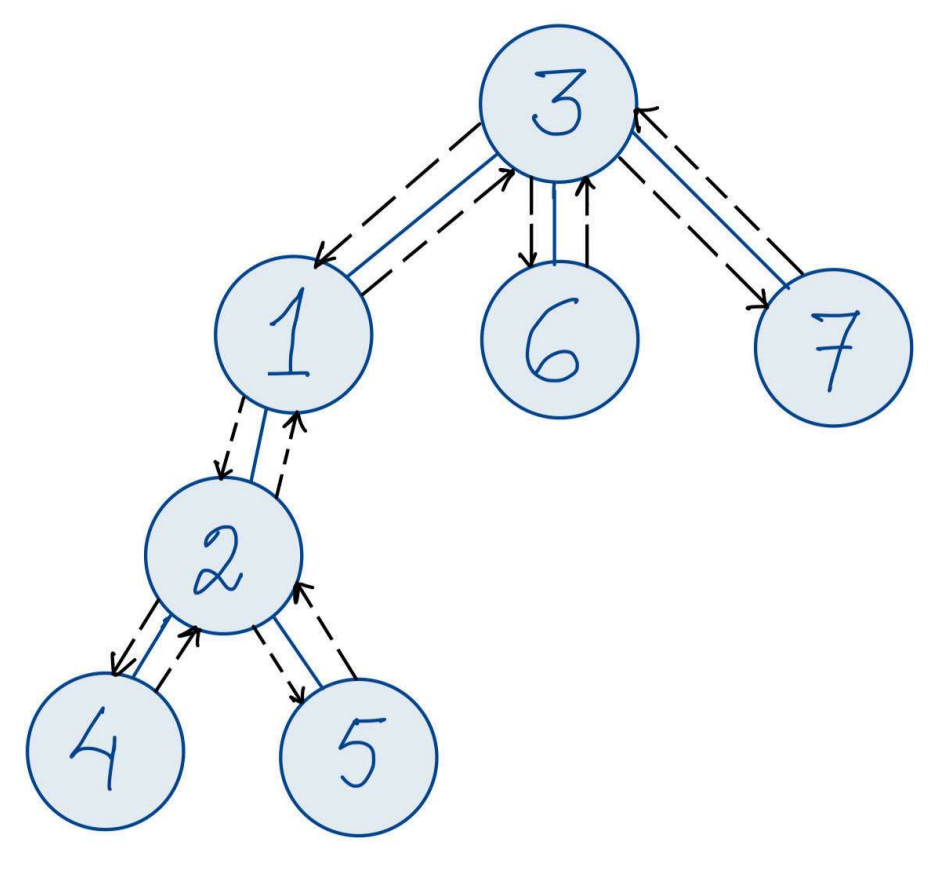
```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    1    1    0    0    0    0
[2,]    1    0    0    1    1    0    0
[3,]    1    0    0    0    0    1    1
[4,]    0    1    0    0    0    0    0
[5,]    0    1    0    0    0    0    0
[6,]    0    0    1    0    0    0    0
[7,]    0    0    1    0    0    0    0
```



Output after Applying dfs() function

▶ Code

```
[1] 3 1 2 4 5 6 7
```



Time Complexity of DFS:

When we use Depth-First Search (DFS), we go through each node and edge exactly one time. **If we come across a node we've already visited, we backtrack, which means we stop exploring that path.** This helps us avoid repeating the same steps and reduces the overall time it takes to finish.

The time complexity of DFS is O(V+E) because:

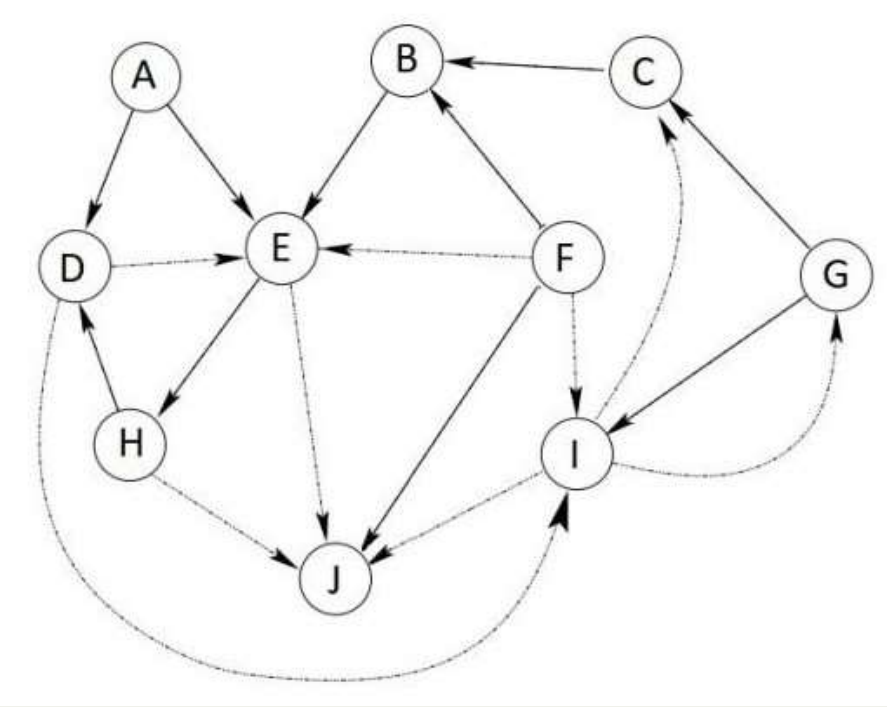
- It visits each vertex once, so that's V times.
- It checks each edge once when moving from one vertex to another, which adds E to the complexity.

So, the overall time it takes for DFS is proportional to the number of vertices (V) plus the number of edges (E), which makes it a linear complexity algorithm.

Applications of DFS

1) Finding Connected Components:

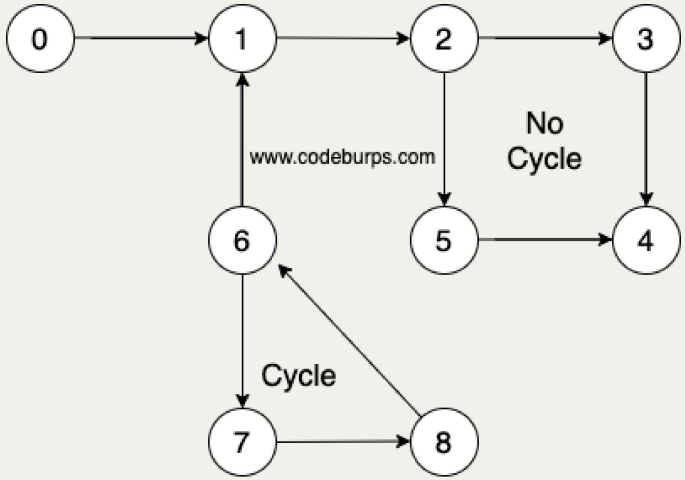
- DFS is a useful tool for discovering connected groups within a graph. A connected group is a bunch of points that are all linked to each other. **By running DFS on each point that hasn't been explored yet, we can pinpoint all the connected groups in a graph.**
- Imagine you have a group of friends in a social network. Each friend represents a point, and friendships between them are the connections. If you wanted to find all the groups of friends who are connected to each other (meaning you can reach any friend from any other friend through mutual connections), you could use DFS.



2) Detecting Cycles in a Graph:

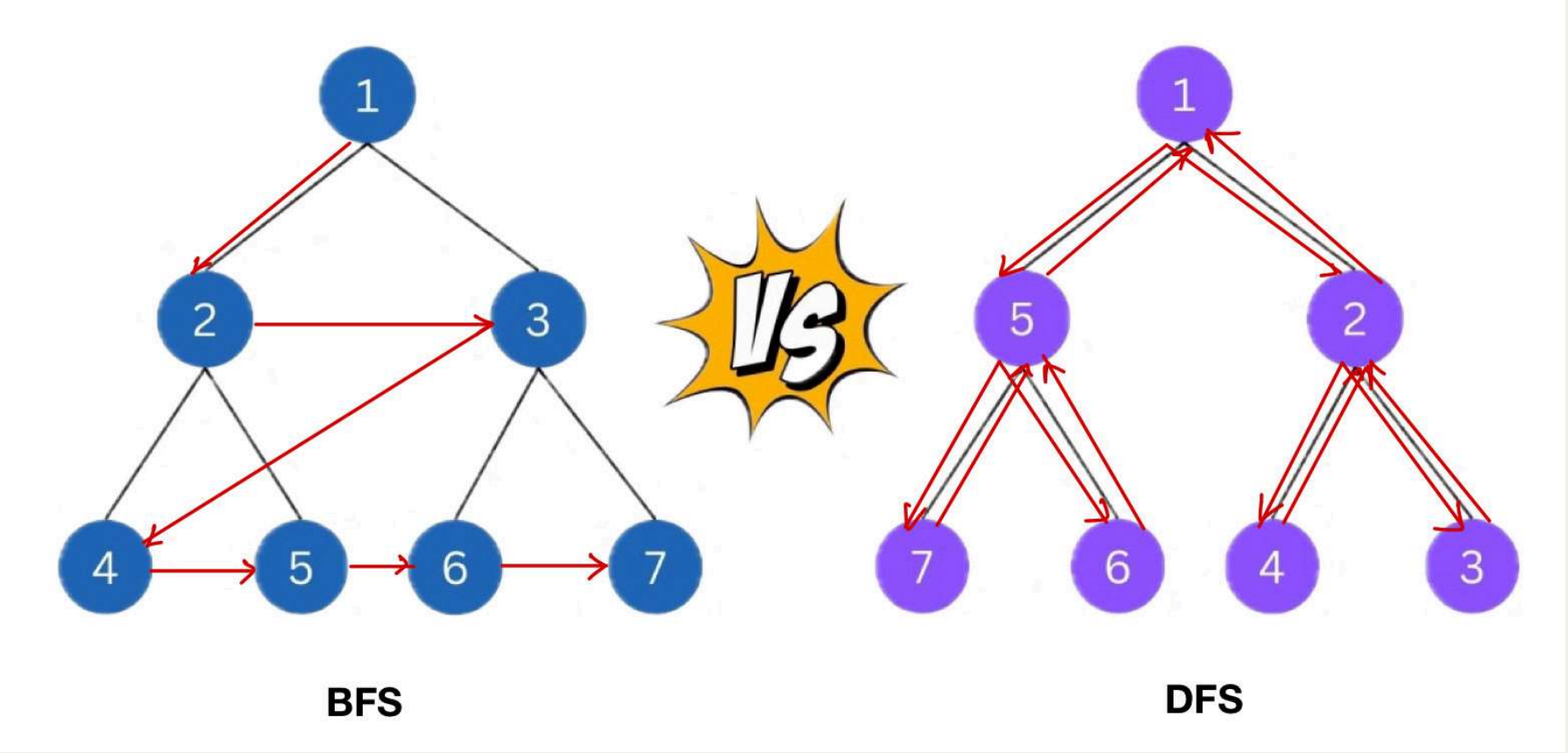
- DFS is not only useful for finding connected components but also for detecting cycles within a graph. A cycle happens when there's a path that starts and ends at the same point in a graph.
- Think about planning a road trip with your friends, where each destination represents a point in the graph, and the roads between them are the connections. If you accidentally end up back at the same place you started without planning to, then you've encountered a cycle in your trip plan.

- So, in real life, if you're planning a trip and accidentally find yourself back where you started without intending to, you've likely encountered a cycle in your trip plan. Similarly, in the code, **if DFS encounters a point that it's already visited, except if it's the starting point, then it knows there's a cycle in the graph.**



Detect cycle in a directed graph

Comparison Between BFS and DFS



	BFS	DFS
Data Structure	BFS uses Queue data structure for finding the shortest path.	DFS uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.

Implementing BFS, DFS Algorithm in Solving Sudoku

About Sudoku:

- Sudoku is a logic -based, combinatorial number-placement puzzle. A classic Sudoku consists of 9 × 9 grid with nine 3 × 3 sub-grids (also called “blocks”) that compose the grid. The objective is to fill the 9 × 9 grid without violating the rule that, each of the 9 columns, each of the 9 rows, and each of the nine 3 × 3 sub-grids contain all of the digits from 1 to 9.
- In a sudoku problem some of the boxes are prefilled and the task is to fill the empty boxes.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Implementing Searching Algorithm to Solve Sudoku:

- The searching algorithms are applied on a graph or tree consisting some nodes and edges. In a problem of Sudoku solving, we define the nodes to represent a ‘state’ of the game.
- A state of the game is meant to be a snapshot of the Sudoku board, every edge in the tree represents an ‘action’, i.e. entering a possible value in an empty cell.
- The initial state of the game is considered as the root node of the tree.
- Starting from a given node S, i.e. a state of the board, child nodes of S could consist of derived states which have first empty cell of S filled with a digit that does not violate the sudoku rules. The search of first empty cell is done according to a left-to-right, top-to-bottom scan.
- Our actual goal is completing the board, i.e. finding a node whose state of the board comes without empty cells.

Implementing BFS Algorithm in Solving Sudoku:

BFS algorithm to solve sudoku:

- Check legality of the initial state. If it is legal, print the state and break.
- Otherwise, initialize a queue and put the node generated by the initial state into it.
- While the queue is not empty:
  - Pop the first node from the queue.
  - Find the first empty cell of the state of the node.
  - Get all possible values between 1 and 9 that can be entered in that cell.
  - Enter each possible value in that cell to generate a child node of current node.
  - If state of any of the child nodes is legal, then print the state and break.
  - Otherwise, store the child nodes in the queue.
- If all possible nodes are explored but none of them is legal, then no solution exists.

Code

- Code

Explanation of the code involving BFS to solve sudoku:

- The function “BFS_solve” takes the initial sudoku board as argument. From this initial sudoku board, the class “Problem” is generated.
- The function “BFS” uses this “Problem” as argument
- Using state of sudoku board involved in “Problem”, another class “Node” is generated, which is taken as the root node, defined as “start”.
- Then legality of the state is checked using the function “check_legal” (defined in the class “Problem”), taking state as argument.
  - In this function legality of each row, column and block are checked. If any of the row-sum or column-sum or block-sum is NOT EQUAL to sum of the numbers 1 to up to 9 (i.e., 45) or ther exists any duplicate row, column or block, then the state is not legal, i.e., it needs to be modified. So, the function returns FALSE.
  - Otherwise, the state is legal, i.e., no sudoku rule is violated. So, the function returns TRUE.
- If “check_legal” returns TRUE, then the initial state is the final solution and is returned by “BFS”. Otherwise, we opt for next steps.
- As BFS works using a queue following FIFO principle, so an empty queue is created, where the root node “start” is stored.
- While the queue is non-empty, the following steps are followed:
  - First node in the queue is picked out and is expanded using the function “expand” to obtain i child nodes.
  - The function “expand” belongs to the class “Node” and it takes the class “Problem” as argument. In this function,

- From “Problem”, using the function “get_spot” having state as an argument, the position of first empty cell of the state is obtained in terms of (row, column), where the search for first empty cell is done by scanning from left to right, top to bottom.
- From “Problem”, using the function “filter_values” having two argument vectors of all possible values and the used values, the values that can be entered in that empty cell without violating the Sudoku rules are developed.
- From “Problem”, using the function “action” having state as argument, the vectors (possible value, row, column) for the first empty cell of the state are stored in the list “result”, which is again stored as “actions” within the function “expand” in the class “node”.
- From “Problem”, using the function “result” having state and action as arguments, all possible next states are produced and stored in “next_state” using each action present in the list “actions”, i.e., the possible value present in the vector action is entered in the position given by (row, column) in action.
- Using “next_state” and action vector, the child nodes are produced and stored in “children”. The function “expand” returns “children”.

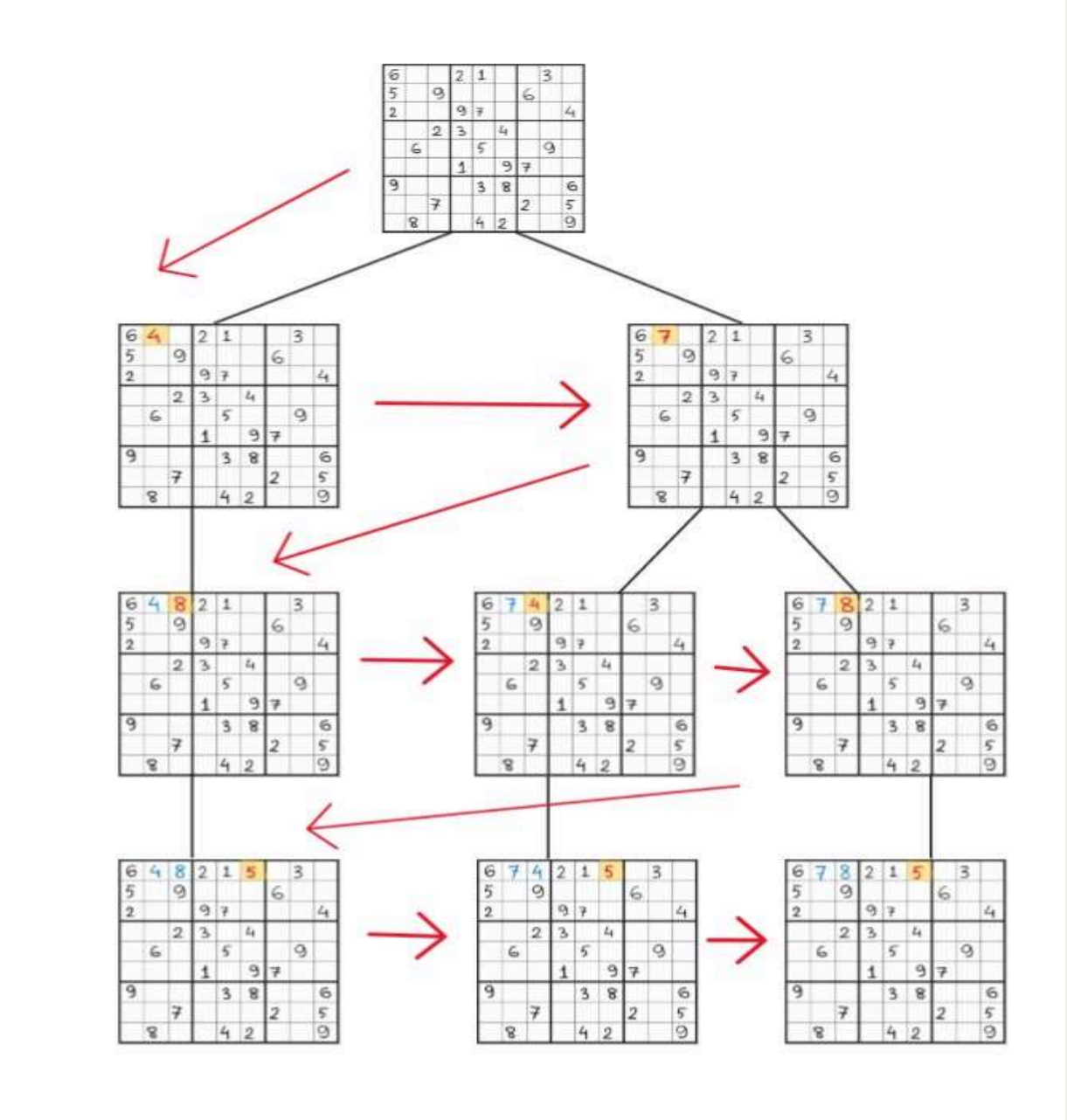
- If any of the child in the returned child nodes is legal, i.e., taking the state of child as argumen “check_legal” returns TRUE, then the state is returned by “BFS”, otherwise, the child nodes a stored in the queue.
- Since BFS works using a queue, so all the nodes in the same level are explored before explorin the child nodes of any one of them.
- If all possible nodes are explored, but no node has a legal state, then Null is returned from the function “BFS”.

Example:

6			2	1			3	
5		9				6		
2			9	7				4
		2	3		4			
	6			5			9	
			1		9	7		
9				3	8			6
		7				2		5
	8			4	2			9

6	4	8	2	1	5	9	3	7
5	7	9	4	8	3	6	2	1
2	1	3	9	7	6	8	5	4
7	9	2	3	6	4	5	1	8
3	6	1	8	5	7	4	9	2
8	5	4	1	2	9	7	6	3
9	2	5	7	3	8	1	4	6
4	3	7	6	9	1	2	8	5
1	8	6	5	4	2	3	7	9

Visualizing How BFS solves the sudoku:



Solving The Example using our Code:

▶ Code

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    6    0    0    2    1    0    0    3    0
 [2,]    5    0    9    0    0    0    6    0    0
 [3,]    2    0    0    9    7    0    0    0    4
 [4,]    0    0    2    3    0    4    0    0    0
 [5,]    0    6    0    0    5    0    0    9    0
 [6,]    0    0    0    1    0    9    7    0    0
 [7,]    9    0    0    0    3    8    0    0    6
 [8,]    0    0    7    0    0    0    2    0    5
 [9,]    0    8    0    0    4    2    0    0    9
```

▶ Code

```
Found solution
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    6    4    8    2    1    5    9    3    7
 [2,]    5    7    9    4    8    3    6    2    1
 [3,]    2    1    3    9    7    6    8    5    4
 [4,]    7    9    2    3    6    4    5    1    8
 [5,]    3    6    1    8    5    7    4    9    2
 [6,]    8    5    4    1    2    9    7    6    3
 [7,]    9    2    5    7    3    8    1    4    6
 [8,]    4    3    7    6    9    1    2    8    5
 [9,]    1    8    6    5    4    2    3    7    9
Elapsed time:  0.21508  seconds
```

Implementing DFS Algorithm in Solving Sudoku:

DFS algorithm to solve sudoku:

- Check legality of the initial state. If it is legal, print the state and break.
- Otherwise, initialize a stack and put the node generated by the initial state into it.
- While the stack is not empty:
  - Pop the last node from the stack. If the state of the node is legal, then print the state and break.
  - Otherwise, Find the first empty cell of the state.
  - Get all possible values between 1 and 9 that can be entered in that cell.
  - Enter each possible value in that cell to generate a child node of current node and store the child nodes in the stack.
- If all possible nodes are explored but none of them is legal, then no solution exists.

Code

- Code

Explanation of the code involving DFS to solve sudoku:

- The function **"DFS_solve"** takes the initial sudoku board as argument. From this initial sudoku board, the class **"Problem"** is generated.
- The function **"DFS"** uses this **"Problem"** as argument
- Using state of sudoku board involved in **"Problem"**, another class **"Node"** is generated, which is taken as the root node, defined as **"start".**
- Then legality of the state is checked using the function **"check_legal"** (defined in the class **"Problem"**), taking state as argument.

  o In this function legality of each row, column and block are checked. If any of the row-sum or column-sum or block-sum is NOT EQUAL to sum of the numbers 1 to up to 9 (i.e., 45) or there exists any duplicate row, column or block, then the state is not legal, i.e., it needs to be modified. So, the function returns FALSE.

  - Otherwise, the state is legal, i.e., no sudoku rule is violated. So, the function returns TRUE.
- If **"check_legal"** returns TRUE, then the initial state is the final solution and is returned by **"DFS"** Otherwise, we opt for next steps.
- As DFS works using a stack following LIFO principle, so an empty stack is created, where the roo node **"start"** is stored.
- While the stack is non-empty, the following steps are followed:
  - Last node in the stack is picked out and legality of the state of the node is checked using function **"check_legal"**
  - If **"check_legal"** returns TRUE, then the present state is returned by **"DFS"**.
  - Otherwise, the present node is expanded using the function **"expand"** to obtain its child node

- The function **“expand”** belongs to the class **“Node”** and it takes the class **“Problem”** as argument. In this function,
  - From **“Problem”**, using the function **“get_spot”** having state as an argument, the position of first empty cell of the state is obtained in terms of (row, column), where the search for first empty cell is done by scanning from left to right, top to bottom.
  - From **“Problem”**, using the function **“filter_values”** having two argument vectors of all possible values and the used values, the values that can be entered in that empty cell without violating the Sudoku rules are developed.
  - From **“Problem”**, using the function **“action”** having state as argument, the vectors (possible value, row, column) for the first empty cell of the state are stored in the list **“result”**, which is again stored as **“actions”** within the function **“expand”** in the class **“node”**.
  - From **“Problem”**, using the function **“result”** having state and action as arguments, all possible next states are produced and stored in **“next_state”** using each action present in the list **“actions”**, i.e., the possible value present in the vector action is entered in the position given by (row, column) in action.
  - Using **“next_state”** and action vector, the child nodes are produced and stored in **“children”**. The function **“expand”** returns **“children”** and it is stored in the stack.
  - Since DFS works using a stack, so before exploring the other nodes in the same level, child nodes of a node are explored.
- If all possible nodes are explored, but no node has a legal state, then Null is returned from the function **“DFS”**.

- The returned object from **“DFS”** is stored in **“solution”** in the function **“DFS_solve”**

Example:

6			2	1			3	
5		9				6		
2			9	7				4
		2	3		4			
	6			5			9	
			1		9	7		
9				3	8			6
		7				2		5
	8			4	2			9

6	4	8	2	1	5	9	3	7
5	7	9	4	8	3	6	2	1
2	1	3	9	7	6	8	5	4
7	9	2	3	6	4	5	1	8
3	6	1	8	5	7	4	9	2
8	5	4	1	2	9	7	6	3
9	2	5	7	3	8	1	4	6
4	3	7	6	9	1	2	8	5
1	8	6	5	4	2	3	7	9

Solving The Example using our Code:

▶ Code

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    6    0    0    2    1    0    0    3    0
 [2,]    5    0    9    0    0    0    6    0    0
 [3,]    2    0    0    9    7    0    0    0    4
 [4,]    0    0    2    3    0    4    0    0    0
 [5,]    0    6    0    0    5    0    0    9    0
 [6,]    0    0    0    1    0    9    7    0    0
 [7,]    9    0    0    0    3    8    0    0    6
 [8,]    0    0    7    0    0    0    2    0    5
 [9,]    0    8    0    0    4    2    0    0    9
```

▶ Code

```
Found solution
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    6    4    8    2    1    5    9    3    7
 [2,]    5    7    9    4    8    3    6    2    1
 [3,]    2    1    3    9    7    6    8    5    4
 [4,]    7    9    2    3    6    4    5    1    8
 [5,]    3    6    1    8    5    7    4    9    2
 [6,]    8    5    4    1    2    9    7    6    3
 [7,]    9    2    5    7    3    8    1    4    6
 [8,]    4    3    7    6    9    1    2    8    5
 [9,]    1    8    6    5    4    2    3    7    9
Elapsed time:  0.122257  seconds
```

Comparing Efficiency of DFS and BFS in solving Classic Sudoku:

1. BFS requires a higher number of nodes than DFS to find a solution, because BFS expands all possible numbers that can be placed in each cell, while DFS expands the number of possibilities in depth.
2. DFS finds a solution faster than BFS as number of nodes produced by BFS is more wasteful than DFS.
3. If a sudoku problem has more than one solution, then BFS is able to find it, DFS cannot.

Thus, DFS is more efficient than BFS in solving a classic sudoku having unique solution.

References:

- Implementation of DFS using adjacency matrix - GeeksforGeeks
- BFS Graph Algorithm(With code in C, C++, Java and Python) (programiz.com)
- (PDF) Comparison Analysis of Breadth First Search and Depth Limited Search Algorithms in Sudoku Game (researchgate.net)
- Depth First Search (Backtracking) Algorithm to Solve a Sudoku Game | Algorithms, Blockchain and Cloud (helloacm.com)
- Sudoku: Depth and Breadth First Search | Nigel Chin (kychin.netlify.app)
- Sudoku #1: Depth-First Search - Blog (kevinkle.in)
- Backtracking Algorithm for Sudoku (opengenus.org)

THANK YOU !