Artificial Intelligence
DSE 3252

Problem Solving

ROHINI R RAO & PADMASHREE G
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2025

Problem Solving Agents

Are goal based agents that use atomic representations

Goal formulation

- First step in problem solving
- based on the current situation and the agent's performance measure
- Environment is represented by states
- Goal state in one in which the goal is satisfied

The agent's task is to find out how to act, now and in the future, so that it reaches a goal state

Problem formulation

- is the process of deciding what actions and states to consider, given a goal

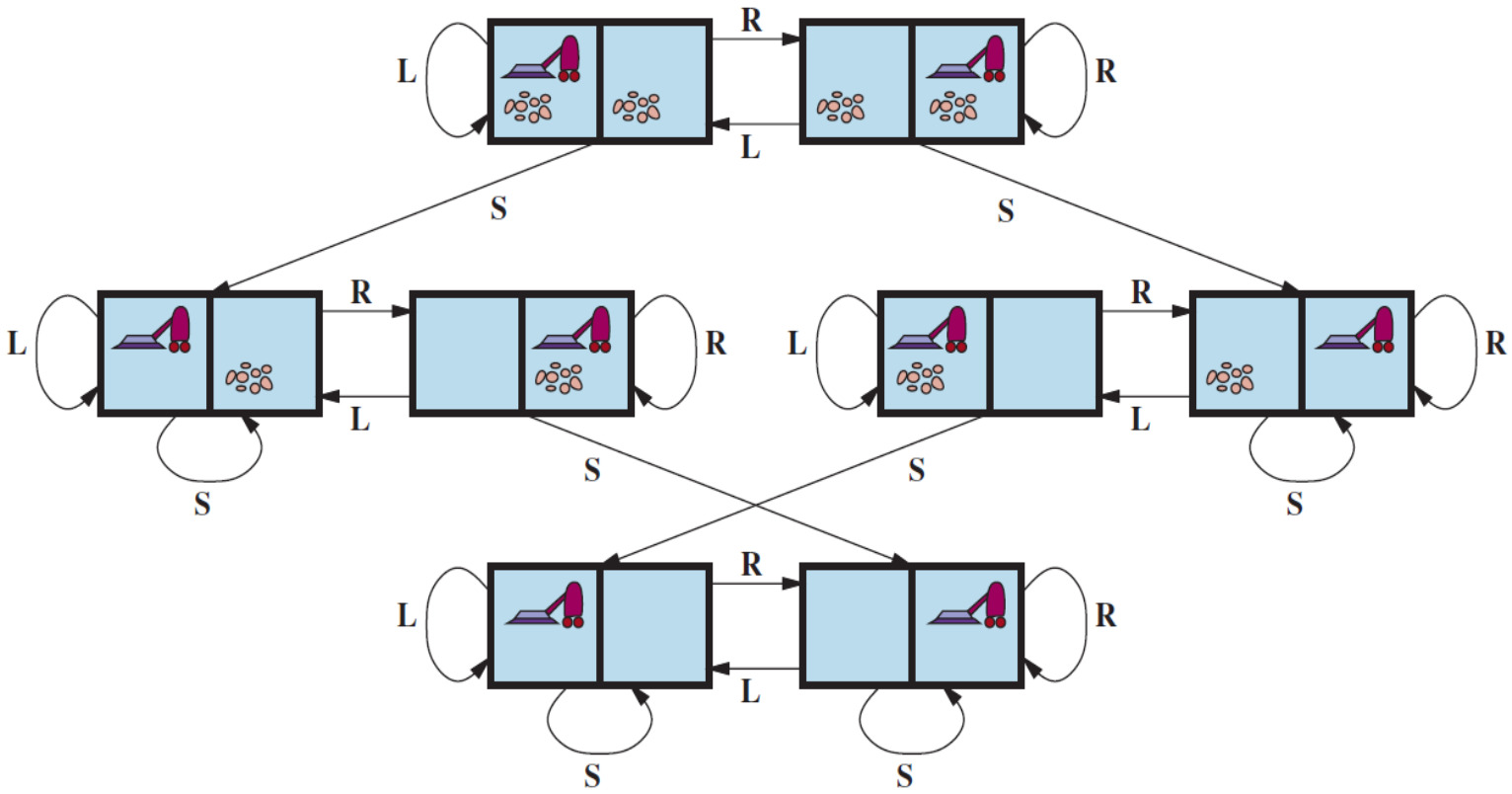
an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value

State Space

- forms a **directed network or graph** in which the nodes are states and the links between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.

- A **solution** to a problem is an action sequence that leads from the initial state to a goal state
- Solution quality is measured by the **path cost** function
- an **optimal solution** has the lowest path cost among all solutions

Example 1 – Vacuum Agent State Transition Diagram



Well-defined problems and solutions

A **Problem** can be defined formally:

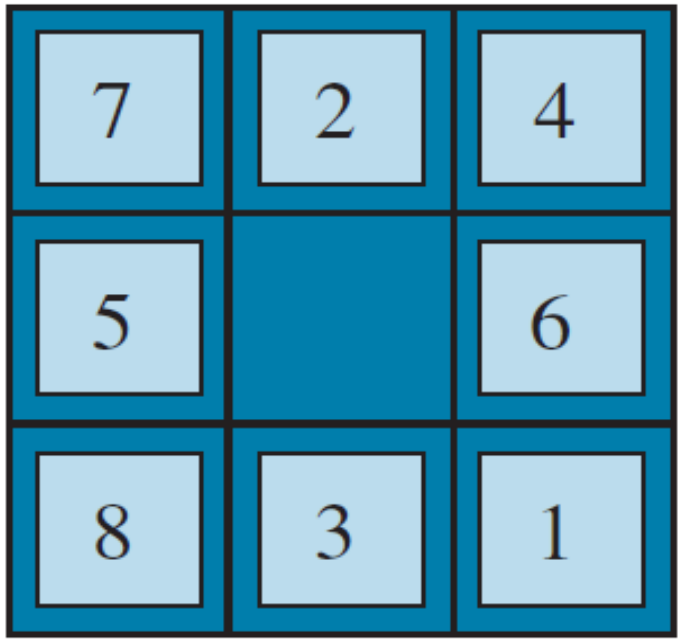
1. The **initial state** that the agent starts in
2. A description of the possible **actions**
3. **Transition model** available to the agent
   - A description of what each action does;
4. The **goal test**, which determines whether a given state is a goal state
5. A **path cost** function that assigns a numeric cost to each path.
   - **Successor** to refer to any state reachable from a given state by a single action

- **State space** of the problem are defined by the initial state, actions, and transition model

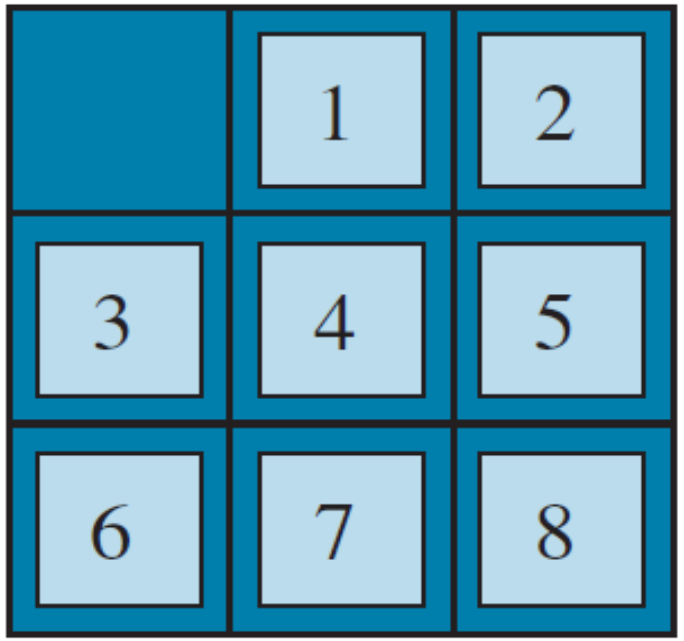
Problem Formulation
Example 1 - Vacuum Agent

- **States:** The state is determined by both the agent location and the dirt locations
  - There are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: each state has just three actions: *Left*, *Right*, and *Suck*
- • **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect.
- • **Goal test**: This checks whether all the squares are clean.
- • **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Problem Formulation
Example 2 – 8 puzzle problem



Start State



Goal State

Problem Formulation
Example 2 – 8 puzzle problem

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

- Initial state: Any state can be designated as the initial state.
- Actions: The simplest formulation defines the actions as movements of the blank space-Left, Right, Up, or Down.
- Transition model: Given a state and action, this returns the resulting state
- Goal test: This checks whether the state matches the goal configuration
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

“formulate, search, execute”

- The process of looking for a sequence of actions that reaches the goal is called **Search**
- **Search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence
- Once a solution is found, the actions are carried out in the **execution** phase
- The possible action sequences starting at the initial state form a **Search Tree:**
  - nodes correspond to states in the state space of the problem.
  - with the initial state NODE at the root
  - the branches are actions
- The set of all leaf nodes available for expansion at any given point is called the **Frontier (Open List)**
- Loopy path
  - result in **repeated state**
  - are a special case of the more general concept of **redundant paths**
- To avoid exploring redundant paths remember where one has been.
- Tree-Search algorithm can be augmented with a data structure called the **explored set (Closed list**), which remembers every expanded node

Partial Search Tree

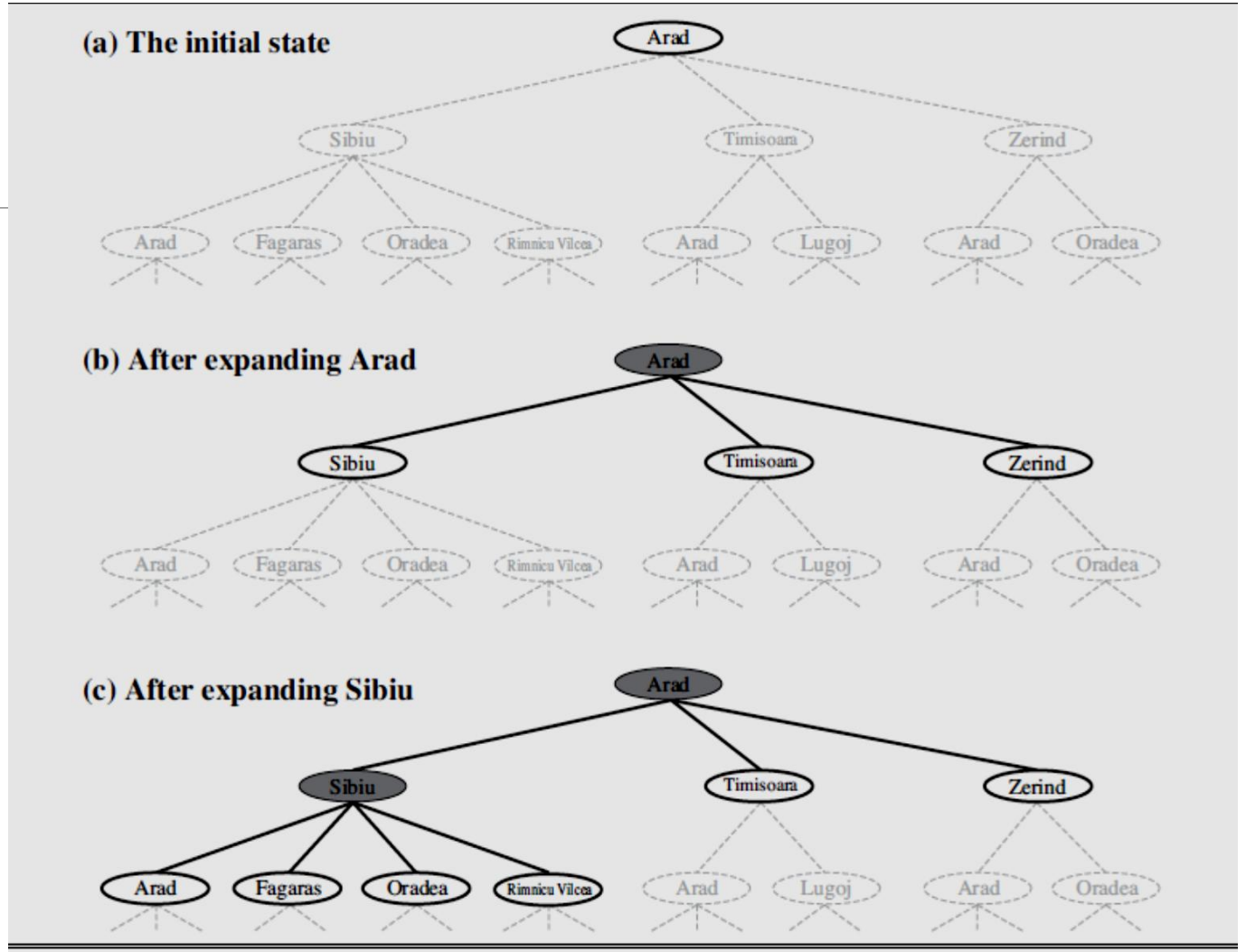


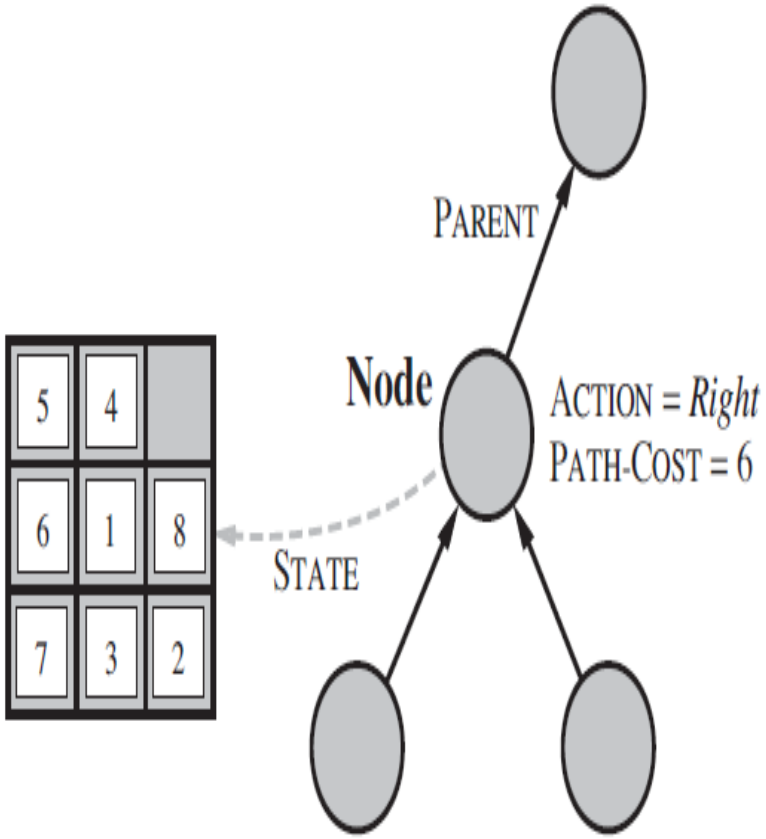
Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Infrastructure for search algorithms

For each node n of the tree, structure should contains 4 **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.

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*)



“formulate, search, execute”

- The process of looking for a sequence of actions that reaches the goal is called **Search**
- **Search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence
- Once a solution is found, the actions are carried out in the execution phase

```
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
```

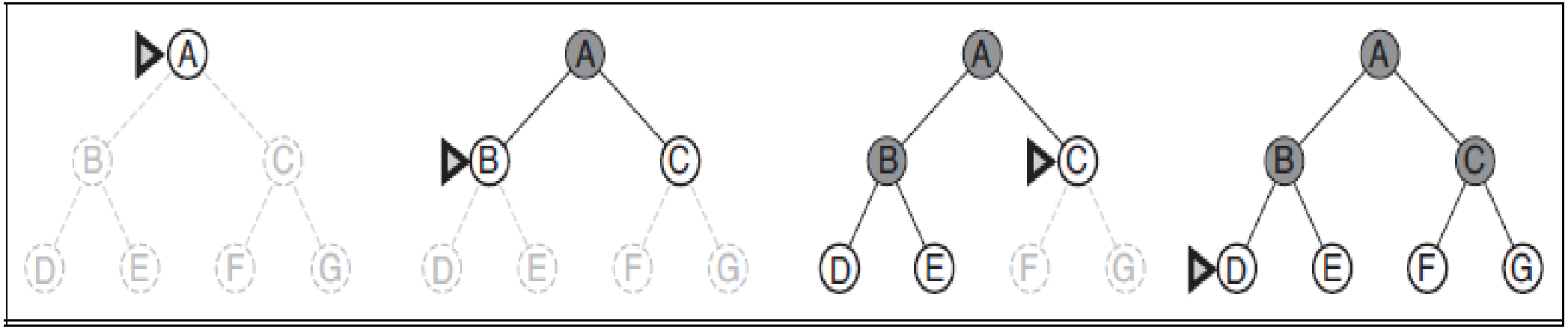
```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return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

Uninformed search strategies
Breadth First Search

Shallowest unexpanded node is chosen for expansion

- the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- Using a **FIFO queue** for the frontier
- goal test is applied to each node when it is *generated*



Breadth First Search

```
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)
```

Measuring problem-solving performance Breadth First Search (BFS)

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Yes, if the shallowest goal node is at some finite depth d & branching factor b is finite.

Optimality: Does the strategy find the optimal solution?

- Yes, if the path cost is a nondecreasing function of the depth of the node.

Time complexity: How long does it take to find a solution?

- The root of the search tree generates b nodes at the first level,, for a total of b^2 at the second level and so on. Suppose that the solution is at depth d.
- Then the total number of nodes generated is $b + b^2 + b^3 \cdot \cdot \cdot + b^d = O(b^d)$

Space complexity: How much memory is needed to perform the search?

- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier

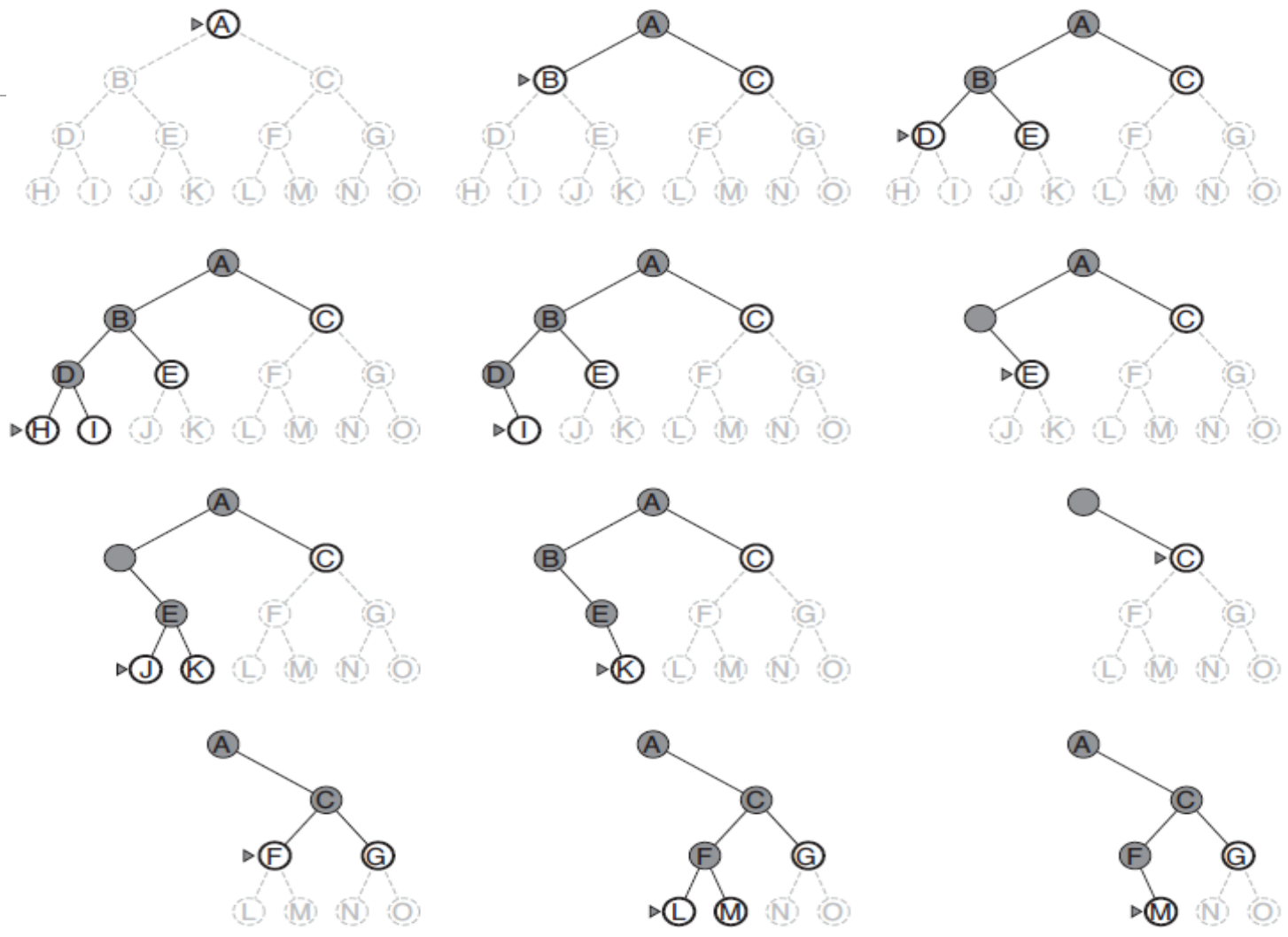
Time and memory requirements for BFS

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Depth First Search

- Instance of graph search which uses LIFO queue
- always expands the *deepest* node in the current frontier of the search tree
- The search proceeds immediately to the **deepest level** of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the **frontier**, so then the search "backs up" to the next deepest node that still has unexplored successors.

Depth First Search



Measuring problem-solving performance Depth First Search (BFS)

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces
- The tree-search version, on the other hand, is not complete

Optimality: Does the strategy find the optimal solution?

- both versions are nonoptimal

Time complexity: How long does it take to find a solution?

- all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node;

Space complexity: How much memory is needed to perform the search?

- requires storage of only O(bm) nodes

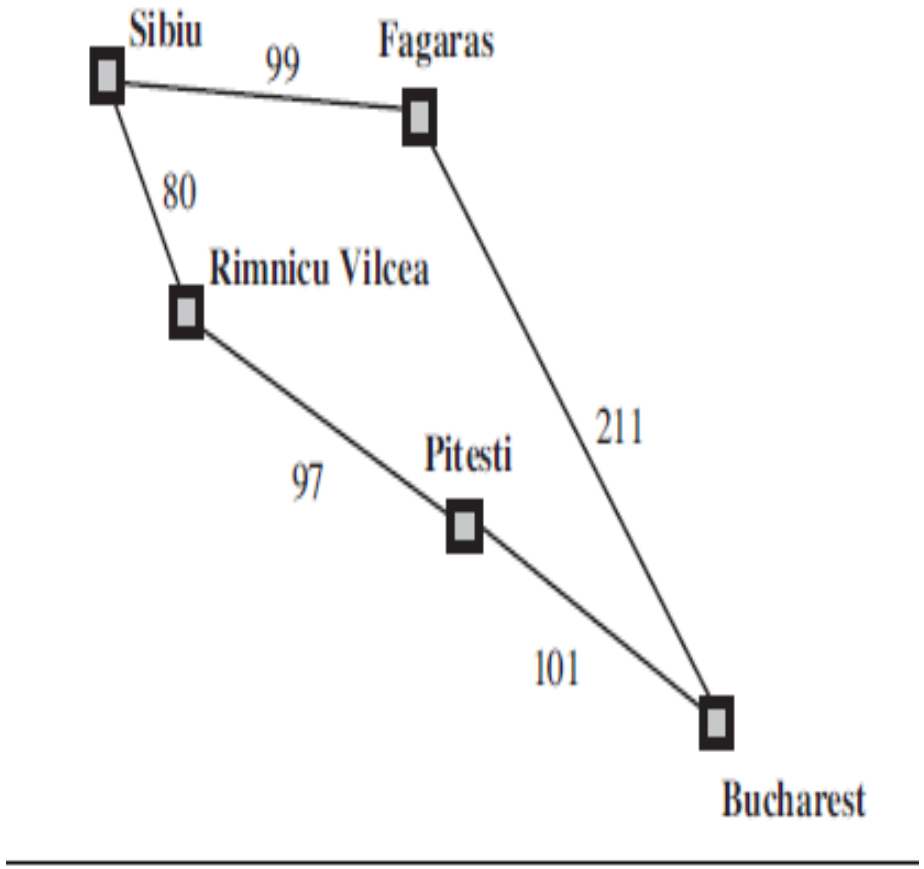
Uniform Cost Search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n)
- Store the frontier as a priority queue ordered by g
- Difference from Breadth-first search
  - the goal test is applied to a node when it is selected for expansion rather than when it is first generated
  - a test is added in case a better path is found to a node currently on the frontier

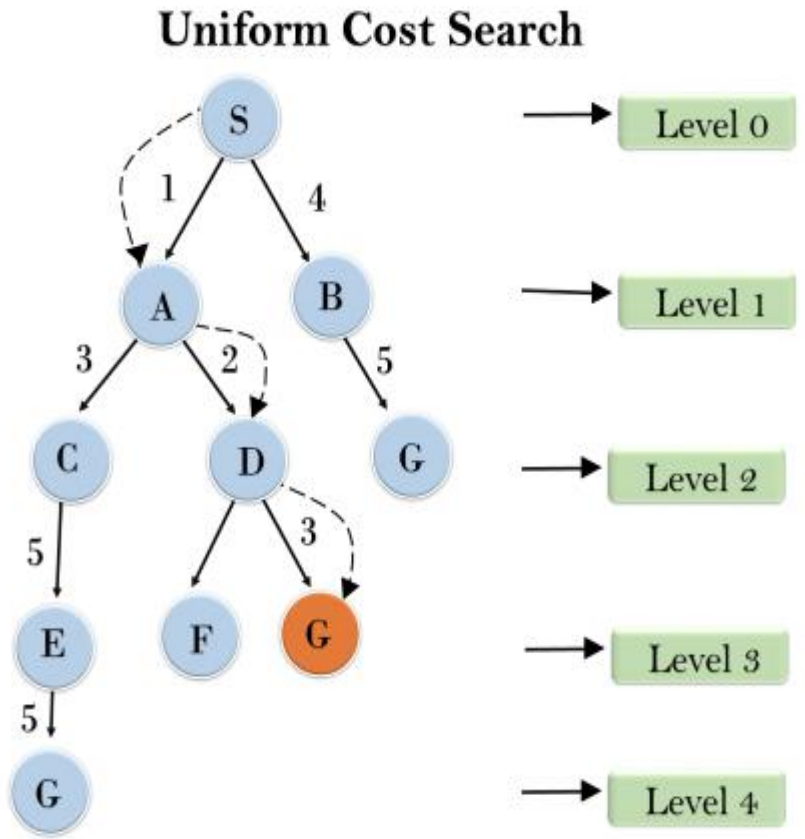
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      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
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform Cost Search



Uniform Cost Search - Example



Measuring problem-solving performance Uniform Cost

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Completeness is guaranteed provided the cost of every step exceeds some small positive constant €
- May get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions

Optimality: Does the strategy find the optimal solution?

- Yes

Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?

- let C *be the cost of the optimal solution and assume that every action costs at least € Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Comparing Uninformed cost strategies

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

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Informed (Heuristic) Search Strategies

- The general approach is called **best-first search**.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n).
- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The choice of f determines the search strategy.
- Most best-first algorithms include as a component of f a **heuristic function**, denoted h(n):
- h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

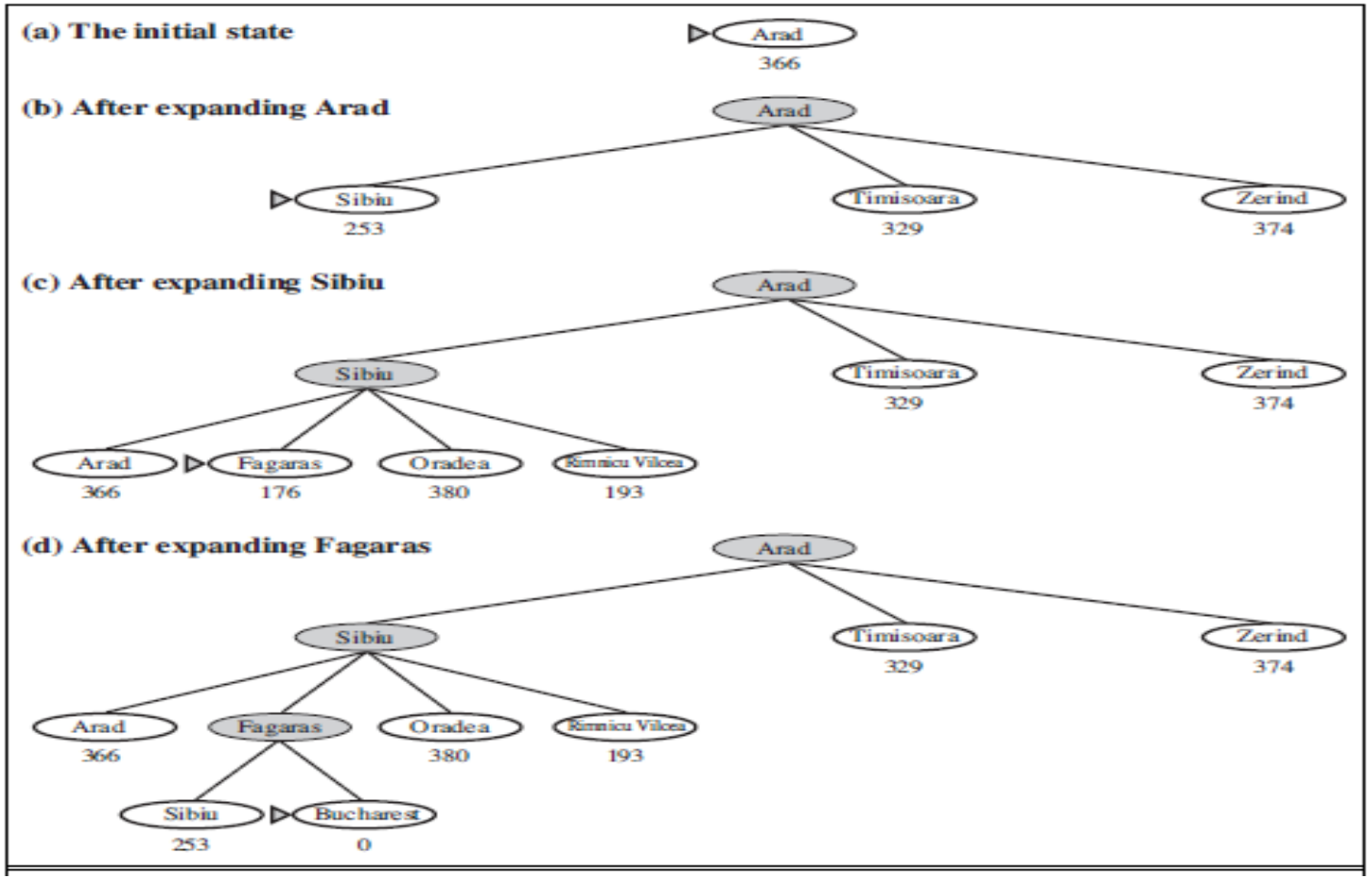
Greedy best-first search

- tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n)

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

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

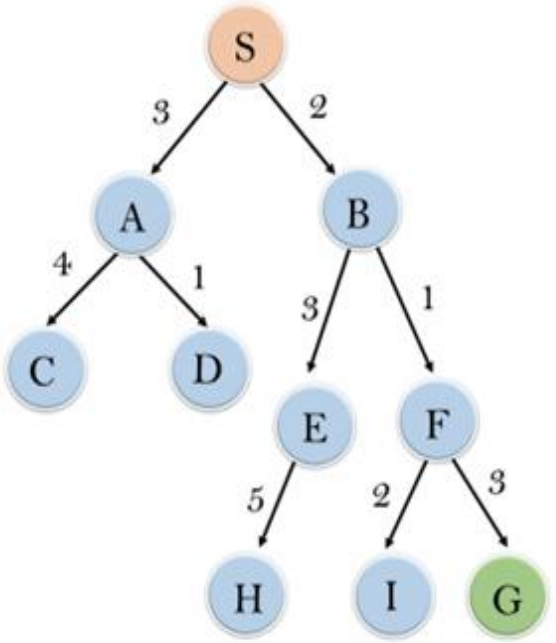
Greedy best-first search



Greedy Best First Algorithm

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and place it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:**
  - Check each successor of node n, and find whether any node is a goal node.
  - If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:**
  - For each successor node, checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list.
  - If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Greedy Best First Search - Example



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Measuring problem-solving performance Greedy Best First Search

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Greedy best-first tree search is incomplete even in a finite state space, much like depth-first search

Optimality: Does the strategy find the optimal solution?

- No

Time complexity: How long does it take to find a solution?

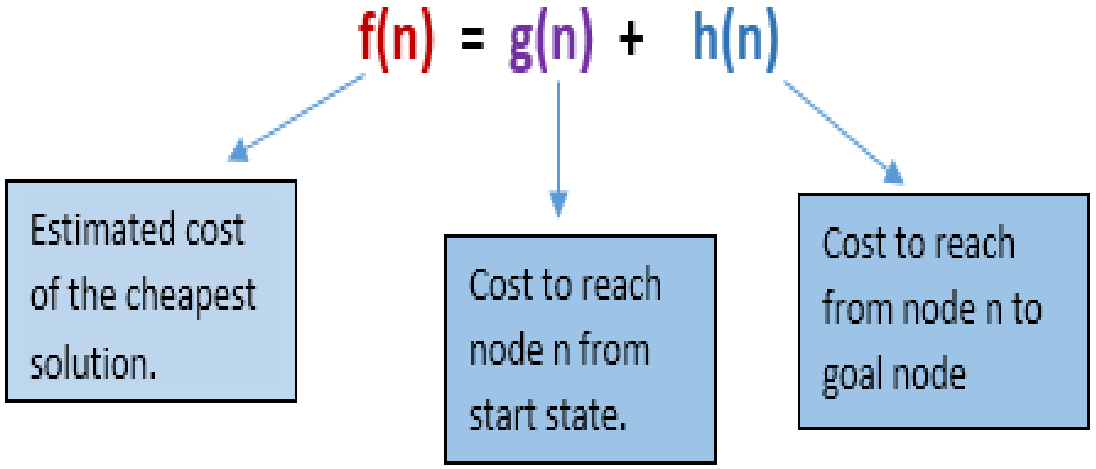
- The worst case time complexity of Greedy best first search is $O(b^m)$.

Space complexity: How much memory is needed to perform the search?

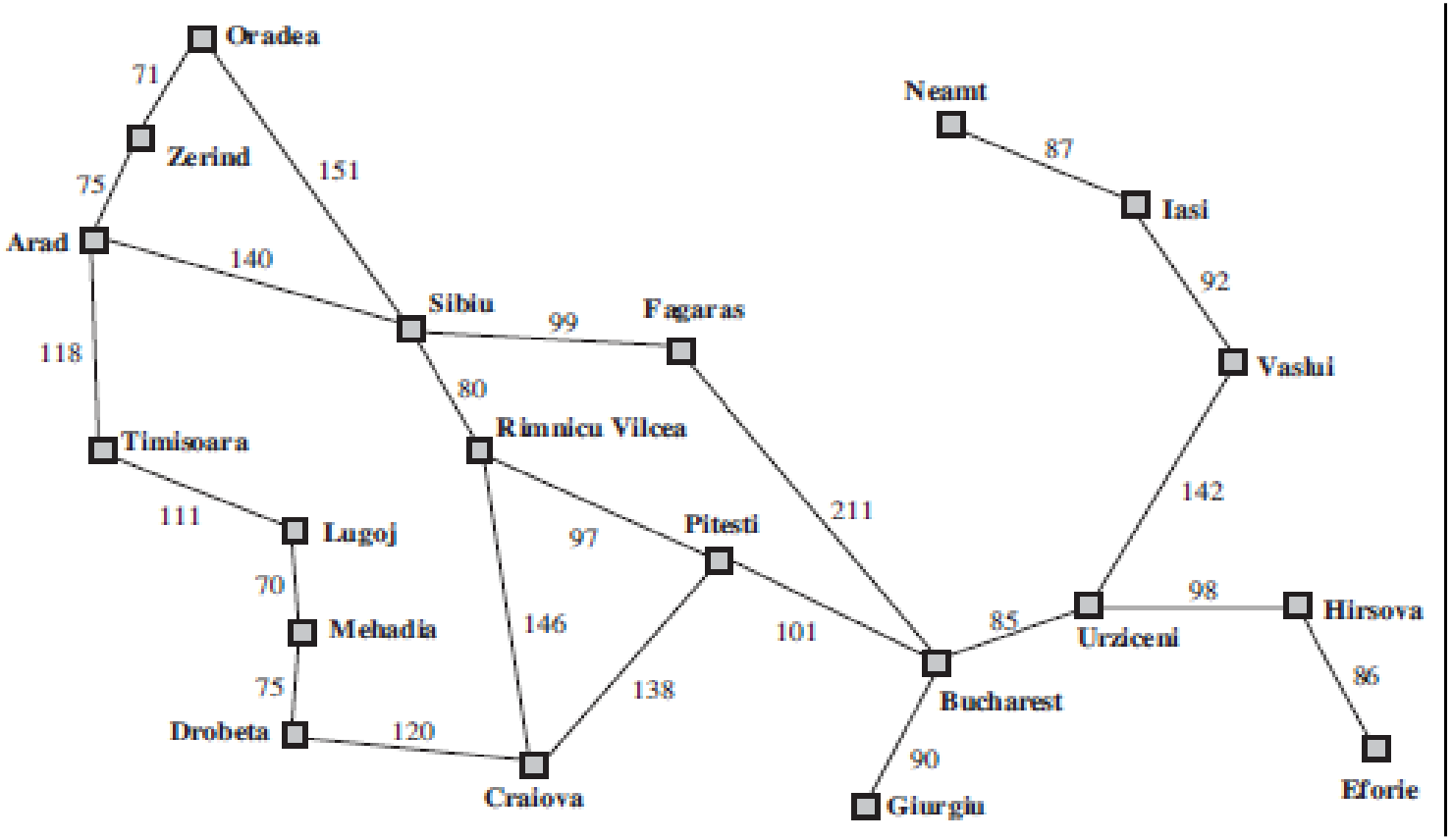
- The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

A* search: Minimizing the total estimated solution cost

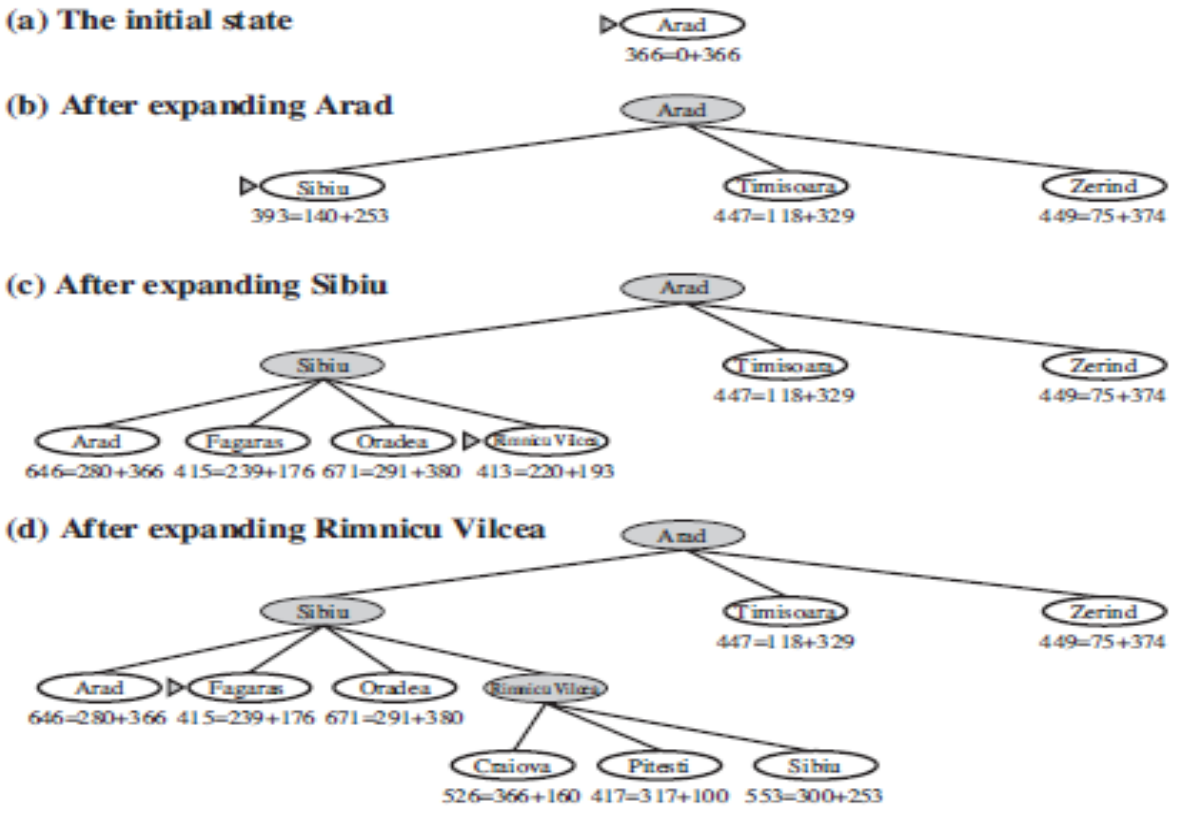
It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:



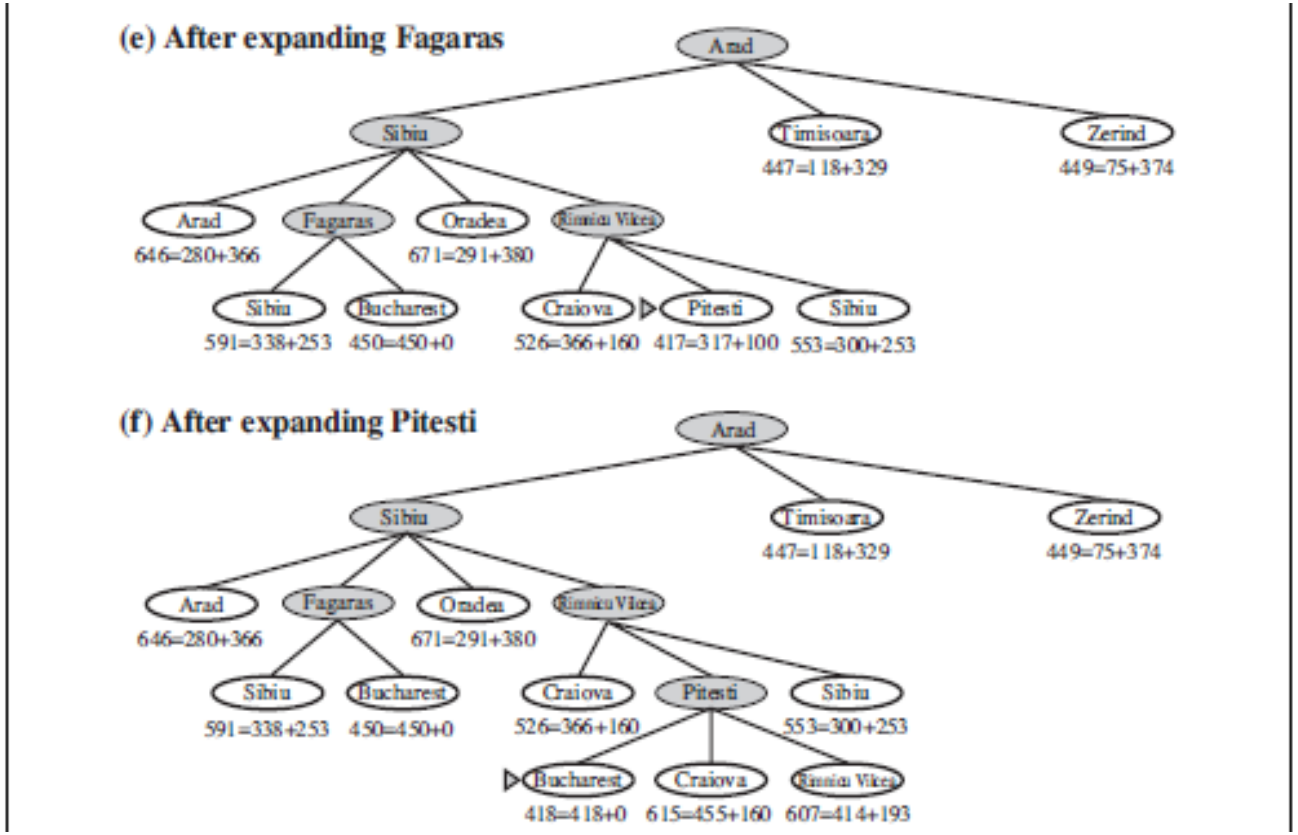
A* - Combines SLD with Path Cost



Stages in A*


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

Stages in A*


(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3:

- Select the node from the OPEN list which has the smallest value of evaluation function (g+h)
- if node n is goal node then return success and stop

Step 4:

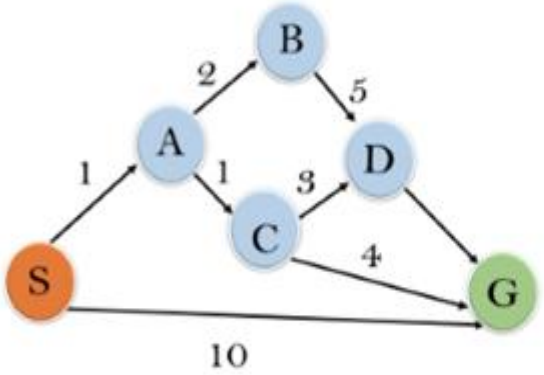
- Expand node n and generate all of its successors, and put n into the closed list.
- For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5:

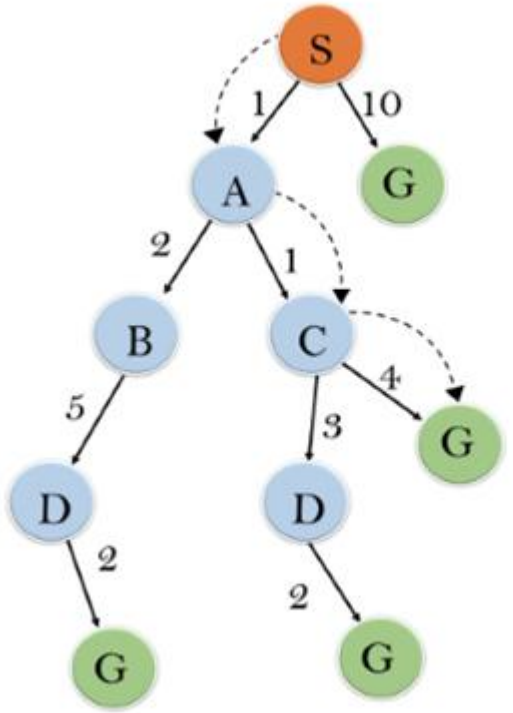
- Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

A* Search - Example



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Measuring problem-solving performance
A*

Completeness: Is the algorithm guaranteed to find a solution when there is one?

- Complete as long as , Branching factor is finite & Cost at every action is fixed

Optimality: Does the strategy find the optimal solution?

A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** h(n) should be an admissible heuristic for A* tree search.
  - An admissible heuristic is optimistic in nature, never overestimates the cost to reach the goal
- **Consistency:** for only A* graph-search.
  - A heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of
  - reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'

$$h(n) \leq c(n, a, n') + h(n') \; .$$

Time complexity: How long does it take to find a solution?

- Depends on Heuristic Function and based on depth d of solution is $O(b^d)$.

Space complexity: How much memory is needed to perform the search?

- $O(b^d)$

A* - Features

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Local search algorithms

In many optimization problems, the path to the goal is irrelevant

the goal state itself is the solution

State space = set of "complete" configurations

Find configuration satisfying constraints

- Examples: n-Queens, VLSI layout, airline flight schedules

Local search algorithms

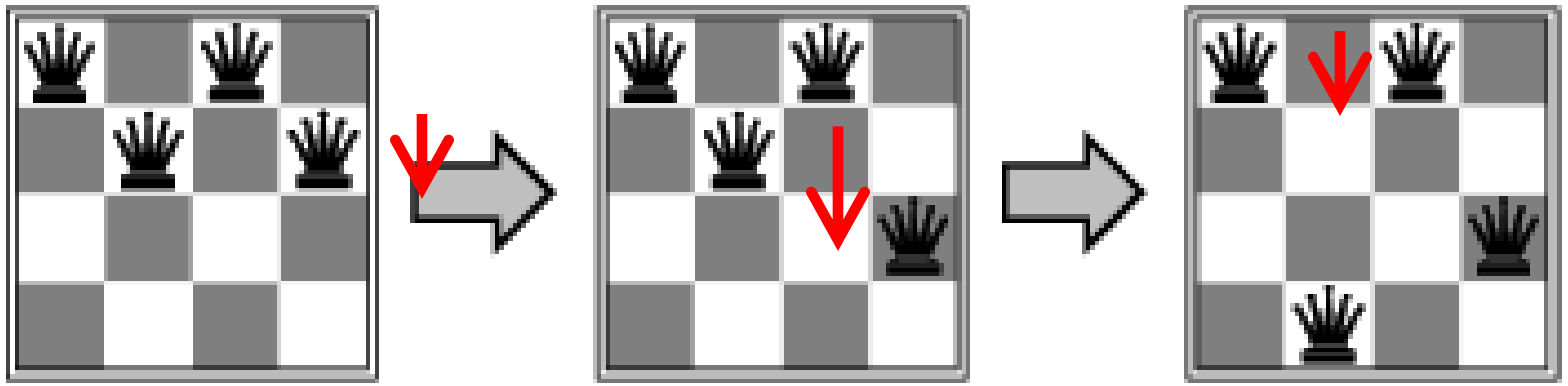
- Keep a single "current" state, or small set of states
- Iteratively try to improve it / them
- Very memory efficient
  - keeps only one or a few states
  - You control how much memory you use

Example: *n*-queens

Goal: Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal

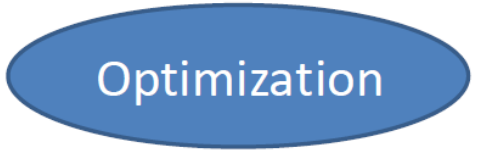
Neighbor: move one queen to another row

Search: go from one neighbor to the next…



Satisfaction vs Optimization

reach the goal node
Constraint satisfaction



optimize(objective fn)
Constraint Optimization

Local Search and Optimization

Local search

- Keep track of single current state
- Move only to neighboring states
- Ignore paths

Advantages:

- Use very little memory
- Can often find reasonable solutions in large or infinite (continuous) state spaces.

"Pure optimization" problems

- All states have an objective function
- Goal is to find state with max (or min) objective value
- Does not quite fit into path-cost/goal-state formulation
- Local search can do quite well on these problems.

Algorithm design considerations

How do you represent your problem?

What is a “complete state”?

What is your objective function?

- How do you measure cost or value of a state?

What is a “neighbor” of a state?

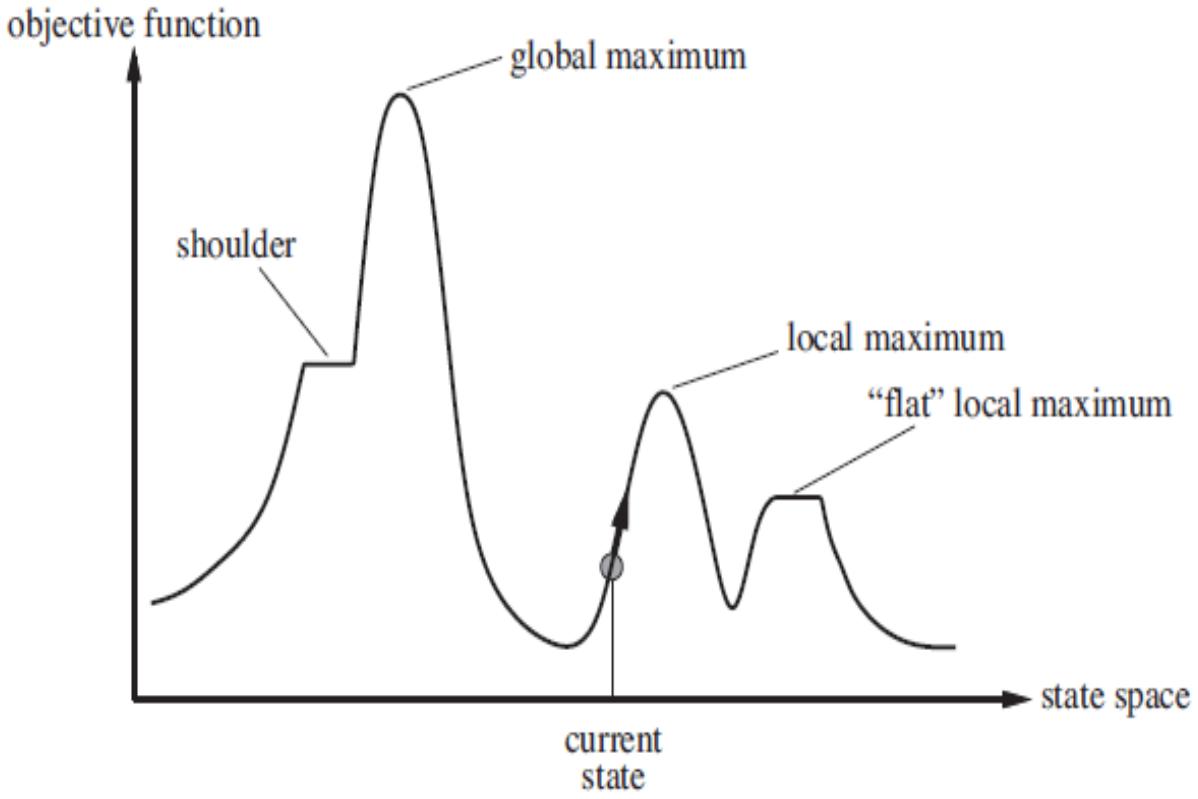
- Or, what is a “step” from one state to another?
- How can you compute a neighbor or a step?

Are there any constraints you can exploit?

Local search

- Search so far - observable, deterministic, known environments where the solution is a sequence of actions.
- Local Search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state.
- Suitable for
  - problems in which all that matters is the solution state, not the path cost to reach it.
  - solving pure optimization problems, in which the aim is to find the best state according to an objective function.
- Algorithms include
- **Simulated Annealing** based on statistical physics
- **Genetic Algorithm** based evolutionary biology.
- Not systematic, they have two key advantages:
  - (1) they use very little memory—usually a constant amount
  - (2) can find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

State Space Landscape



A landscape has

- Location- defined by the state
- Elevation- defined by the value of the heuristic cost function or objective function
- If elevation corresponds to
  - cost, then the aim is to find the lowest valley—a **global minimum**
  - to an objective function, then the aim is to find the highest peak—a **global maximum**
- C**omplete** local search algorithm always finds a goal if one exists
- **Optimal** algorithm always finds a global minimum/maximum.

Algorithm for Simple Hill Climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.
Step 2: Loop Until a solution is found or there is no new operator left to apply.
Step 3: Select and apply an operator to the current state.
Step 4: Check new state:

If it is goal state, then return success and quit.

Else if it is better than the current state then assign new state as a current state.

Else if not better than the current state, then return to step2.

Step 5: Exit.

Hill Climbing Search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Local maxima:

is a peak that is higher than each of its neighboring states but lower than the global maximum

Ridges:

result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

Plateau

is a flat area of the state-space landscape.
It can be a flat local SHOULDER maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.

Steepest-Ascent hill climbing

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
    a) Let SUCC be a state such that any successor of the current state will be better than it.
    b) For each operator that applies to the current state:
        a) Apply the new operator and generate a new state.
        b) Evaluate the new state.
        c) If it is goal state, then return it and quit, else compare it to the SUCC.
        d) If it is better than SUCC, then set new state as SUCC.
        e) If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

Variation : Stochastic hill climbing

- does not examine for all its neighbor before moving.
- selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Adversarial Search

In **multiagent environments**, each agent needs to consider the actions of other agents and how they affect its own welfare

The unpredictability of these other agents can introduce **contingencies** into the agent's problem-solving process

Competitive environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

Mathematical **game theory**

- a branch of economics
- views any multiagent environment as a game, provided that the impact of each agent on the others is "significant,"
- regardless of whether the agents are cooperative or competitive

In AI, games are

- deterministic, turn-taking, two-player,
- **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.

this opposition between the agents' utility functions that makes the situation adversarial

Game defined as a kind of search problem

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.

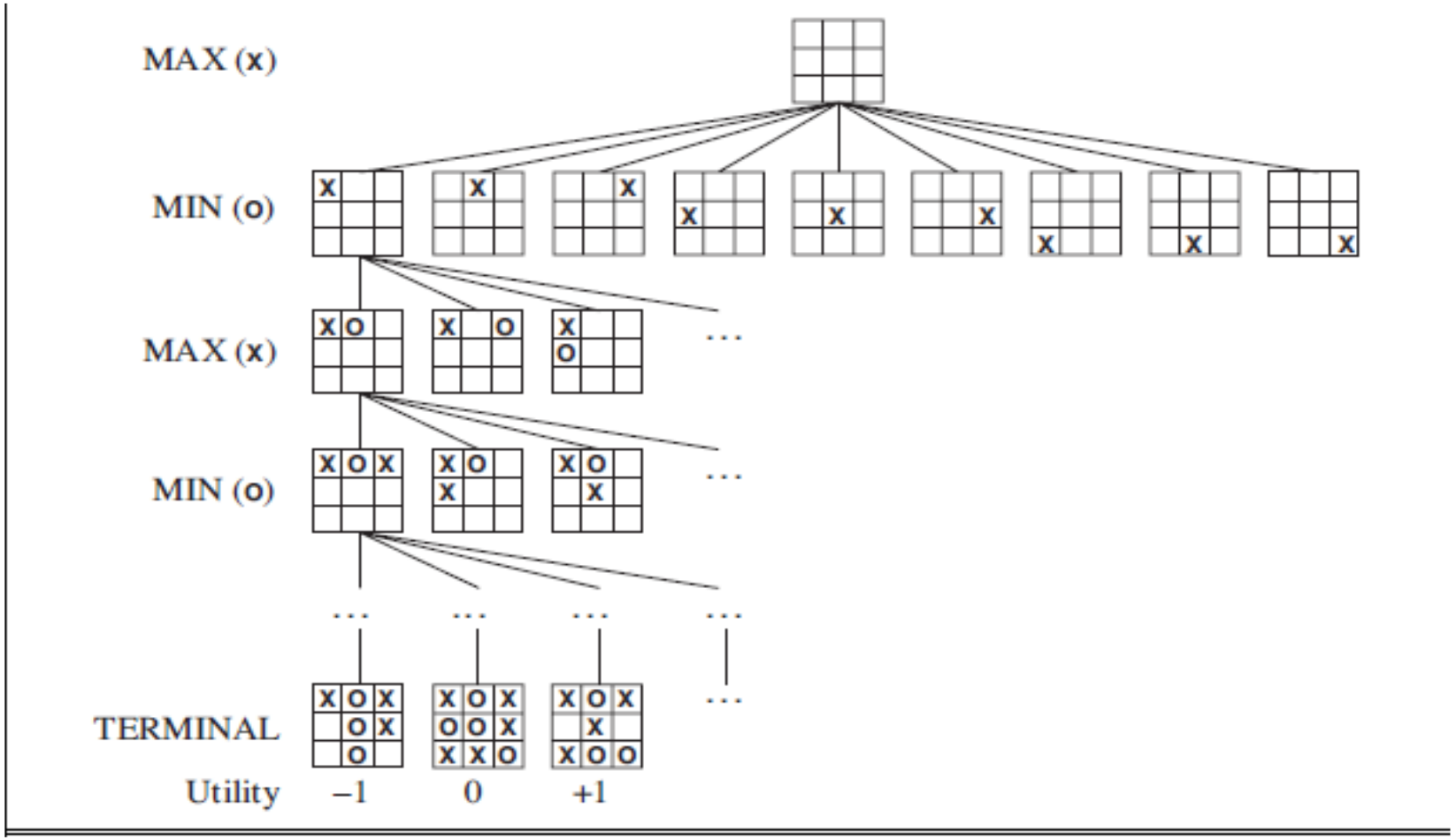
TERMINAL-TEST(s):

- A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.

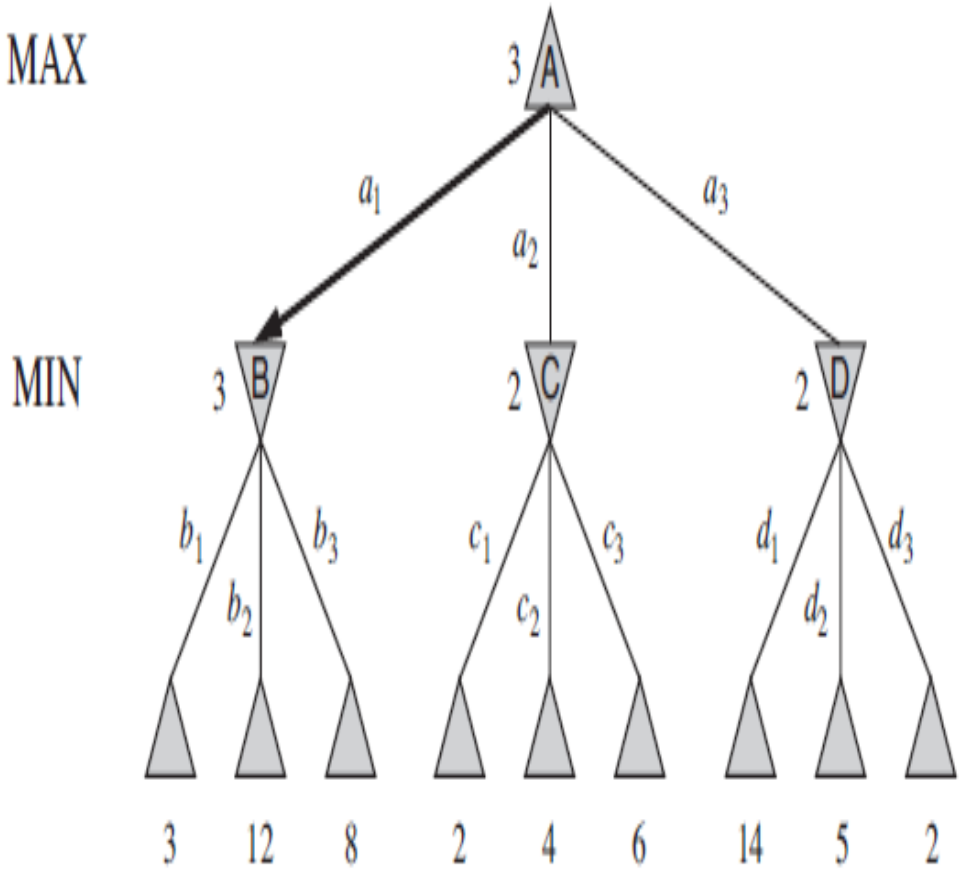
UTILITY(s, p):

- A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.
- In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2
- A zero-sum game - where the total payoff to all players is the same for every instance of the game.
- Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½ + ½

Partial Game Tree for tic-tac-toe



Optimal Decisions in Games



MAX must find a contingent **strategy**

- which specifies MAX's move in the initial state
- then MAX's moves in the states resulting from every possible response by MIN and so on

Assumption: optimal play for MAX assumes that MIN also plays optimally

Optimum strategy is determined by the minimax value of each node.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax Algorithm

computes the minimax decision from the current state.

It uses a simple recursive computation of the minimax values of each successor state

The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.

performs a complete depth-first exploration of the game tree

If the maximum depth of the tree is m and there are b legal moves at each point, then

- time complexity of the minimax algorithm is $O(b^m)$
- The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time

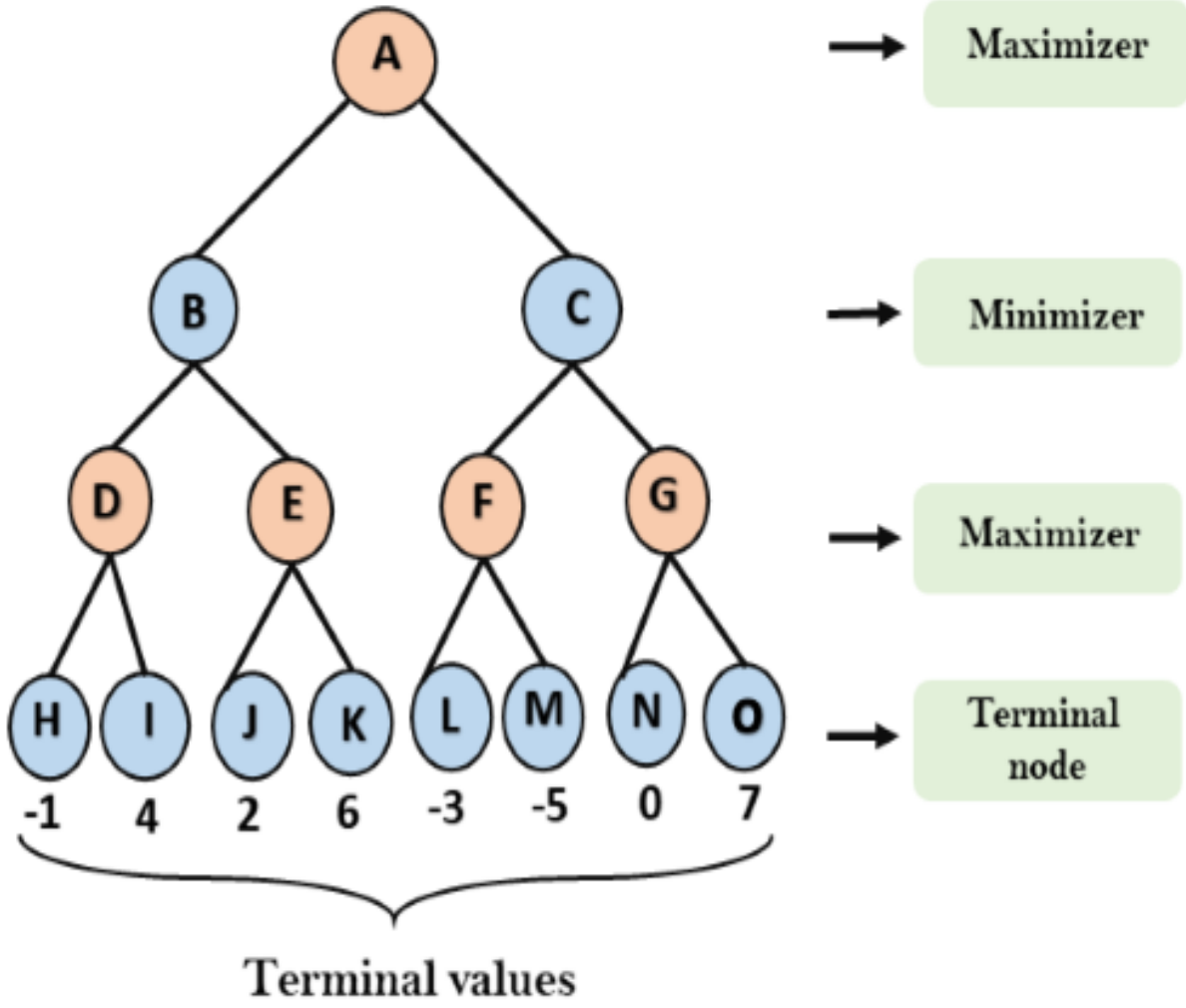
```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

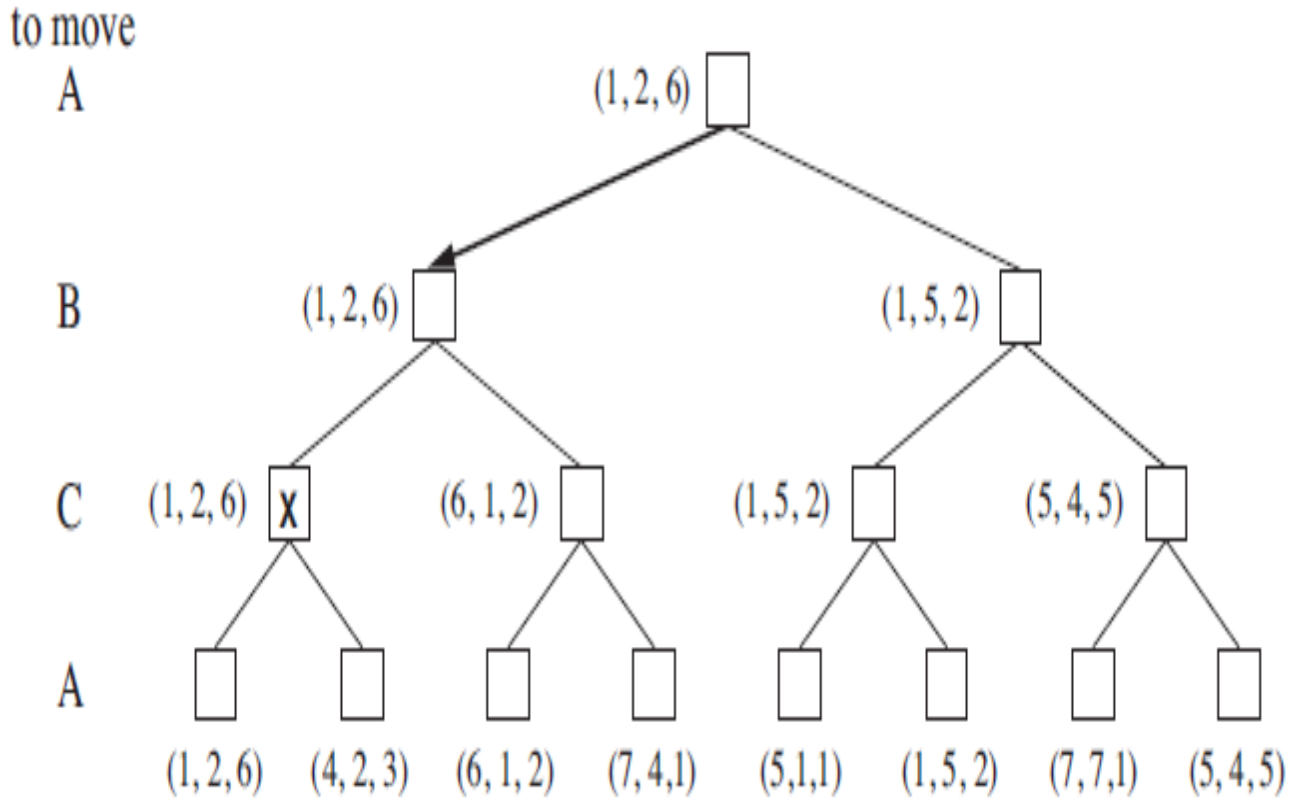
Psuedocode for Mini-Max Algorithm

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then  return static evaluation of node
 if MaximizingPlayer then      // for Maximizer Player
    maxEval= -infinity
    for each child of node do
        eval= minimax(child, depth-1, false)
        maxEval= max(maxEval,eval)       //gives Maximum of the values
    return maxEval
  else                    // for Minimizer player
   minEval= +infinity
   for each child of node do
       eval= minimax(child, depth-1, true)
       minEval= min(minEval, eval)       //gives minimum of the values
    return minEval
```



- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7

3 plies of Game tree with 3 players

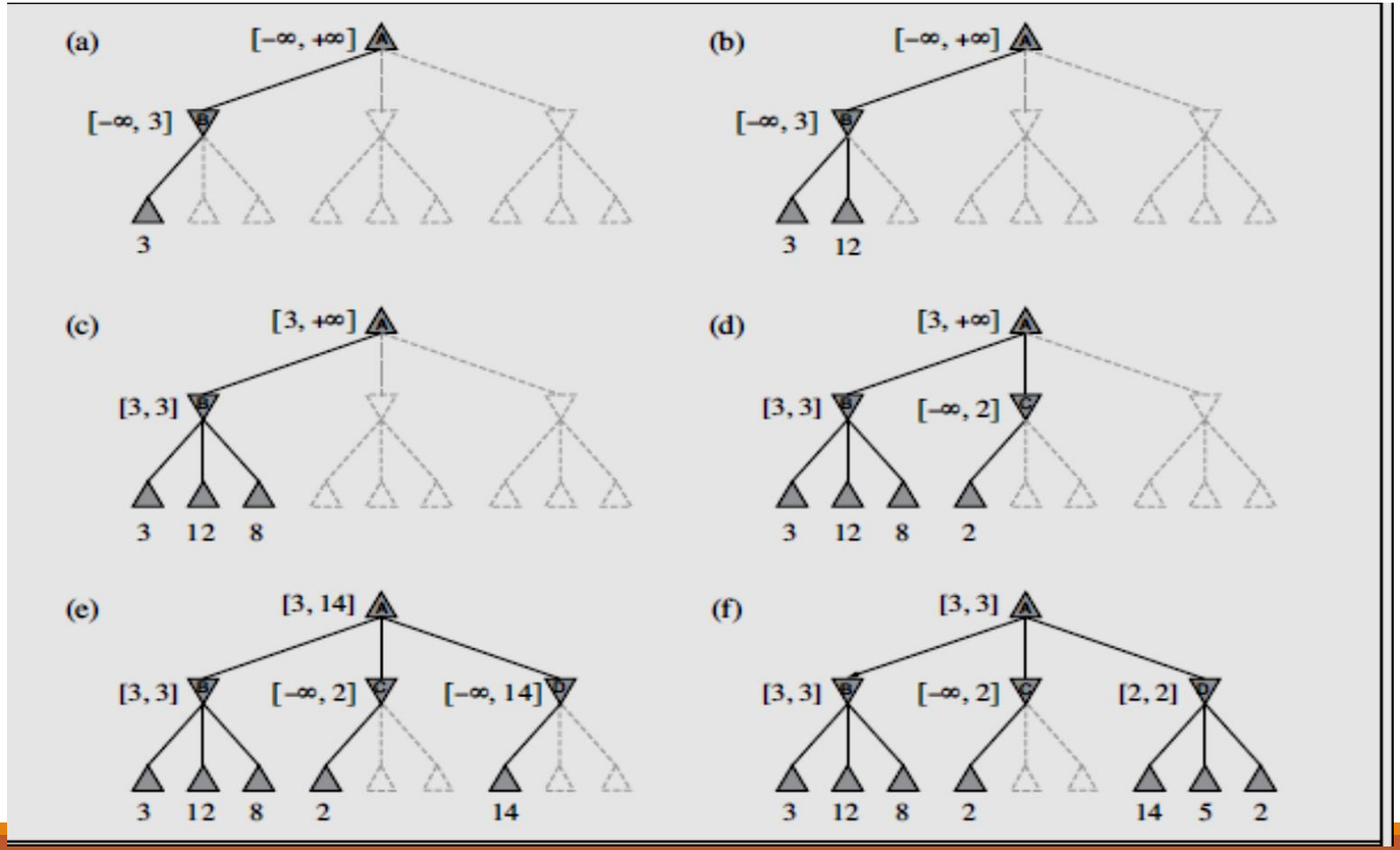


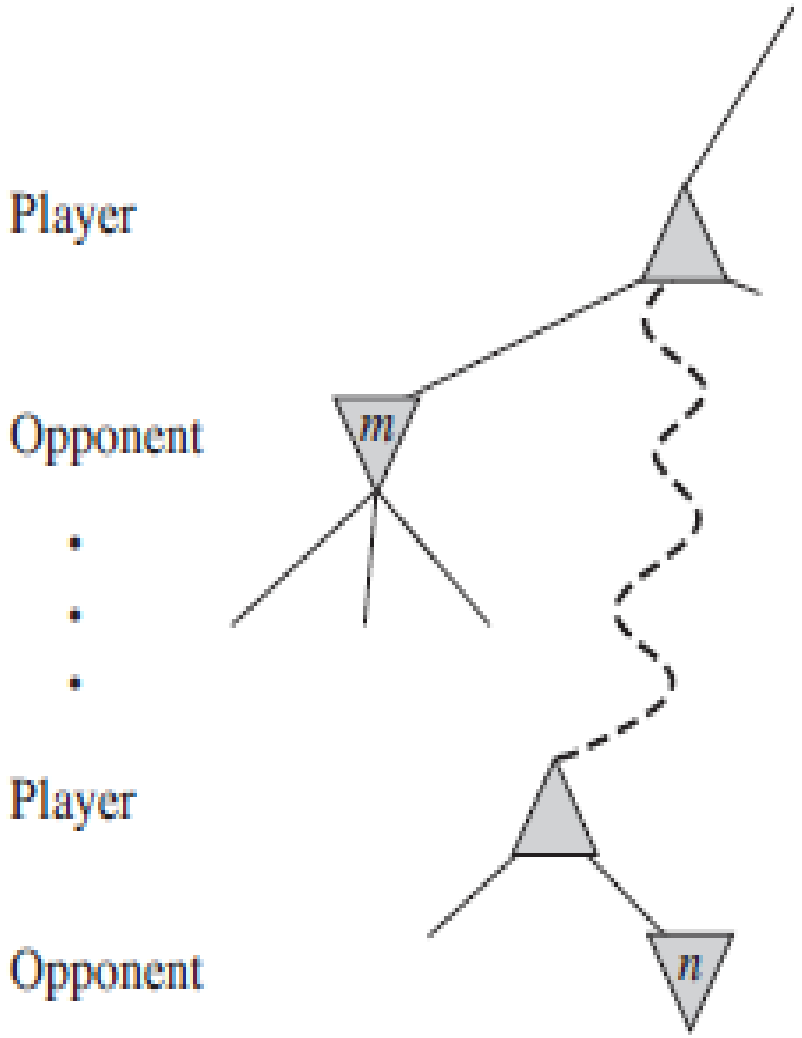
Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds.

Measuring Problem Solving Performance Mini-Max algorithm:

- **Complete-**
  - Yes. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-**
  - is optimal if both opponents are playing optimally.
- **Time complexity-**
  - As it performs DFS for the game-tree, so the time complexity is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity**
  - Space complexity of Mini-max algorithm is also similar to DFS which is **$O(b^m)$**.

Stages in the calculation of the optimal decision for the game tree





general case for alpha–beta pruning

If Player has a better choice m either at the parent node of n or at any choice point further up,

then n *will never be reached in actual play.*

So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Alpha Beta Pruning

The problem with minimax search is that the number of game states it has to examine is

exponential in the depth of the tree

Alpha–beta pruning

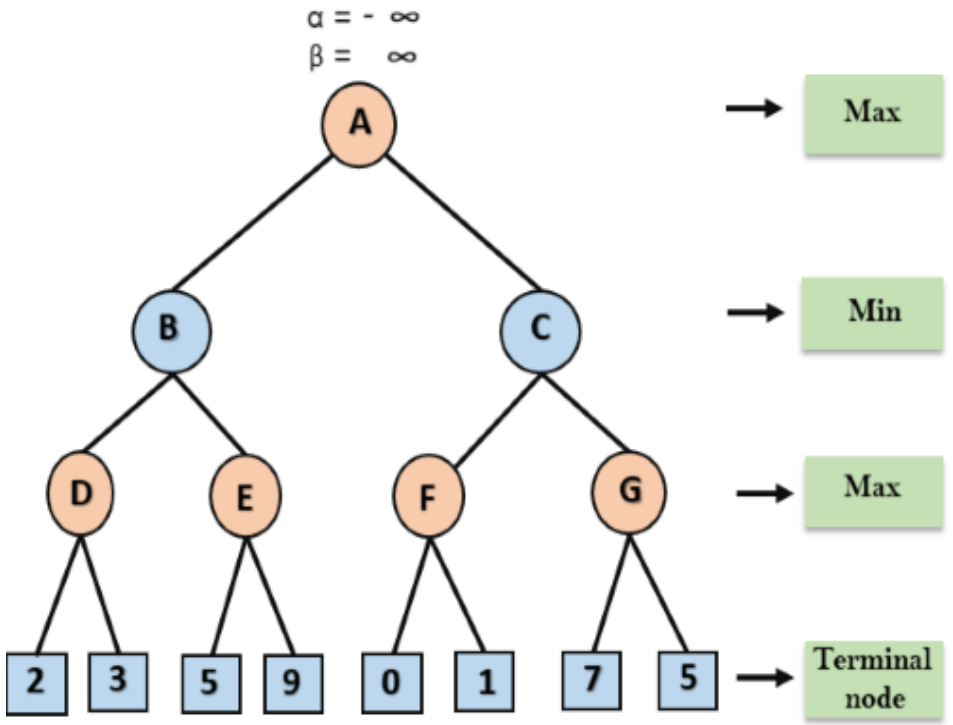
- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branchs that cannot possibly influence the final decision
- Simplification of Minimax formula

$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2)) \\ &= \max(3,\min(2,x,y),2) \\ &= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \le 2 \\ &= 3.\end{aligned}$$

- the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.
- Alpha–beta pruning has two parameters that describe bounds on the backed-up values that appear anywhere
- along the path:
- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

```
function minimax(node, depth, alpha, beta, maximizing player)
if depth ==0 or node is a terminal node then  return static evaluation of node
 if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha   break
 return maxEva
 else                    // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha    break
 return minEva
```

Alpha Beta Pruning



Alpha Beta Search Algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Improvements on Alpha-Beta Search

Killer Move Heuristics:

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined
- If we can examine first the successors that are likely to be best, then alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax
- **Dynamic move-ordering** – moves that were found to be best in the past.
- Best moves called **killer-moves**

Transposition table

- repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position
- The hash table of previously seen positions is maintained in a transposition table
- Minimax & its related algorithms wont work for games like Chess or Go
- Claude Shannon's paper **Programming a Computer for Playing Chess** (1950) proposed
  - **Type A strategy** – (Wide but shallow) Considers all possible moves to a certain depth and then uses heuristic evaluation function to estimate utility of states in that depth
  - **Type B strategy** – (Deep but narrow) ignores moves that look bad and follows promising paths as far as possible

Heuristic Alpha-Beta Tree Search

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN.} \end{cases}$$

Cutoff test

- should return true if terminal states but otherwise can cutoff search based on depth or any other heuristic

Eval function

- returns an estimate of expected utility of state s to player p
- Should be strongly correlated to actual chances of winning
- Example: Weighted Linear function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)\,,$$

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010