Theoretical Foundations of Artificial Intelligence

Contents

Unit I – Introduction to Artificial Intelligence

UNIT I: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

What is Artificial Intelligence?

Def: AI is the simulation of human intelligence in machines. It involves creating systems that can perform tasks that typically require human intelligence, **like**: reasoning, learning, problem-solving, perception, language understanding.

Reasoning: Adapting the solution to context – understanding the context and constraints and deciding what action to take

Ex1:

Overtaking Another Vehicle

- **Perception**: Sensors detect a slow-moving truck ahead.
- **Reasoning**:
  - Check if the left lane is clear.
  - If safe, decide to change lane.
  - If not safe, continue behind the truck.
- **Action**: Executes overtaking safely.

Ex2:

Emergency Vehicle Detection

- **Perception**: Microphone + camera detect a siren and flashing ambulance lights.
- **Reasoning**:
  - Identify → emergency vehicle has right of way.
  - Decide → slow down, move aside to let it pass.
- **Action**: Car shifts lanes.

Learning: Machine Learning – Netflix recommendations, email spam filters etc

Problem Solving: AI's ability to find solutions to complex or unfamiliar tasks.

Ex1: Chess Playing AI

Perceptron: Ability to use sensors (vision, sound, touch, etc.) to understand the environment.

Ex1: **Camera sensor** Mounted all around the car (front, rear, sides, 360°) detect: traffic lights, road signs, pedestrians, lane markings, other vehicles.

Ex2: **LiDAR sensor** (Light Detection and Ranging): Uses lasers to scan the environment in 3D. Creates a point cloud (3D map of surroundings). Detect: exact position of objects, distance, size, and shape.

Ex3: **Radar sensor** (Radio Detection and Ranging): Uses radio waves to detect speed and distance of objects. Works well in bad weather (rain, fog, snow) where cameras may fail.

Example: Radar detects a fast-approaching car from behind at high speed.

Language Understanding: Understanding Human Language

Ex1: Amazon Alexa device/Smart TV Remote, YouTube etc understanding our voice commands
Ex2: Chatbots
Ex3: Google Translate

Goals of AI

- Create machines that can think and act like humans
- Solve complex problems efficiently
- Automate human tasks

Applications of AI

- Self-driving cars
- Virtual assistants (Siri, Alexa)
- Smart recommendations in Entertainment (Netflix, Amazon)
- Medical Image Diagnosis
- Content Generation – AI Anchors, Voice Cloning, Songs generation
- Fraud Detection (in finance)
- Etc

Types of AI

Weak AI / ANI (Artificial Narrow Intelligence) - excels/expert at one specific task (may excel humans too in that domain)

ex1: Identify robbery in a bank
ex2: Car driving
ex3: Predict house price
ex4: MRI photo - analysis - predict disease present or not
ex5: Chess playing AI software

Strong AI / AGI (" General "): Equal to Humans in Intelligence and problem-solving ability - Still theoretical, this doesn't exist yet practically.

Should do all the following: Cooking, drive a car and drop and pickup children from school, play table tennis, detect bank robbery, show emotions, social interaction, take old people to hospital.

Super AI / ASI (" Super "): Far greater than humans, could outperform the best human minds in every field. It is only a concept, still very far from achieving

Ex1: Diagnose any disease instantly and accurately. Provide personalized treatment plans for everyone. Perform surgeries with near-zero error.

Ex2: Teach any subject at an adaptive level for each learner.

Ex3: Search for extraterrestrial life and communicate with them.

Ex4: Tell a city **“It will rain 12 mm in your city on Sept 28th”** with 95–99% confidence.

AI Problems

Types of AI Problems:

1. **Classification Problems**: Categorizing data (email spam detection)
2. **Prediction Problems**: Forecasting future events (weather prediction)
3. **Optimization Problems**: Finding the best solution (route planning)

   Example: Route Planning (Self-Driving Cars / Google Maps)

   Problem: A car needs to go from Point A to Point B.

   - There are many possible routes.
   - Each route has a different travel time, distance, traffic condition, and fuel cost.
   - AI must pick the **optimal route**.

   Step-by-Step:

   1. **Input Data**
      - Road map, distances, traffic updates, speed limits, construction zones.
   2. **Possible Solutions**
      - Route 1: Shortest distance but heavy traffic.
      - Route 2: Longer distance but smooth traffic.
      - Route 3: Highway route, fast but with toll charges.
   3. **Optimization Goal**
      - Minimize **travel time** (or fuel cost, or toll fees, depending on user preference).
   4. **AI Algorithm**
      - Uses **optimization algorithms** (e.g., Dijkstra's Algorithm, A* Search, Genetic Algorithms).
   5. **Output (Best Route)**
      - Example: Google Maps suggests → "Route 2, 25 min (fastest route now)."
4. **Pattern Recognition**: Identifying patterns in data

   Pattern recognition in AI is the ability to detect regularities, structures, or similarities in data

   It's like the AI asking: "Does this new thing I see match something I've seen before?"

   Ex1: AI uses pattern recognition to **analyse pixels in an image** and classify what's inside.
   AI detects patterns in shapes, edges, textures.

Example: Cat → pointy ears, whiskers, oval face.

Ex2: Customer Segmentation = Pattern Recognition in Business

5. **Natural Language Processing**: Understanding human language
6. **Expert Systems/Tasks:** Medical diagnosis, financial forecasting etc

Problem Characteristics:

These **Problem Characteristics** are used in AI to understand the **nature of a problem** and to decide what kind of AI approach/algorithm is suitable.

- **Decomposability**: Can the problem be broken into smaller parts?

  Example: Breaking **image recognition** into smaller tasks (edge detection → object detection → classification).

- **Solution Steps (Reversibility)**: Can we go back if we make a wrong step?

  Examples:

  Reversible: Maze-solving robot → if it takes a wrong turn, it can backtrack and try again.
  Irreversible: Surgery planning by a medical AI → once an incision is made, it cannot be reversed. The AI must plan carefully in advance.

- **Predictability**: Is the problem's universe predictable?

  Examples:

  Predictable: Solving a math equation → rules don't change.
  Unpredictable: Driving a self-driving car in traffic → pedestrians, weather, and other drivers' actions are uncertain.

- **Knowledge Requirements**: What knowledge is needed to solve it?

  Examples:

  Knowledge-intensive: Medical diagnosis AI → must know anatomy, diseases, treatments.
  Knowledge-light: Sorting numbers in ascending order → requires only basic algorithmic knowledge.

3. AI Techniques

Major AI Techniques:

1. **Search Techniques**: *AI often works by searching through possible solutions* until it finds the best one.
   Ex1: Google Maps finding the **shortest path** from your home to the airport.
   Ex2: A chess-playing AI trying different moves ahead to see which path leads to **winning**.

2. **Knowledge Representation**: AI needs a way to **store knowledge** (facts, rules, relationships) so it can use it for reasoning.
   Ex1: Facts: "Parrot is a bird", "Birds can fly".
   Rule: If something is a bird → it can usually fly.
   AI can then answer: "Can a parrot fly?" → Yes.
- **Ex2:** Medical system: storing symptoms (fever, cough) and rules (If fever + cough → possible flu/influenza virus).

  Note:

  - When the influenza virus infects your body, your **immune system fights back**.
  - This immune response leads to **fever** (body raises temperature to kill the virus).
  - The virus also irritates your **respiratory system** (nose, throat, lungs), which causes **cough, sore throat, and runny nose**.

3. **Machine Learning**: Instead of giving rules directly, AI learns patterns from **experience/data**.
   Ex1: Predicting **house prices** from past data: size, location, number of rooms → price.
   Ex2: Email spam filter: Learns from past emails marked as spam or not.

4. **Neural Networks**: Inspired by the **human brain**, neural networks are made of layers of "neurons" that process inputs and learn complex patterns.

   Ex1: Voice assistants (Siri, Alexa) recognizing your speech.
   Ex2: Self-driving cars detecting pedestrians and traffic lights (images).
   Ex3: Action Recognition in a CC Camera recorded footage - live or recorded

5. **Expert Systems**: These are AI systems built on **expert knowledge** in a field, usually with **rules**
   Ex1: MYCIN (early AI system): Suggested antibiotics for infections.
   Ex2: A **legal expert system**: Stores laws and gives advice based on user queries.
   Ex3: A **car repair expert system**: Knows rules like "If engine doesn't start + battery is dead → replace battery".

   Note: Is DALL.E an expert system – because it can draw images

   What is DALL·E?

   - DALL·E is a **generative AI model** (built on neural networks).
   - It is **trained on millions of images and captions**.
   - It **learns patterns** in images and text during training.
   - When you give a text prompt ("A cat riding a bicycle"), it **generates** a new image by using those learned patterns.

   A **drawing expert system** would have rules like: "If user says 'house', draw a rectangle + triangle on top."
   DALL·E doesn't use such rules. It **learns** from millions of house images and then creates one that looks realistic.

   So, **DALL·E is not an expert system**.

4. Defining Problems as State Space Search

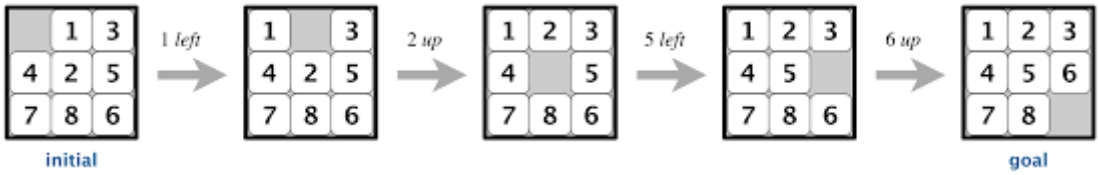
State Space Representation:

Here, a problem is divided into different states (situations) (Eg: Goods carrying Robot Moving in a Godown)

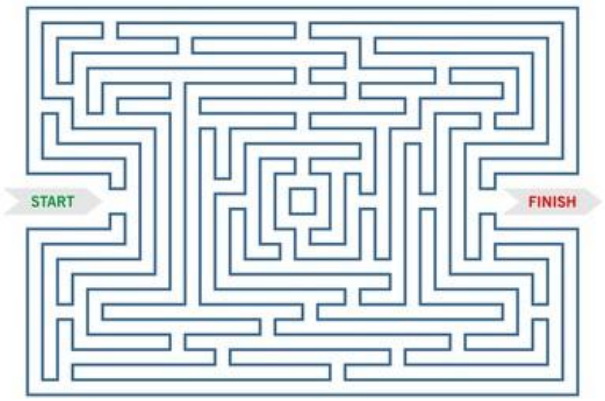
- **State**: A representation of the problem at a given moment.
- **State Space**: All possible states of the problem.
- **Initial State**: Where we start
- **Goal State**: Where we want to reach
- **Search**: The process of exploring the state space from the initial state to the goal state.
- **Operators/Actions**: What we can do to change states to **reach Goal state** from **initial state**
- **Path**: Sequence of states from initial to goal

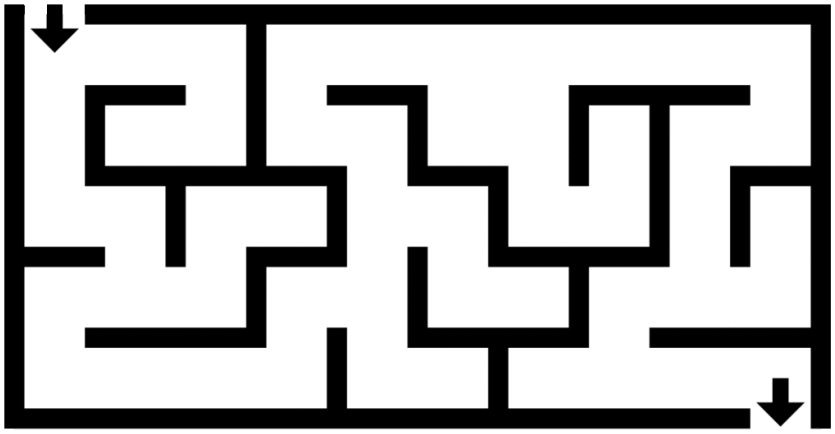
Example1 – Puzzle Solvers - 8-Puzzle:

- State Space: All possible board arrangements
- State: A particular arrangement at a given point of time
- Initial State: The puzzle's starting arrangement.
- Goal State: The arrangement we want to reach.
- Operators:
  Legal moves we can apply: sliding a tile into the empty space.
  Eg: If empty space (_) is in the bottom middle:
  7 _ 8
  Possible moves: slide 7 right, or 8 left, or 5 down (if 5 is above _).
- Search: The process of **exploring states** (trying different moves) to get from initial state to goal state.
- Path: The **sequence of states** (or moves) from initial state to goal state.



Example2: Maze with start and goal





Ex3: Robot Motion Planning

Ex4: Drone route with time window and moving obstacles

5. Production Systems

A Production System is a **rule-based problem-solving model** used in Artificial Intelligence.

It works like:
IF (Condition) → THEN (Action)

Example Rule:
IF temperature > 80°F AND humidity > 70%
THEN turn on air conditioning

Components of Production Systems:

1. Production Rules
2. Working Memory / Global Database: Current state information
3. Control Strategy: How to apply rules
4. Rule applier

Production Rules:

Knowledge is stored as rules.
Format: IF condition THEN action.

Working Memory / Global Database:

Stores the current state information
Keeps track of facts and information at any given time.

Control Strategy: How to apply rules:

Decides **which rule to apply next** when many rules match.
Example: Depth-first, Breadth-first, or heuristics.

Rule applier/Execution:

The "brain" that applies the selected rule to working memory.
Repeats until the goal state is reached.

Example1: Imagine an **AI cooking assistant** that decides how to make tea:

- **Rules**:
  1. IF water is not boiled → THEN boil water
  2. IF tea leaves are available AND water is boiled → THEN add tea leaves
  3. IF tea leaves added → THEN add milk & sugar
  4. IF tea is ready → THEN serve
- **Working Memory**:
  Starts with {water not boiled, tea leaves available}
- **Control Strategy**:
  Apply rule 1 first → then rule 2 → then rule 3 → then rule 4
- **Execution**:
  Step by step, the system reaches the **goal state**: *Tea is ready!*

Example2: An AI assistant for flu diagnosis:

- **Rules**:
    1. Rule 1 (base evidence):
       IF fever AND cough → THEN possible flu
    2. Rule 2 (stronger evidence):
       IF fever AND cough AND headache AND fatigue → THEN flu more likely
    3. Rule 3
       IF flu suspected → THEN suggest rest + fluids
    4. Rule 4
       IF severe symptoms (high fever, breathing difficulty) → THEN recommend doctor visit

- **Working Memory**:
  Patient symptoms {fever, cough, headache}

- **Execution**:
    - Rule 1 fires → flu suspected
    - Rule 2 fires → flu more likely
    - Rule 3 fires → suggest rest & fluids
    - Rule 4 fires → recommend doctor visit
    - Goal state → *Advice provided to patient*

Example3: An AI deciding moves in chess:

- **Rules**:
    1. IF opponent queen/mantri threatens king → THEN move king to safe square
    2. IF checkmate possible in 1 move → THEN make that move
    3. IF no immediate threat → THEN move piece to gain material advantage
- **Working Memory**:
  Current board state
- **Execution**:
    - Checks rules in order
    - If in danger → defend
    - If chance to win → attack
    - Otherwise → improve position
    - Goal state → *Best move chosen*

Key Idea

- **Cooking Tea Example** → Daily-life routine (sequence of actions).
- **Medical Diagnosis Example** → Expert system (healthcare).
- **Chess Example** → Game-playing AI (decision-making).

All follow the **Production System pattern**:

Rules + Working Memory + Control Strategy = Goal Achievement.

Control Strategies in a Production System:

- **Forward Chaining**: Start with facts, derive conclusions
- **Backward Chaining**: Start with goal, work backwards

In Detail:

Forward Chaining:

Start with the facts → Apply rules → Keep deriving new facts → Until you reach the goal.

- It's **data-driven reasoning**.
- Used when you have **lots of known facts** and want to see what conclusions can be drawn.

Example (Medical Diagnosis – Flu)

- **Facts (Working Memory)**: {Fever, Cough}
- **Rules**:
    1. IF Fever AND Cough → THEN Flu suspected
    2. IF Flu suspected → THEN Suggest rest & fluids

Execution:

- Start with {Fever, Cough}
- Apply Rule 1 → add {Flu suspected}
- Apply Rule 2 → add {Suggest rest & fluids}
- Goal reached: Provide advice to patient

Note: Here, the reasoning **moves forward** from facts → conclusions.

Backward Chaining

Start with the goal → Check which rules could prove it → See if conditions are satisfied → Work backwards until you reach known facts.

- ***It's goal-driven reasoning.***
- Used when you want to prove or disprove a specific hypothesis.

Example (Chess Move Decision)

- Goal: Can I checkmate in 1 move?
- Rule: IF Move X → THEN Checkmate
- To prove goal, system checks: Is there any Move X available?
- If yes → apply the move → Goal achieved

Note: Here, reasoning starts at the goal and **works backwards** to facts.

Forward Vs Backward Chaining

Feature	**Forward Chaining**	**Backward Chaining**
Direction	Facts → Goal	Goal → Facts
Type	Data-driven	Goal-driven
Best Used When	Many facts, few goals	Few specific goals
Example	Medical Diagnosis	Chess moves

Production System Categories

They are classified by how they apply rules:

- **Monotonic:** Once a fact is known, it remains true. Rules only add knowledge.
- **Non-Monotonic:** New knowledge can make old facts invalid. More realistic but complex.
- **Partially Commutative:** If a sequence of rules leads to a goal, any order of those rules will work. Simplifies search.

Monotonic Production System

- **Definition**: Once you apply a rule, it never prevents other rules from being applied later.
- In other words, adding new facts never invalidates earlier inferences.
- The knowledge base **only grows or stays the same**.

Example:

- Suppose we have rules about family relations:
  - IF X is parent of Y THEN X is ancestor of Y.
  - IF X is ancestor of Y AND Y is ancestor of Z THEN X is ancestor of Z.
- Once you add a fact like *"Ramesh is parent of Sita"*, it will always remain true and can only lead to more inferences.

Applications: Logical reasoning, mathematical theorem proving.

Non-Monotonic Production System

- **Definition**: Applying a rule can **block or invalidate** other rules that might have been applicable before.
- Knowledge is **not permanent**; new information can **retract** old conclusions.

Example: Traffic Rules

- **Rule 1**: IF green light → THEN car may go.
- **Rule 2**: IF green light AND traffic police says STOP → THEN car must not go.

Case 1:

- Fact: green light.
- Inference: car may go.

Case 2 (new info added):

- Fact: green light AND traffic police says STOP.
- Now Rule 2 overrides Rule 1.
- The earlier conclusion car may go is **invalidated**.

Why is it Non-Monotonic?

- In monotonic systems, once car may go was inferred, it would **always remain true**.
- But here, adding new info (police says STOP) **changes the conclusion**.
- Knowledge is **revised** dynamically.

Partially Commutative Production System

- **Definition**: The order of rule application **doesn't matter** — different rule sequences may still lead to the same final state.

- But unlike monotonic systems, **not all rules are order-independent**, only some are.

Example:

- Suppose we have two rules:
  - Rule1: IF tea leaves → THEN add hot water.
  - Rule2: IF tea prepared → THEN pour into cup.
- Whether you *add sugar before or after pouring* doesn't affect the final "tea in cup with sugar".
- But if you *pour into cup before adding hot water*, the result changes.
- Hence, **partially commutative**.

Applications: Problem-solving systems where certain actions commute (e.g., puzzles, planning tasks).

6. Uninformed Search (Blind Search)

Search Strategies in AI

In AI, many problems can be thought of as search problems:

Examples:

Ex1: Robot Path Planning: A robot in a warehouse must move from its start position to a goal position.
Ex2: Puzzle Solving: Solve an 8-puzzle
Ex3: Route Finding (GPS Navigation): Find the **best route** from home to office.
Ex4: Chess Game Playing: Choose the best move to win.

In all these examples:

- You have a start state (where you are now).
- You want a goal state (solution).
- You need to find a path (sequence of actions).

A search strategy decides *how* we explore possible states to reach the goal.

UnInformed Search:

It is like "searching in the dark without a map."

Breadth-First-Search (BFS)

Note: Here we use a Queue DS

A BFS starts at some arbitrary vertex of a graph and **explores/visits the neighbouring vertices first**, before moving to the **next level neighbours (move outward).**

(OR)

BFS explores level by level (all neighbors first, then move outward).

BFS Algorithm (Step by Step)

1. **Start at the root (or starting node)** → put it in a **queue**.

2. **Mark it as visited** (so you don't visit again).
3. While the queue is **not empty**:
    - Remove the **front node** from the queue.
    - Visit it (process it).
    - Add all its **unvisited neighbors** to the **back of the queue**.
    - Mark those neighbors as visited.
4. Repeat until the queue becomes empty.

Note: Since a Queue follows FIFO order, the vertex entered first will be visited first and their neighbours will be added in the queue first.

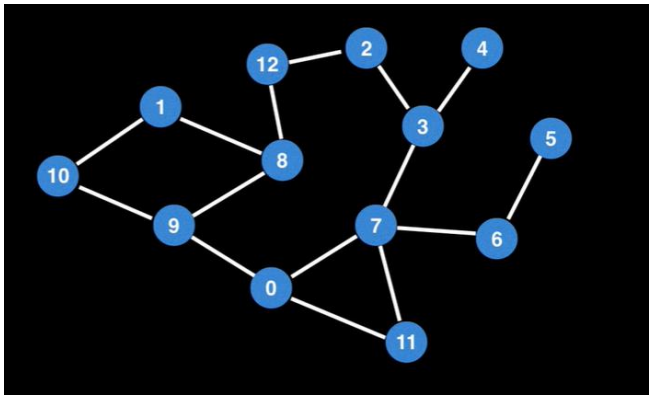
Comparing it with a Real-life scenario:

Imagine you are at a **party**, and you want to talk to everyone there, starting with the people closest to you.

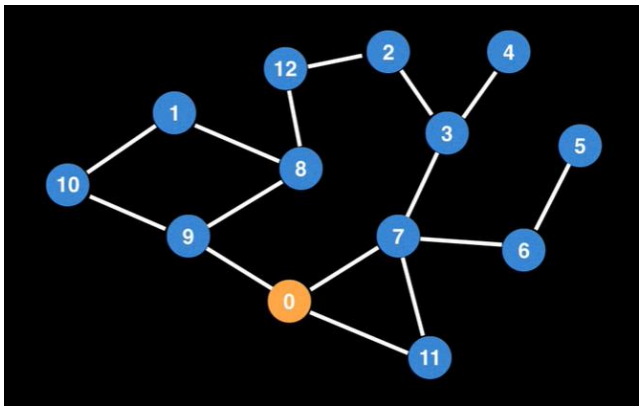
1. You first **talk to everyone nearby** (people standing closest to you, within your immediate circle).
2. After finishing with this group, you move on to the **next group of people** who are a little farther away (friends of your immediate circle).
3. You keep moving to the next group, exploring all people in each layer before moving to the next layer.

Key Point: You explore everyone level by level, ensuring you meet everyone in one circle of distance before moving on to people farther away.

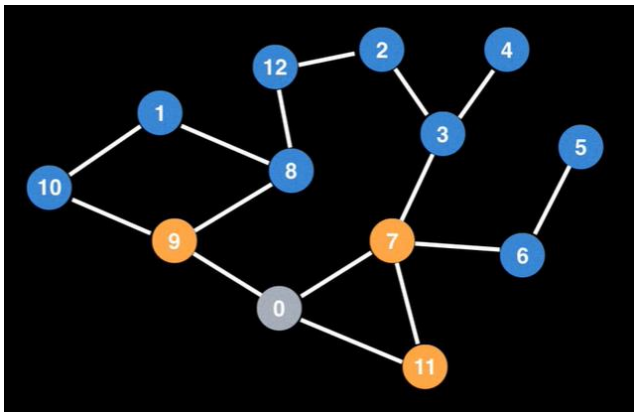
Ex1: Let's consider the following Graph:



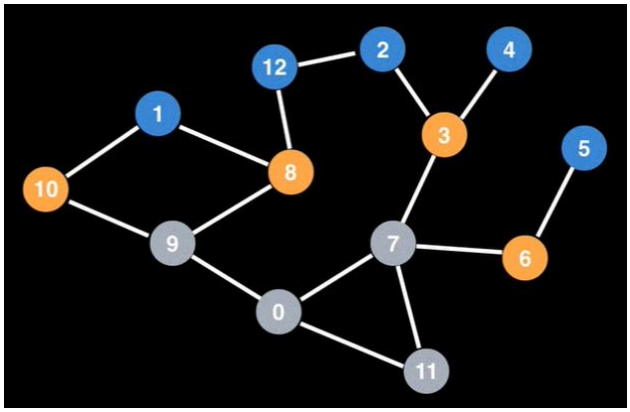
Let’s begin with an arbitrary vertex: Vertex 0



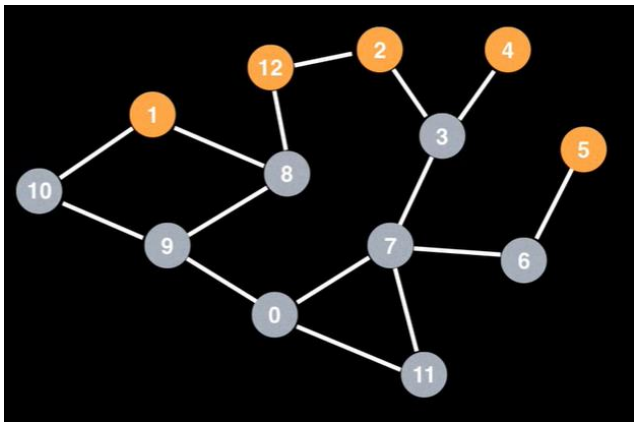
Visit all the neighbours of Vertex 0



Then, we would visit all the unvisited neighbours of vertices 9, 7 and 11 in the order they were added to the Queue. i.e., If 9 is added, we shall start with 9 itself.



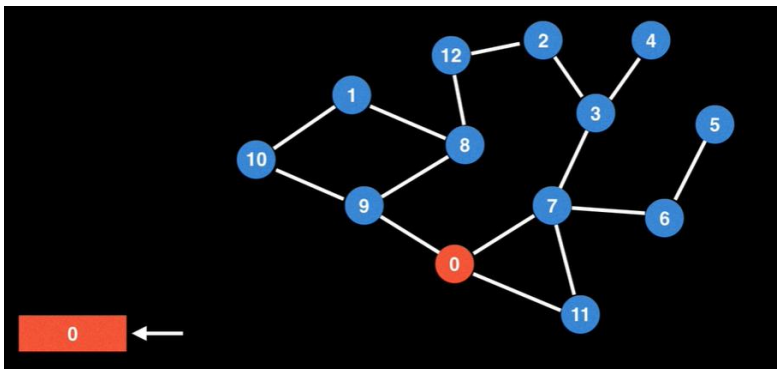
Then we would visit all the unvisited neighbours of 10, 8, 3 and 6



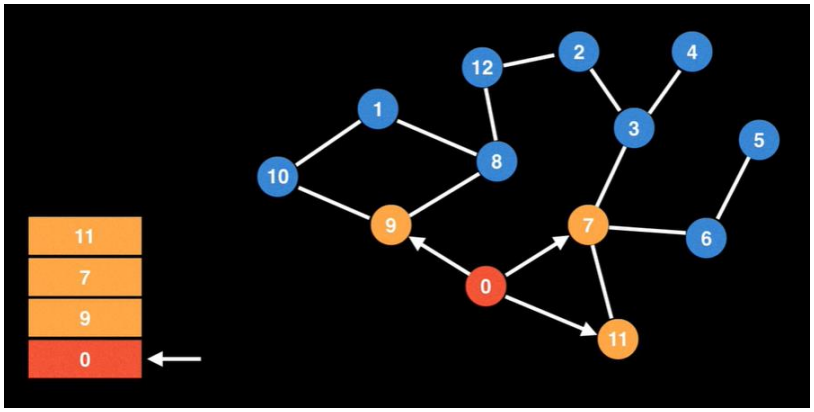
Note: The vertices that should be visited next are stored in a Queue Data Structure.

Result: 0, 9, 7, 11, 10, 8, **(of 9)** 6, 3, **(of 7)** 1, **(of 10)** 12, **(of 8)** 5, **(of 6)** 2, 4 **(of 3)**,

Step1. Current vertex is 0



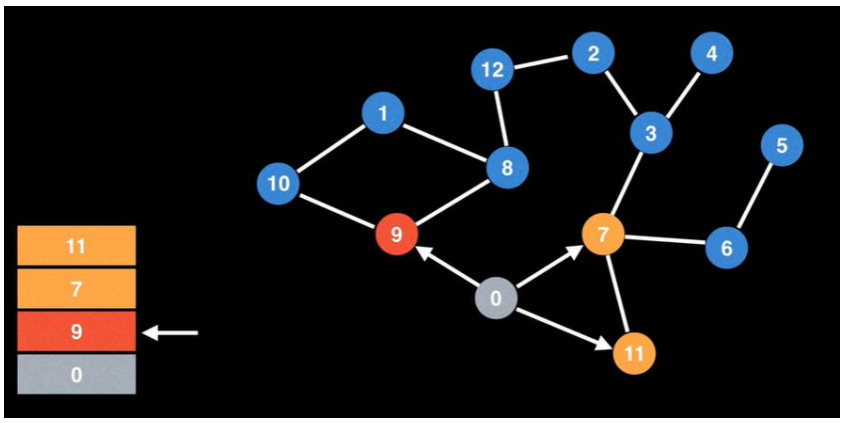
Step2. Enqueuing all the neighbouring vertices of vertex 0, i.e., 9, 7 and 11

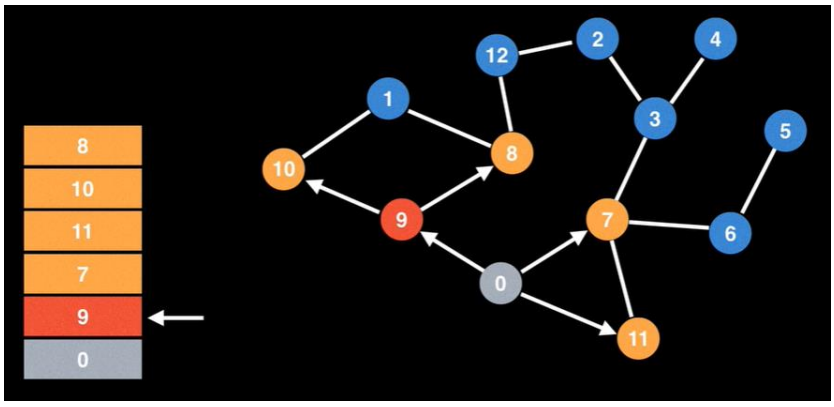


Step3. Enqueuing all the neighbouring vertices of 9, 7 and 11

Current Vertex is 9:

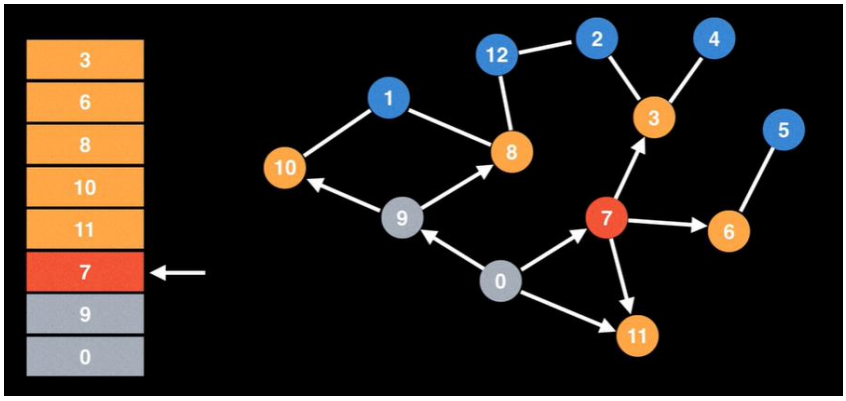
For vertex 9, neighbours are 10 and 8





Current Vertex: 7

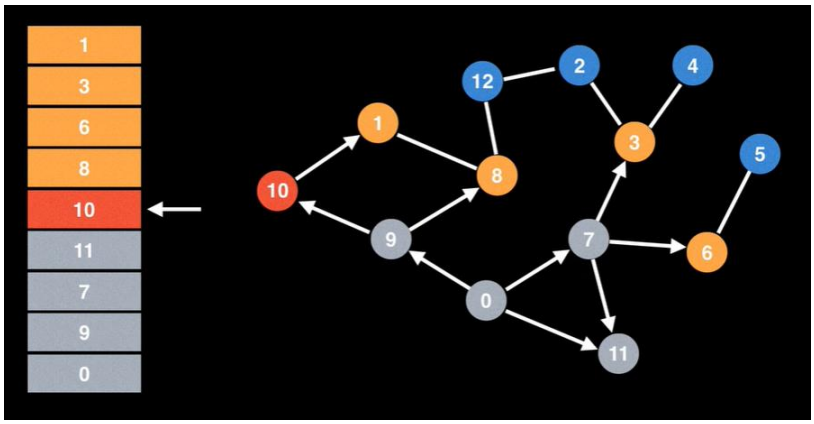
For vertex 7, the neighbours are 3 and 6



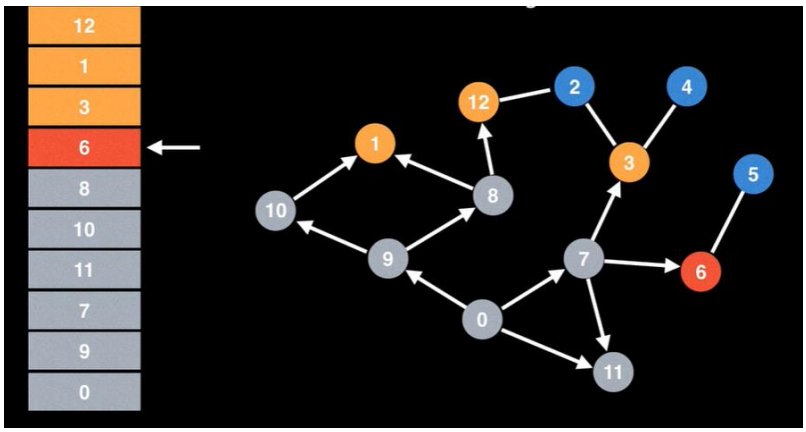
For vertex 11 there are no neighbours

Step4. Current vertex 10:

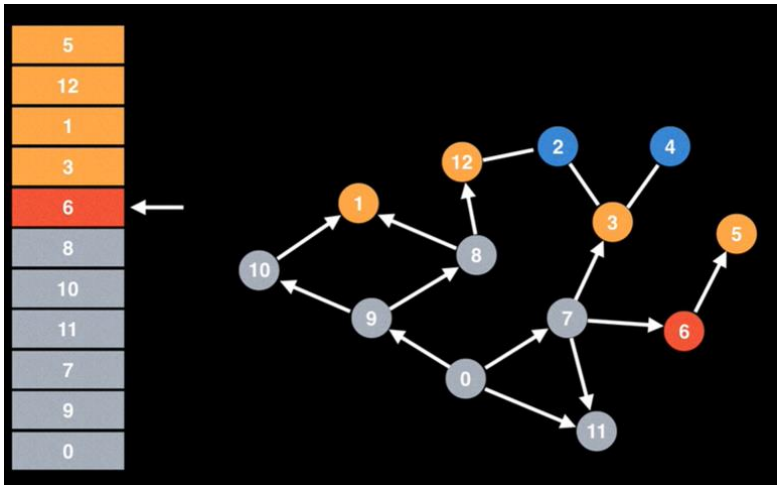
For vertex 10, the unvisited neighbours are vertex 1



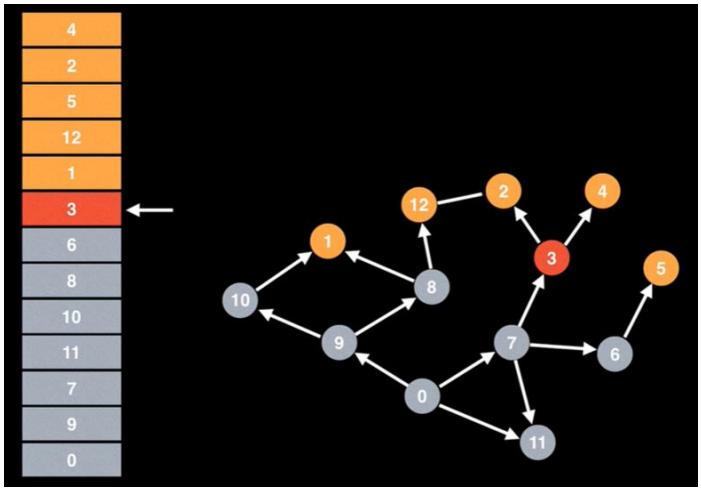
Current vertex: 8



Current vertex 6:



Current vertex 3:



Current vertex 1: No un visited neighbours

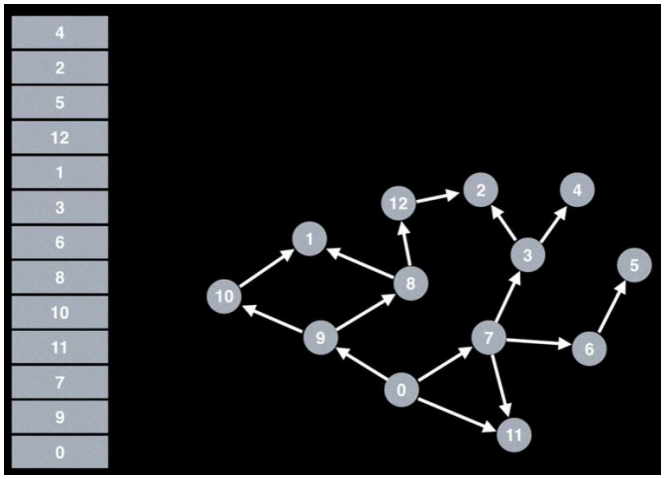
Current vertex 12: No un visited neighbours

Current vertex 5: No un visited neighbours

Current vertex 2: No un visited neighbours

Current vertex 4: No un visited neighbours

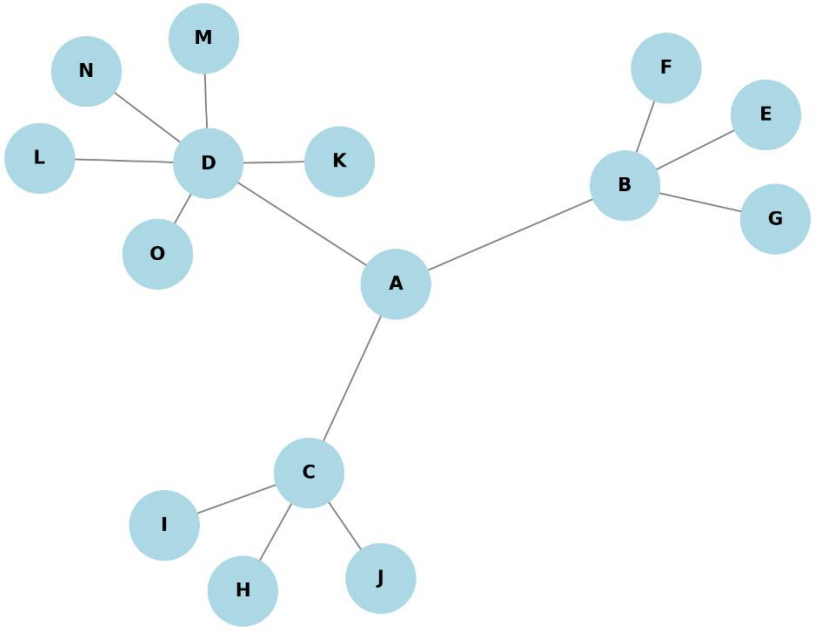
Final Order of Visiting the vertices:



Result: 0, 9, 7, 11, 10, 8, 6, 3, 1, 12, 5, 2, 4

Note: The output may vary based on which adjacent vertex is visited first.
Eg: It may be: 0, 9, 11, 7, 10, 8, 3, 6, 1, 12, 2, 4, 5

Example2: Let us consider the following Graph:



Graph structure from the above figure:

- **A** → connected to {B, C, D}
- **B** → connected to {E, F, G}
- **C** → connected to {H, I, J}
- **D** → connected to {K, L, M, N, O}

BFS Traversal Order (starting from A):

1. Start at **A**
2. Visit A's neighbors → **B, C, D**
3. Visit B's neighbors → **E, F, G**
4. Visit C's neighbors → **H, I, J**

5. Visit D's neighbors → **K, L, M, N, O**

Because, there are no more neighbours for E, F, G, H, I, J, K, L, M, N, O , the process stops.

BFS sequence:
A → B, C, D → E, F, G, H, I, J, K, L, M, N, O

Depth First Search (DFS)

Note: Here we use a Stack DS

A DFS plunges **Depth first** into a Graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

(OR)

DFS goes **deep into one branch first**, then backtracks.

DFS Algorithm (Step by Step)

1. **Start at the root (or starting node)** → put it in a **stack** (or use recursion).
2. **Mark it as visited**.
3. While the stack is **not empty**:
    - Pop the **top node** from the stack.
    - Visit it (process it).
    - Push all its **unvisited neighbors** onto the stack
      (usually from **right to left**, so the leftmost is visited first).
    - Mark those neighbors as visited.
4. Repeat until stack is empty.

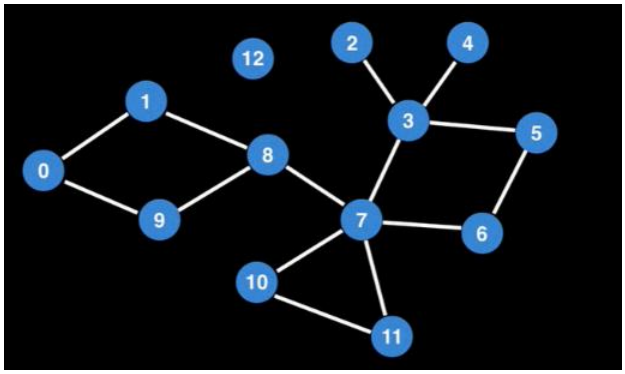
Comparing it with a Real-life Scenario:

Imagine you are exploring a **cave system** with multiple tunnels and branches, and your goal is to explore every part of the cave.

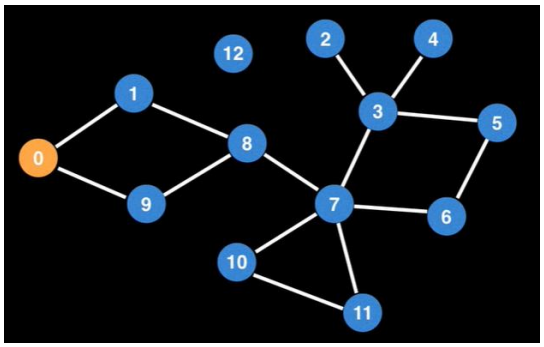
1. You enter the cave and choose one tunnel at the entrance.
2. You keep walking **deeper into that tunnel**, following it until you reach a dead end or no further tunnels exist.
3. Once you've explored the end of that tunnel, you **backtrack** to the previous junction where there was an unexplored tunnel.
4. You then choose the next unexplored tunnel and **go as deep as possible** again, repeating this process.
5. You continue this until all the tunnels have been explored.

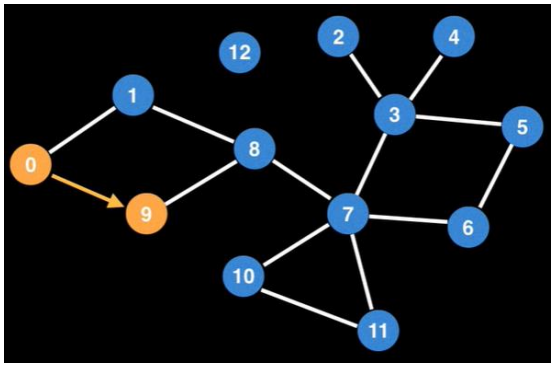
Key Point: In DFS, just like in cave exploration, you focus on **fully exploring one path** as far as it goes before returning to the last branch point and trying the next unexplored path.

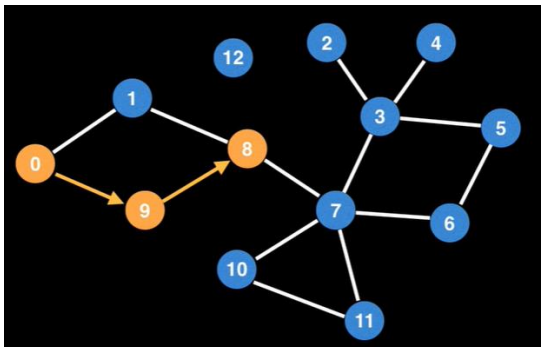
Ex1: Let's consider the following Graph:

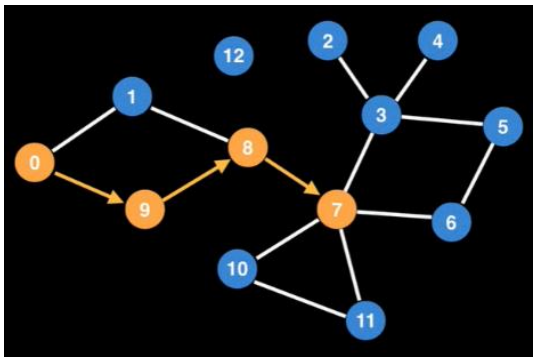


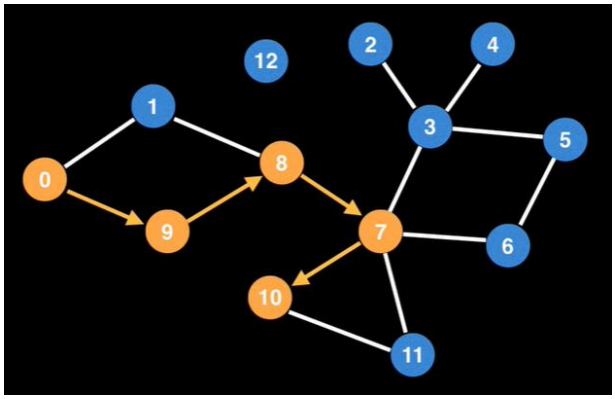
Step1.

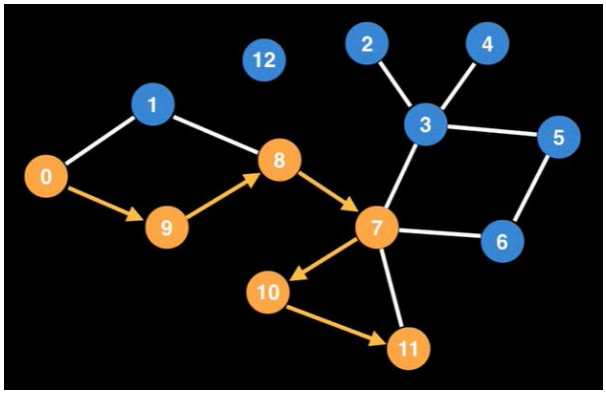


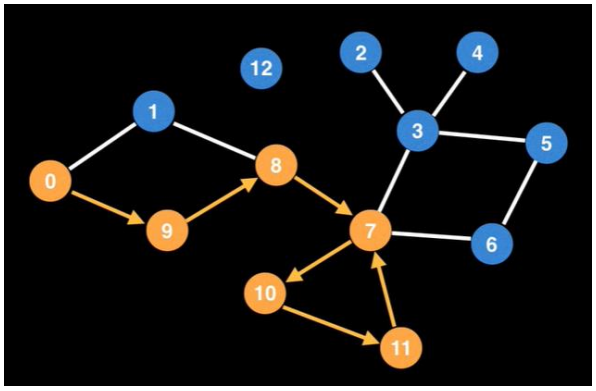

12
2
4
1
3
5
0
8
9
7
6
10
11


12
2
4
1
3
5
0
8
9
7
6
10
11


12
2
4
1
3
5
0
8
9
7
6
10
11

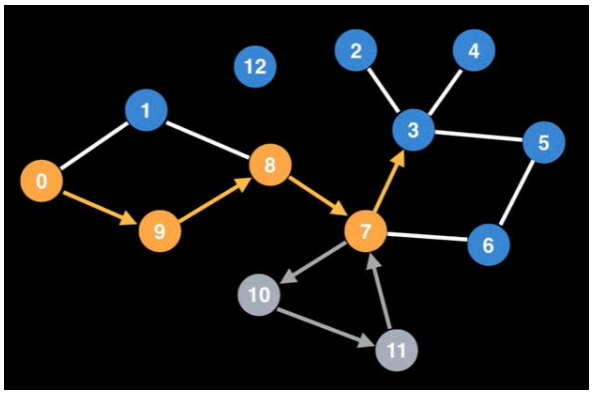

12
2
4
1
3
5
0
8
9
7
6
10
11



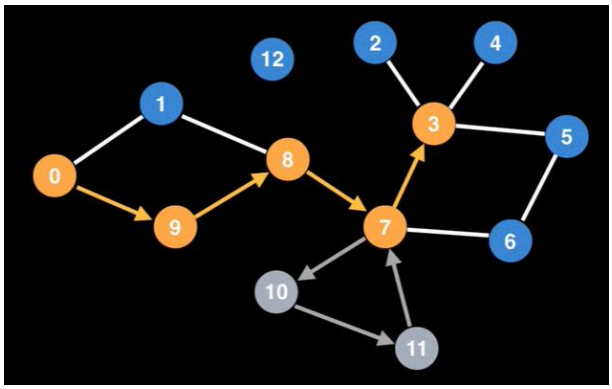


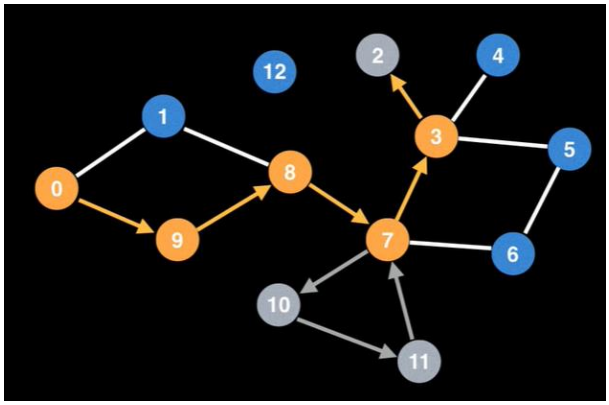
Step2. Now Backtrack:

Note: Back tracking means coming back to the same vertex from which we reached the current vertex (Last In First Out - LIFO). Reached dead-end at Node 11. Then, backtracks to Node 7.

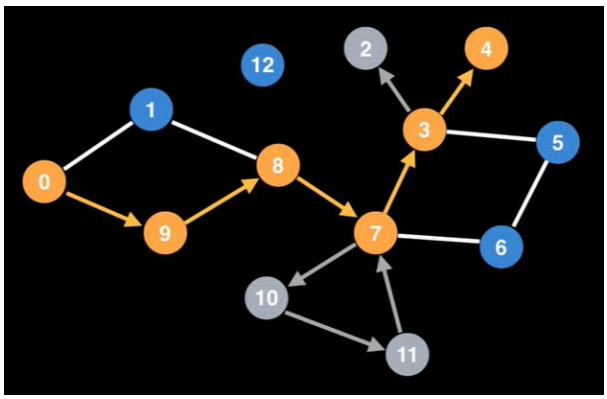


Step3. We haven't finished exploring vertex 7 yet, so let's explore it



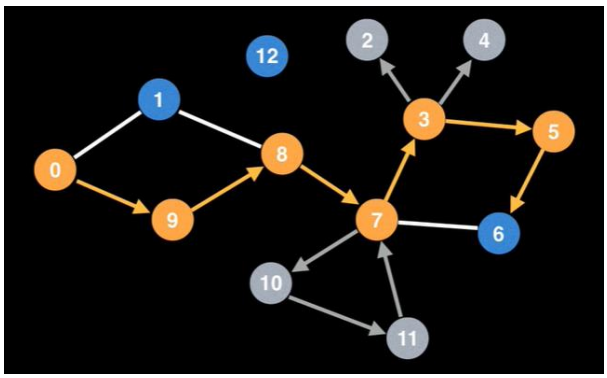


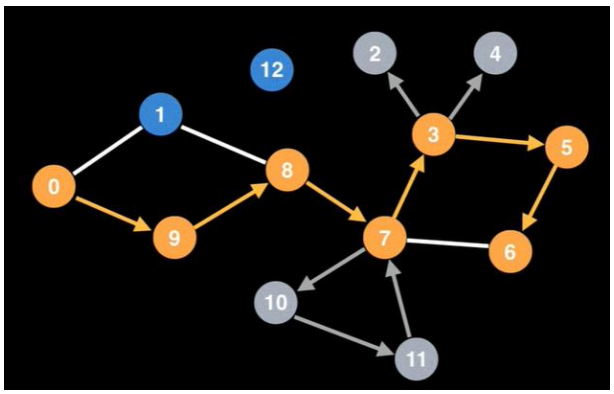
Step4. Vertex 2 is a Dead-end. Again backtracking…

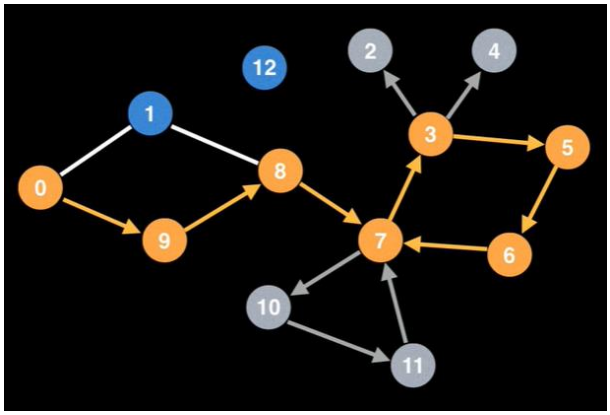


Vertex 4 is also a dead-end.

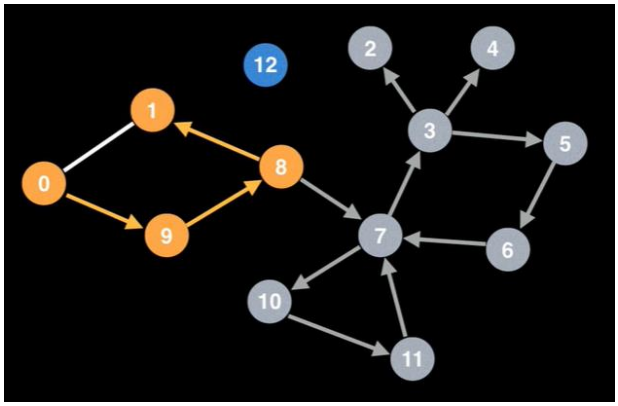
Step5. Now explore the other unvisited vertices (5) of vertex 3

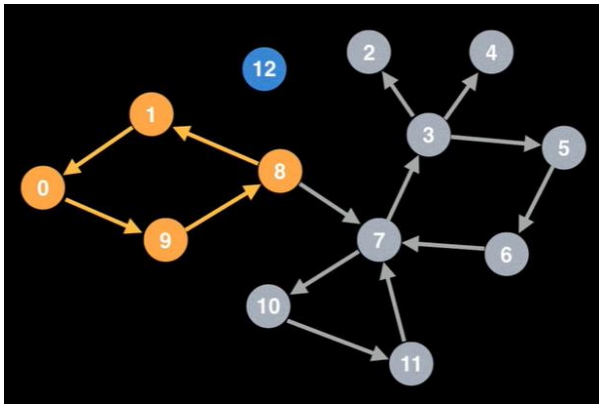




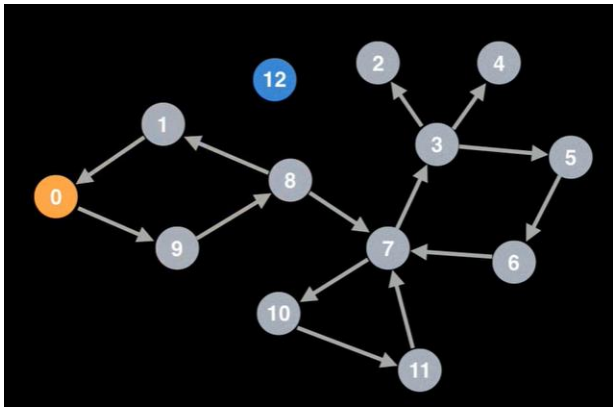


Step6. Backtrack all the way to vertex 8

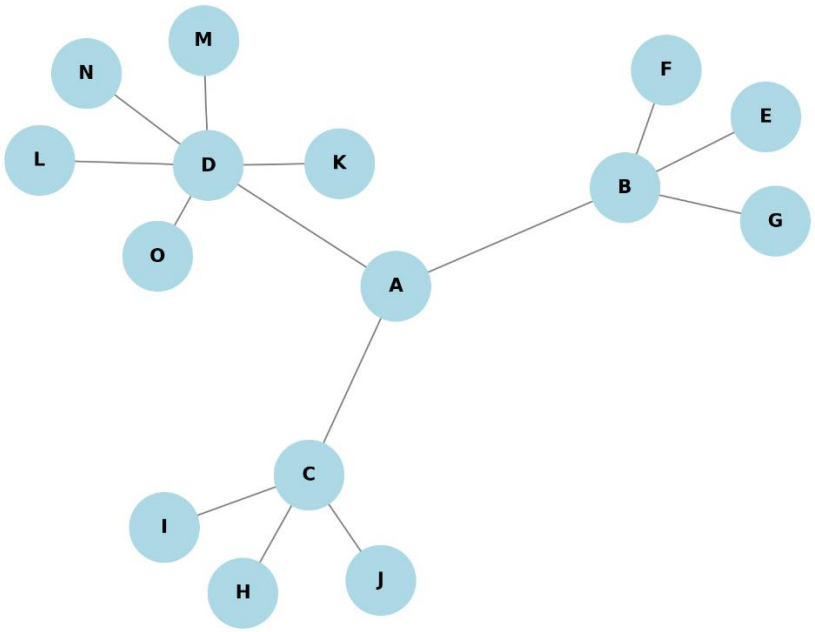




Step7. Again backtrack:



Example2:



Graph connections (from the above figure):

- **A** → {B, C, D}
- **B** → {E, F, G}
- **C** → {H, I, J}
- **D** → {K, L, M, N, O}

DFS Traversal (starting at A)

Let's assume we visit children in the order **from B, C and D**:

1. Start at **A**
2. Go to **B**
    - From B → go to **E** (leaf) → backtrack
    - Next → **F** (leaf) → backtrack
    - Next → **G** (leaf) → backtrack to B → **back to A**
3. From A, go to **C**
    - From C → go to **H** (leaf) → backtrack
    - Next → **I** (leaf) → backtrack
    - Next → **J** (leaf) → backtrack to C → **back to A**
4. From A, go to **D**
    - From D → go to **K** (leaf) → backtrack
    - Next → **L** (leaf) → backtrack
    - Next → **M** (leaf) → backtrack
    - Next → **N** (leaf) → backtrack
    - Next → **O** (leaf) → backtrack to D → **back to A**

DFS order:

A → B → E → F → G → C → H → I → J → D → K → L → M → N → O

DFS using a Stack (start at A)

We'll push neighbors **in reverse order** (so that leftmost gets visited first).

Step 0:

- Start with stack = [A]

Step 1:
- Pop A → visit **A**
- Push its neighbors (D, C, B) → stack = [D, C, B]

Step 2:
- Pop B → visit **B**
- Push its neighbors (G, F, E) → stack = [D, C, G, F, E]

Step 3:
- Pop E → visit **E**
- E has no children (so backtracks) → stack = [D, C, G, F]

Step 4:
- Pop F → visit **F**
- No children → stack = [D, C, G]

Step 5:
- Pop G → visit **G**
- No children → stack = [D, C]

Step 6:
- Pop C → visit **C**
- Push its neighbors (J, I, H) → stack = [D, J, I, H]

Step 7:
- Pop H → visit **H**
- No children → stack = [D, J, I]

Step 8:
- Pop I → visit **I**
- No children → stack = [D, J]

Step 9:
- Pop J → visit **J**
- No children → stack = [D]

Step 10:
- Pop D → visit **D**
- Push its neighbors (O, N, M, L, K) → stack = [O, N, M, L, K]

Step 11:
- Pop K → visit **K**
- No children → stack = [O, N, M, L]

Step 12:
- Pop L → visit **L**
- No children → stack = [O, N, M]

Step 13:
- Pop M → visit **M**
- No children → stack = [O, N]

Step 14:
- Pop N → visit **N**
- No children → stack = [O]

Step 15:
- Pop O → visit **O**
- No children → stack = []

Note: The order of visits could have gone in lot of different ways as well.

Breadth-First Search (BFS):

- Explores all nodes at current depth before going deeper. Explores level-by-level (all neighbours first)
- **Advantages**: Finds shortest path
- **Disadvantages**: Uses lots of memory
- **Example**: In Google Maps

Depth-First Search (DFS):

- Explores one path as deep as possible, then backtracking
- **Advantages**: Uses less memory
- **Disadvantages**: May not find optimal solution, can get stuck
- **Example**: Exploring a maze by following one path all the way until dead-end, then backtracking

Space Complexity of BFS and DFS:

Imagine you are **searching for your friend** in your neighbourhood. Imagine you start from **A**, and **A** has 1000 neighbours:

Wide Graph

```
    A
 / / | \  ... (1000 branches)
B   C   D ...
```

BFS:

- Step 1: Put **A** in queue
- Step 2: Visit A → enqueue its **1000 neighbors**
- Queue now holds ~1000 nodes → **O (width of graph)**

DFS:

- Step 1: Push **A**
- Step 2: Go to **B**, then stop (if B has no children) → at most **2 nodes in stack** (A + B)
- Much smaller memory usage

Conclusion: BFS eats a lot of memory if the graph is *wide*.

Deep Graph:

A → B → C → D → E → ... → Z

BFS:

- Step 1: Put A in queue
- Step 2: Visit A → enqueue B
- Then enqueue C, D, E one by one… but at most **2 nodes at a time in queue** (current + next).

DFS:

- Step 1: Push A
- Step 2: Push B
- Step 3: Push C

- … until Z → stack has **all 26 nodes** at once

Conclusion: DFS eats a lot of memory if the graph is *deep*.

For BFS:

Time Complexity: O(V+E)
Space Complexity: O(V)

For DFS:

Time Complexity: O(V+E)
Space Complexity: O(V)