AI Module2

Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015

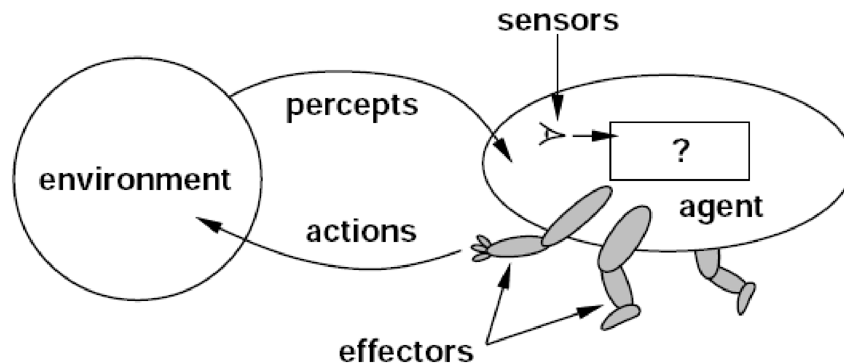
Topics:

1. **A**gent
2. **P**roblem-solving agents,
3. **E**xample problems,
4. **S**earching for Solutions,
5. **U**ninformed Search Strategies:
    a. **B**readth First search,
    b. **D**epth First Search
    c. **D**epth Limited Search
    d. **I**terative deepening depth first search
6. Time and Space Complexity

Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015

2.1 Agent

In the context of AI, **an agent is a system or program** that perceives its environment through sensors, makes decisions or takes actions to achieve specific goals, and is capable of autonomy. Agents can be physical entities like robots or virtual entities like software programs.
- (Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Novig)



Sources: (Pattie Maes, MIT Media Lab) , (Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Novig),

Types of Agents

Agent Type	Description
Simple Reflex Agents	Select actions based on the current percept, without considering the history of past percepts.
Model-Based Reflex Agents	Maintain an internal model of the world, considering the history of percepts for decision-making.
Goal-Based Agents	Designed to achieve specific objectives, using internal goals to determine actions.
Utility-Based Agents	Evaluate actions based on a utility function, quantifying the desirability of different outcomes.
Learning Agents	Improve performance over time through learning from experience.
Logical Agents	Use logical reasoning for decision-making in environments with explicit knowledge representation.
Reactive Agents	Select actions based on the current situation, suitable for real-time, dynamic environments.
Deliberative Agents	Consider multiple possible actions, consequences, and plan ahead to achieve goals thoughtfully.
Mobile Agents	Have the ability to move through the environment, commonly used in robotics and autonomous systems.
Collaborative Agents	Work together, communicate, and coordinate actions to achieve common goals.
Rational Agents	Make decisions that maximize expected utility, aiming for the best outcome given knowledge and goals.

2.2 Problem Solving Agents

Problem Solving Agent is a type of goal based intelligent agent in artificial intelligence that is designed to analyse a situation, identify problems or goals, and then take actions to achieve those goals. These agents are designed to address and solve complex problems or tasks in its environment.

In the city of Arad, Romania, an agent on a touring holiday has a performance measure with various goals, such as improving suntan, language skills, exploring sights, and avoiding hangovers. The decision problem is complex, involving tradeoffs and guidebook analysis. However, if the agent has a nonrefundable ticket to fly out of Bucharest the next day, it is logical for the agent to prioritize the goal of reaching Bucharest. This simplifies the decision problem, as actions not leading to Bucharest can be disregarded. Goal formulation, the first step in problem-solving, helps organize behaviour by narrowing down objectives based on the current situation and the agent's performance measure.

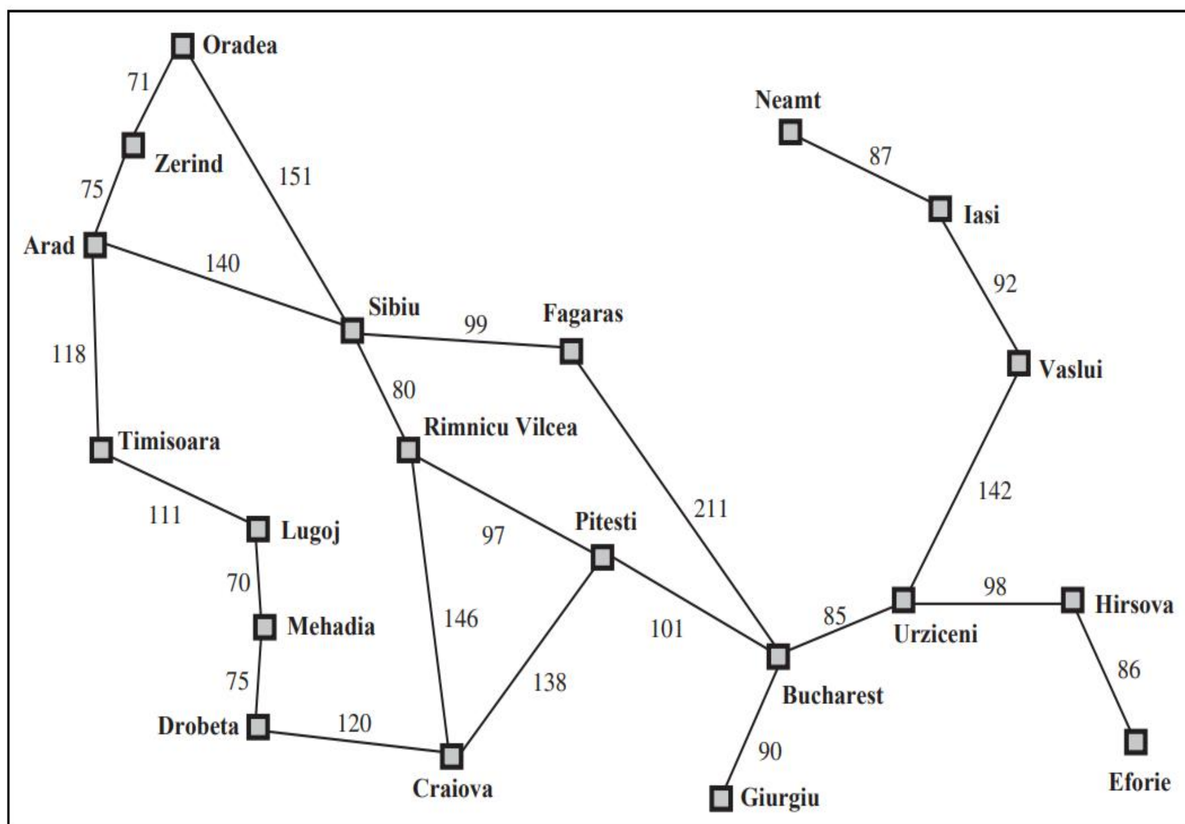


Figure 3.2 A simplified road map of part of Romania.

2.2.1 Steps followed by Problem Solving Agents:

Problem-solving agents follow a series of steps to analyse a situation, formulate goals, and take actions to achieve those goals. The typical steps followed by problem-solving agents are:

1. **Perception**: Gather information about the current state of the environment through sensors or perceptual mechanisms.

2. **Goal Formulation**: Define the objectives or goals that the agent is trying to achieve. This involves specifying what the agent is aiming for in the given situation.

3. **Problem Formulation**: Convert the vague goals into a specific, actionable problem. This step defines the current state, the desired state, and the possible actions that can be taken to move from the current state to the desired state.

4. **Search:** Explore possible sequences of actions to find a solution to the formulated problem. This involves considering different paths and evaluating their feasibility and desirability.

5. **Action Selection**: Choose the best sequence of actions based on the results of the search. The selected actions should lead the agent from the current state to the desired state.

6. **Execution**: Implement the chosen actions in the real world. This involves interacting with the environment and carrying out the planned sequence of actions using actuators.

7. **Learning (Optional):** In some cases, problem-solving agents may incorporate learning mechanisms to improve their performance over time. Learning can be based on feedback from the environment or from the consequences of past actions.

8. **Feedback and Iteration**: If the goals are not fully achieved or if the environment changes, the agent may need to iterate through the problem-solving process. This involves revisiting the perception, goal formulation, and problem formulation steps to adapt to new information.

Hence, we employ a straightforward "**formulate, search, execute**" framework for the agent, illustrated in Figure 3.1. Following the formulation of a goal and a corresponding problem, the agent initiates a search procedure to find a solution. Subsequently, the agent follows the solution's guidance for its actions, executing the recommended next steps—usually starting with the initial action of the sequence—and removing each completed step. Once the solution has been implemented, the agent proceeds to formulate a new goal.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

2.2.2 Well Defined Problems and Solutions:

A problem can be defined formally by five components:

1. **State Representation**: Encompasses the initial state from which the agent begins its problem-solving journey, represented, for example, as "*In(Arad)*."
2. **Actions and Applicability**: Describes the set of possible actions available to the agent in a given state, denoted as ACTIONS(s). For instance, in the state *In(Arad)*, applicable actions include {*Go(Sibiu), Go(Timisoara), Go(Zerind)*}.
3. **Transition Model**: Specifies the consequences of actions through the transition model, represented by the function RESULT(s,a), which yields the state resulting from performing action a in state s. For example, RESULT(*In(Arad),Go(Zerind))=In(Zerind)*.

4. **Goal Specification and Test**: Defines the goal state or states and includes a test to determine whether a given state satisfies the goal conditions. In the example, the goal is represented as the singleton set *{In(Bucharest)}*.
5. **Cost Functions**: Encompasses both the path cost function, assigning a numeric cost to each path, and the step cost, denoted as $c(s,a,s')$, which represents the cost of taking action **a** in state **s** to reach state s'. The cost functions play a crucial role in evaluating and optimizing the performance of the agent's solution.

These **five components** collectively provide a comprehensive definition of a problem and serve as inputs to problem-solving algorithms. The solution to the problem is an action sequence that leads from the initial state to a state satisfying the specified goal conditions, with the quality of the solution evaluated based on the assigned costs.

2.2.3 Formulating Problems

The previous section introduced a problem formulation for reaching Bucharest, involving the initial state, actions, transition model, goal test, and path cost. However, this formulation is a theoretical model, lacking real-world intricacies.

The chosen state description, such as "In(Arad)," simplifies the complex reality of a cross-country trip, excluding factors like travel companions, radio programs, and weather. This simplification, known as abstraction, is essential.
In addition to abstracting the state, actions must also be abstracted. Driving, for instance, involves numerous effects beyond changing location, such as time consumption, fuel usage, and pollution generation. The formulation only considers location changes, omitting actions like turning on the radio or slowing down for law enforcement.

Determining the appropriate level of abstraction requires precision. Abstract states and actions correspond to large sets of detailed world states and sequences. A valid abstraction allows expanding abstract solutions into detailed ones. The abstraction is useful if executing abstract actions is easier than the original problem, ensuring they can be carried out without extensive search or planning by an average agent. Constructing effective abstractions involves removing unnecessary details while maintaining validity and ensuring ease of execution. Without this ability, intelligent agents would struggle to navigate the complexities of the real world.

2.3 Example Problems: Toy problems and Real-World Problems

The problem-solving methodology has found application in diverse task environments, encompassing a broad range of scenarios. We categorize these scenarios into two types**: toy problems and real-world problems**.

A toy problem is designed to showcase or test various problem-solving techniques, featuring a precise and concise description. This allows different researchers to use it for comparing algorithm performances.

On the other hand, **a real-world problem** is one that holds significance for people, lacking a single universally agreed-upon description. However, we can provide a general sense of their formulations.

2.3.1 Toy Problems

2.3.1.1. Vacuum World Problem:

The problem can be formalized as follows:

1. **States:** The state is defined by the agent's location and the presence of dirt in specific locations. The agent can be in one of two locations, each potentially containing dirt. Consequently, there are 8 possible world states (2×2^2). For a larger environment with n locations, there would be $n \cdot 2^n$ states.
2. **Initial state**: Any state can serve as the initial state.
3. **Actions**: In this uncomplicated environment, each state presents three actions: Left, Right, and Suck. More extensive environments might also include Up and Down.
4. **Transition model**: Actions produce expected effects, except for instances where moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square result in no effect. The comprehensive state space is depicted in Figure 3.3.
5. **Goal test**: This assesses whether all squares are clean.
6. **Path cost**: Each step incurs a cost of 1, making the path cost equivalent to the number of steps taken in the path.

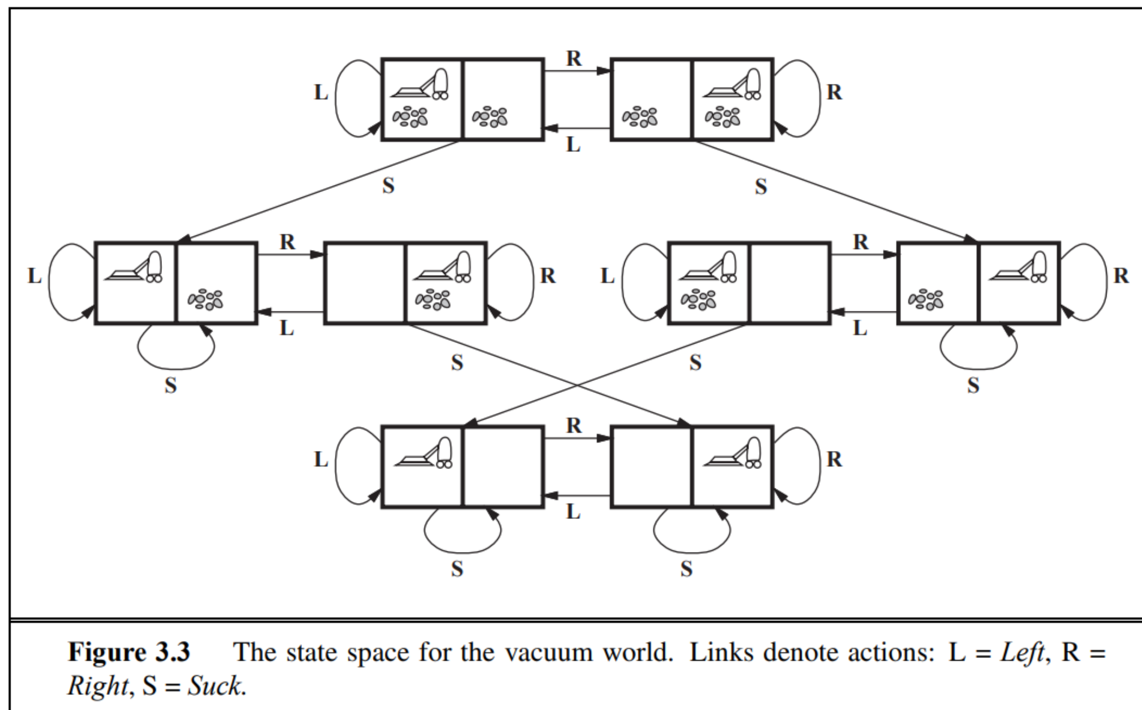


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

2.3.1.2 Eight Puzzle:

The 8-puzzle, illustrated in Figure 3.4, features a 3×3 board with eight numbered tiles and an empty space. Tiles adjacent to the empty space can slide into it, and the goal is to achieve a specified configuration, as depicted on the right side of the figure. The standard formulation is outlined as follows:

1. **States:** A state description indicates the position of each of the eight tiles and the empty space within the nine squares.
2. **Initial state:** Any state can be designated as the initial state.
3. **Actions:** In its simplest form, actions are defined as movements of the empty space—Left, Right, Up, or Down. Different subsets of these actions are possible based on the current location of the empty space.
4. **Transition model**: Given a state and an action, the model returns the resulting state. For instance, applying Left to the starting state in Figure 3.4 would switch the positions of the 5 and the empty space.
5. **Goal test:** This checks if the state aligns with the specified goal configuration shown in Figure 3.4. Other goal configurations are also conceivable.
6. **Path cost:** Each step incurs a cost of 1, making the path cost equivalent to the number of steps taken in the path.

The abstraction in this formulation is evident in the treatment of actions, which are abstracted to their fundamental movements—Left, Right, Up, or Down—depending on the position of the empty space.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Figure 3.4 A typical instance of the 8-puzzle.

2.3.1.2 Eight Queens Problem:

The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

1. **States:** Any arrangement of 0 to 8 queens on the board is a state.
2. **Initial state**: No queens on the board.
3. **Actions:** Add a queen to any empty square.
4. **Transition model**: Returns the board with a queen added to the specified square.
5. **Goal test:** 8 queens are on the board, none attacked.

2.3.1.4 Math's Sequences:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5 .$$

The problem definition is as follows:

1. **States:** Positive numbers.
2. **Initial state**: 4.
3. **Actions:** Apply factorial, square root, or floor operation (factorial for integers only).
4. **Transition model**: As given by the mathematical definitions of the operations.
5. **Goal test**: State is the desired positive integer

2.3.2 Real Time Problems

1. Route Finding Problem
2. Touring Problem
3. Traveling Salesperson Problem
4. VLSI Layout
5. Robot Navigation

Consider the **airline travel problems that must be solved by a travel-planning Web site**:

1. **States:** Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
2. **Initial state**: This is specified by the user's query.
3. **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
4. **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
5. **Goal test**: Are we at the final destination specified by the user?

6. **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

2.4 Searching for Solutions:

- The **SEARCH TREE** possible action sequences starting at the initial state form a search tree with the initial state NODE at the root; the **branches** are **actions** and the **nodes** correspond to states in the state space of the problem.
- **Expanding** the current state applying each legal action to the current state, thereby **generating** a new set of state.

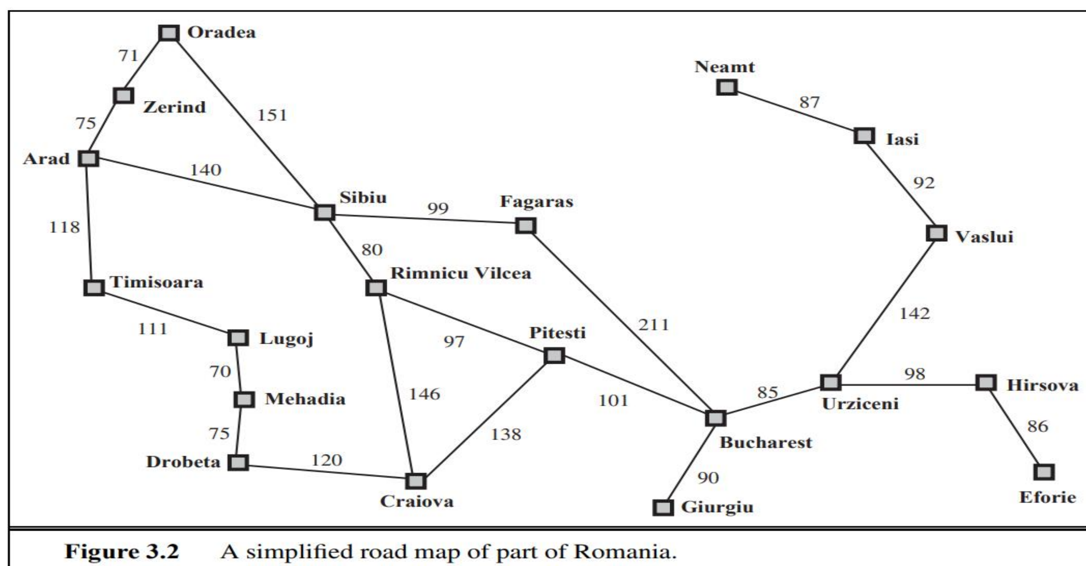
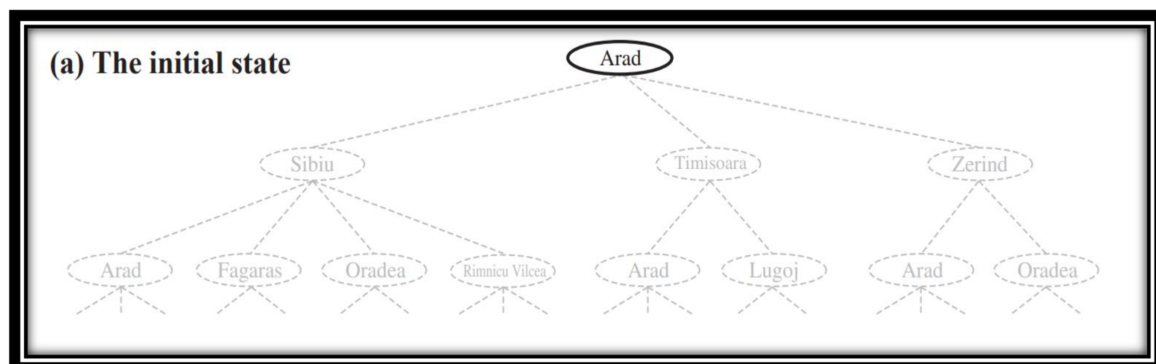
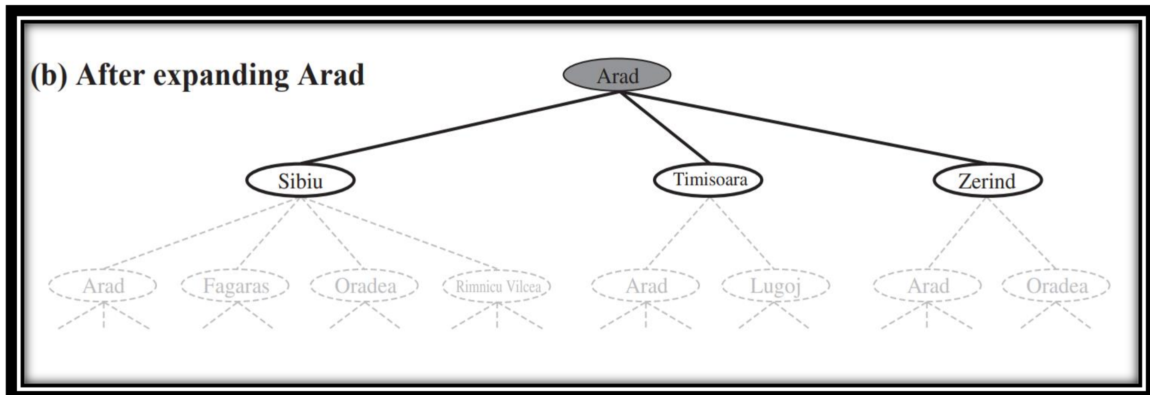


Figure 3.2 A simplified road map of part of Romania.

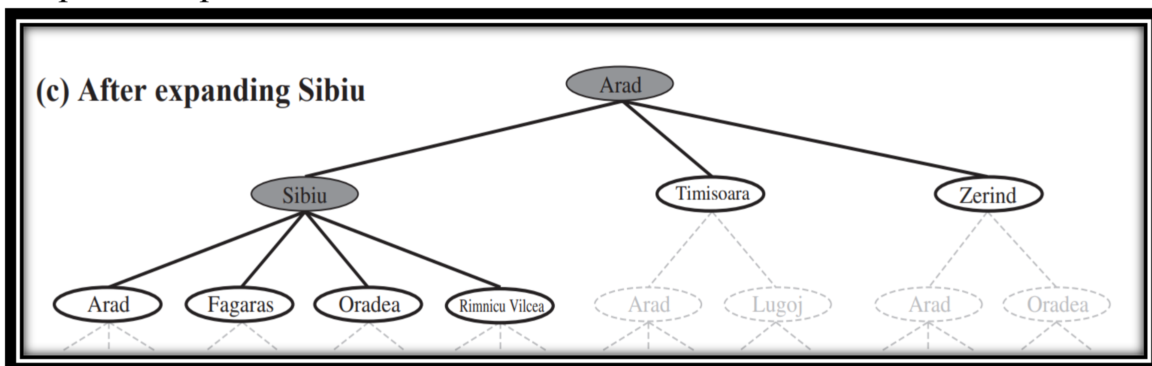
Partial search trees for finding a route from Arad to Bucharest is as shown in the following figures. Initially the searching for route starts from the route node "Arad".



The set of all leaf nodes available for expansion at any given point is called the frontier. Arad node has frontier { Sibiu, Timisoara, Zerind}.Expansion of nodes will be done from left to right.



In the following figure the left most node Sibiu will be further expanded to explore the path to reach the Goal i.e Bucharest.



But after expanding Sibiu , the following frontier will be obtained : {Arad, Fagaras, Oradea, Rimincu Vilcea} . Since Arad is already visited, searching continues from Fagaras.

General Pseudocode for Tree Search is as illustrated below:

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

General Pseudocode for Graph Search is as illustrated below:

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Following figure illustrates the snapshots of sequences of search trees generated by a graph search on the Romania problem (i.e. To find the route from Arad to Bucharest).

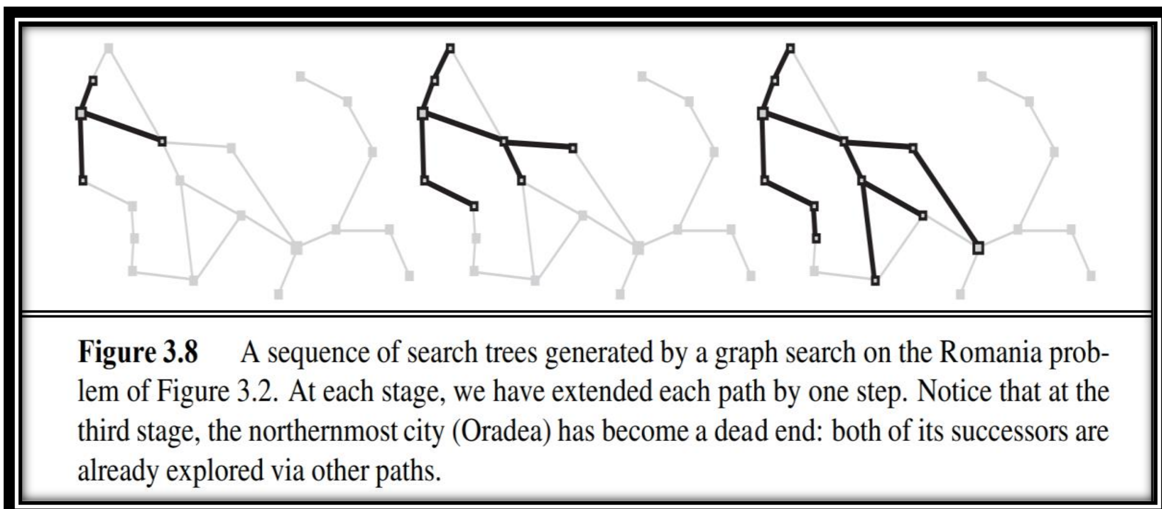


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Following figure illustrates the separation property of Graph search illustrated on a rectangular grid problem:

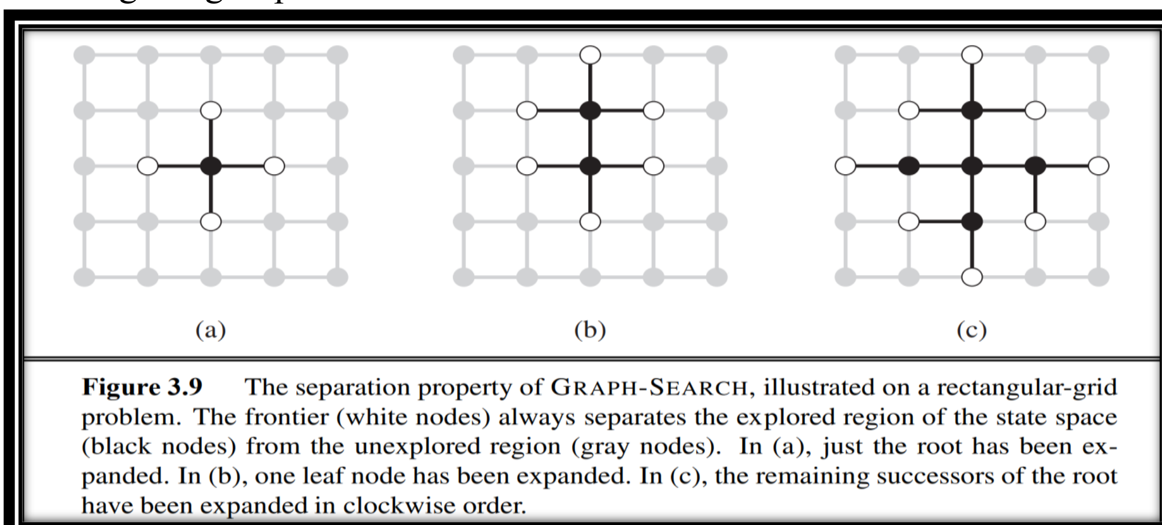


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

2.4.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed.
For each **node n** of the tree, we have a structure that contains four components:

- **n.STATE**: the state in the state space to which the node corresponds;
- **n.PARENT**: the node in the search tree that generated this node;
- **n.ACTION**: the action that was applied to the parent to generate the node;
- **n.PATH-COST**: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

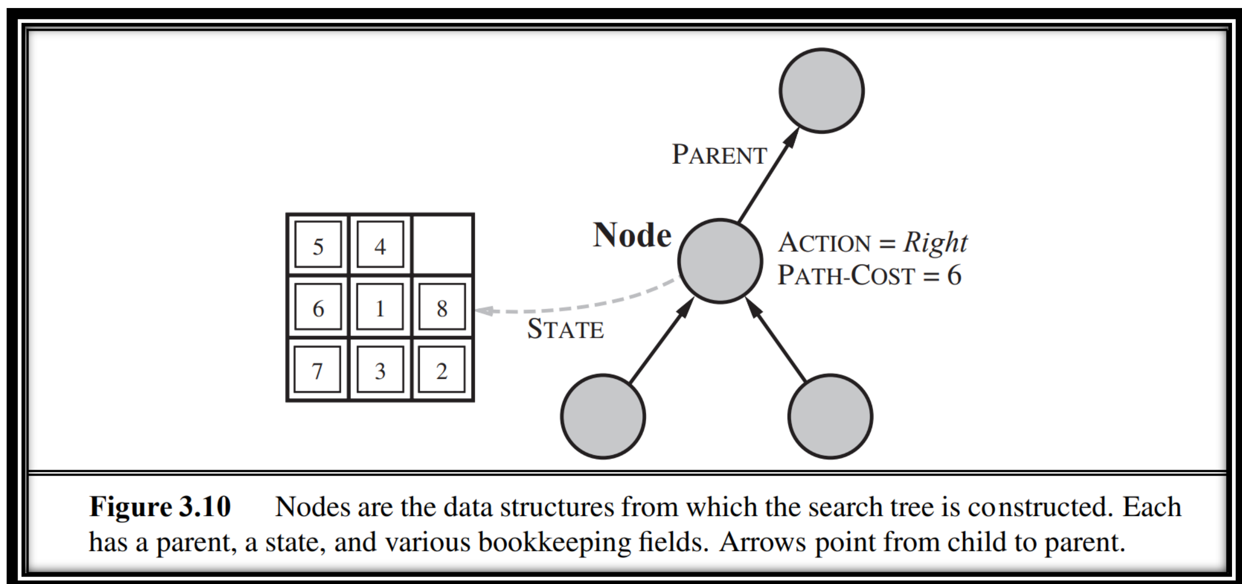


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Following figure illustrates the generic pseudocode for any child node in search tree:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

2.4.2 Measuring problem-solving performance:

We can evaluate an algorithm's performance in four ways:

1. **COMPLETENESS:** Is the algorithm guaranteed to find a solution when there is one?
2. **OPTIMALITY:** Does the strategy find the optimal solution?
3. **TIME COMPLEXITY:** How long does it take to find a solution?
4. **SPACE COMPLEXITY:** How much memory is needed to perform the search?

2.5 Uninformed and Informed Search Strategies:

In the context of search algorithms in artificial intelligence and computer science, the terms "informed" and "uninformed" refer to different strategies for exploring a search space. These strategies are commonly used in algorithms designed to find solutions to problems, typically within a graph or a state space.

Uninformed Search: Also known as blind search, uninformed search algorithms do not have any additional information about the problem other than the connectivity of the states or nodes in the search space.

The algorithms explore the search space without considering the specific characteristics of the problem or the goal location. Uninformed search methods are generally simpler and may explore a large portion of the search space, which can be inefficient for certain types of problems.

Examples:
1.Breadth-first search
2.Uniform-cost search
3.Depth-first search
4.Depth-limited search
5.Iterative deepening depth-first search
6.Bidirectional search

Informed Search: Informed search algorithms, also called heuristic search algorithms, use additional knowledge about the problem to guide the search more efficiently towards the goal.

These algorithms make use of heuristics, which are rules or estimates that provide information about the desirability of different paths. The heuristics help in selecting paths that are more likely to lead to a solution.

A classic example of an informed search algorithm is **A* (A-star),** which incorporates a heuristic function to evaluate the desirability of different paths and combines it with information about the cost incurred so far.

Examples:

1.Greedy best-first search
2.A* search: Minimizing the total estimated solution cost
3.Memory-bounded heuristic search
4.AO* Search
5.Problem Reduction
6.Hill Climbing

Key differences

Characteristic	Uninformed Search	Informed Search
Information Utilization	No additional knowledge	Additional knowledge (heuristics)
Decision Making	Based solely on structure of the search space	Uses heuristics to intelligently guide the search
Goal State	May require exhaustive exploration to find the goal state	More efficient, guided towards potential solution paths
Search Algorithms	Examples: Breadth-First Search Depth-First Search	Examples: A*, Greedy Best-First Search, etc.
Completeness	Completeness depends on the specific algorithm	Completeness depends on the specific algorithm and heuristic
Optimality	May not guarantee the most optimal solution	A* is optimal under certain conditions
Efficiency	May be less efficient for certain problems due to exhaustive exploration	Generally more efficient due to heuristic guidance
Examples	BFS, DFS, Uniform Cost Search, etc.	A*, Greedy Best-First Search,etc.

Enqueue and Dequeue Operations

In the context of any queue-based algorithm, enqueue and dequeue are operations related to adding elements to the queue and removing elements from the front of the queue, respectively.

Enqueue Operation: Adding an element to the end of the queue.

Example: If the queue is [A, B, C] and you enqueue the node D, the queue becomes [A, B, C, D].

Dequeue Operation: Removing an element from the front of the queue.

Example: If the queue is [A, B, C, D] and you dequeue, the result is A, and the queue becomes [B, C, D].

These operations are fundamental for maintaining the order of exploration in BFS. A queue is a First-In-First-Out (FIFO) data structure, meaning that the first element added to the queue will be the first one to be removed. The queue ensures that nodes are processed in a level-wise manner, which is essential for BFS to systematically explore the tree or graph.

2.5.1 Breadth-First Search (BFS)

Breadth-First Search (BFS), a strategy where we start from a root node, expand it to generate its children, and then put those children in a queue (i.e, FIFO) to expand then later. This means all nodes at some depth level d of the tree get expanded before any node at depth level d+1 gets expanded. The goal test is applied when nodes are immediately detected (i.e., before adding it to the queue) because there's no benefit to continue checking nodes. BFS is complete and optimal, but it also suffers from horrible space and time complexity.

Algorithm: Breadth-First Search (BFS): Breadth-First Search is an uninformed search algorithm that explores all the nodes at the current depth before moving on to nodes at the next depth level. It starts at the root node and explores each neighbour before moving to the next level.

Description:

1. Initialize a queue with the initial state (usually the root node).
2. While the queue is not empty:
   a. Dequeue a node from the front of the queue.
   b. If the node contains the goal state, return the solution.
   c. Otherwise, enqueue all the neighbouring nodes that have not been visited.
3. If the queue becomes empty and the goal state is not found, then there is no solution.

Pseudocode for BFS:

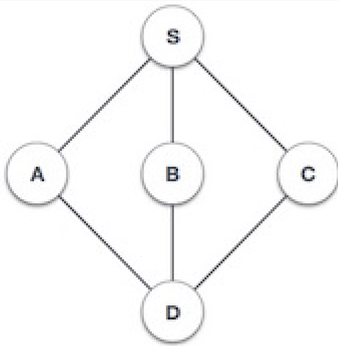
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Example 1: BFS on binary tree

Step	Details	Visited Nodes	QUEUE
1	Initialization: Start with the root node A.	V = {}	a
2	Node **a**, Visited, Dequeue node **a** and enqueue neighbour nodes **b** and **c** to queue	V = {a}	(empty) b c b
3	Node **b** visited, deque node **b** and enqueue neighbour nodes **d** and **e** to queue.	V = {a,b}	c d c e d c
4	Node **c** visited, deque node **c** and enqueue neighbour nodes **f** and **g** to queue	V = {a,b,c}	e d f e d g f e d
5	Node **d** visited, deque node **d**	V = {a,b,c,d}	g f e
6	Node **e** visited, deque node **e**	V = {a,b,c,d,e}	g f
7	Node **f** visited, deque node **f**	V = {a,b,c,d,e,f}	g
8	Node **g** visited, deque node **g**	V = {a,b,c,d,e,f,g}	(empty)

Example 2: BFS on Simple Graph

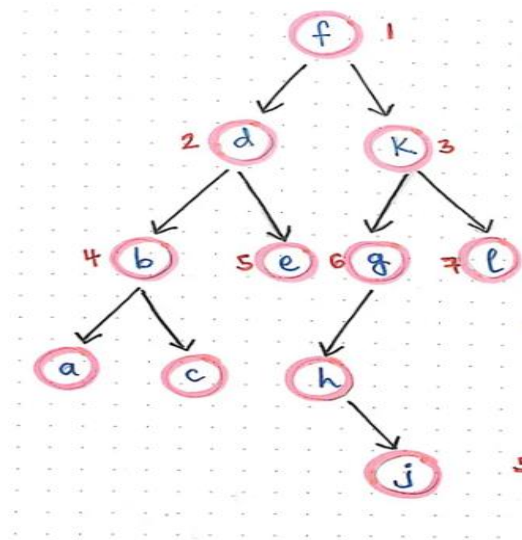


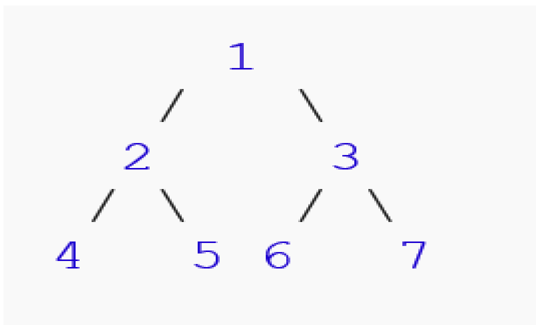
Step	Description	Visited Nodes	Queue
1	Initialize the queue.	V= {}	S
2	Node **S** Visited, Dequeue node **S** and enqueue neighbour nodes **A, B** and **C** to queue.	V= {S}	A B A C B A
3	Node **A** Visited, Dequeue node **A** and enqueue neighbour node **D**	V= {S,A}	D C B
4	Node **B** Visited, Dequeue node **B**	V= {S,A,B}	D C
5	Node **C** Visited, Dequeue node **C**	V= {S,A,B,C}	D
6	Node **D** Visited, Dequeue node **D**	V= {S,A,B,C,D}	

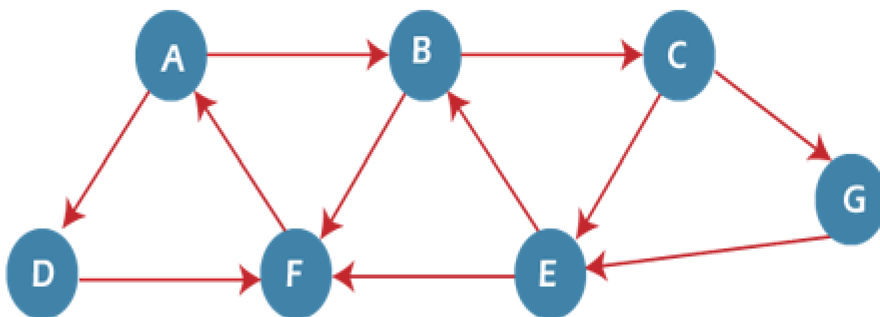
Breadth First Search: S A B C D

.

Exercise: Apply BFS on Following







2.5.2 Depth First Search

Depth-First Search (DFS) is a tree traversal algorithm that explores as far as possible along each branch before backtracking. In the context of a binary tree, DFS can be implemented using **recursion or a Stack (LIFO QUEUE**). Let's go through the DFS algorithm on a binary tree with an example.

DFS Algorithm on Binary Tree:

Recursive Approach:

- Start at the root node.
- Visit the current node.
  - Recursively apply DFS to the left subtree.
  - Recursively apply DFS to the right subtree.

Stack-Based Approach:

Depth-First Search (DFS) can also be implemented using a Last-In, First-Out (LIFO) queue/Stack. The basic idea is to push nodes onto the queue and explore as deeply as possible before backtracking. In the context of a binary tree, the LIFO queue simulates the stack used in the recursive and stack-based approaches. Here's how the DFS algorithm works on a binary tree using a LIFO queue with an example:

DFS Algorithm with LIFO Queue:

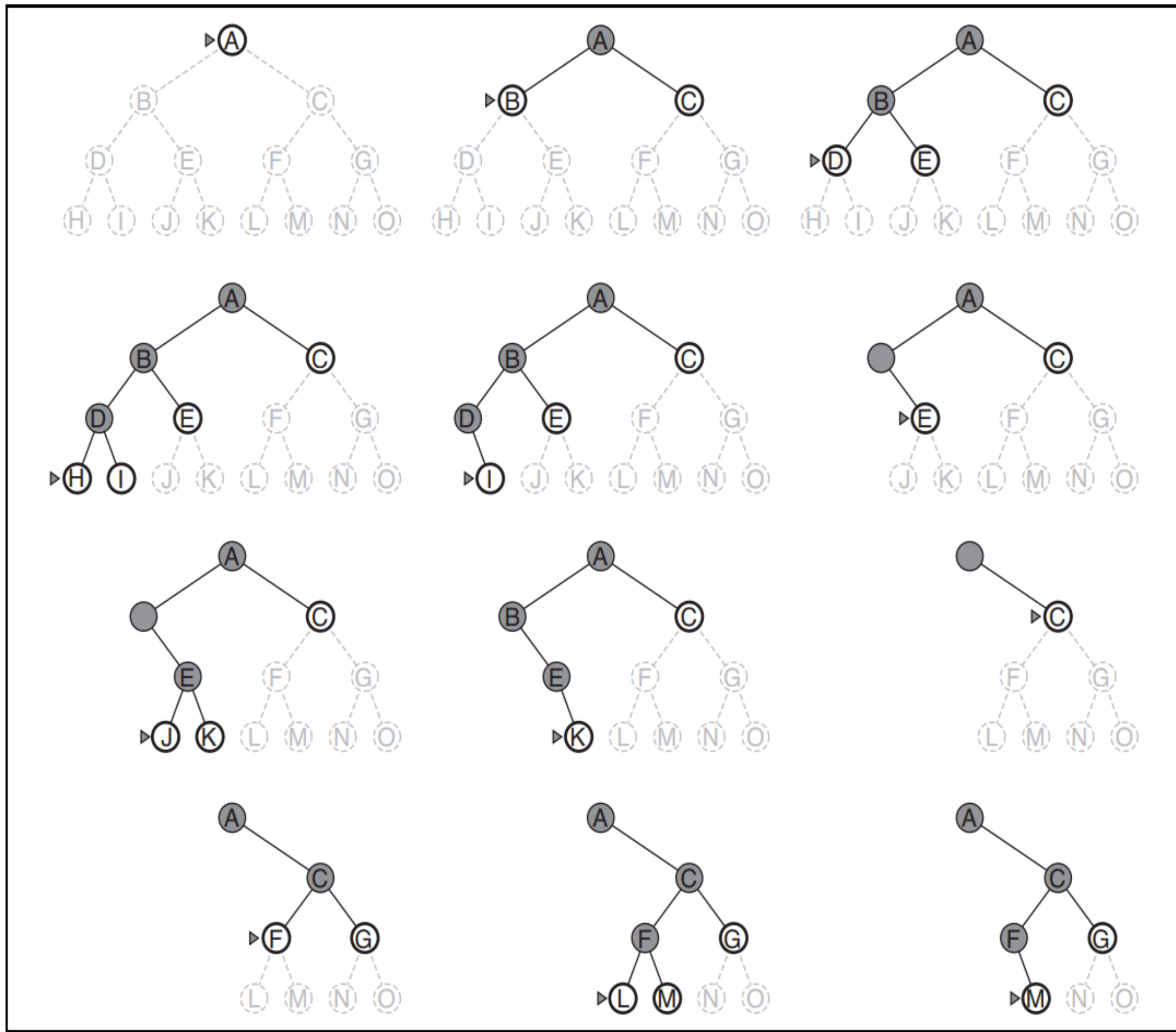
Initialization:
Push the root node onto the LIFO queue.

Traversal:

While the LIFO queue is not empty:

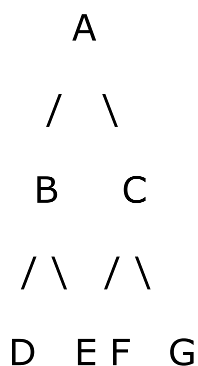
- Pop a node from the LIFO queue.
- Visit the popped node.
- Push the **right child** onto the LIFO queue (if exists).
- Push the **left child** onto the LIFO queue (if exists).

Figure below illustrates the Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.



DFS Resultant Path for the Goal **M** in the above figure is **A->C->F->M**

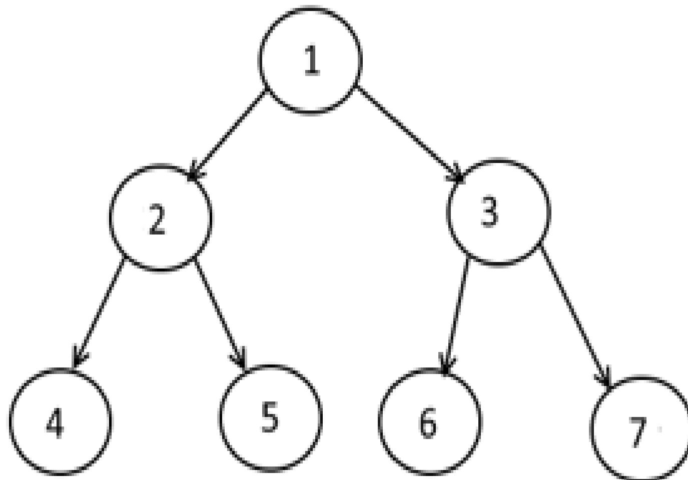
Example: Consider the following binary tree:



Step	Description	Visited Nodes	LIFO QUEUE
1	Push 'A' onto the LIFO queue	{}	[A]
2	Pop 'A', visit it, and push its children 'B' and 'C' onto the LIFO queue	{A}	[C, B]
3	Pop 'B', visit it, and push its children 'E' and 'D' onto the LIFO queue.	{A, B}	[C, E, D]
4	Pop 'D', visit it (no children to push)	{A,B,D}	[C, E]
5	Pop 'E', visit it (no children to push)	{A,B,D,E}	[C]
6	Pop 'C', visit it, and push its children 'F' and 'G'	{A,B,D,E,C}	[G,F]
7	Pop 'F', visit it (no children to push)	{A,B,D,E,C,F}	[G]
8	Pop 'G', visit it (no children to push)	{A,B,D,E,C,F,G}	[]

Resulting DFS Traversal Order: **A,B,D,E,C,F,G**

Exercise: Apply the DFS for the following (Assume the Goal is 6)



2.5.3 Depth Limited Search

Depth-Limited Search is a modification of the traditional Depth-First Search (DFS) algorithm, designed to address the challenges posed by **infinite state spaces**. In infinite state spaces, DFS can get stuck exploring paths indefinitely, leading to an **infinite loop**. To mitigate this issue, a depth limit (ℓ) is introduced in Depth-Limited Search, restricting the exploration depth of the algorithm.
Here are key points about Depth-Limited Search based on the provided discussion:

1. **Purpose of Depth Limit (ℓ):**
   - The depth limit (**ℓ**) is a predetermined value that restricts the depth of exploration.
   - Nodes at depth **ℓ** are treated as if they have no successors, addressing the infinite-path problem.

2. **Completeness and Optimality**:
   - Depth-Limited Search introduces a potential source of incompleteness if **ℓ<d**, where d is the depth of the shallowest goal. This occurs when the shallowest goal is beyond the depth limit.
   - It can be non-optimal if **ℓ>d**.
   - The time complexity of Depth-Limited Search is $\mathbf{O(b^{\ell})}$, and its space complexity is $\mathbf{O(b^{\ell})}$, where b is the branching factor.

3. **Comparison with Depth-First Search**:
   - Depth-First Search can be viewed as a special case of Depth-Limited Search with $\ell = \infty$.

4. **Setting Depth Limits**:
   - Depth limits can be based on problem-specific knowledge. For example, the diameter of the state space can be used as a better depth limit for more efficient search.
   - In some cases, a good depth limit may not be known until the problem is solved.

5. **Implementation:**
   - Depth-Limited Search can be implemented as a modification of the general tree or graph-search algorithm.
   - It can also be implemented as a simple recursive algorithm.

6. Termination Conditions:

- Depth-Limited Search can terminate with two types of failure: the standard failure value (indicating no solution) and the cutoff value (indicating no solution within the depth limit).

Figure below illustrate the Pseudocode for depth limited tree search:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Algorithm:

Initialization:

- Set the depth limit to l
- Push the root node onto the LIFO queue.

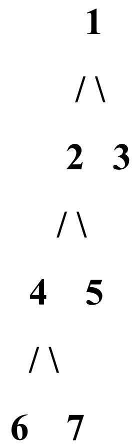
Traversal:

d = 0

While the LIFO queue is not empty and d<=l and goal is not reached:

- Pop a node from the LIFO queue.
- Visit the popped node.
- Push the **right child** onto the LIFO queue (if exists).
- Push the **left child** onto the LIFO queue (if exists).
- Update the depth if goal is not reached

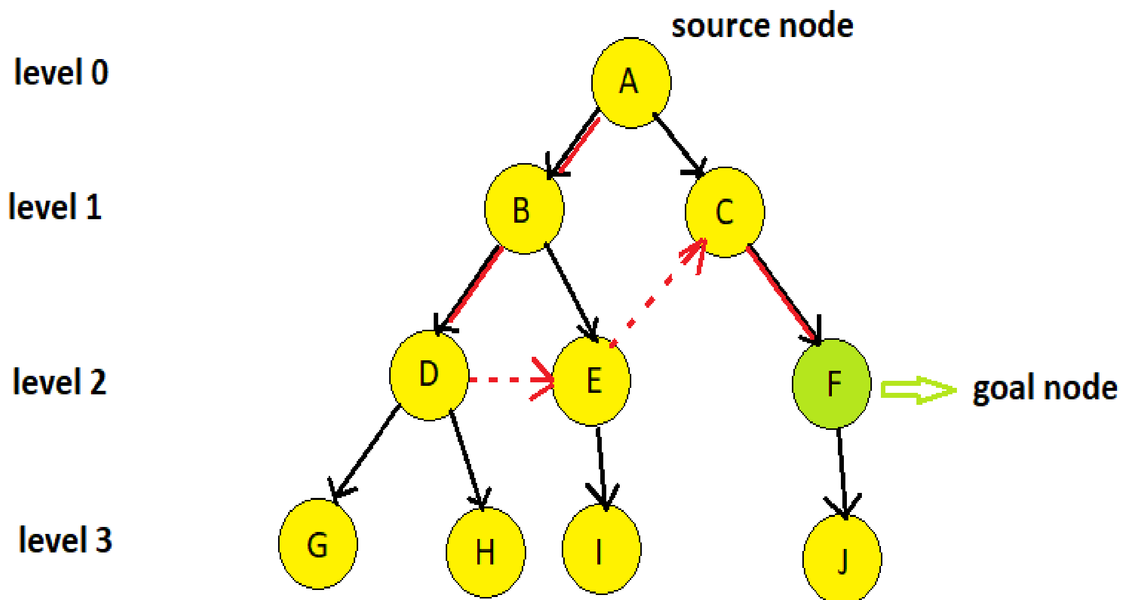
Example1: Let's consider a simple example of Depth-Limited Search (DLS) on a tree. In this example, we'll use a tree where each node has a value, and the goal is to find a specific target value. We'll set a depth limit (ℓ) to control the depth of exploration. Consider the following tree:



Step	Description	Visited Nodes	LIFO QUEUE	Depth d
1	Push '1' onto the LIFO queue	{}	[1]	0 < 2
2	Pop '1', visit it, and push its children '2' and '3' onto the LIFO queue	{1}	[3,2]	1<2
3	Pop '2', visit it, and push its children '5' and '4' onto the LIFO queue.	{1, 2}	[3,5,4]	2=2
4	Pop '4', visit it (no children to push)	{1,2,4}	[3, 5]	2
5	Pop '5', visit it (Goal reached)	{1,2,4,5}	[3]	2

In this example, the search terminates successfully, finding the goal node with the value 5 within the depth limit of 2. If the depth limit were **set to 1**, the search would not have found the goal, demonstrating how the depth limit affects the exploration of the tree.

Example2:



DFS: A->B->D->G->H->E->I->C->F

Depth Limit = 2

DLS: A->B->D->E->C->F

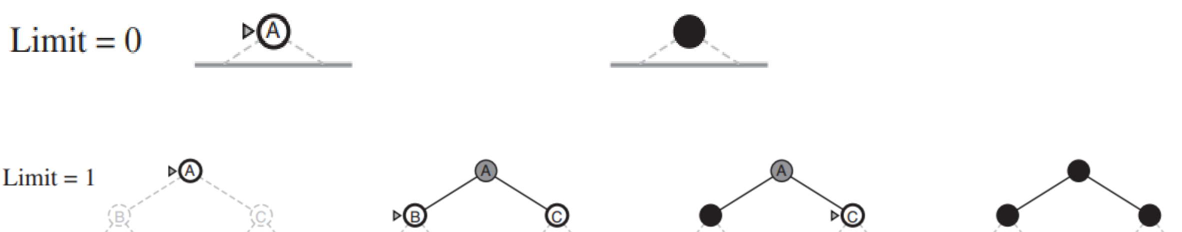
2.5.4 Iterative deepening depth first search

Iterative Deepening Depth-First Search (IDDFS) is a search strategy that combines the benefits of depth-first search and breadth-first search. It systematically performs depth-first search up to a certain depth, incrementally increasing the depth limit in subsequent iterations until the goal is found. This approach ensures completeness and optimality while avoiding the excessive memory usage associated with full breadth-first exploration.

Iterative deepening search, also known as iterative deepening depth-first search, is a versatile strategy commonly employed in conjunction with depth-first tree search to determine the optimal depth limit. This method incrementally raises the limit, starting from 0 and progressing to 1, 2, and so forth, until a goal state is reached, typically when the depth limit attains the value of 'd,' representing the depth of the shallowest goal node. The algorithmic process is illustrated in Figure below: *The iterative deepening search algorithm, which repeatedly applies depthlimited search with increasing limits. It terminates when a solution is found or if the depthlimited search returns failure, meaning that no solution exists.*

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative deepening search combines the advantages of both depth-first and breadth-first search approaches. Similar to depth-first search, it maintains modest memory requirements with a precise complexity of O(bd), where 'b' is the branching factor. Similar to breadth-first search, it achieves completeness when the branching factor is finite and optimality when the path cost is a non-decreasing function of the node's depth. Figure 3.19 illustrates four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, culminating in the discovery of the solution in the fourth iteration.



Limit = 2

Limit = 3

Although iterative deepening search may seem inefficient due to the generation of states multiple times, the cost is not prohibitive. This is because, in a search tree with a consistent branching factor, the majority of nodes are concentrated at the bottom level.

The algorithm generates nodes on the bottom level (depth 'd') once, those on the next-to-bottom level twice, and so forth, up to the children of the root, which are generated 'd' times.

The worst-case total number of generated nodes is given by the formula:

$N(IDS) = (d)b + (d - 1)b^2 + \cdots + (1)b^d$,

resulting in a time complexity of $O(b^d)$, asymptotically equivalent to breadth-first search.

Although there is an additional cost associated with generating upper levels multiple times, it remains manageable.

For instance, with values of 'b = 10' and 'd = 5,' the node counts are :

N(IDS) = 123,450 and

N(BFS) = 111,110.

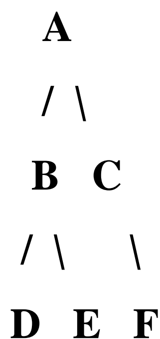
To mitigate repetition concerns, a suggested hybrid approach involves running breadth-first search until memory is nearly exhausted and then switching to iterative deepening from nodes in the frontier.

In general, iterative deepening is the preferred uninformed search method for large search spaces with unknown solution depths. It mirrors breadth-first search by exploring a complete layer of new nodes in each iteration before progressing to the next layer. There is a suggestion to develop an iterative analog to uniform-cost search, inheriting its optimality guarantees while circumventing its memory requirements. The proposed idea involves using increasing path-cost limits instead of increasing depth limits.

Example:

Let's walk through an example of Iterative Deepening Depth-First Search on a binary tree:

Consider the following binary tree:



The goal is to find the node with the value **'F'**. We'll use **Iterative Deepening Depth-First Search** with increasing depth limits.

Iteration 1 (Depth Limit = 0):

- Start at the root node A and perform depth-first search up to depth 0.
- Explore only the root node A.
- No goal found.

Iteration 2 (Depth Limit = 1):

- Start again at the root node A.
- Explore nodes A, B, and C up to depth 1.
- No goal found.

Iteration 3 (Depth Limit = 2):

- Start again at the root node A.
- Explore nodes A, B, C up to depth 2.
- Explore node B's children D and E, and node C's child F.
- Goal 'F' is found.

In this example, IDDFS successfully finds the goal node 'F' by incrementally increasing the depth limit in each iteration. The search is complete, and the solution is found in an optimal manner.

The key advantage of IDDFS is that it guarantees completeness and optimality, similar to breadth-first search, while maintaining the low memory requirements of depth-first search. It is particularly useful in scenarios where memory is limited, and full breadth-first exploration is not practical.

2.6 Time and Space Complexity

Algorithm	Time	Space	Complete	Optimal
Breadth First Search	$O(b^d)$	$O(b^d)$	YES	YES
Depth First Search	$O(b^d)$	O(bd)	NO	NO
Depth Limited Search	$O(b^l)$	O(bl)	NO	NO
Iterative deepening Search	$O(b^d)$	O(bd)	YES	YES