Artificial Intelligence

B.Tech(AIDS)_V Semester

Academic Year: 2024-2025

Dr D. L. Sreenivasa Reddy

Email: dlsrinivasareddy_aids@cbit.ac.in

Syllabus

UNIT - I

Introduction: The Foundation of AI, The History of AI, The State of art.**Intelligent agents**: Agent and Environments, Good Behavior, Nature of Environments, Structure of Agents

UNIT - II

Search Algorithms: State space representation, Search graph and Search tree. Random search, Search with closed and open list, Depth first and Breadth first search. Heuristic search, Best first search. A* algorithm, problem reduction, constraint satisfaction, Game Search, minmax algorithm, alpha beta pruning, constraint satisfaction problems.

UNIT - III

Knowledge & Reasoning: Knowledge-Based Logic Agents, Logic, First-Order Logic, Syntax-Semantics in FOL, Simple usage, Inference Procedure, Inference in FOL, Reduction, Inference Rules, Forward Chaining, Backward Chaining, Resolution.

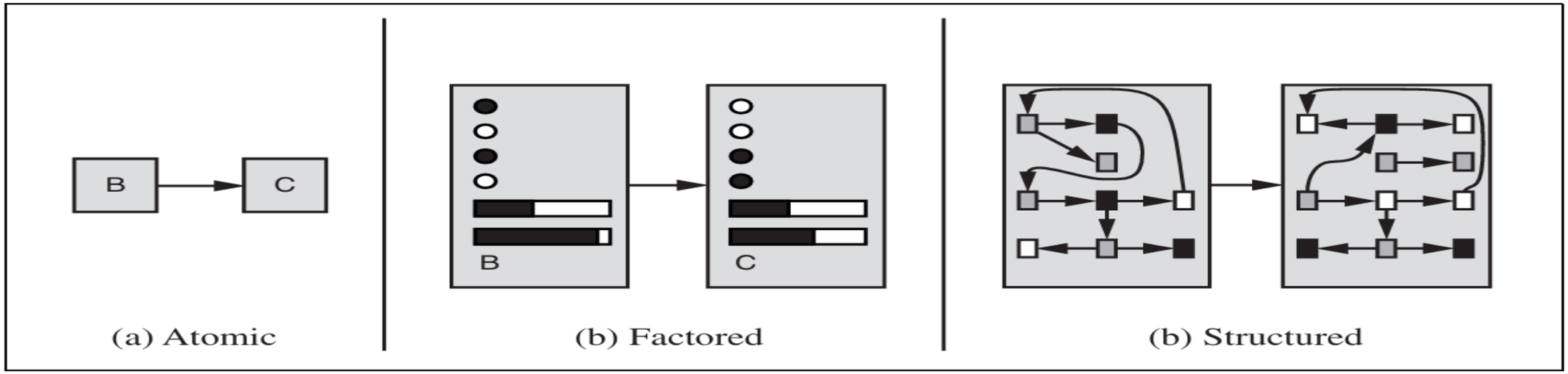
UNIT - IV

Probabilistic Reasoning: Representing knowledge in an Uncertain Domain, The semantics of Bayesian networks, efficient representation of conditional distribution, Inference in Bayesian Networks, Inference in Temporal Models, Hidden Markov models. **Markov Decision Process:** MDP formulation, utility theory, utility functions, value iteration, policy iteration and partially observable MDPs.

UNIT - V

Reinforcement Learning: Introduction**,** Passive reinforcement learning, Active Reinforcement Learning, Generalization in reinforcement learning, adaptive dynamic programming, temporal difference learning, active reinforcement learning- Q learning.

How the components of agent programs work



The representations along an axis of increasing complexity and expressive power

Atomic: (Block box)(GPS)

Factored: (Autonomous Vehicle)(Variables-values)(Different States have different values)

Factored representations-including constraint satisfaction algorithms, propositional logic, planning Bayesian networks and the machine learning algorithms

Structured: (Connection between Variable to variables)

Relational databases and first-order logic , first-order probability models, knowledge-based learning and much of natural language understanding

Learning Agents

- A learning agent can be divided into four conceptual components
  - Learning element
  - Performance element
  - Critic
  - Problem generator
- The design of the learning element depends very much on the design of the performance element.
- When trying to design an agent that learns a certain capability
- Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent
- The problem generator’s job is to suggest these exploratory actions

SOLVING PROBLEMS BY SEARCHING

- **Problem-solving agent**:

When the correct action to take is not immediately obvious, an agent may need a sequence of actions that form a path to a goal state.

- **Search:**Computational process it undertakes is called **search**
- **Problem-solving agents use**:
  - Atomic representations
  - Factored or structured representations of states are called planning agents
- **Informed algorithms**: in which the agent can estimate how far it is from the goal,
- **Uninformed algorithms**: where no such estimate is available.

Problem-Solving Agents

- Agent can follow this **four-phase** problem-solving process
- **Goal formulation:** Goals organize behavior by limiting the objectives and hence the actions to be considered.
- **Problem formulation**: Problem formulation is the process of deciding what actions and states to consider for a given a goal.
- ***Search:*** *Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.* ***Such a sequence is called a solution***.
- **Execution:** The agent can execute the actions in the solution, one at a time.

- In a fully observable, deterministic, known environment, the solution to any problem is a fixed sequence of actions
- If the **model is correct**, then once the agent has found a solution, it can **ignore its percepts** while it is executing the actions
- **Open-loop system**: ignoring the percepts breaks the loop between agent and environment
- **Closed-loop system:** If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent will use Closed-Loop System. That monitors the percept's

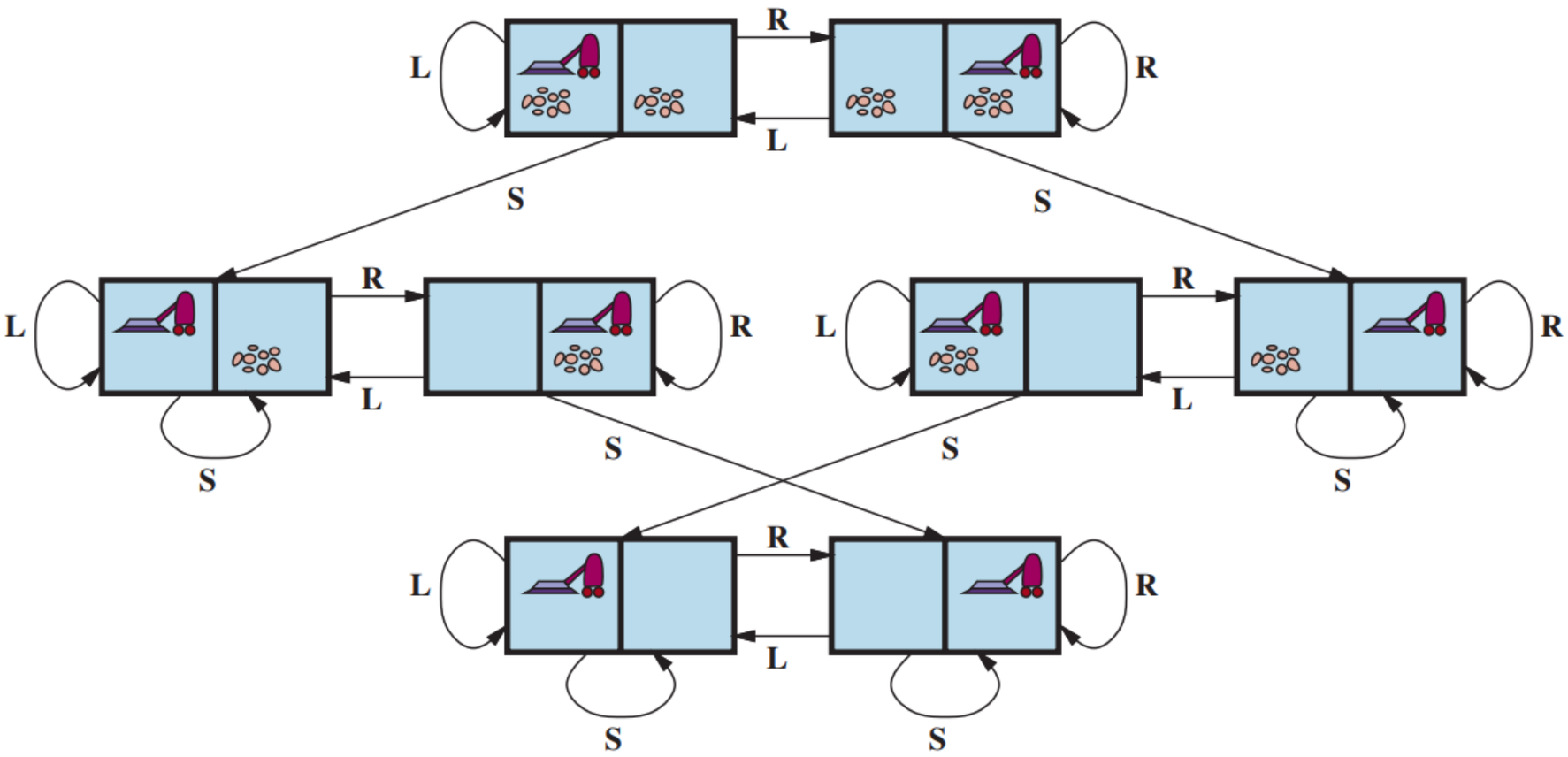
Search Algorithms-Terminology

- **A search problem can be defined formally as follows:**
  - A set of possible **states**(State Space)
  - The **initial state** that the agent starts in
  - A set of one or more **goal states**
  - The **actions** available to the agent. - ACTIONS(s)
  - A **transition model**, which describes what each action does. RESULT(s, a)
  - An **action cost function**, denoted by ACTION-COST(s,a,s ')
  - A sequence of actions forms a **path**
  - **solution** is a path from the initial state to a goal state
  - The **total cost** of a path is the sum of the individual action costs.
  - An **optimal solution** has the lowest path cost among all solutions
  - The **state space** can be represented as **a graph** in which the vertices are states
  - Directed edges between them are **actions**.

Formulating Problems

- The **formulation** of the problem is a model: an abstract mathematical description,not the real thing
- The process of removing detail from a representation is called **abstraction**.
- A good problem formulation has the right level of detail
- **Level of abstraction**:The abstraction is valid if we can elaborate any abstract solution into a solution in the more detailed world

States:
Initial state:
Actions:
Transition model:
Goal test:
Path cost:

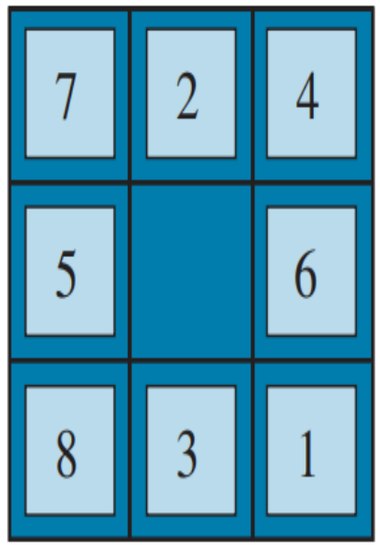


The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = Left, R = Right, S = Suck

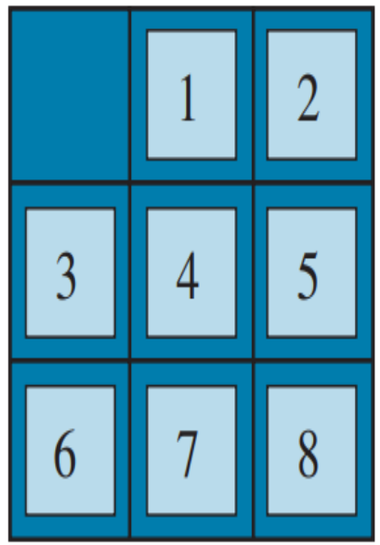
Grid world problem-Vacuum world

- **Grid world problem** is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.
- **States**: A state of the world says which objects are in which cells.
  - objects are the agent and any dirt
  - The agent can be in either of the two cells, and each cell can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states
  - a vacuum environment with n cells has $n \cdot 2^n$ states
- **Initial state**: Any state can be designated as the initial state.
- **Actions:** Suck, move Left, and move Right
- **Transition model**: Suck removes any dirt from the agent's cell; Forward moves the agent ahead one cell in the direction it is facing
- **Goal states**: The states in which every cell is clean.
- **Action cost**: Each action costs 1.

Sokoban Puzzle

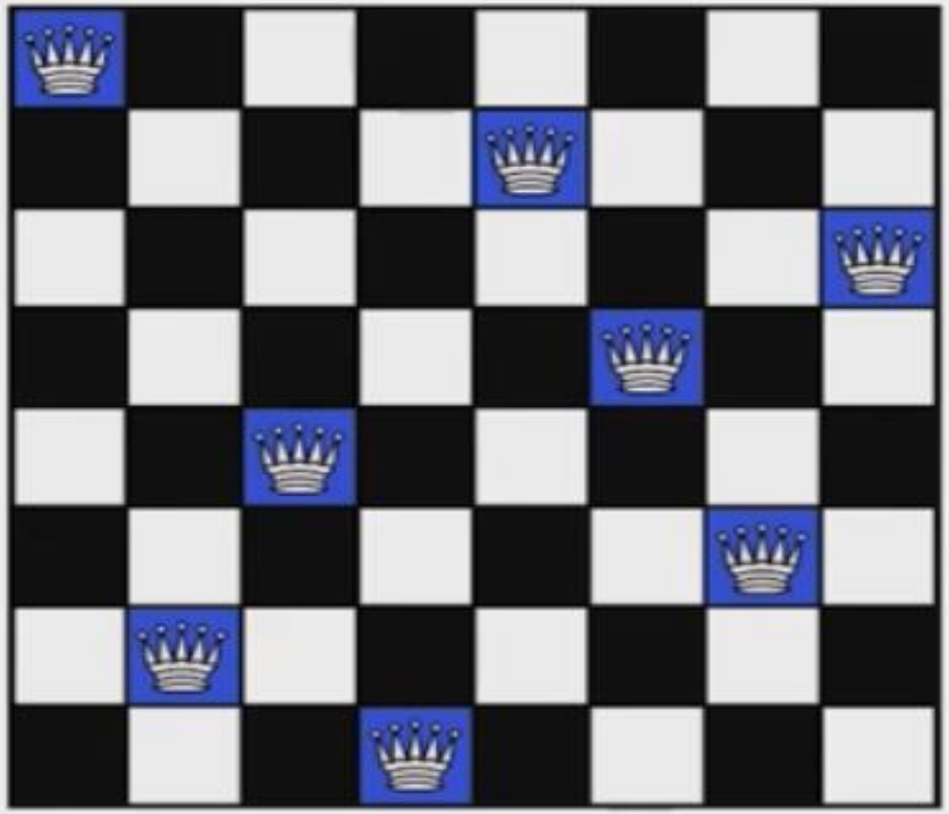


Start State



Goal State

Search agents



The 8-queen problem: on a chess board, place 8 queens so that no queen is attacking any other horizontally, vertically or diagonally.

8-Queens Problem

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- Rule: A queen attacks any piece in the same row, column or diagonal.
- **Incremental formulation:**
  - **States**: Any arrangement of 0 to 8 queens on the board is a state.
  - **Initial state**: No queens on the board.
  - **Actions**: Add a queen to any empty square.
  - **Transition model**: Returns the board with a queen added to the specified square.
  - **Goal test**: 8 queens are on the board, none attacked.
- In this formulation, we have 64.63… 57 ≈ 1.8×10^^14 possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked

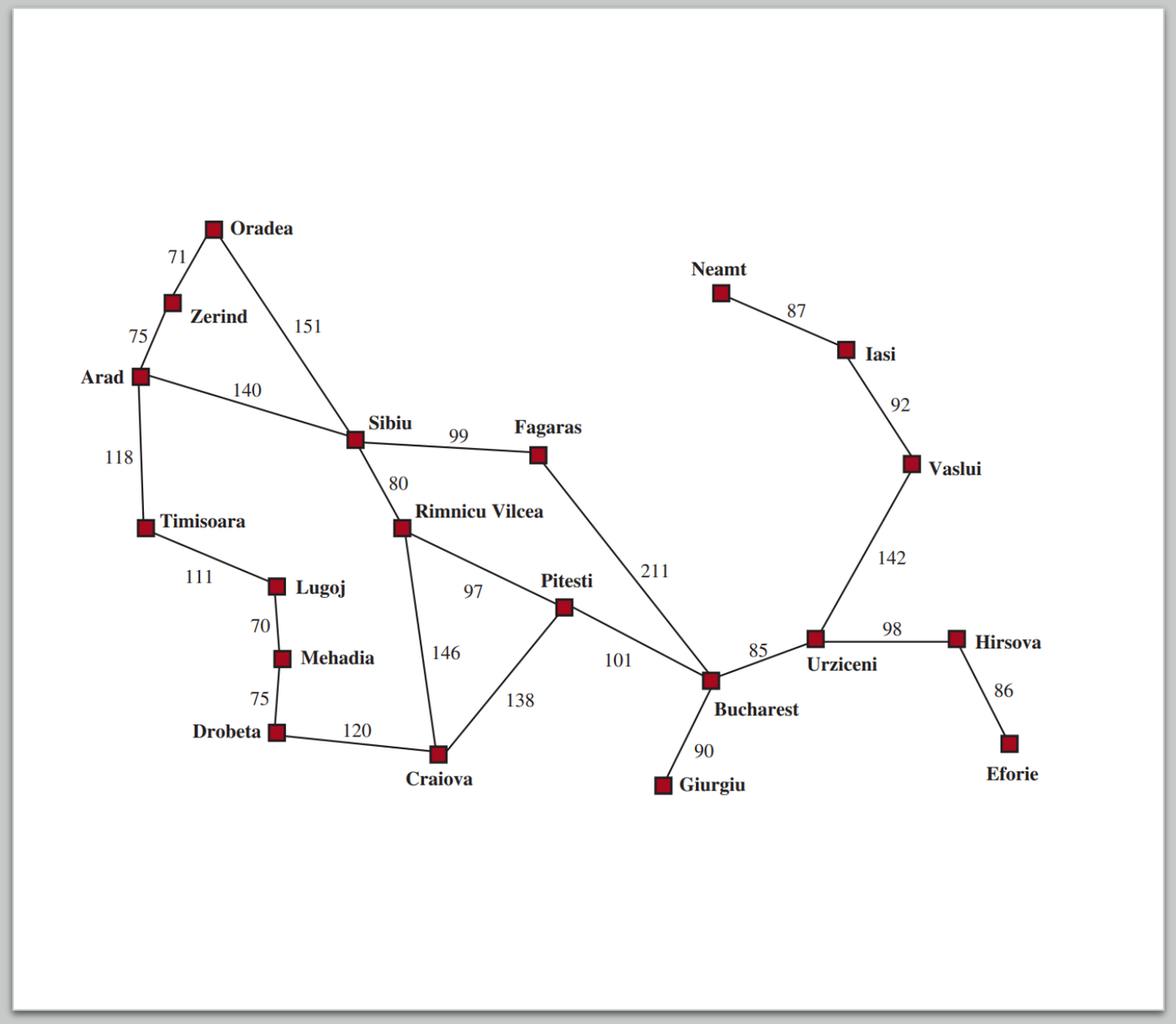
Real-world problems-Airline travel problem

- **States: I**ncludes a location (e.g., an airport) and the current time
- **Initial state:** The user's home or airport
- **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.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new
- **Goal state**: A **destination** city. Sometimes the goal can be more complex, such as "arrive at the destination on a **nonstop** flight."
- **Action cost**:A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Real-world problems

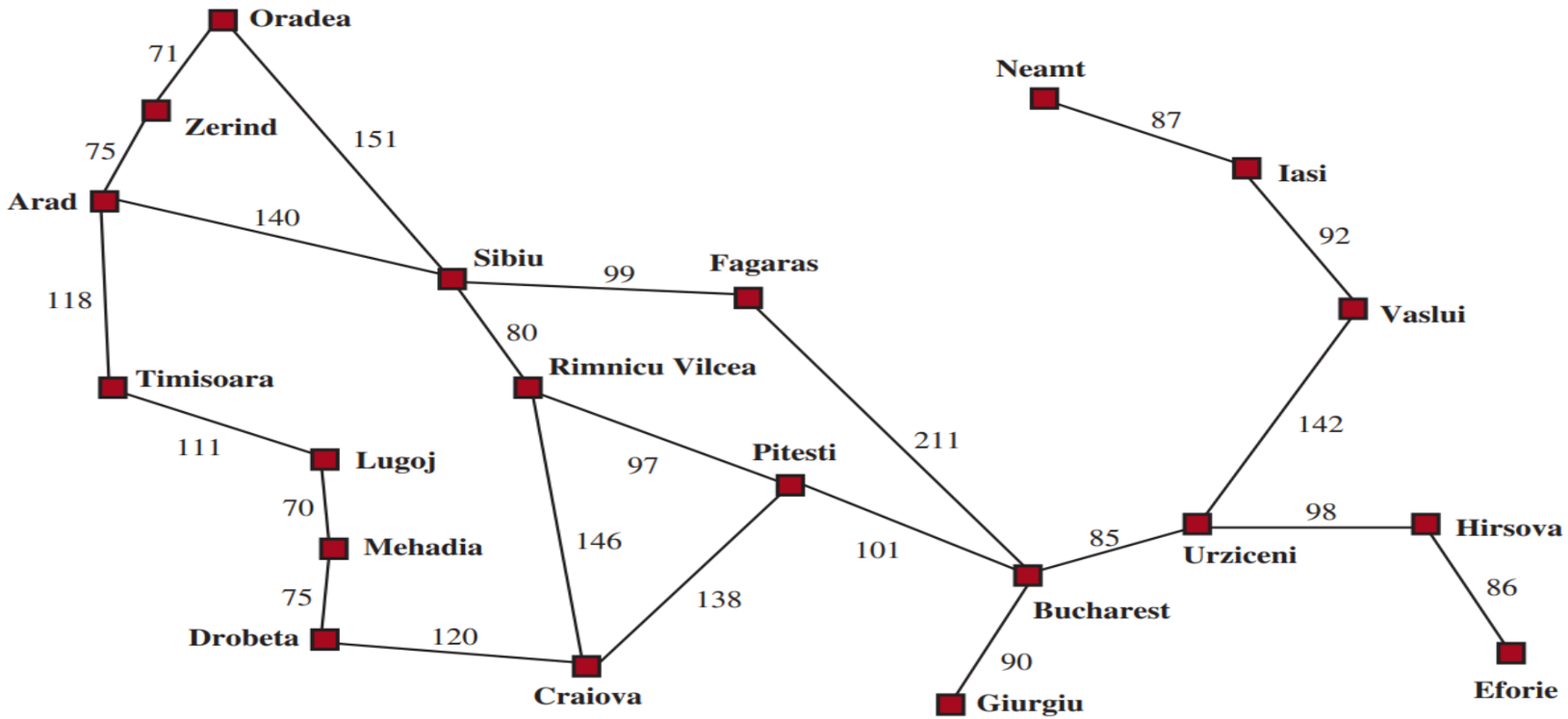
- Touring problem
- Traveling salesperson problem
- VLSI layout
- Robot navigation
- Automatic assembly sequencing

- State Space?
- **initial state?**
- **goal states?**
- **Actions?**
- **transition model?**
- **action cost function?**
- **Path?**
- **Solution?**
- **total cost?**
- **optimal solution?**
- **Repeated state**
- **Cycle**
- **Loopy path**
- **Redundant path**



Search Algorithms

- A search algorithm takes a search problem as input and returns a solution or Failure



- A solution is an action sequence
- The possible action sequences starting at the initial state form a search tree with **the initial state at the root**
- **Expanding** the current state
- **Generating** a new set of states
- **Parent node**
- **Child node**
- **Leaf node**
- **Frontier(open list):** The set of all leaf nodes available for expansion at any given point is called the frontier.
- The **process of expanding nodes** on the frontier continues until either a solution is found or there are no more states to expand.
- **Search strategy: H**ow an algorithm chooses which state to expand next is called **Search Strategy**
- **Repeated state: In(Arad) to Sibu to Arad**
- **Loopy path:** Loopy paths are a special case of the more general concept of redundant paths,
- **Redundant Path:** Which exists whenever there is **more than one way to get from one state to another**.
- **Example:** Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).
- **How to Avoid Redundant paths: Use a separate data structures called** **Explored Set(Closed List)**

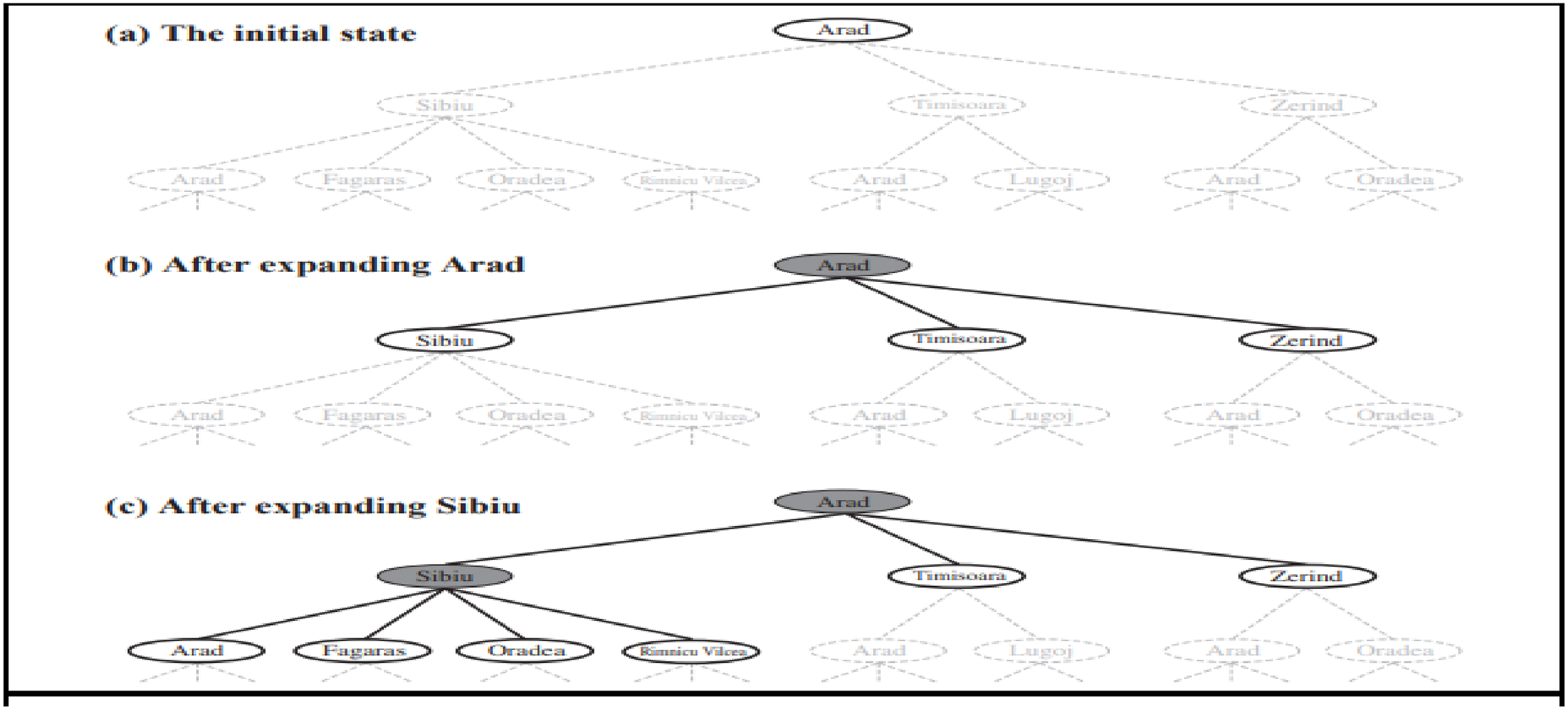
(This remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be dis carded instead of being added to the frontier)

```
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
```

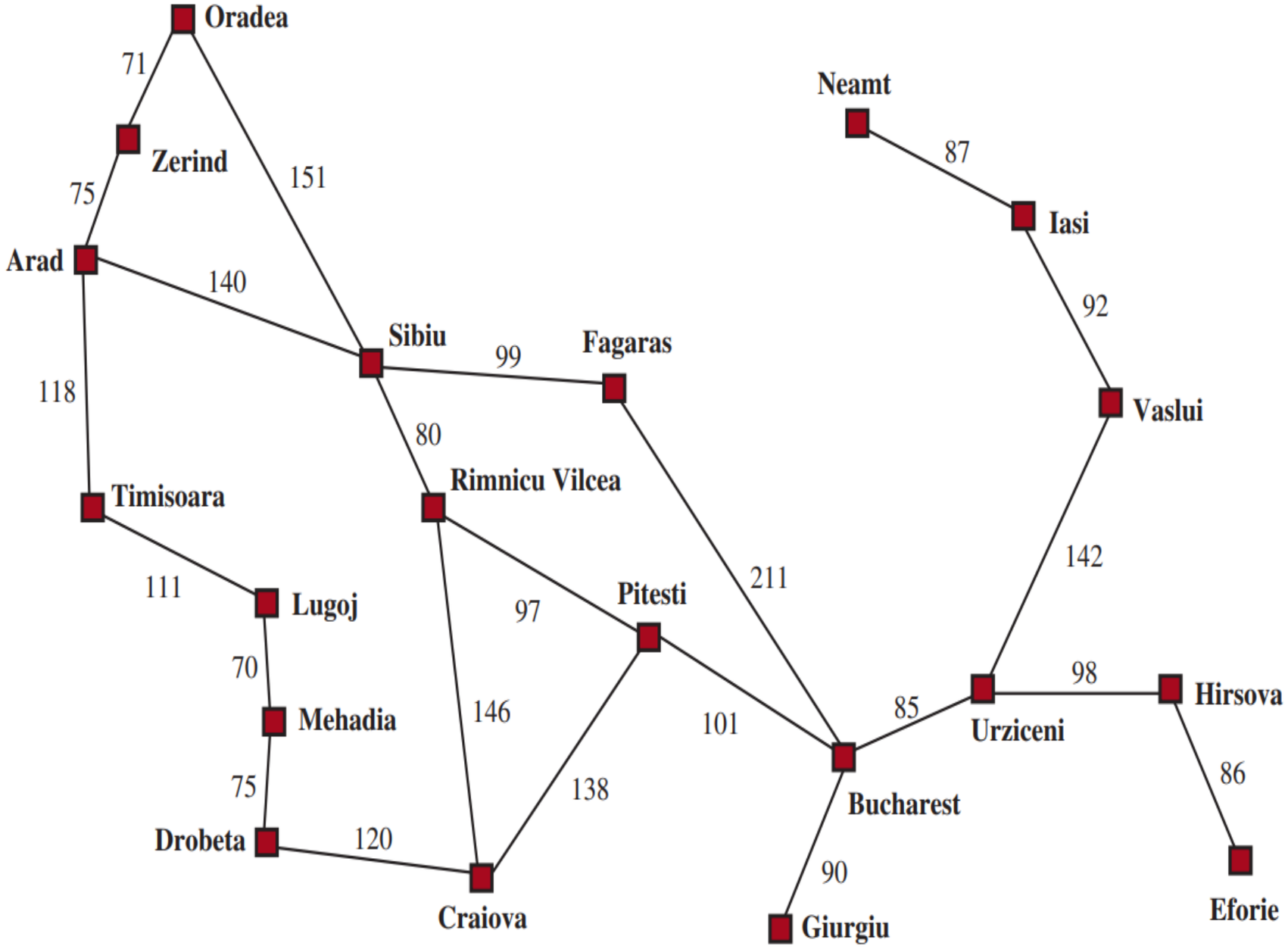
```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        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
```

- GRAPH-SEARCH algorithm contains at most one copy of each state, as growing a tree directly on the state-space graph.
- This algorithm has another nice property: the frontier separates the state-space graph into the explored region and the unexplored region

Conversion of Graph to Tree in Searching Problems



- State Space?
- **initial state?**
- **goal states?**
- **Actions?**
- **transition model?**
- **action cost functic**
- **Path?**
- **Solution?**
- **total cost?**
- **optimal solution?**
- **Repeated state**
- **Cycle**
- **Loopy path**
- **Redundant path**



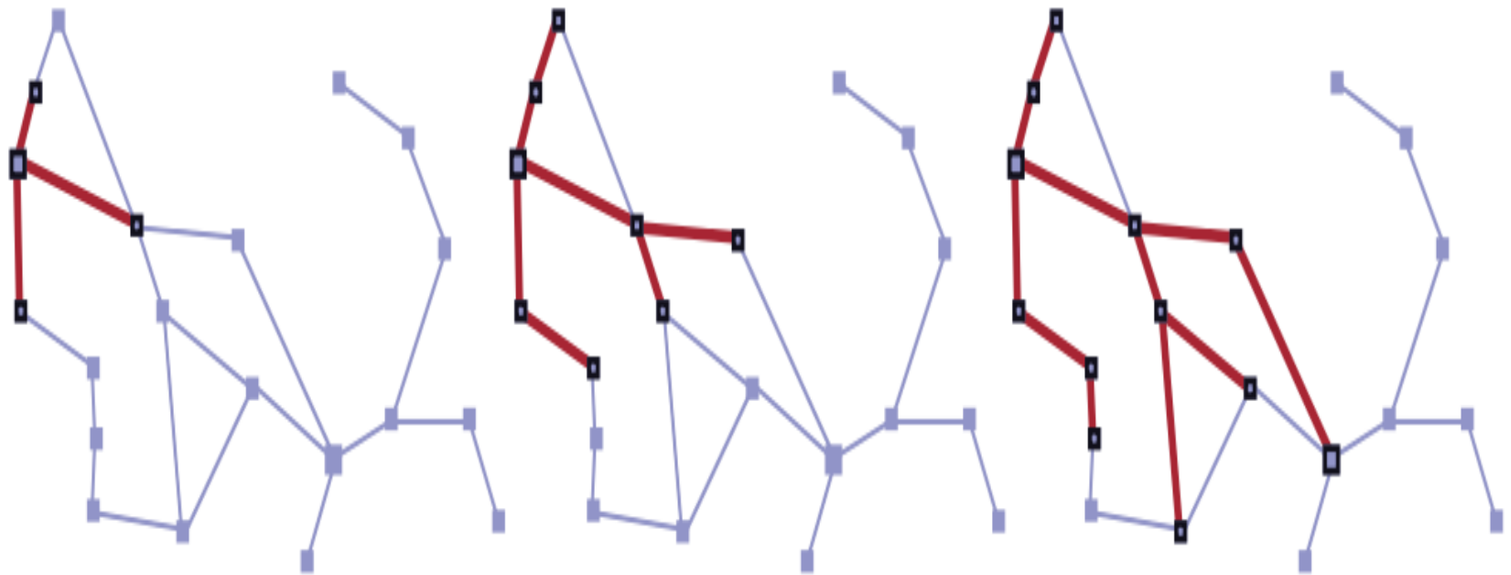
Ex: Searching between Arad to Bucharest

Generating a new node (called a Child node or successor node)

Parent node.

Frontier

Reached



Frontier separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached

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, $action$)

- **A node** is a bookkeeping data structure used to represent the search tree
- **A state** corresponds to a configuration of the world.
- Nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
- Two different nodes can contain the same world state if that state is generated via two different search paths.
- The explored set(Closed Set) can be implemented with a **hash table** to allow efficient checking for repeated states
- Canonical form: logically equivalent states should map to the same data structure

Uninformed Search

- How do we decide which node from the frontier to expand next?
- Choose a node, n, with minimum value of some evaluation function, f(n)
- On each iteration choose a node on the frontier with minimum f(n) value
- It returns it if its state is a goal state,
- otherwise apply EXPAND to generate child nodes.
- Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path
- The algorithm returns either an indication of failure, or a node that represents a path to a goal.

Search data Structures

- Search algorithms require a data structure to keep track of the search tree
- For a node(Data structure with four components)
  - node.STATE: the state to which the node corresponds
  - node.PARENT: the node in the tree that generated this node
  - node.ACTION: the action that was applied to the parent's state to generate this node
  - node.PATH-COST: the total cost of the path from the initial state to this node. g(n)
- PARENT pointers
- Frontier(Data structure to store the frontier)(Queue-priority, FIFO, LIFO)
  - IS-EMPTY(frontier) returns true only if there are no nodes in the frontier.
  - POP(frontier) removes the top node from the frontier and returns it.
  - TOP(frontier) returns (but does not remove) the top node of the frontier.
  - ADD(node, frontier) inserts node into its proper place in the queue.

Measuring problem-solving performance

- Graph search: If redundant
- Tree-like search: If No redundant
- In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infi nite.
- We can evaluate an algorithm's performance in four ways:
  - Completeness
  - Cost optimality
  - Time complexity
  - Space complexity
- Complexity is measured with :
  - **B:**the branching factor or maximum number of successors of any node
  - **D:**the depth of the shallowest goal node (i.e., the number of steps along the path from the root);
  - **M:** the maximum length of any path in the state space.

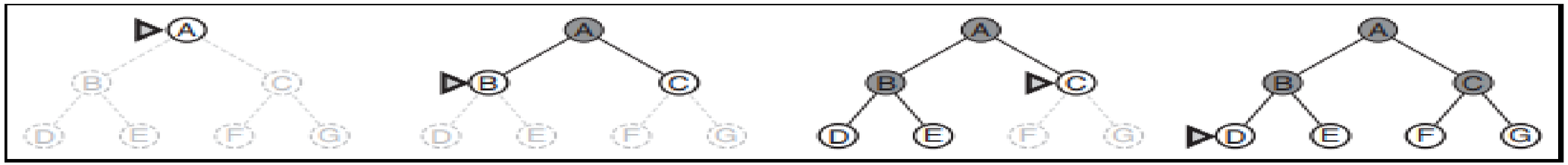
Uninformed Search Strategies(Blind Search) -Breadth-First Search

- An uninformed search algorithm is given no clue about how close a state is to the goal(s)
- When all actions have the same cost, an appropriate strategy is breadth-first search
- The root node is expanded first,
- Then all the successors of the root node are expanded next, then their successors, and so on
- The evaluation function f(n) is the depth of the node—that is, the number of actions it takes to reach the node
- A first-in-first-out queue will be faster than a priority queue

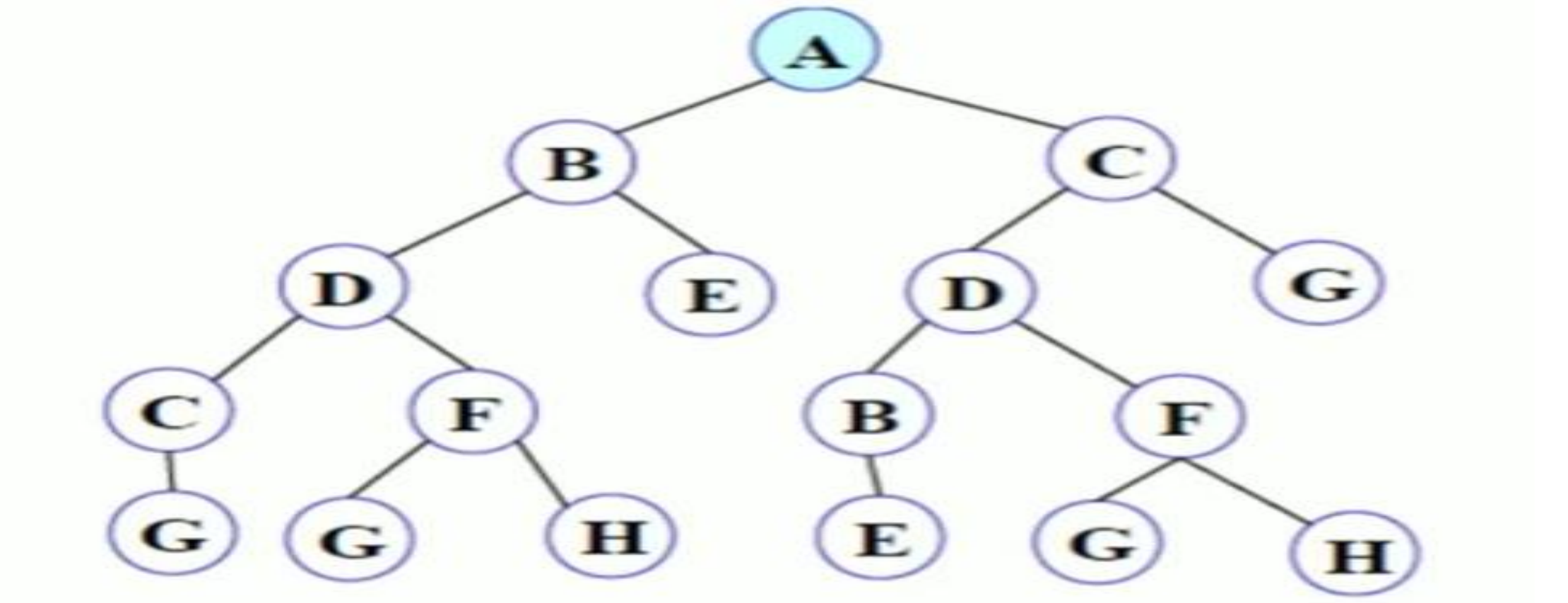
Uninformed Search Strategies-Breadth-First Search

Breadth-First Search – Algorithm

1. Create a variable called NODE-LIST and set it to the initial state.
2. Until a goal state is found, or NODE-LIST is empty:
   a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, then quit.
   b) For element E do the following:
      i. Apply the rule to generate a new state,
      ii. If the new state is a goal state. quit and return this state.
      iii. Otherwise, add the new state to the end of NODE-LIST



Uninformed Search Strategies-Breadth-First Search



Uninformed Search Strategies-Breadth-First Search

Advantages of Breadth first search are:

- One of the simplest search strategies
- BFS is Complete. If there is a solution, BFS is guaranteed to find it.
- If there are multiple solutions, then a minimal solution will be found

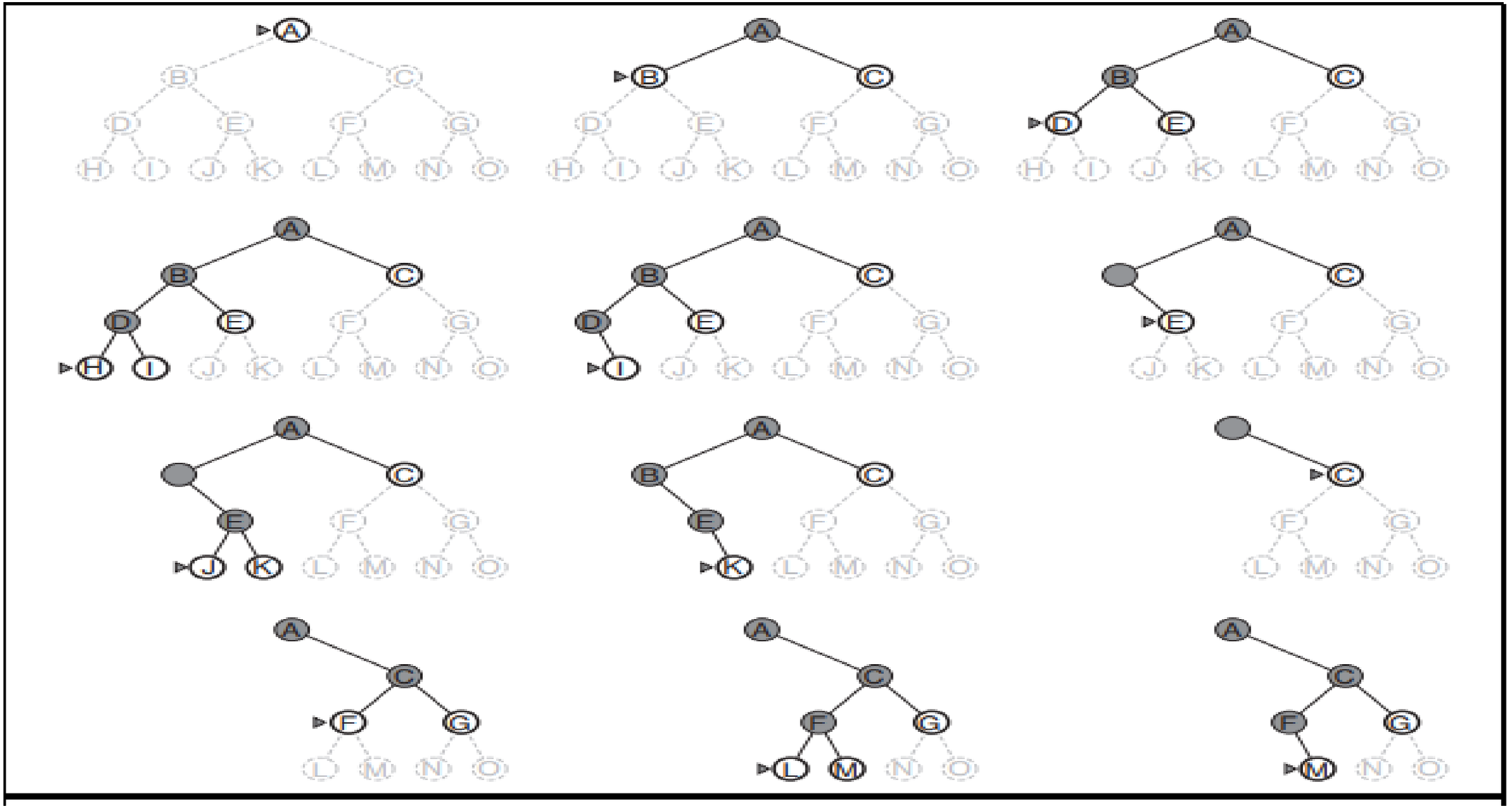
Disadvantages of Breadth first search are :

- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

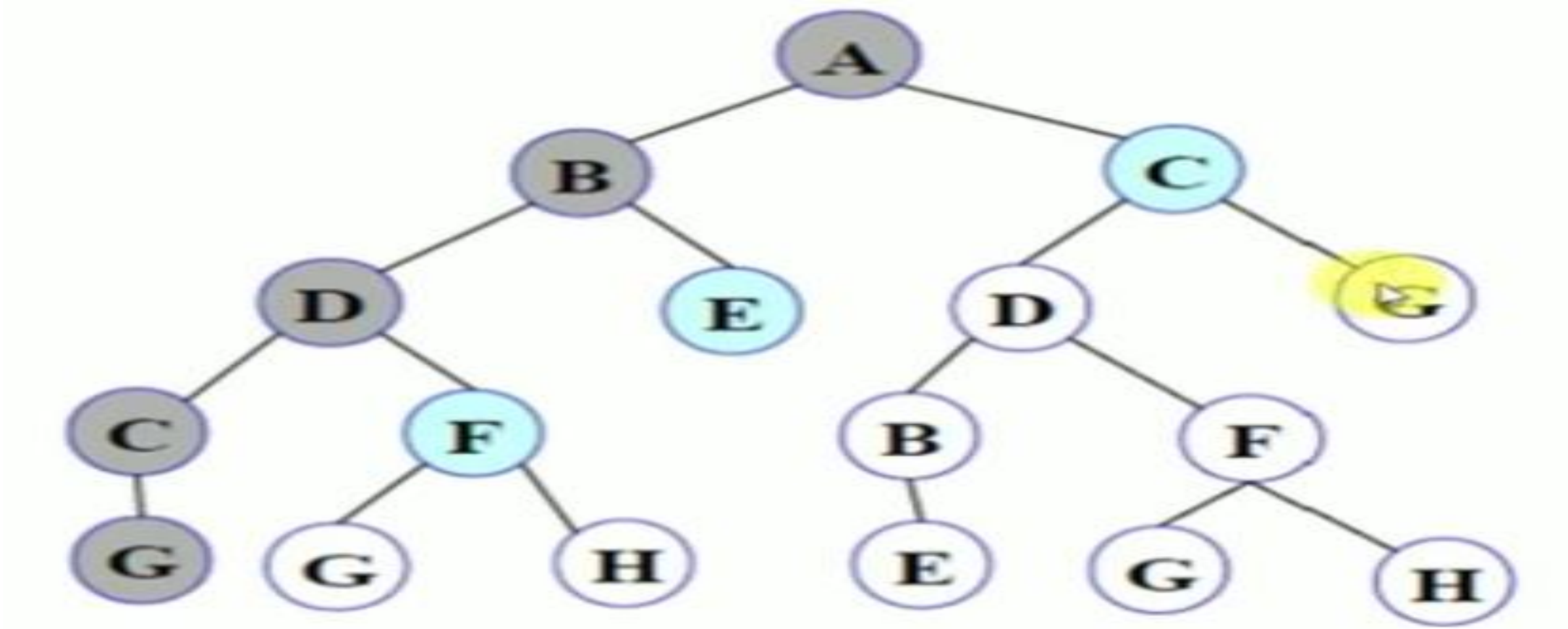
Uninformed Search Strategies-DFS

Depth-First Search Algorithm

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
   a) Generate a successor, E, of the initial state. If there are no more successors, signal failure.
   b) Call Depth-First Search with E as the initial state.
   c) If success is returned, signal success. Otherwise continue in this loop.


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

Uninformed Search Strategies-DFS



Uninformed Search Strategies-DFS

Depth-First Search – Advantages and Disadvantages

Advantages of Depth-first search are:

- Depth-first search requires less memory since only the nodes on the current path are stored.
- The depth-first search may find a solution without examining much of the search space at all.

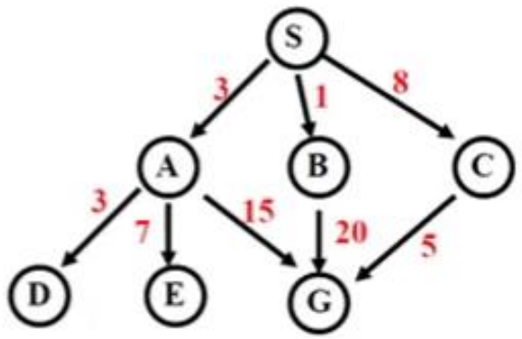
Disadvantages of Depth-first search are:

- May find a sub-optimal solution (one that is deeper or more costly than the best solution)

Uniform Cost Search

Uniform-Cost Search

- Use a priority queue instead of a simple queue
- Insert nodes in the increasing order of the cost of the path so far
- Guaranteed to find an optimal solution!
- This algorithm is called uniform-cost search



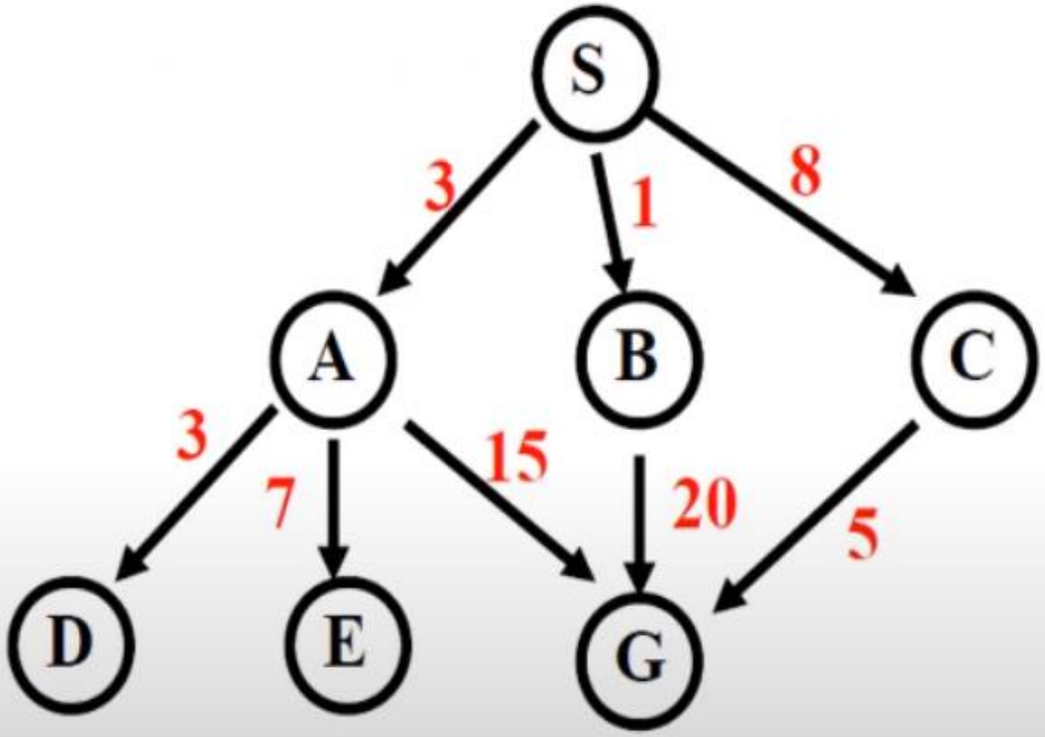
Uniform Cost Search

Uniform-Cost Search...

- Enqueue nodes by path cost.
  - That is, let g(n) = cost of the path from the start node to the current node n.
  - Sort nodes by increasing value of g.

Uniform-Cost Search…

- Expanded node Nodes list
- { S0 }
- S0 { B1, A3, C8 }
- B1 { A3, C8, G21 }
- A3 { D6, C8, E10, G18, G21 }
- D6 { C8, E10, G18, G21 }
- C8 { E10, G13, G18, G21 }
- E10 { G13, G18, G21 }
- G13 { G18, G21 }
- Solution path found is S C G, cost 13
- Number of nodes expanded (including goal node) = 7



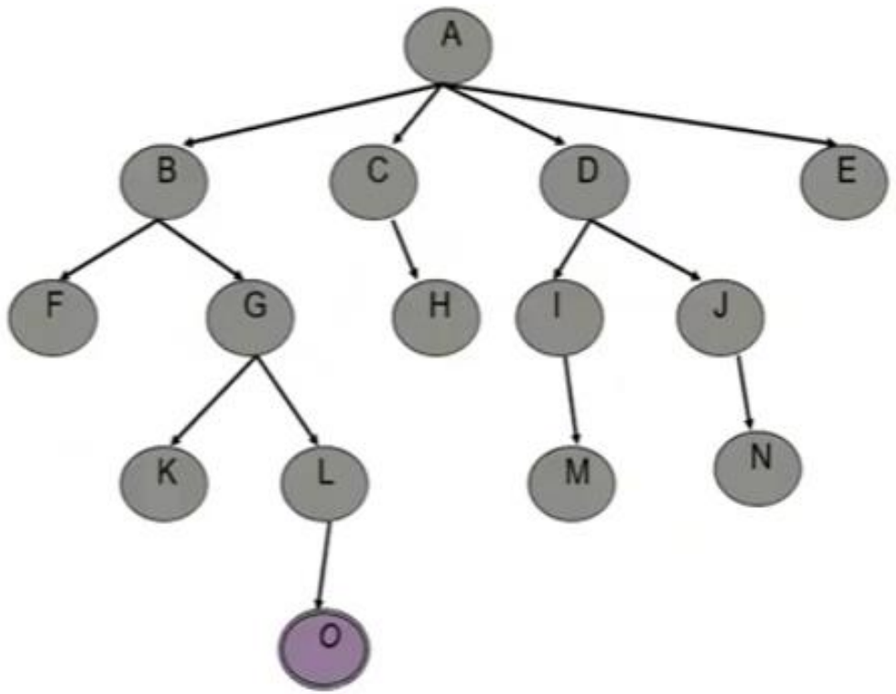
Depth Limited Search

Depth Limited Search (DLS)

- The failure of Depth-First Search is the infinite state spaces, that can be overcome by supplying depth-first search with a predetermined depth limit.
- The depth limit 'l' i.e. the cut-off value or the maximum level of the depth to overcome the disadvantages of depth first search.
- Nodes at depth are treated as if they have no successors.
- This approach is called Depth-Limited Search.
- The depth limit solves the infinite-path problem.
- Notice that depth-limited search can terminate with two kinds of failure:
  - the standard failure value indicates no solution;
  - the cutoff value indicates no solution within the depth limit.

Depth Limited Search (DLS)…

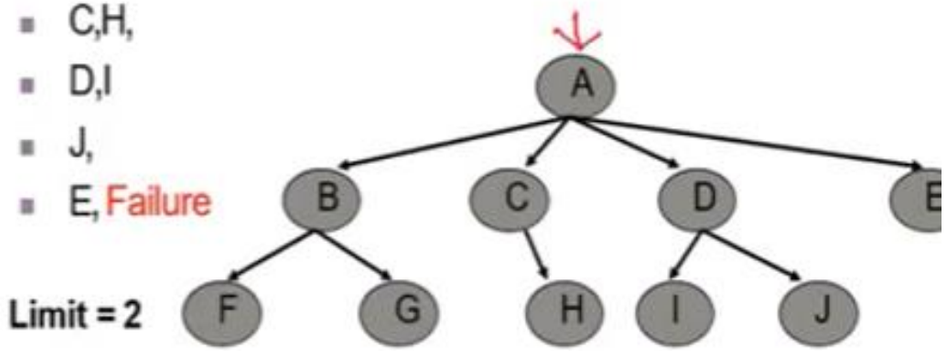
- Limit 0
- Limit 1
- Limit 2

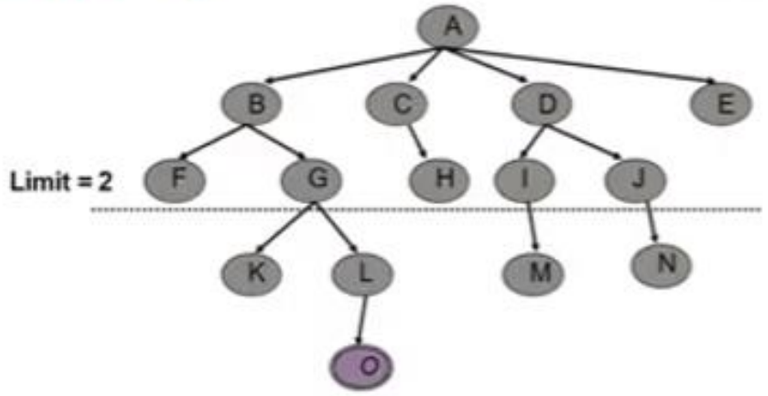


Depth Limited Search (DLS)...

- DLS algorithm returns Failure (no solution)
- The reason is that the goal is beyond the limit (Limit =2): the goal depth is (d=4)

- A,B,F,
- G,
- C,H,
- D,I
- J,
- E, Failure





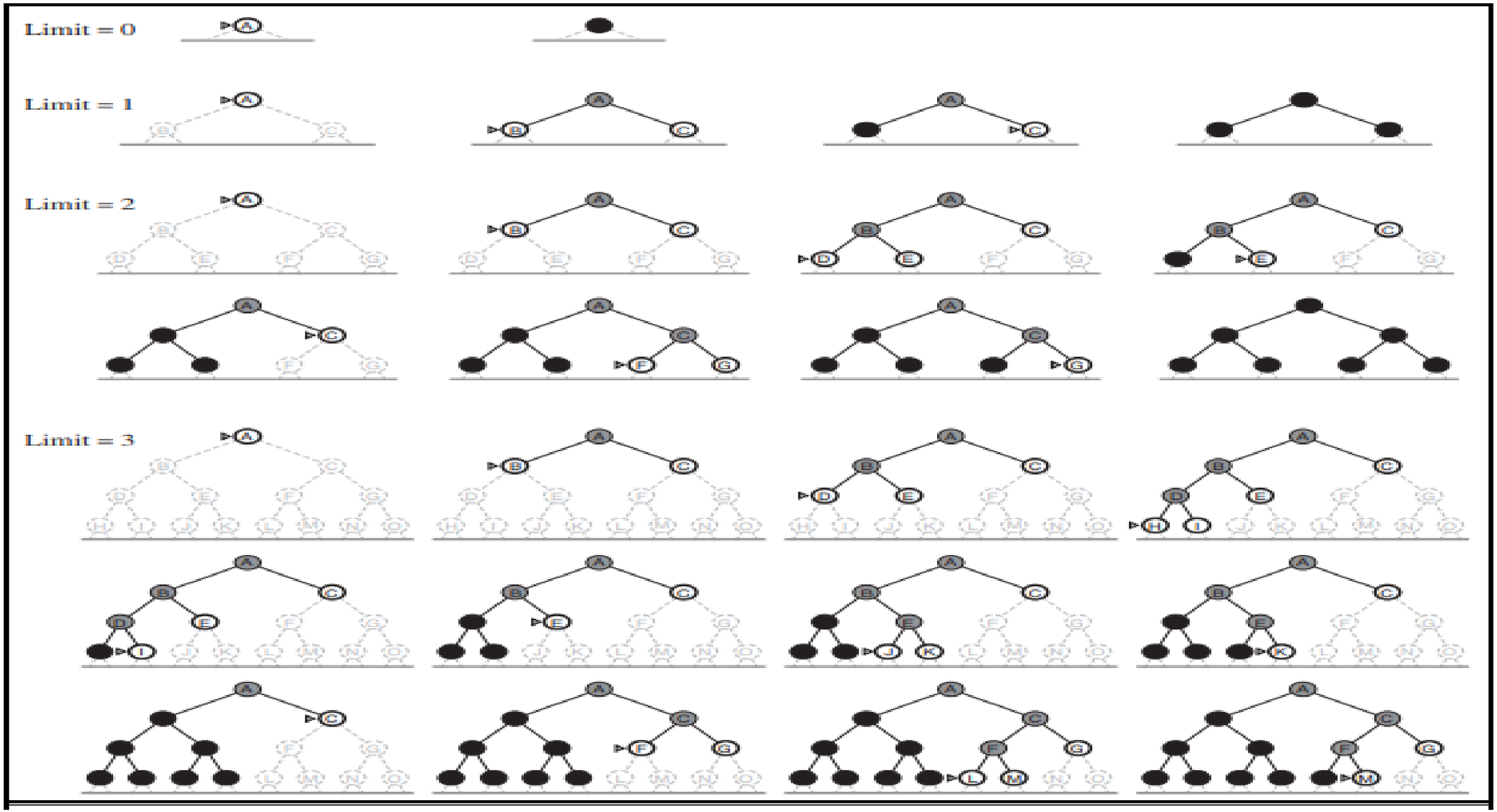
Performance measure of DLS

- The DLS solves the infinite path problem, but it introduces many other problems.
- Completeness : it is incomplete if $l < d$.
- Optimality : it is non-optimality if $l < d$, because not guaranteed to find the shortest solution in the search technique.
- Time and Space complexity:
- Time : $O(b^l)$
- Space : $O(bl)$

Iterative deepening depth-first search

- **Iterative deepening search (or iterative deepening depth-first search) is a general strategy**
- **It is often used in combination with depth-first tree search, that finds the best depth limit**
- **It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.**
- **This will occur when the depth limit reaches d, the depth of the shallowest goal node.**

```
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
```

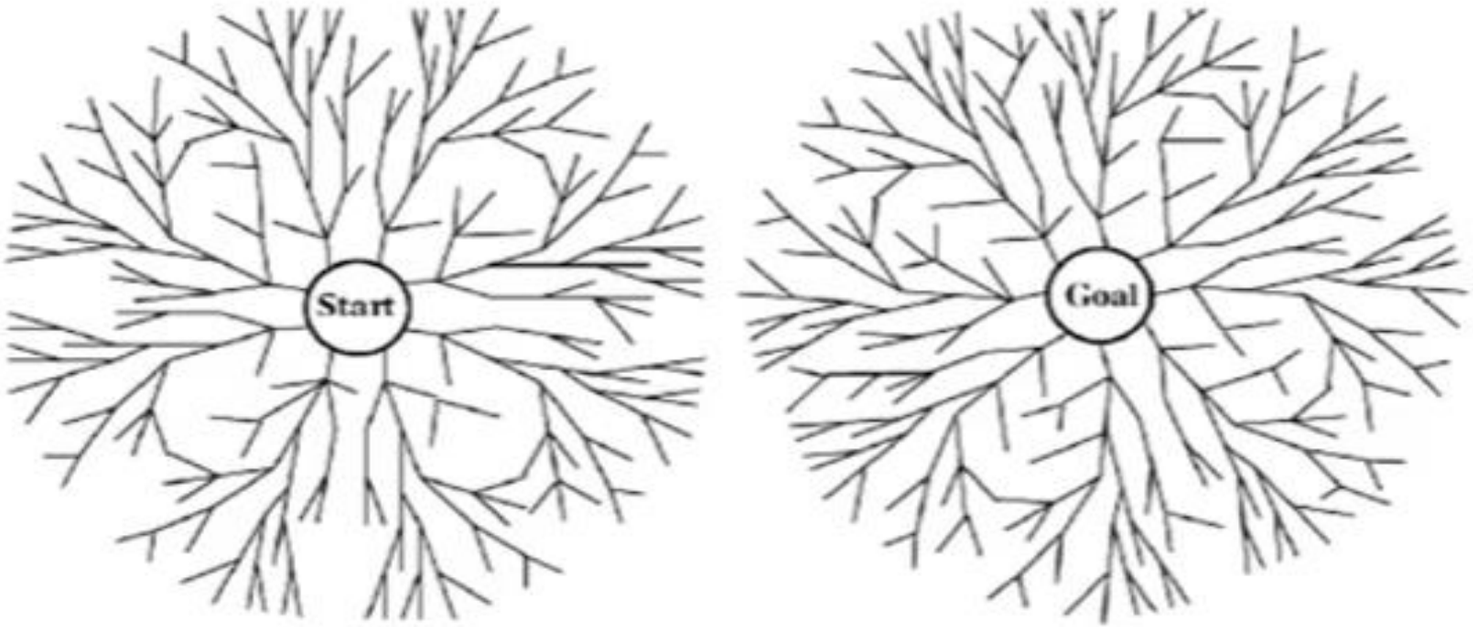

Limit = 0
Limit = 1
Limit = 2
Limit = 3

- **Benefits:**
- Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too cost
- if b = 10 and d = 5, the numbers are

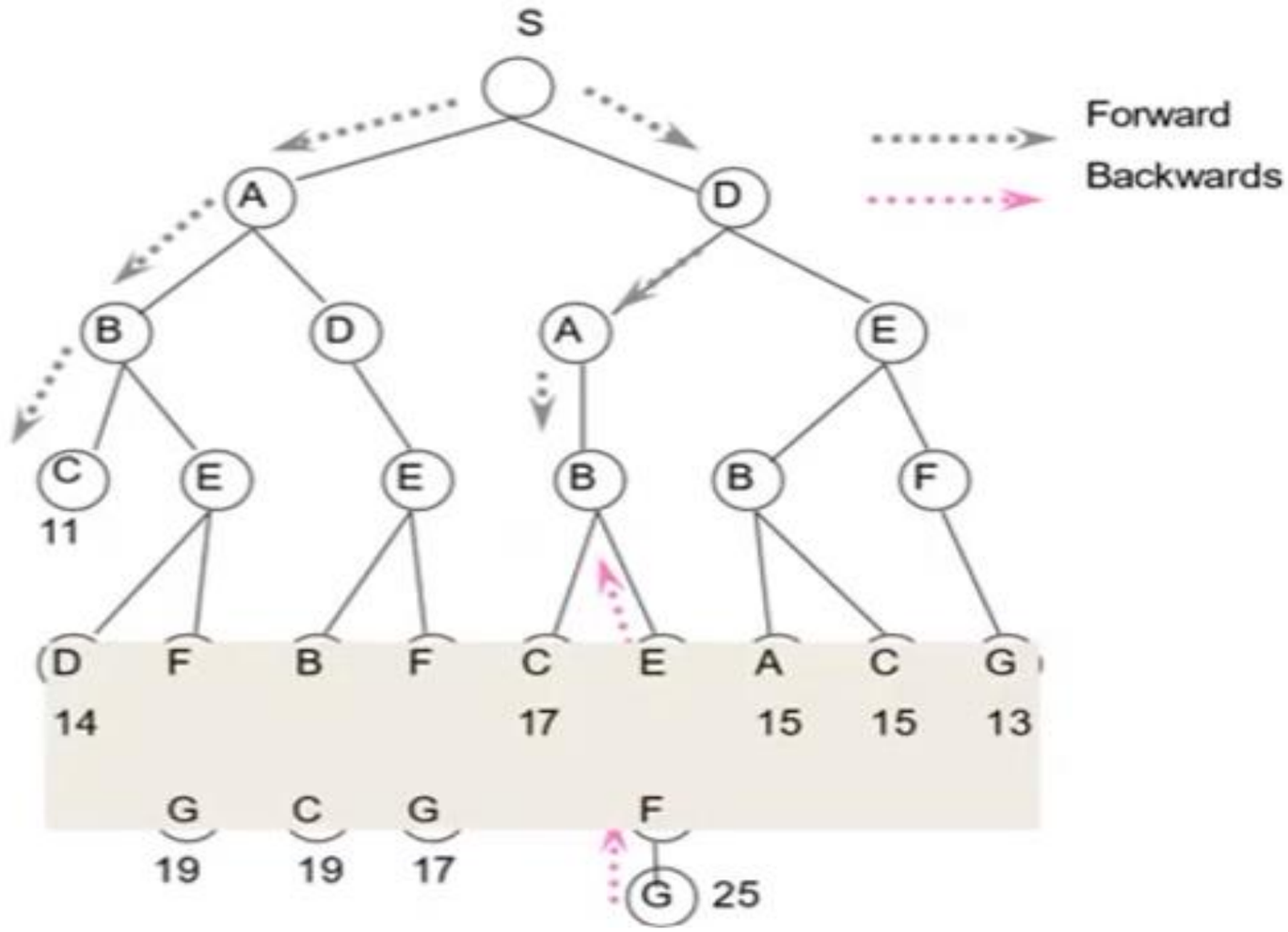
$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 .$$

Bi-directional Search

- Bi-directional searching, there are two searching, one from the start state toward the goal and another from the goal state toward the start.
- This algorithm Stops when both search meet in the middle.
- Can (sometimes) lead to finding a solution more quickly.



Bi-directional Search...



Bi-directional Search...

- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.
- Motivation: $b^{d/2} + b^{d/2}$ is much less than b^d
- Can be implemented using BFS or iterative deepening (but at least one frontier needs to be kept in memory)
- Works well only when there are unique start and goal states.

Bi-directional Search...

- Complete : Yes, (if only single goal, otherwise difficult)
- Time complexity : $O(b^{d/2})$ (better when compared to other algorithms).
- Space complexity: $O(b^{d/2})$ (Space requirement is more).
- Optimal : Yes, but not always optimal, even if both searches are BFS
- Check when each node is expanded or selected for expansion

Comparison of Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Informed (Heuristic) Search Strategies

- Domain-specific hints about the location of goals
- Find solutions more efficiently than an uninformed strategy
- The hints come in the form of a heuristic function, denoted h(n)
- Where h(n):= estimated cost of the cheapest path from the state at node n to a goal state.

Greedy Best-First Search

Greedy best-first search

- The evaluation function is f(n) = h(n)
- Were, h(n)= estimated cost from node n to the goal.

- Greedy search ignores the cost of the path that has already been traversed to reach n
- Therefore, the solution given is not necessarily optimal

- Greedy best-first search is a form of best-first search that expands first the node with the Greedy best-first search lowest h(n) value
- The node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly
- Straight-line distance h_{SLD}
- Evaluation function, f(n)=The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first.
- heuristic functionh(n) = estimated cost of the cheapest path from the state at node n to a goal state.

Search Strategies-Best-First Search

Best-First Search in Artificial Intelligence

- Best First Search algorithm combines the advantages of both DFS and BFS into a single method.
- At each step of the BFS search process, we select the most promising of the nodes we have generated so far.
- This is done by applying an appropriate heuristic function to each of them.
- We then expand the chosen node by using the rules to generate its successors

Best-First Search

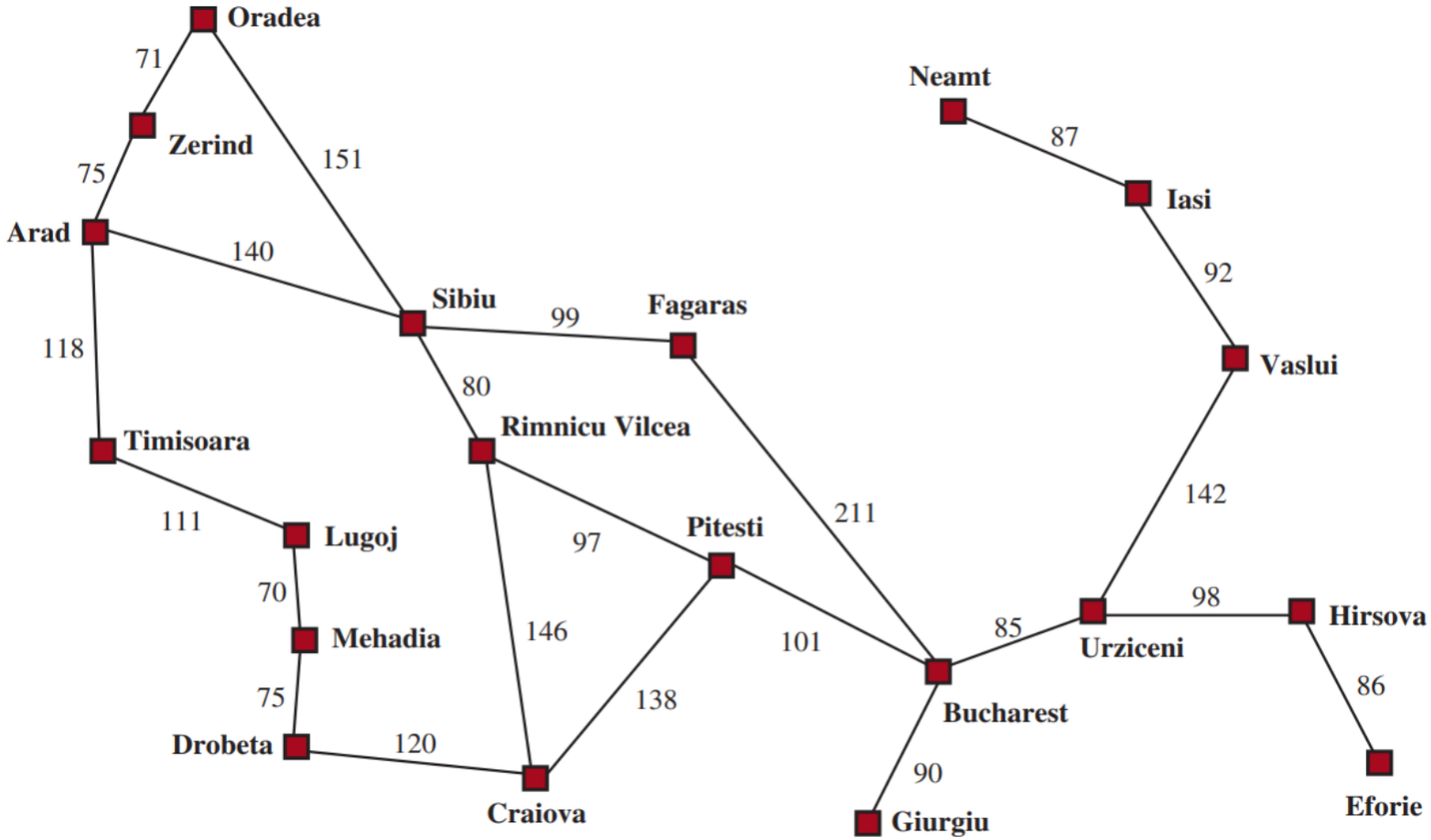
Best-First Search in Artificial Intelligence

- To implement such a graph-search procedure, we will need to use two lists of nodes:
  - **OPEN** — nodes that have been generated and have had the heuristic function applied to them, but which have not yet been examined (i.e., had their successors generated).
  - **CLOSED** — nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been

Algorithm: Best-First Search

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left on OPEN do:
   a) Pick them best node on OPEN.
   b) Generate its successors.
   c) For each successor do:
      i. if it has not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

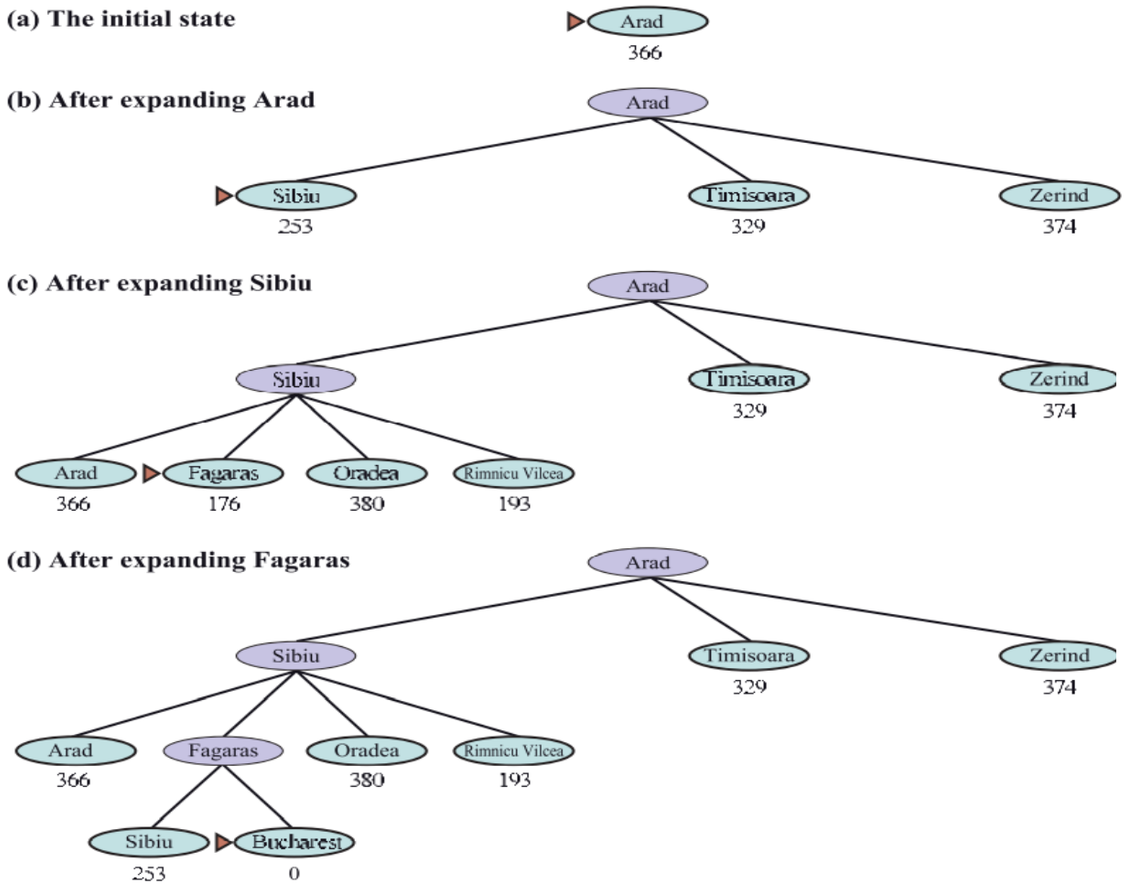
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



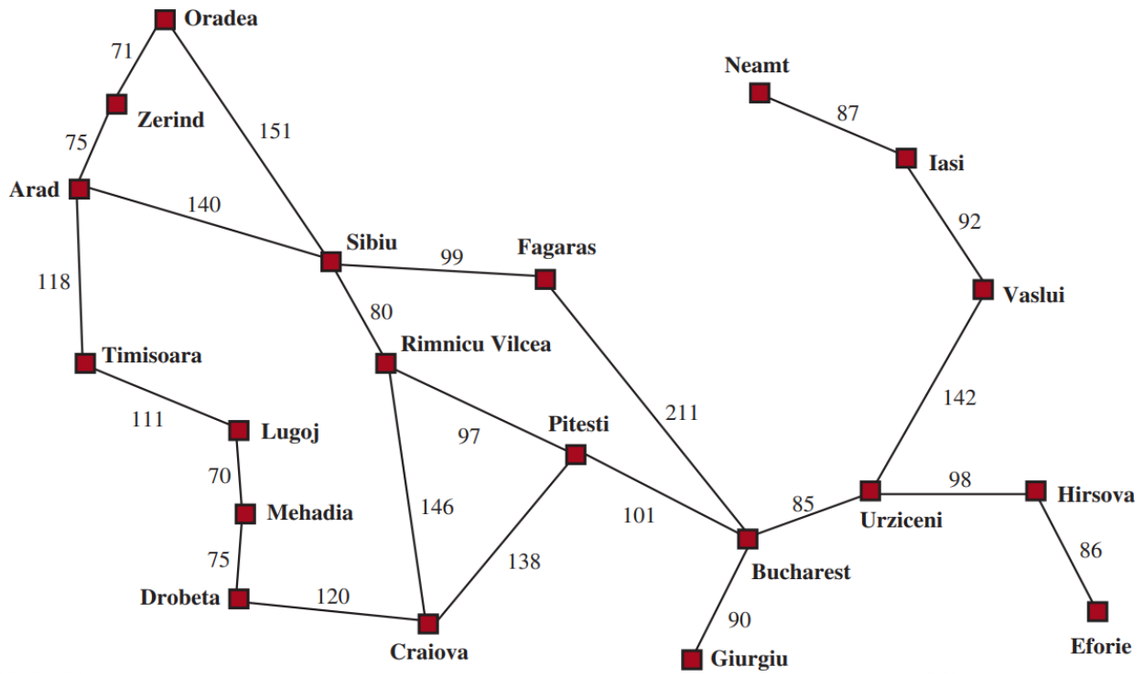
Greedy best-first search

- f(n) = estimate of cost from *n* to *goal*
- e.g., $f_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal.

Greedy Best-First Search



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



Greedy Best First Search

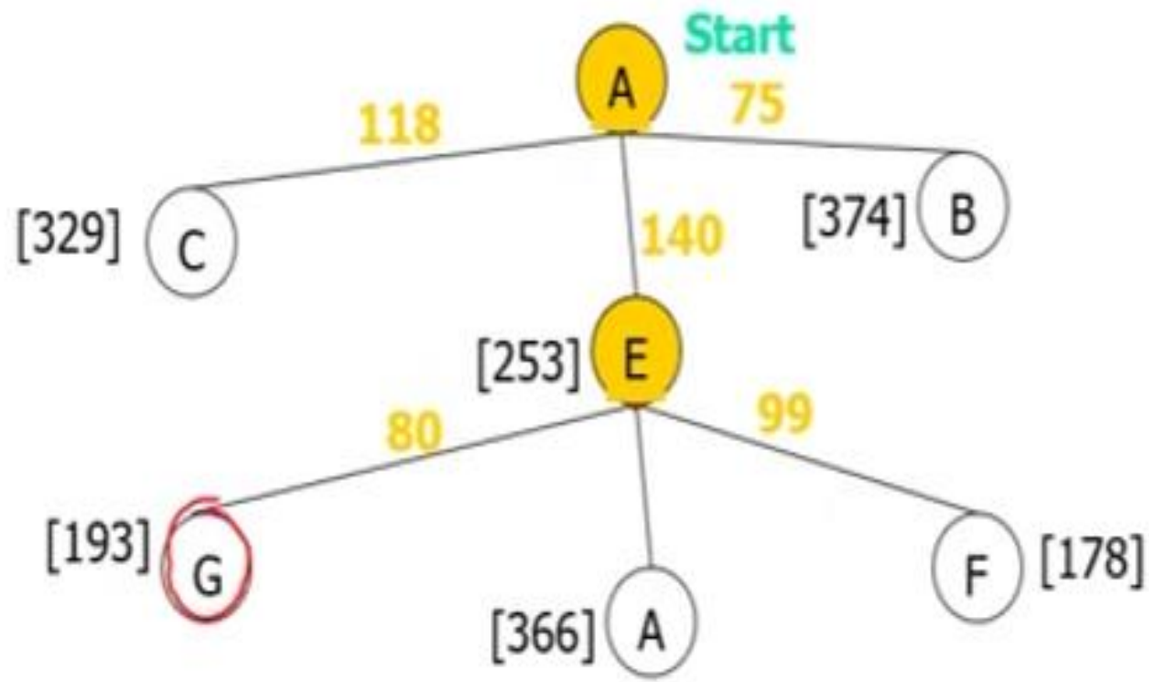
Properties of greedy best-first search

- Complete? No – can get stuck in loops.
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ - keeps all nodes in memory
- Optimal? No

  e.g. Arad → Sibiu → Rimnicu
  Virea → Pitesti → Bucharest is shorter!

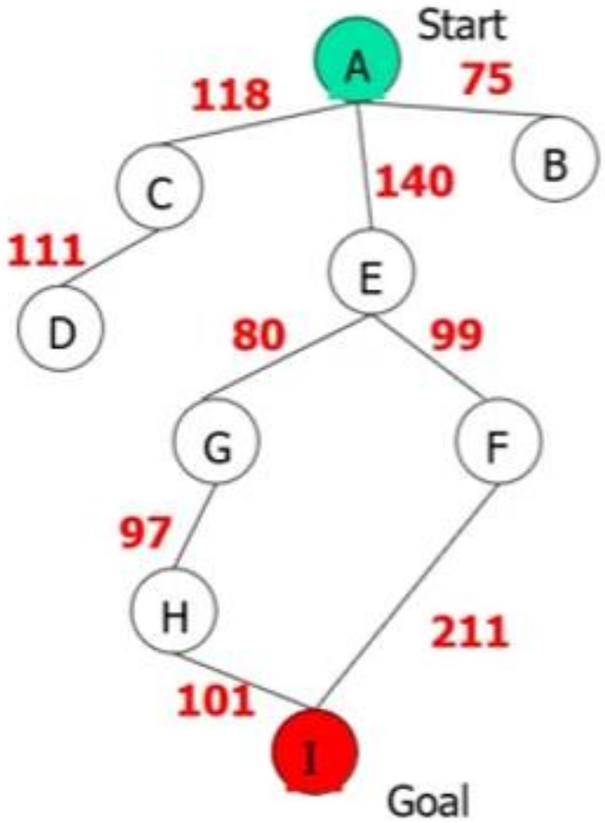
where m is the maximum depth of the search space

Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Best First Search



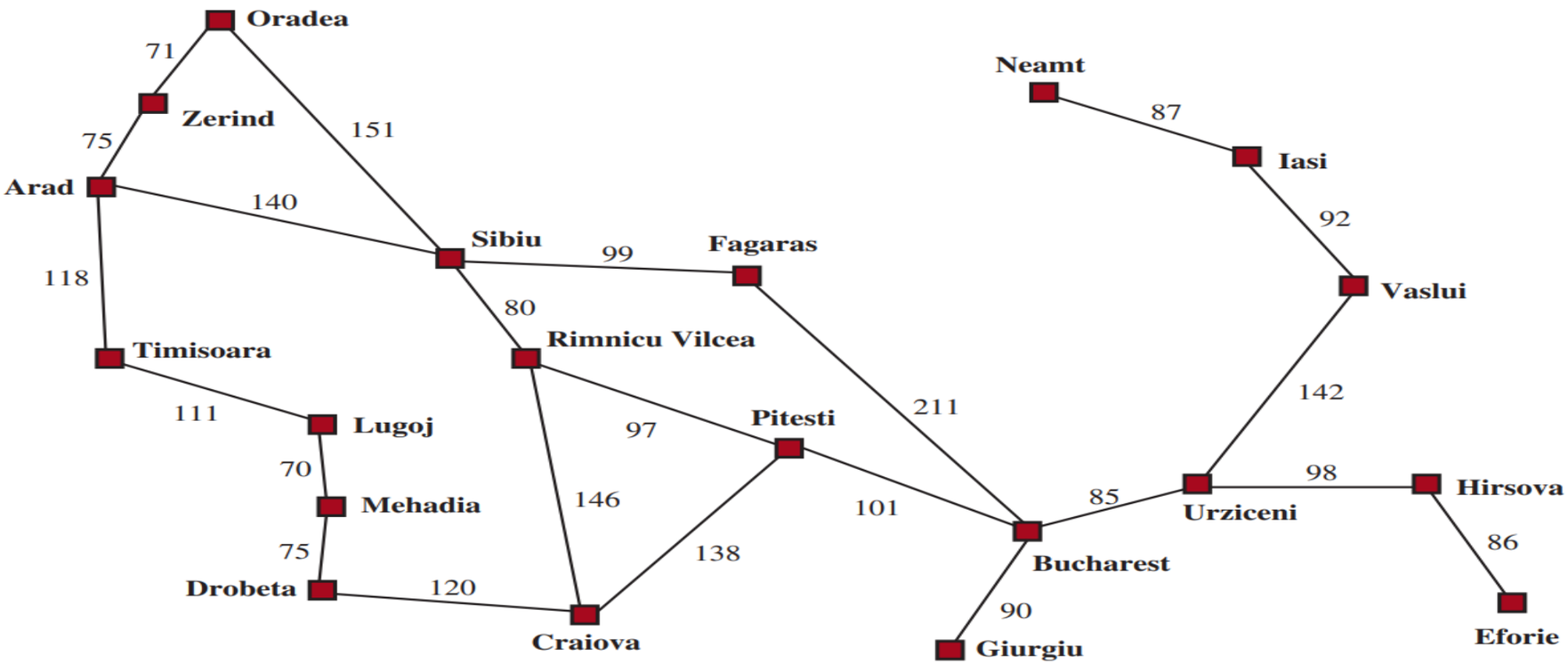
State	Heuristic: h(n)
A	366
B	374
** C	250
D	244
E	253
F	178
G	193
H	98
I	0

f(n) = h (n) = straight-line distance heuristic

A* Search

- Greedy Best First Search minimizes a heuristic h(n) which is an estimated cost from a current state n to the goal state.
- Greedy Best First Search is efficient but it is not optimal and not complete.
- Uniform Cost Search minimizes the cost g(n) from the initial state to current state n.
- Uniform Cost Search is optimal and complete but not efficient.
- **A* Search**: Combine Greedy Best First Search and Uniform Cost Search to get an efficient algorithm which is complete and optimal.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

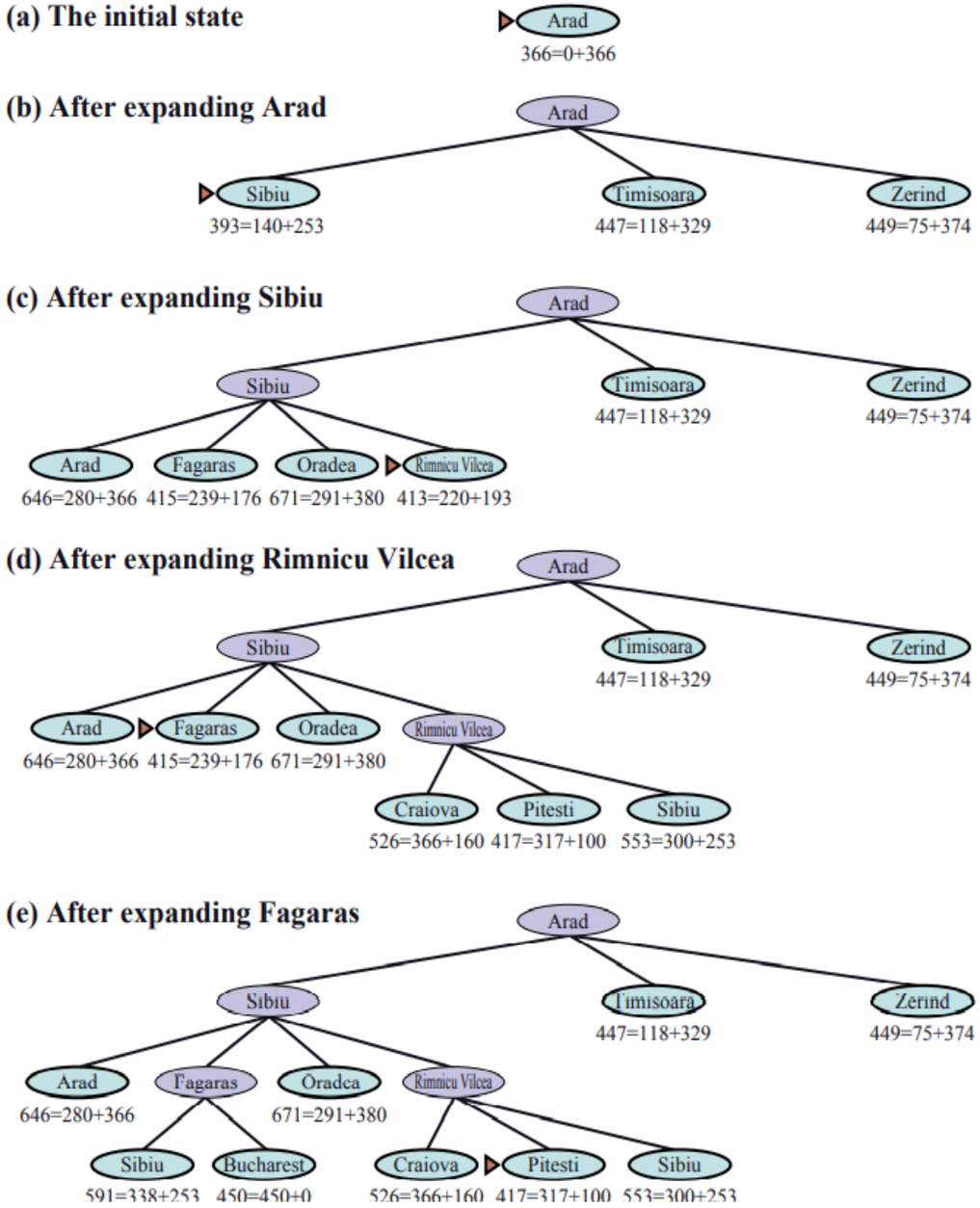


A* Search Algorithm

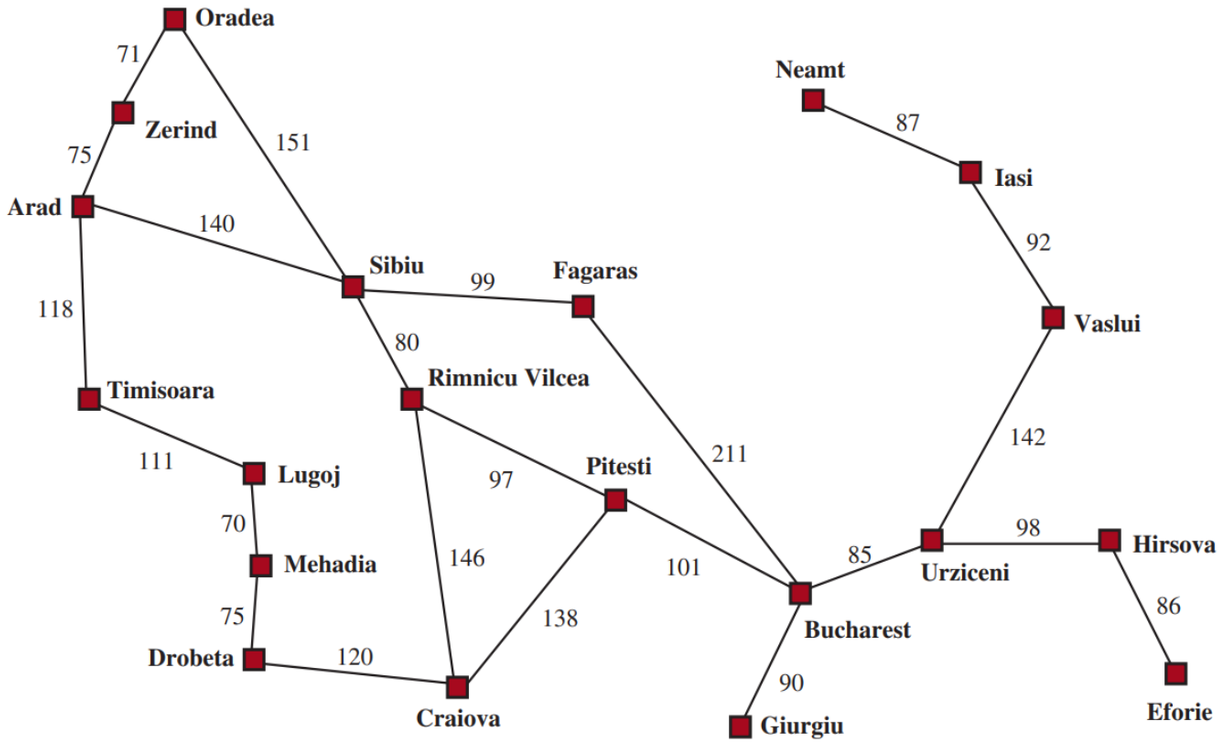
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal
- Best First search has $f(n)=h(n)$

Conditions for optimality: Admissibility and consistency

- The first condition requires for optimality is that h(n) be an **admissible heuristic.**
- **Admissible heuristic** is one that never overestimates the cost to reach the goal
- g(n) is the actual cost to reach n along the current path
- f(n) = g(n) + h(n)
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



A* Search Algorithm

- Complete? Yes (unless there are infinitely many nodes with f $\leq f(G)$, i.e. path-cost > ε)
- Time/Space? Exponential b^d

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$
- Optimal? Yes
- *Optimally Efficient*: Yes (no algorithm with the same heuristic is guaranteed to expand fewer nodes)

ADVERSARIAL SEARCH AND GAMES

- Multi agent environments
- Contingencies(Circumstances)
- Impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive
- Competitive environments : Goals are in conflict=Adversarial Search
  - Ex: Games
- Two or more agents have conflicting goals, giving rise to adversarial search problems.
- **Zero-sum games of perfect information**: **Deterministic, fully observable environments** in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
- Utility functions that makes the situation adversarial.
- state of a game is **easy to represent**, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules

Zero-sum Games

- Adversarial: Pure competition.
- Agents have different values on the outcomes.
- One agent maximizes one single value, while the other minimizes it.

Embedded thinking...

Embedded thinking or backward reasoning!

- One agent is trying to figure out what to do.
- How to decide? He thinks about the consequences of the possible actions.
- He needs to think about his opponent as well...
- The opponent is also thinking about what to do etc.
- Each will imagine what would be the response from the opponent to their actions.
- This entails an embedded thinking.

- Choosing a good move**
- **Pruning :** Ignoring portions of the search tree that make no difference to the final choice
- **Heuristic evaluation functions** allow us to approximate the true utility of a state without doing a complete search.
- **Imperfect information**:
  - Ex playing cards

Games with two players= call MAX and MIN

(points are awarded to the winning player and penalties are given to the loser)

Terminology:

S_0: The **initial state**, which specifies how the game is set up at the start.

PLAYER(s): Defines which player has the move in a state.

ACTIONS(s): Returns the set of legal moves in a state.

RESULT(s, a): The **transition model**, which defines the result of a move.

TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.

UTILITY(s, p): A utility function (also called an **objective function or payoff function**),

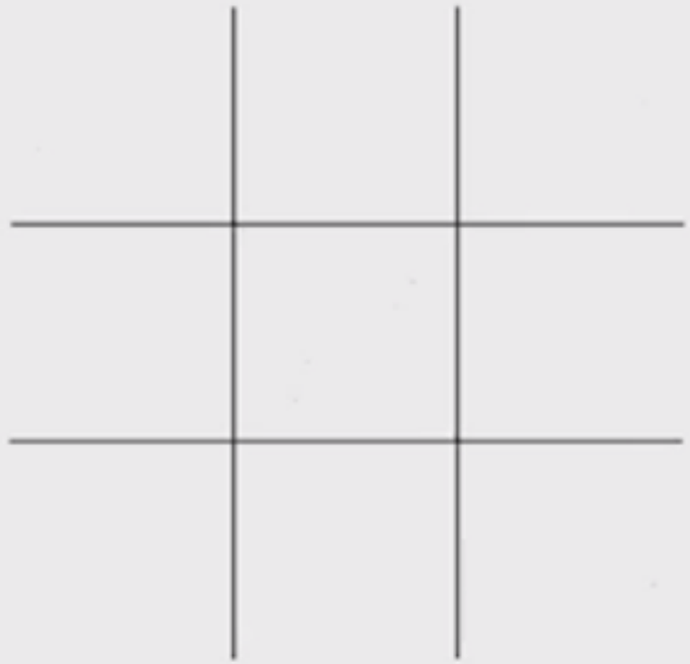
defines the final numeric value for a game that ends in terminal states for a player p**.**

Formalization

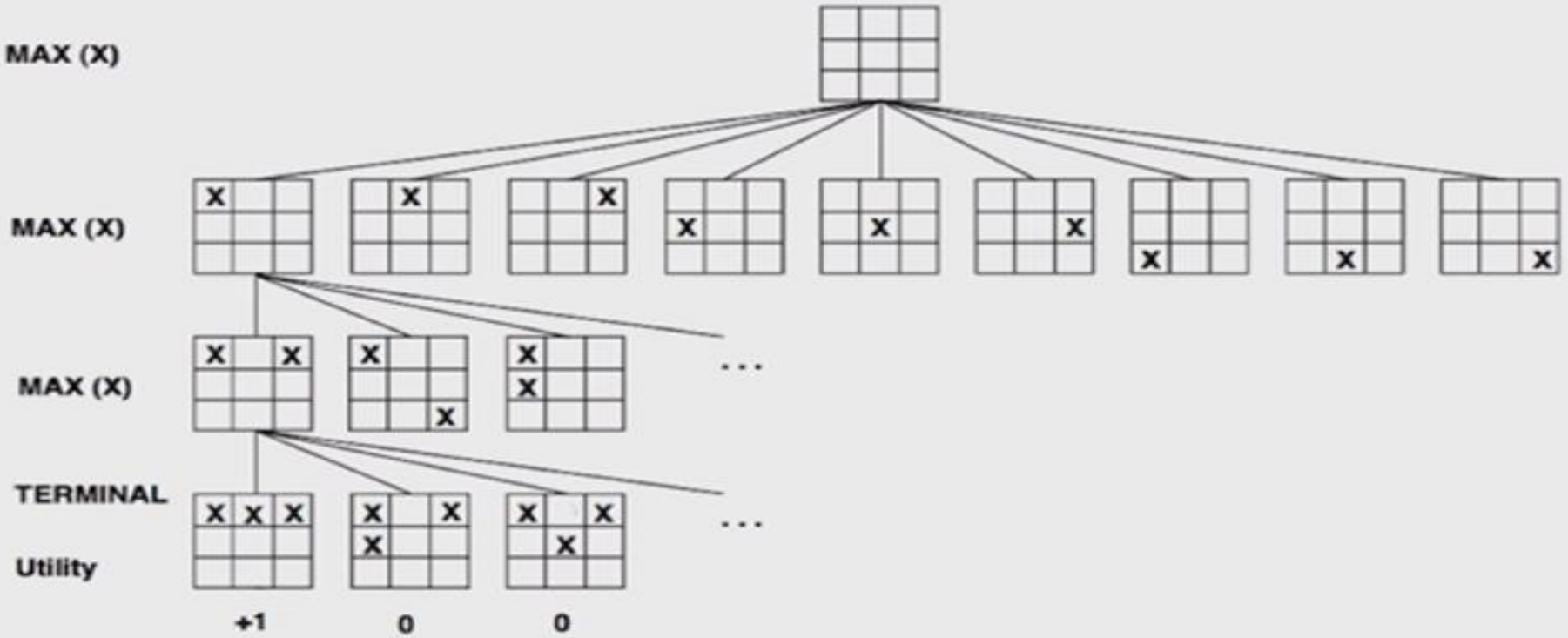
- The **initial state**
- Player(s): defines which player has the move in state s. Usually taking turns.
- Actions(s): returns the set of legal moves in s
- **Transition** function: $S \times A \rightarrow S$ defines the result of a move
- Terminal_test: True when the game is over, False otherwise. States where game ends are called **terminal states**
- $Utility(s, p)$: **utility function** or objective function for a game that ends in terminal state s for player p. In Chess, the outcome is a win, loss, or draw with values +1, 0, 1/2. For tic-tac-toe we can use a utility of +1, -1, 0.

Single player...

Assume we have a tic-tac-toe with one player.
Let's call him Max and have him play three moves only for the sake of the example.



Single player...

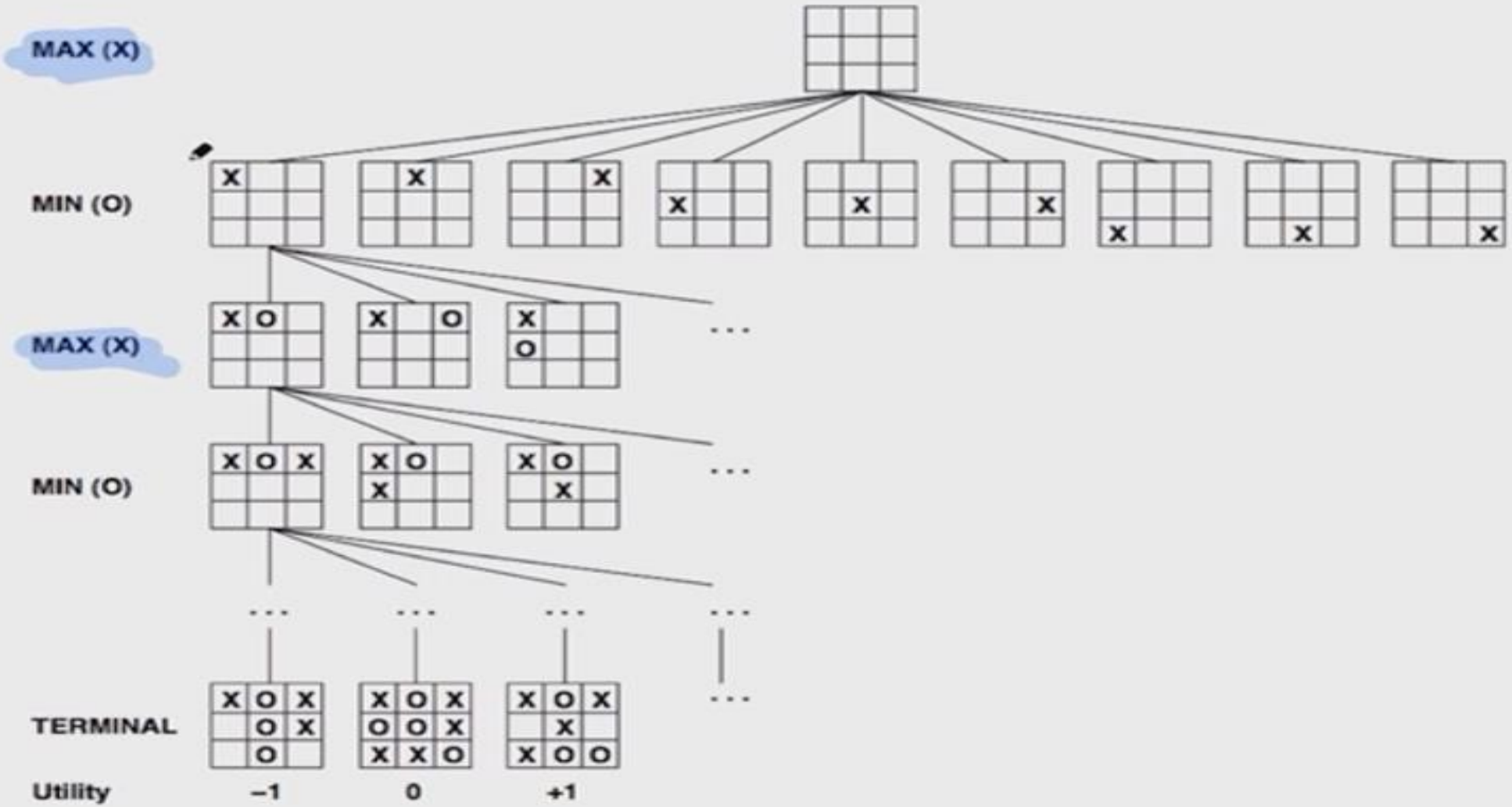


In the case of one player, nothing will prevent Max from winning (choose the path that leads to the desired utility here 1), unless there is another player who will do everything to make Max lose, let's call him Min (the Mean :))

Adversarial search: minimax

- Two players: Max and Min
- Players alternate turns
- Max moves first
- Max maximizes results
- Min minimizes the result
- Compute each node's minimax value's the best achievable utility against an optimal adversary
- Minimax value $\equiv$ best achievable payoff against best play

Minimax example



Adversarial search: minimax

- Find the optimal strategy for Max:
  - Depth-first search of the game tree
  - An optimal leaf node could appear at any depth of the tree
  - Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
  - Propagate minimax values up the tree once terminal nodes are discovered

Adversarial search: minimax

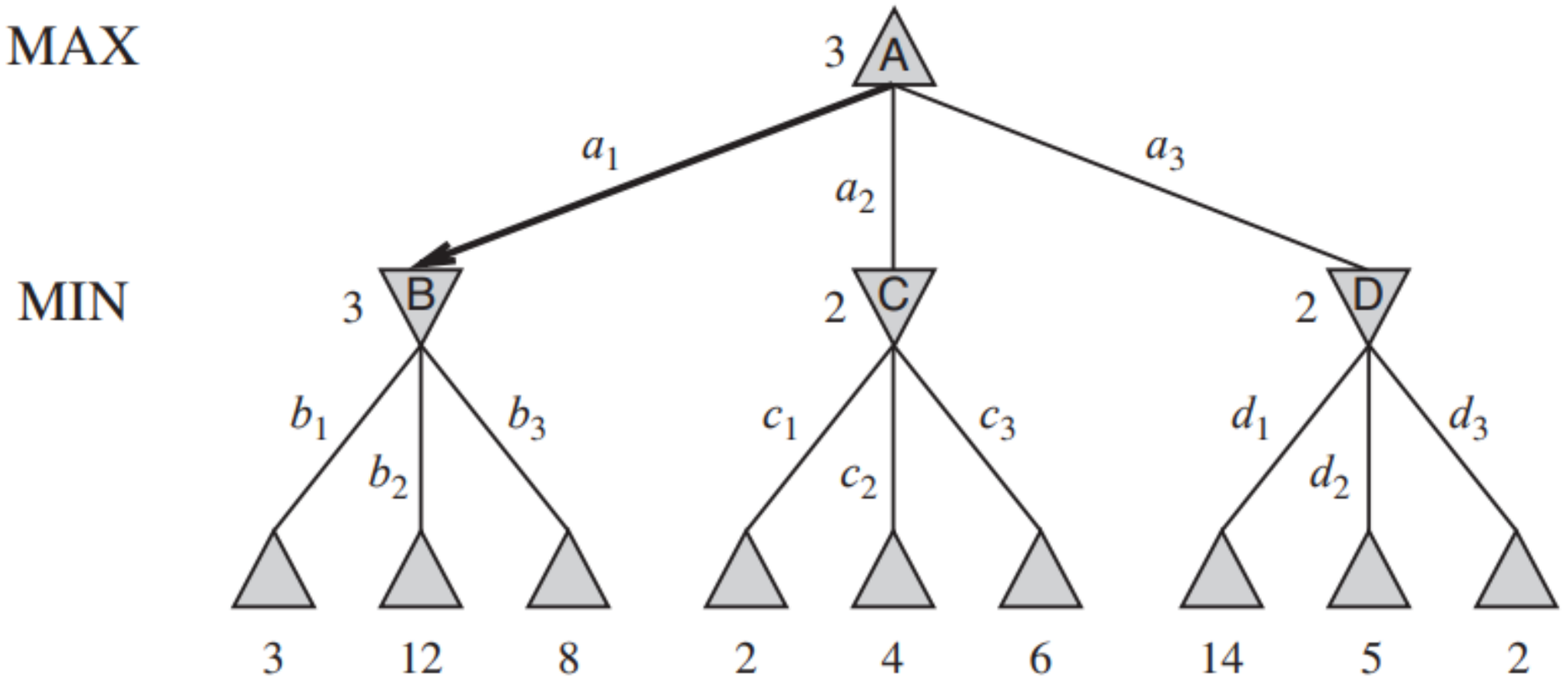
For a state s minimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Min} \end{cases}$$

Adversarial search: minimax

- If state is terminal node: Value is utility(state)
- If state is MAX node: Value is highest value of all successor node values (children)
- If state is MIN node: Value is lowest value of all successor node values (children)

- GAME TREE:
- A tree where the nodes are game states and the edges are moves
- A two-ply game tree. The △ nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ▽ nodes are “MIN nodes



The minimax algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨ _, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨ _, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

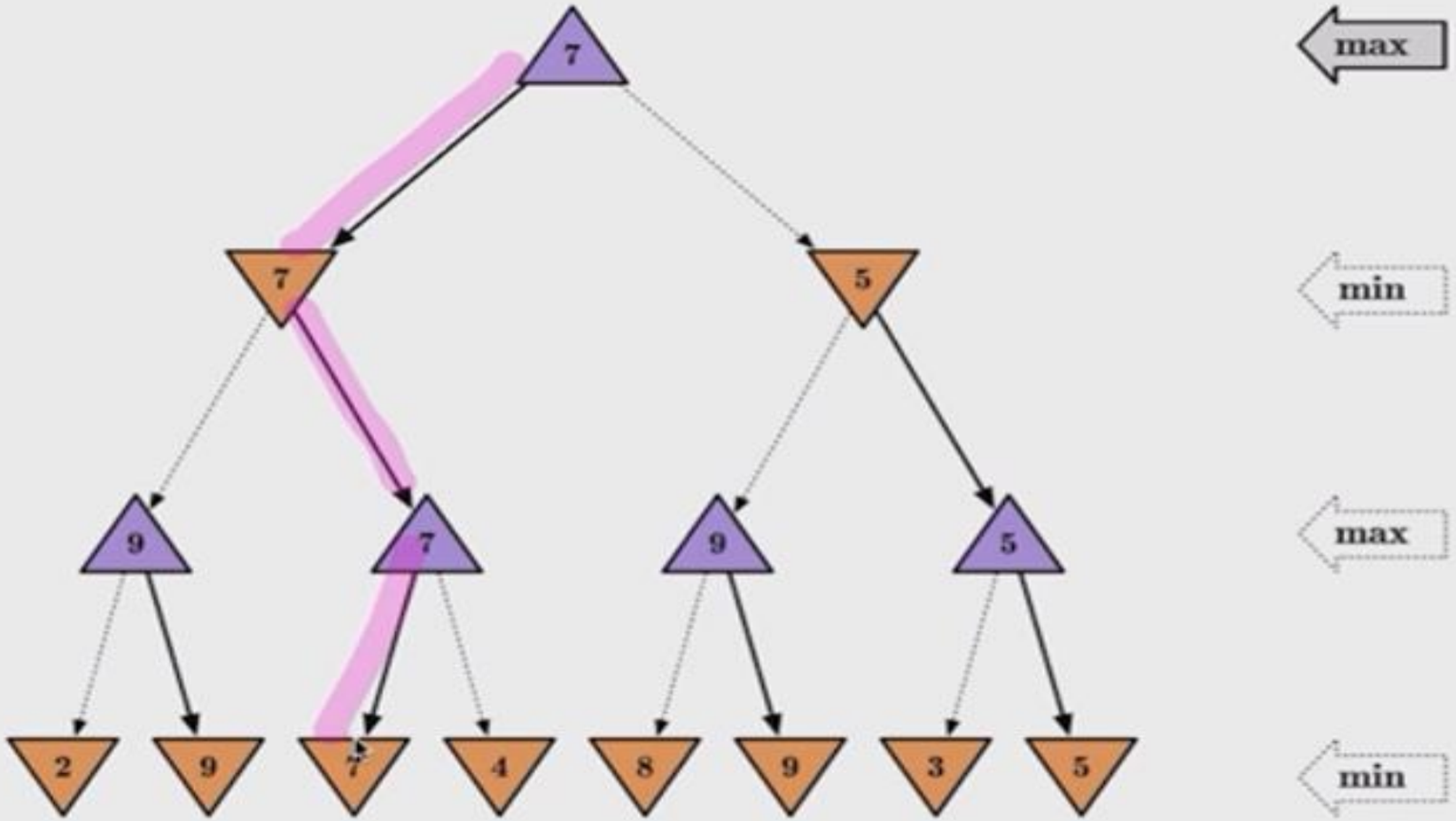
The minimax algorithm

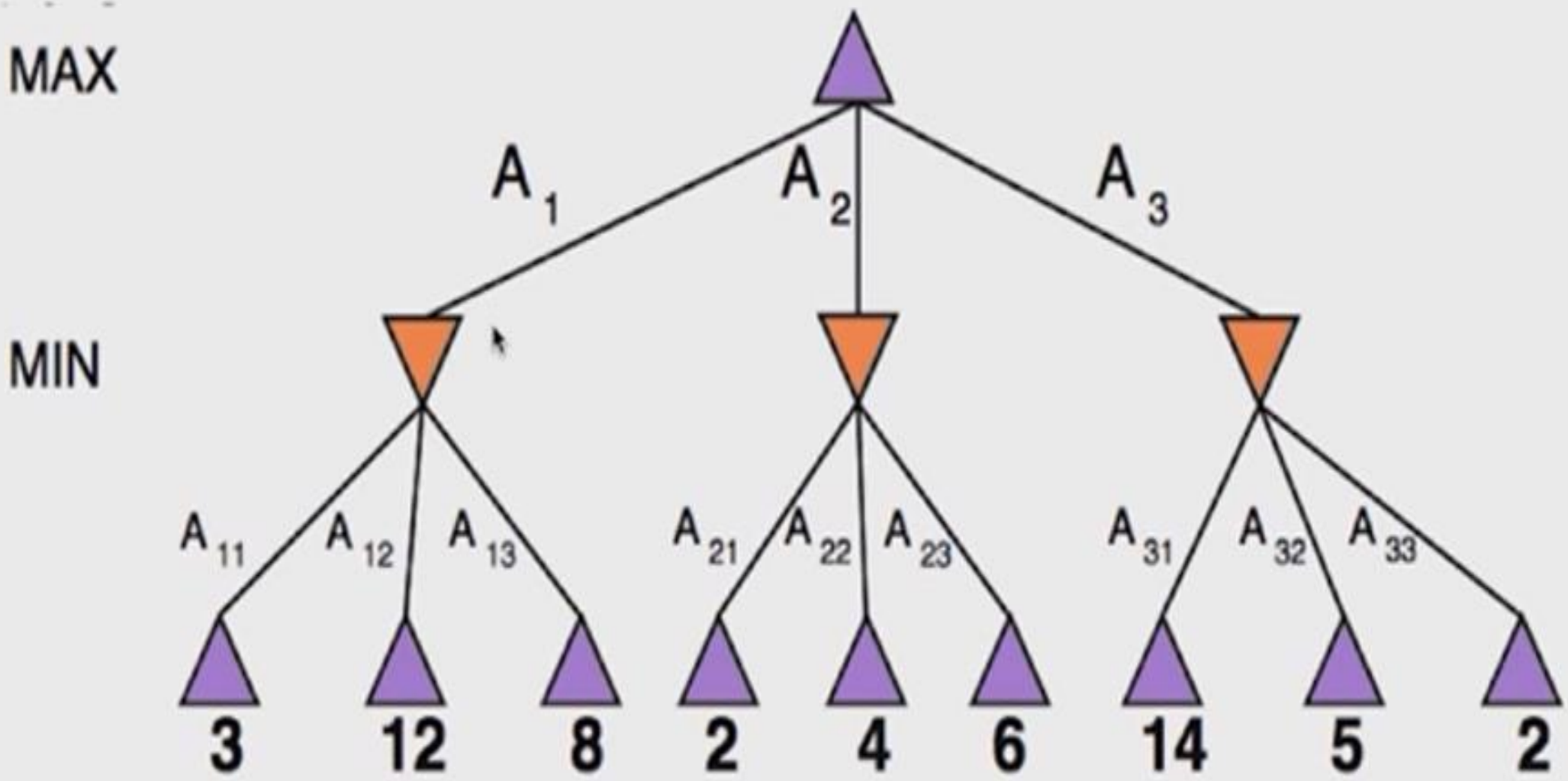
```
/* Find the child state with the lowest utility value */
function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */
function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */
function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

Minimax example





A two-ply game tree.

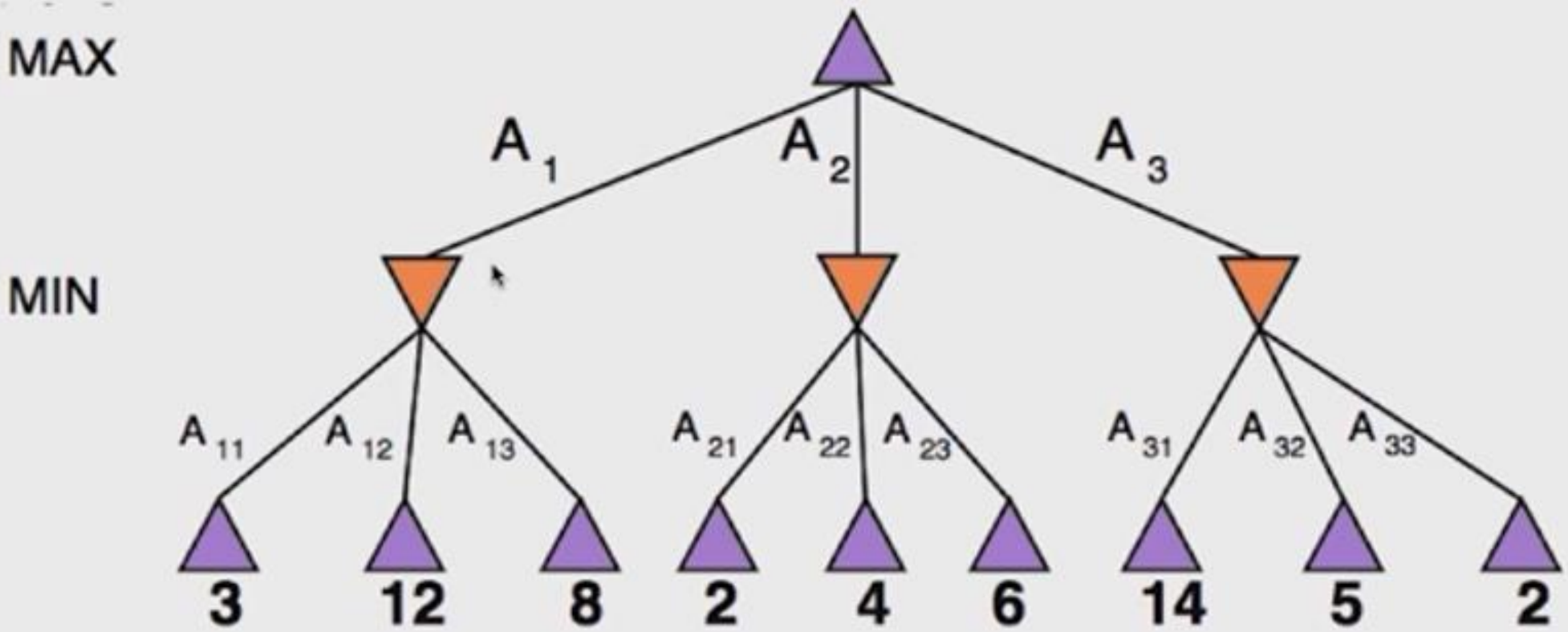
Properties of minimax

- Optimal (opponent plays optimally) and complete (finite tree)
- DFS time: $O(b^m)$
- DFS space: $O(bm)$
  - **Tic-Tac-Toe**
    - ≈ 5 legal moves on average, total of 9 moves (9 plies).
    - $5^9 = 1,953,125$
    - $9! = 362,880$ terminal nodes
  - **Chess**
    - $b \approx 35$ (average branching factor)
    - $d \approx 100$ (depth of game tree for a typical game)
    - $b^d \approx 35^{100} \approx 10^{154}$ nodes
  - **Go** branching factor starts at 361 (19X19 board)

Case of limited resources

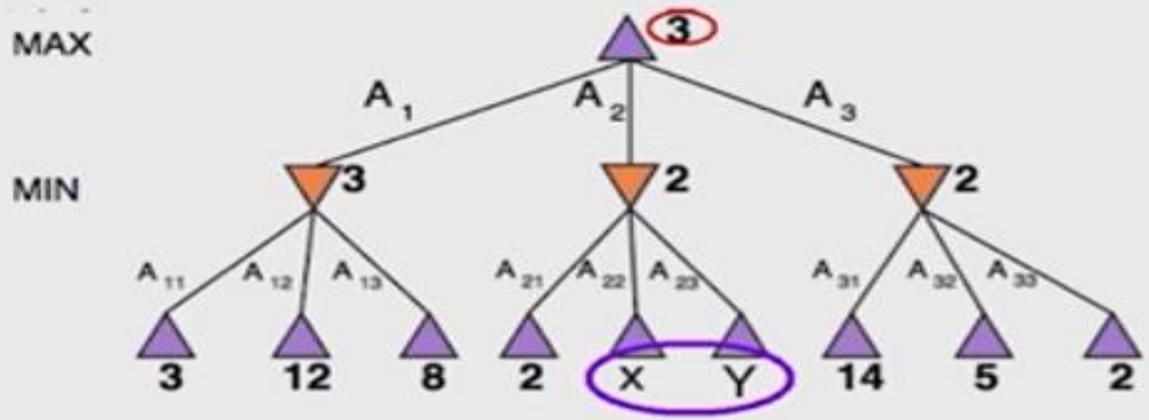
- **Problem:** In real games, we are limited in time, so we can't search the leaves.
- To be practical and run in a reasonable amount of time, minimax can only search to some depth.
- More plies make a big difference.
- **Solution:**
  1. Replace terminal utilities with an evaluation function for non-terminal positions.
  2. Use Iterative Deepening Search (IDS).
  3. Use pruning: eliminate large parts of the tree.

$\alpha - \beta$ pruning



A two-ply game tree.

$\alpha - \beta$ pruning



$$Minimax(root) = max(min(3, 12, 8), min(2, X, Y), min(14, 5, 2))$$

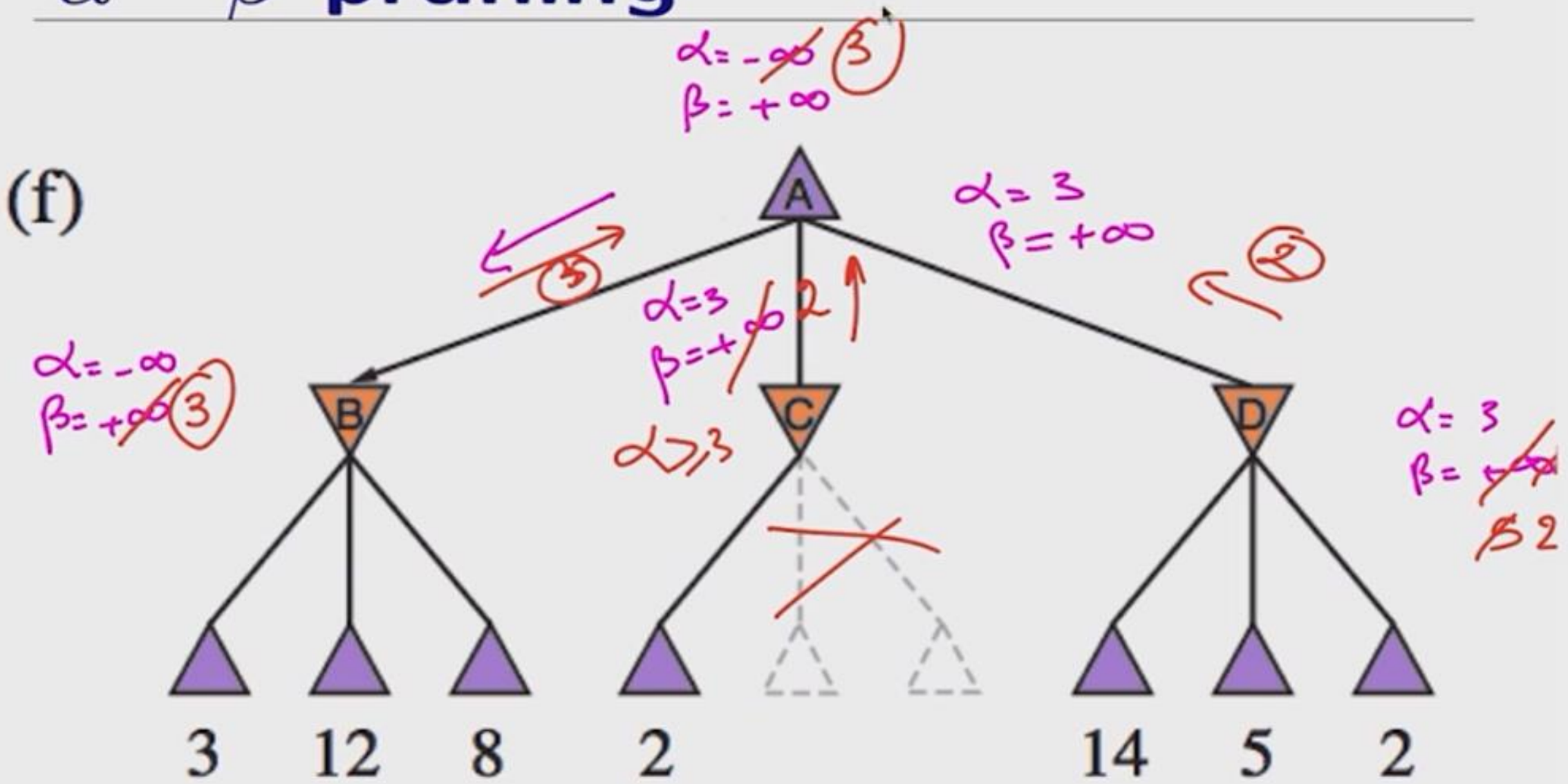
$$= max(3, min(2, X, Y), 2)$$

$$= max(3, z, 2) \qquad \text{where} \quad z = min(2, X, Y) \leq 2$$

$$= 3$$

Minimax decisions are independent of the values of X and Y.

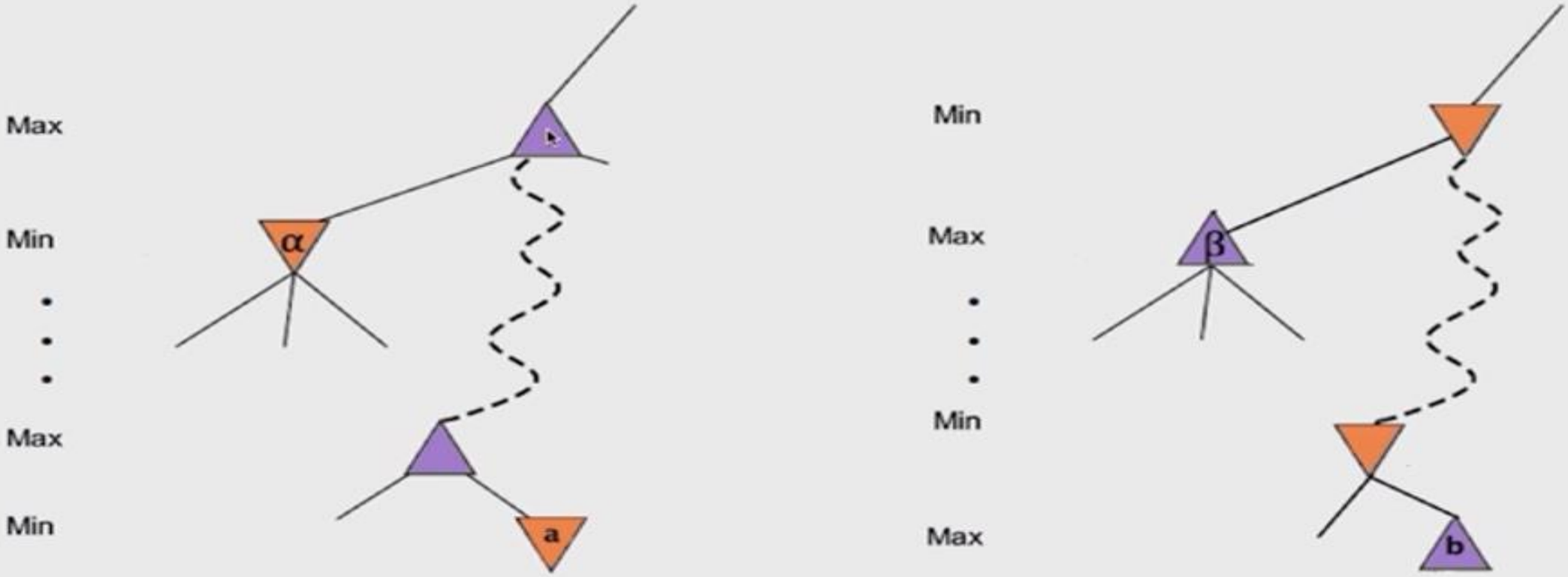
$\alpha - \beta$ pruning



$\alpha - \beta$ pruning

- **Strategy**: Just like minimax, it performs a DFS.
- **Parameters**: Keep track of two bounds
  - α: largest value for Max across seen children (current lower bound on MAX's outcome).
  - β: lowest value for MIN across seen children (current upper bound on MIN's outcome).
- **Initialization**: $\alpha = -\infty$, $\beta = \infty$
- **Propagation**: Send α, β values *down* during the search to be used for pruning.
  - Update α, β values by *propagating upwards* values of terminal nodes.
  - Update α only at Max nodes and update β only at Min nodes.

$\alpha - \beta$ pruning



- If α is better than a for Max, then Max will avoid it, that is prune that branch.

- If β is better than b for Min, then Min will avoid it, that is prune that branch.

Games: conclusion

- Games are modeled in AI as a search problem and use heuristic to evaluate the game.
- Minimax algorithm choses the best most given an optimal play from the opponent.
- Minimax goes all the way down the tree which is not practical give game time constraints.
- Alpha-Beta pruning can reduce the game tree search which allow to go deeper in the tree within the time constraints.
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice.

Constrained Satisfaction Problem

- **Factored representation**: a set of **variables**, each of which has a **value**
- A problem is solved when each **variable has a value** that satisfies all the ~ idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints
- **CSP Components:**
  - X is a set of variables, $\{X_1,...,X_n\}$.
  - D is a set of domains, $\{D_1,...,D_n\}$, one for each variable.
    - Where Di = {v1,...,vk}, for variable Xi, Different variables can have different domains of different sizes.
  - C is a set of constraints that specify allowable combinations of values.
    - Each constraint Cj consists of a pair < **scope, rel** >

    Scope:=**<(2,4),(2,5),(2,8),(3,4)(3,8)(5,8)>**
  - D1={2,3,5}, D2={4,5,8} <

Scope is a tuple of variables that participate in the **constraint**

rel is a relation that defines the values that those variables can take on.

A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.

Ex: X1 and X2 are the variables

D1= {1,2,3},

D2={4,1,3},

Constraint saying that X1 must be greater than X2.

Rel=<(X1,X2),{(3,1),(3,2),(2,1)}> Or<(X1,X2), X1 > X2>

- CSPs deal with **assignments of values to variables**, , {Xi = vi ,Xj = vj ,...}.
- An assignment that does not violate any constraints is called a **consistent** or legal assignment.
- **A complete Consistent** assignment is one in which every variable is assigned a value, and a solution to a CSP is Complete assignment a consistent
- **A partial assignment** is one that leaves some variables unassigned,
- **A partial solution** is a partial assignment that is consistent.
- **Solving a CSP** is an NP-complete problem in general, although there are important subclasses of CSPs that can be solved very efficiently.
- **Linear Constraints**
- **Non-Linear Constraints**

CSPs definition

- A constraint satisfaction problem consists of **three elements**:
  - A set of **variables**, $X = \{X_1, X_2, \cdots X_n\}$
  - A set of **domains** for each variable: $D = \{D_1, D_2, \cdots D_n\}$
  - A set of **constraints** C that specify allowable combinations of values.
- Solving the CSP: **finding the assignment(s)** that **satisfy all constraints**.
- Concepts: problem formalization, backtracking search, arc consistency, etc.
- We call a solution, a **consistent assignment**.

Example: Map coloring

Variables: X ={WA, NT, Q, NSW, V , SA, T}

Domains: D_i = {red, green, blue}

Constraints: adjacent regions must have different colors;

e.g., WA $\neq$ NT or (W A, N T) $\in$ {(red, green), (red, blue), etc..}

Example: Map coloring

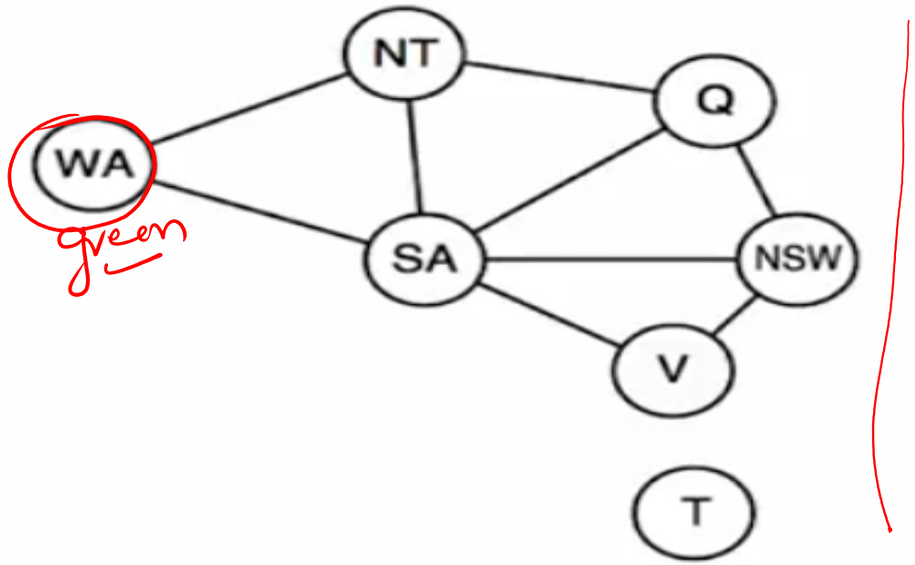
Example:

{WA=red, NT=green, Q=red, NSW=green,V=red, SA=blue,T=green}

Real-world CSPs

- Assignment problems, e.g., who teaches what class?
- Timetabling problems, e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floor planning
- Notice that many real-world problems involve real-valued variables

Constraint graph

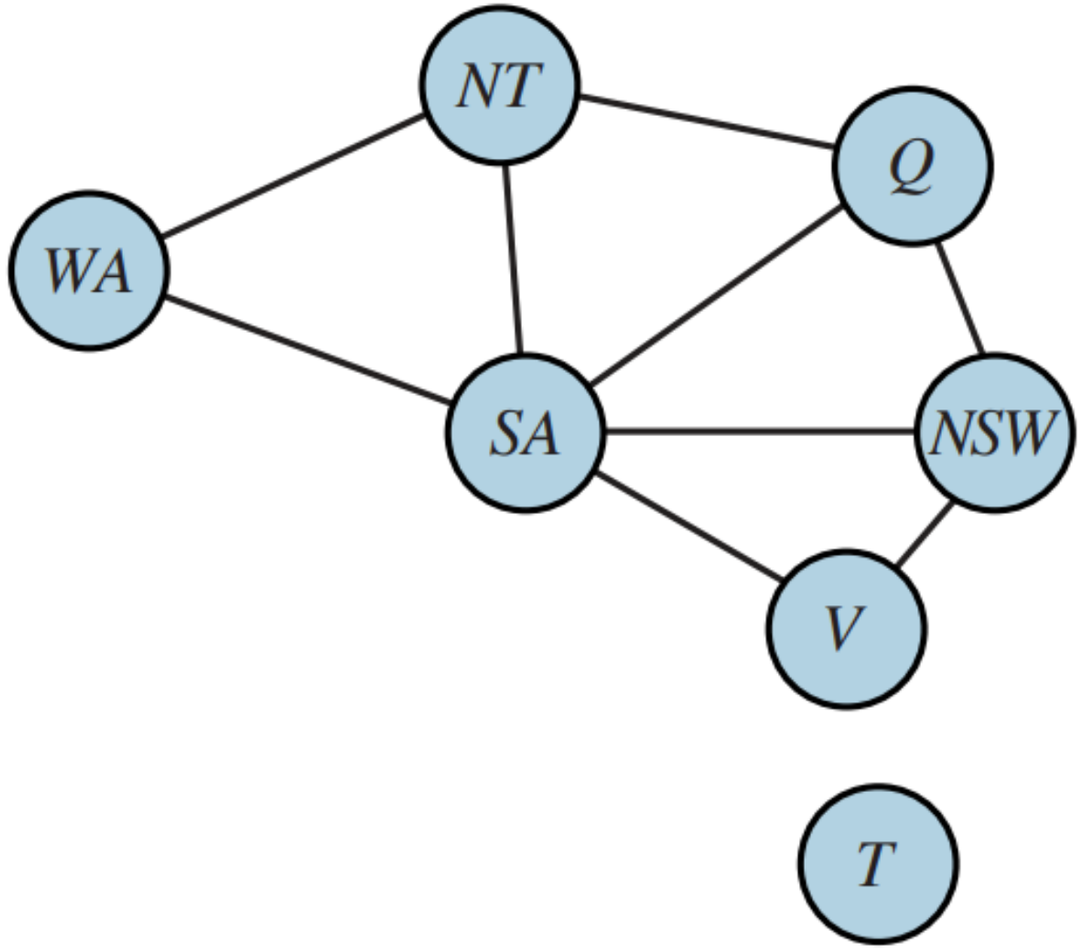


Binary CSP: each constraint relates at most two variables Constraint graph: nodes are variables, arcs show constraints

CSP algorithms: use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

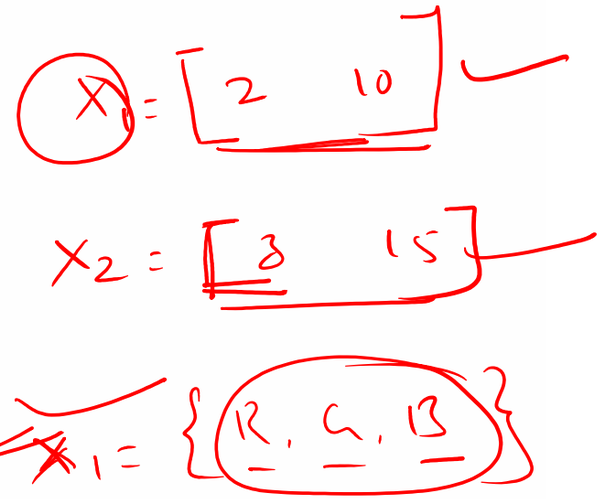
CSP: Example: Map Coloring

- Task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.
- Formulate this as a CSP:
- Variables=?
- Domains=?
- Constraints=?
- Possible solutions=?



Varieties of variables

- **Discrete variables**:
  - Finite domains:
    - assume n variables, d values, then the number of complete assignments is $O(d^n)$.
    - e.g., map coloring, 8-queens problem
  - Infinite domains (integers, strings, etc.):
    - need to use a constraint language,
    - e.g., job scheduling. $T_1 + d \leq T_2$.
- **Continuous variables**:
  - Common in operations research
  - Linear programming problems with linear or non linear equalities

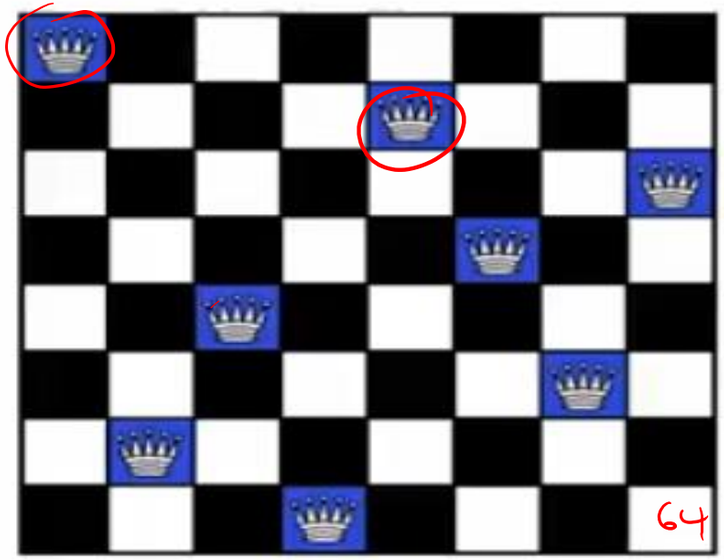


Varieties of constraints

- **Unary constraints**: involve a single variable e.g., SA $\neq$ green
- **Binary constraints**: involve pairs of variables e.g., SA $\neq$ WA
- **Global constraints**: involve 3 or more variables e.g., *Alldiff* that specifies that all variables must have different values (e.g., cryptarithmetic puzzles, Sudoku)
- **Preferences (soft constraints)**:
  - Example: red is better than green
  - Often represented by a cost for each variable assignment
  - constrained optimization problems

Example: 8-queen

8-Queen: Place 8 queens on an 8x8 chess board so no queen can attack another one.



Soln :

64

Variables : $x_1 x_2 x_3 \cdots x_8$

$D_1 = \{1, 2, \ldots 64\}$

$D_2 = \{$ $\}$

$D_3 = \{$ $\}$

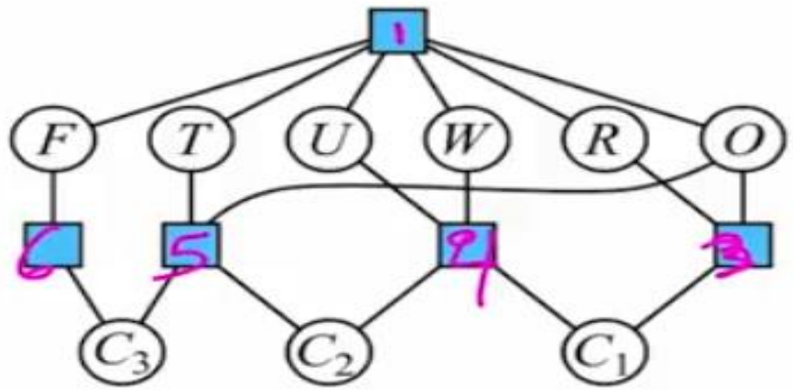
Problem formalization 1:

- One variable per queen, Q_1, Q_2, ..., Q_8.
- Each variable could have a value between 1 and 64.
- Solution: $Q_1 = 1$, $Q_2 = 13$, $Q_3 = 24$, ..., $Q_8 = 60$.

Example Cryptarithmetic

$C_3\ C_2\ C_1$

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



Variables: $X = \{F, T, U, W, R, O, C_1, C_2, C_3\}$

Domain: $D = \{0, 1, 2, \cdots, 9\}$

Constraints:

- Alldiff(F, T, U, W, R, O)
- $T \neq 0, \quad F \neq 0$
- $O + O = R + 10 * C_1$
- $C_1 + W + W = U + 10 * C_2$
- $C_2 + T + T = O + 10 * C_3$
- $C_3 = F$

n-ary constraint

6-ary

Summary

- CSPs are a special kind of search problems:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) is an important mechanism in CSPs.
- It does additional work to constrain values and detect inconsistencies.
- Tree-structured CSPs can be solved in linear time
- Further exploration: How can local search be used for CSPs?
- **The power of CSPs: domain-independent, that is you only need to define the problem and then use a solver that implements CSPs mechanisms.**