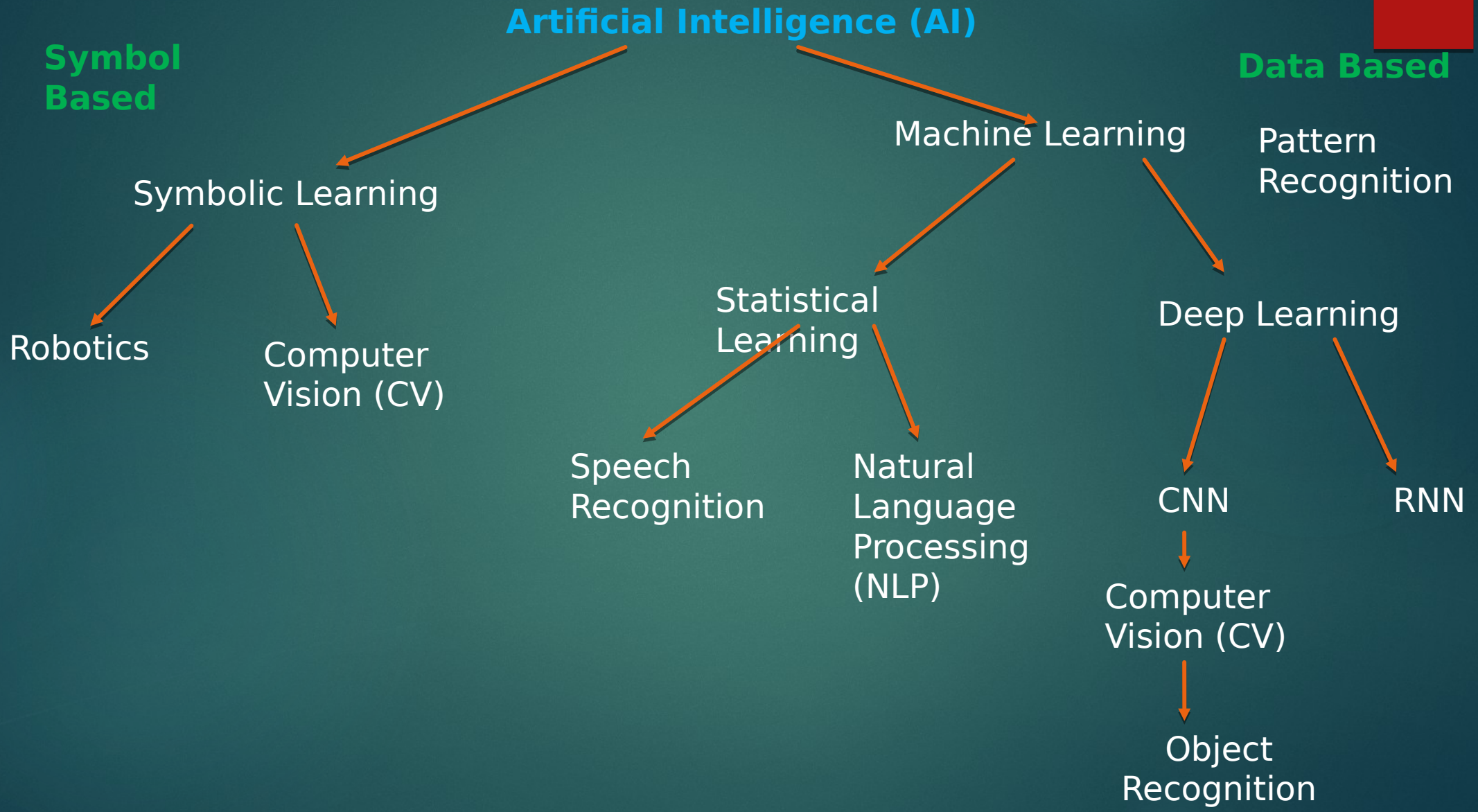
Artificial Intelligence

INTERIM SEMESTER 2021-22
BPL
CSE3007-LT-AB306
FACULTY: SIMI V.R.

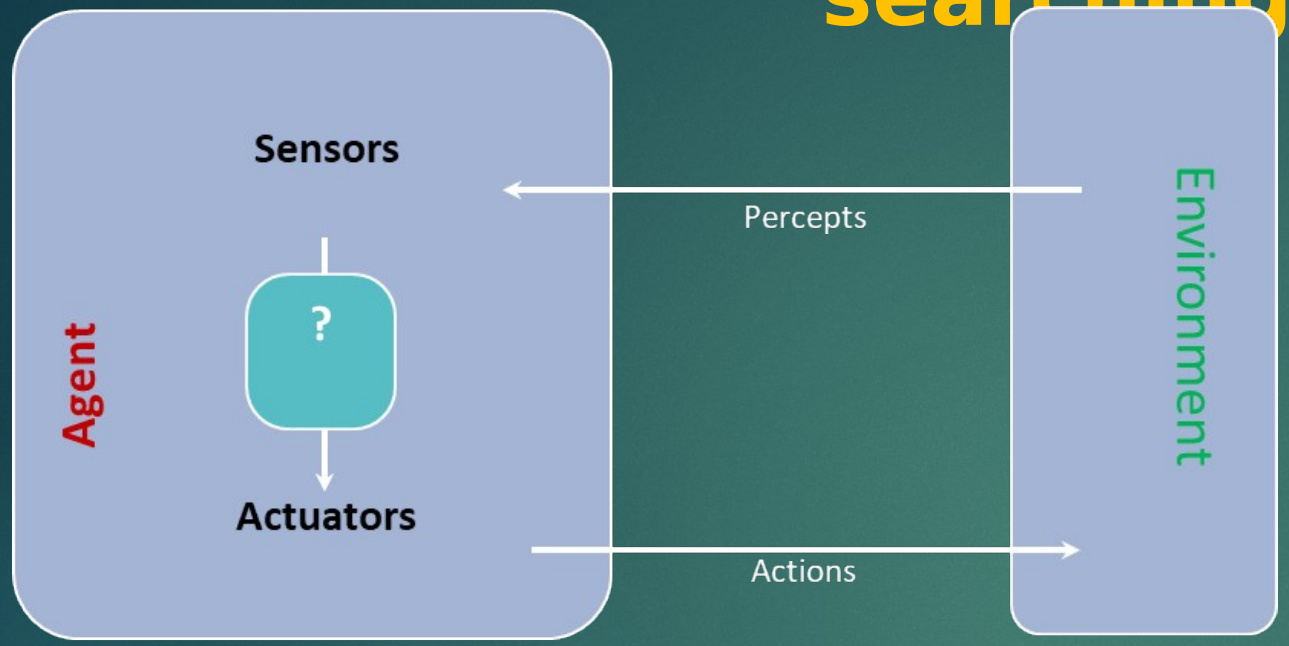
Computer Science

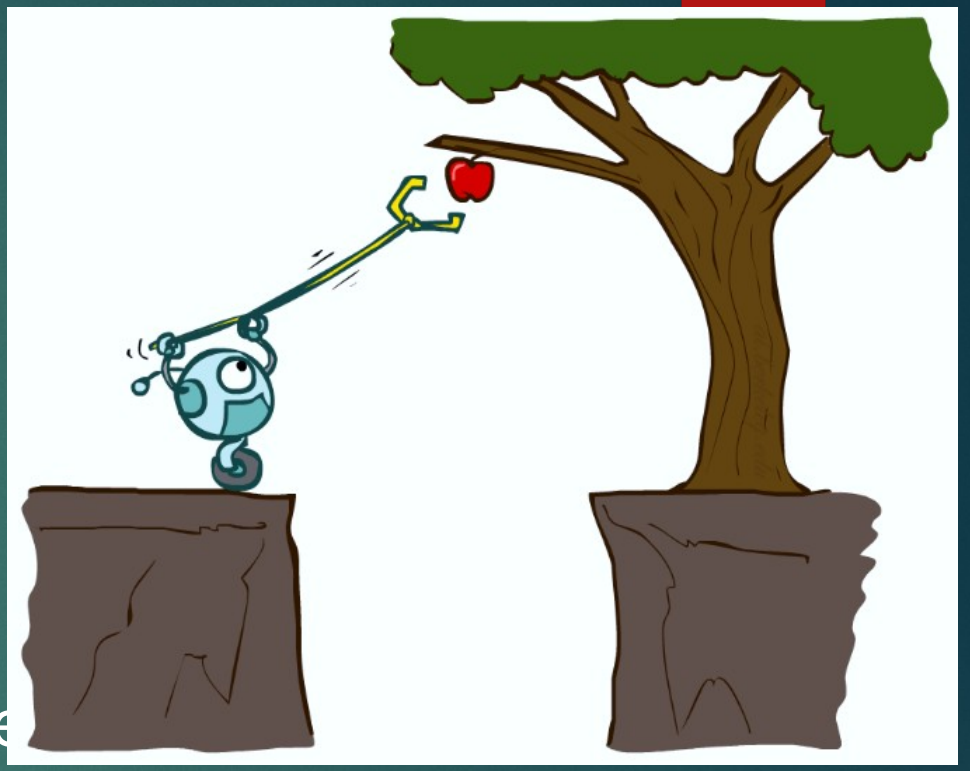


What can AI do today?

- Robotic vehicles
- Speech recognition
- Autonomous planning and scheduling
- Game playing
- Spam fighting
- Logistics planning
- Robotics
- Machine Translation

PROBLEM SOLVING-Solving problems by searching





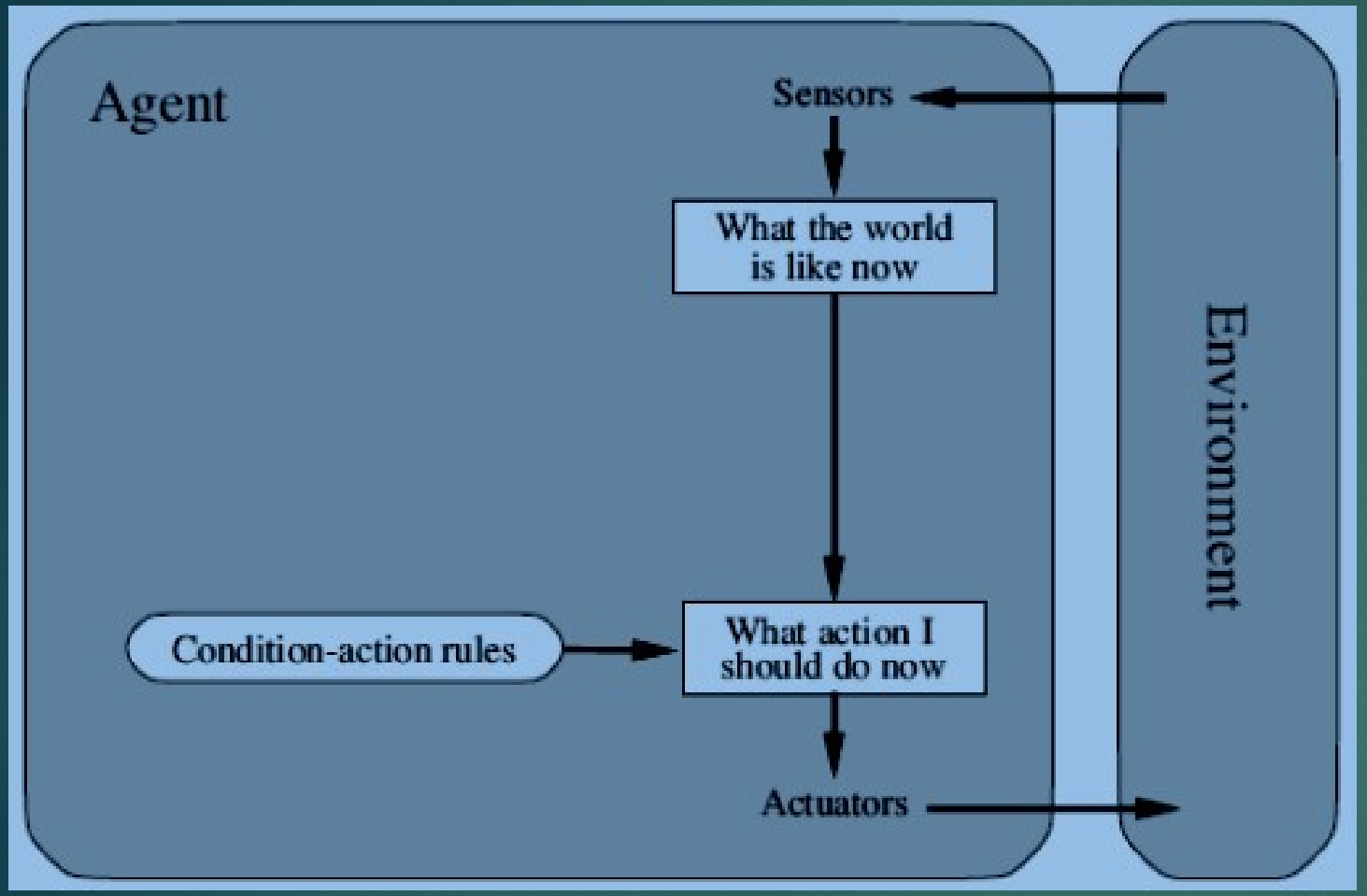
Problem-solving agent - Goal-based age

Goal formulation - is the first step in problem solving.

Problem formulation - is the process of deciding what actions and states to consider

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

Problem-solving agent



Schematic diagram of a simple reflex agent.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

- A simple reflex agent.
- It acts according to a rule whose condition matches the current state, as defined by the percept.

Examples of agent types and their PEAS descriptions

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Problem solving agents

- **The agent function** for an agent specifies the action taken by the agent in response to any percept sequence.
- **The performance measure** evaluates the behaviour of the agent in an environment.
- **An agent** is something that perceives and acts in an environment.
- **A rational agent** is one that does the right thing-conceptually speaking, every entry in the table for the agent function is filled out correctly. It acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.
- **A task environment** specification includes the performance measure, the external environment, the actuators, and the sensors.

  In designing an agent, the first step must always be to specify the task environment as fully as possible. They can be fully or partially observable, single-agent or multiagent, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, and known or unknown.
- **The agent program** implements the agent function. There exists a variety of basic agent-program designs reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.
- **Simple reflex agents** respond directly to percepts, whereas model-based reflex agents maintain internal state to track aspects of the world that are not evident in the current percept.
- **Goal-based agents** act to achieve their goals
- **Utility-based agents** try to maximize their own expected “happiness.”
- All agents can improve their performance through **learning**.

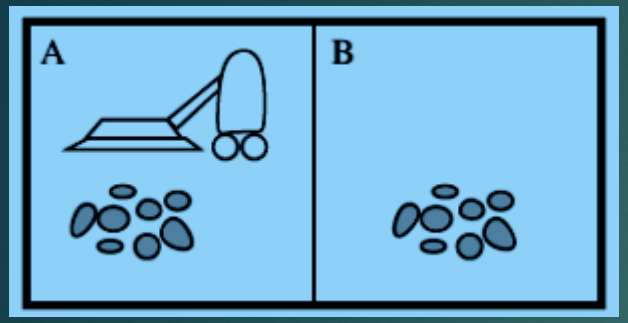
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  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
```

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

Toy problems - Vacuum world

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, 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**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **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.

Intelligent Agents



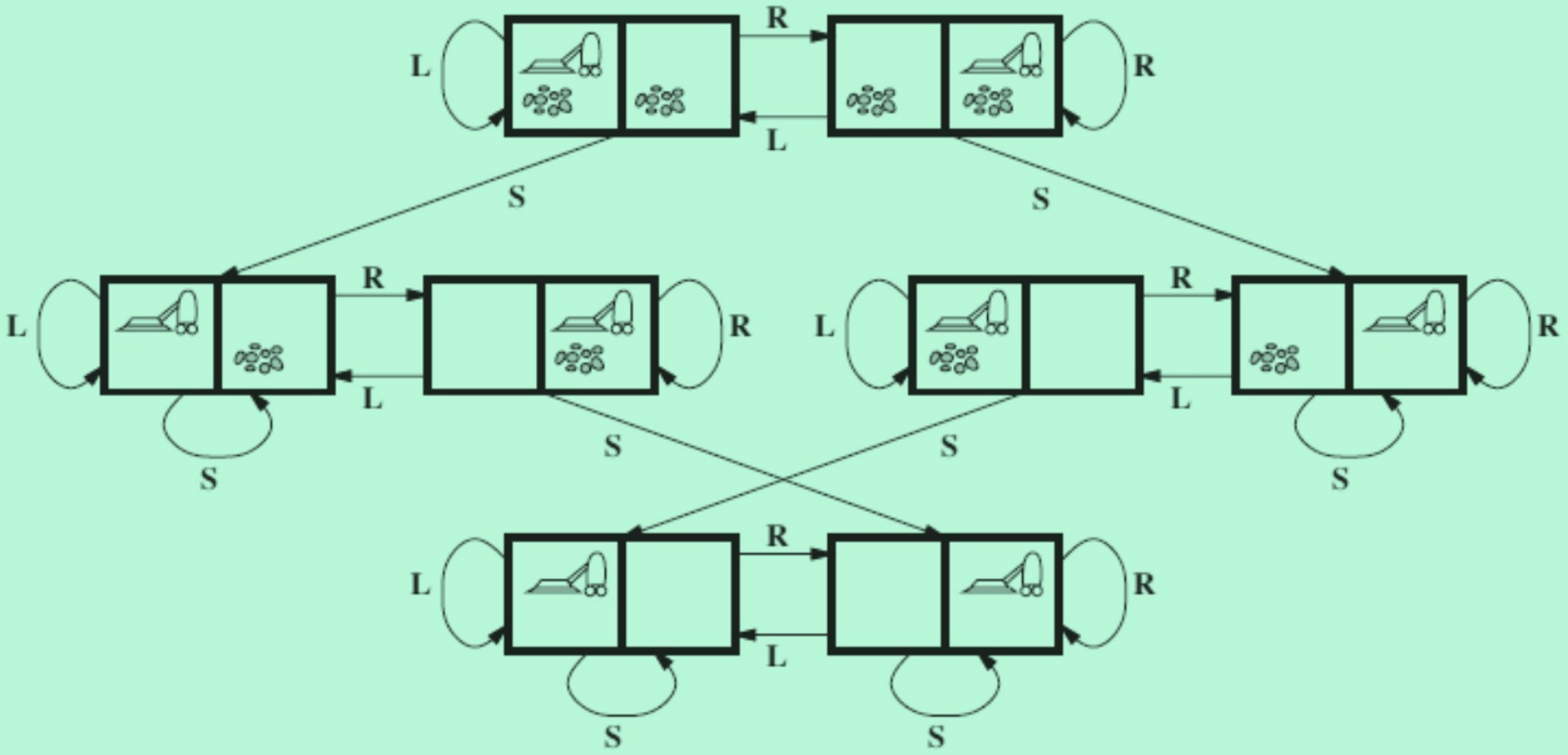
A vacuum-cleaner world with just two locations.

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
⋮	⋮

Partial tabulation of a simple agent function for the vacuum-cleaner world

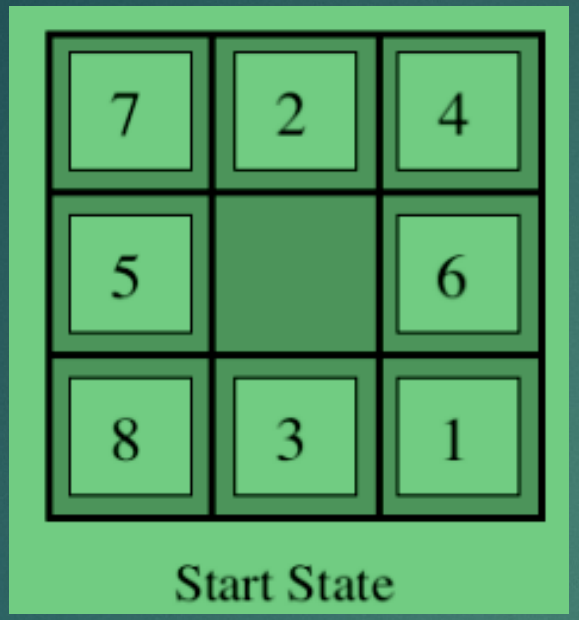
Example—the vacuum-cleaner world: This world is so simple that we can describe everything that happens. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing.

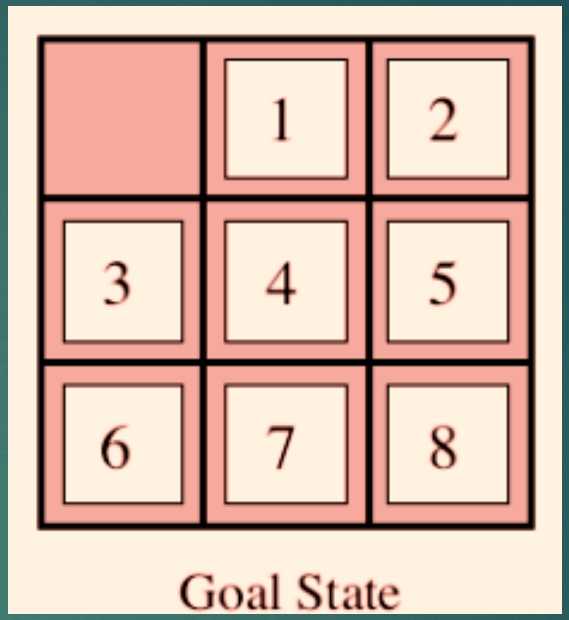
One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.



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

8-puzzle (Sliding-block puzzles)



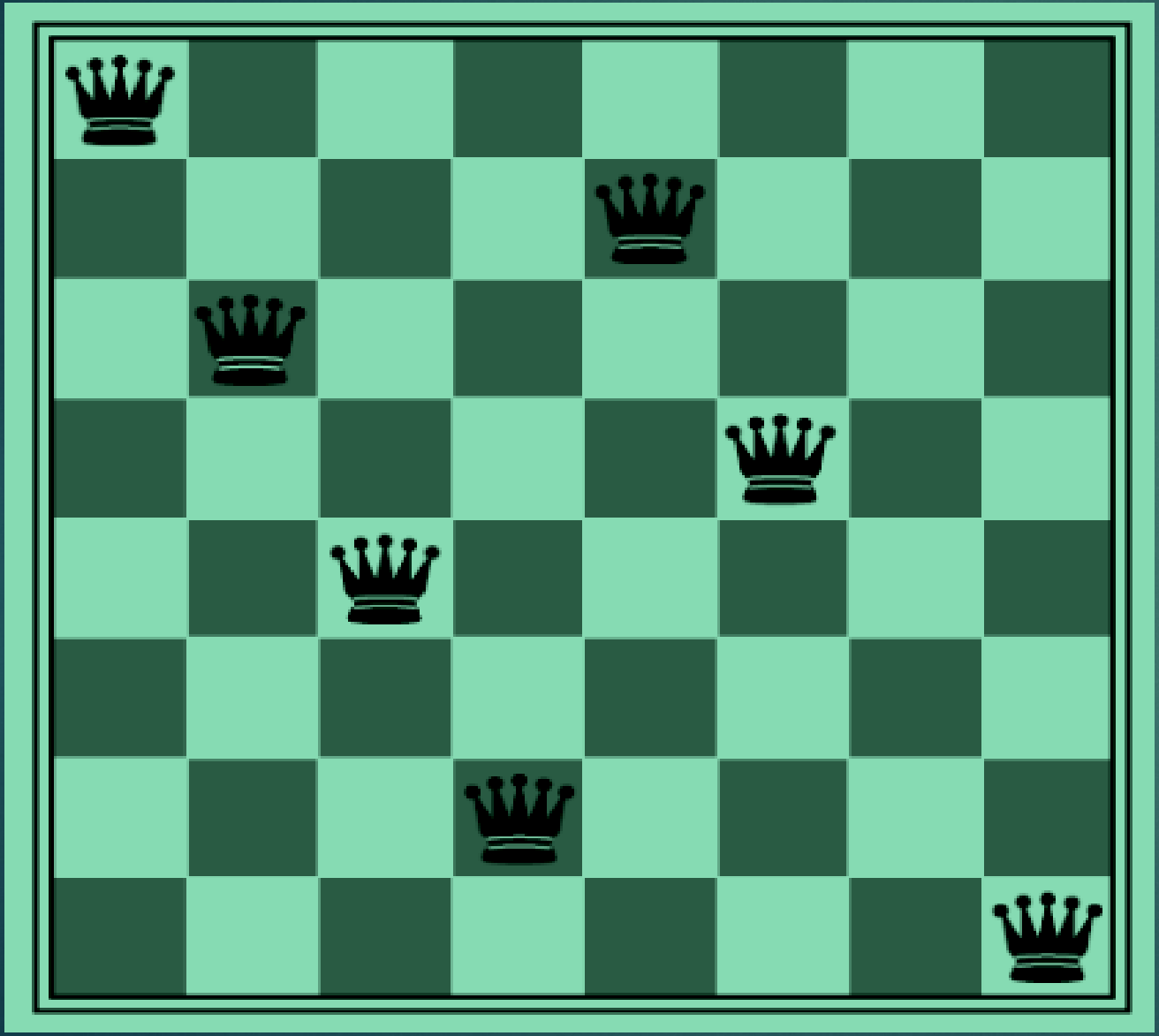


- Which are often used as test problems for new search algorithms in AI.
- An instance is shown in Figure.
- Consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The object is to reach a specified goal state, such as the one shown on the right of the figure.
- Sliding-block puzzles - NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described here.

8-puzzle: The standard formulation

- **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. Note that any given goal can be reached from exactly half of the possible initial states
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

8-queens problem



Goal: is to place eight queens on a chessboard such that no queen attacks any other.
(A queen attacks any piece in the same row, column or diagonal.)

An attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

8-queens problem

1. Incremental formulation

- Involves operators that *augment* the state description, starting with an empty state.
- Each action adds a queen to the state.

2. A complete-state formulation

- Starts with all 8 queens on the board and moves them around.
- In either case, the path cost is of no interest because only the final state counts.

The 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.

Real-world problems

Route-finding problem

- Web sites
- In-car systems that provide driving directions
- Routing video streams in computer networks
- Military operations planning
- Airline travel-planning systems

Touring problems

Traveling salesperson problem (TSP)

VLSI layout problem

Robot navigation

Automatic assembly sequencing

Airline travel problems solved by a travel-planning web site

States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.

Initial state: This is specified by the user’s query.

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.

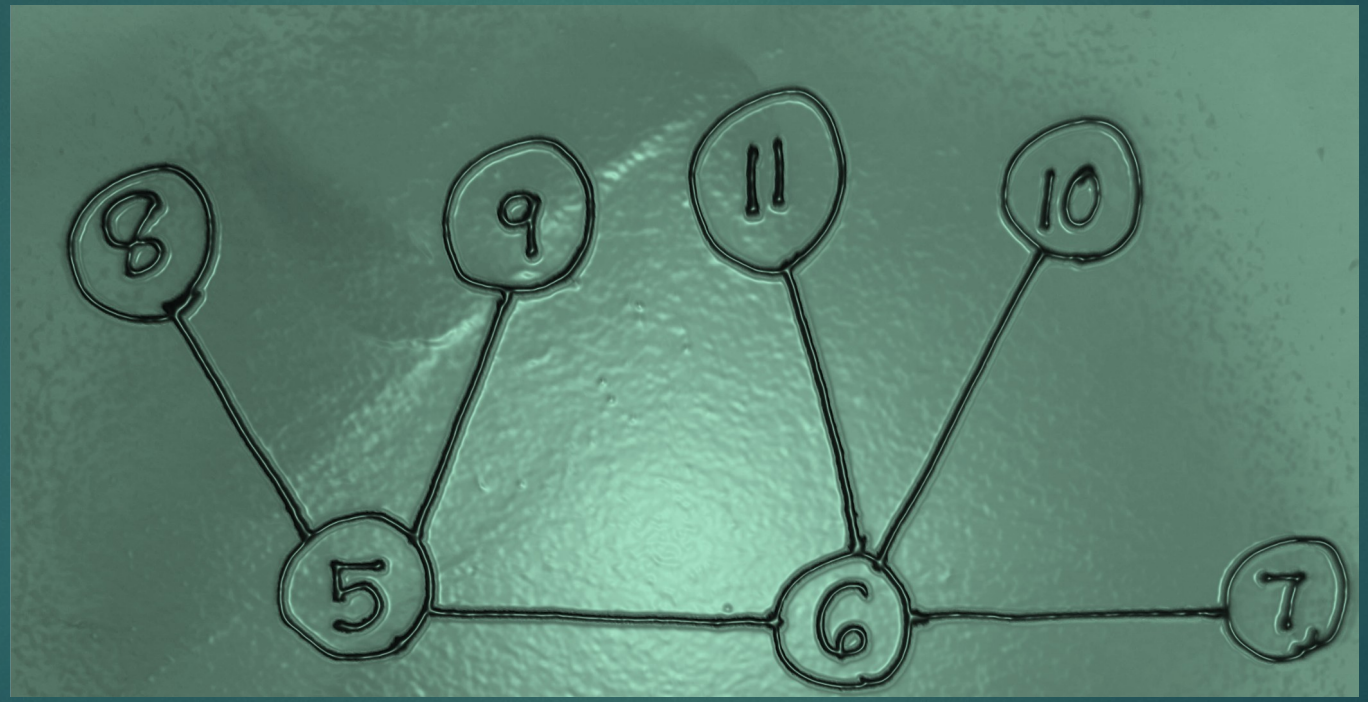
Transition model: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.

Goal test: Are we at the final destination specified by the user?

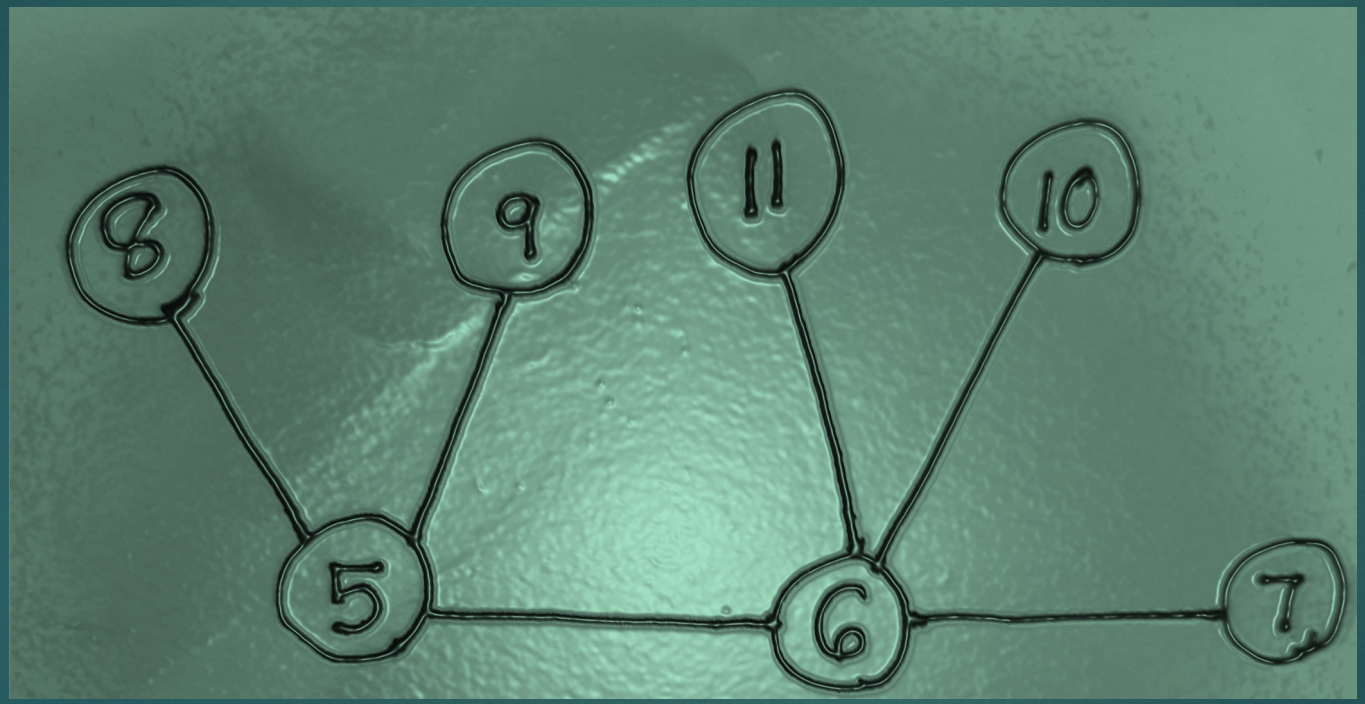
Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

BREADTH-FIRST-SEARCH (BFS)

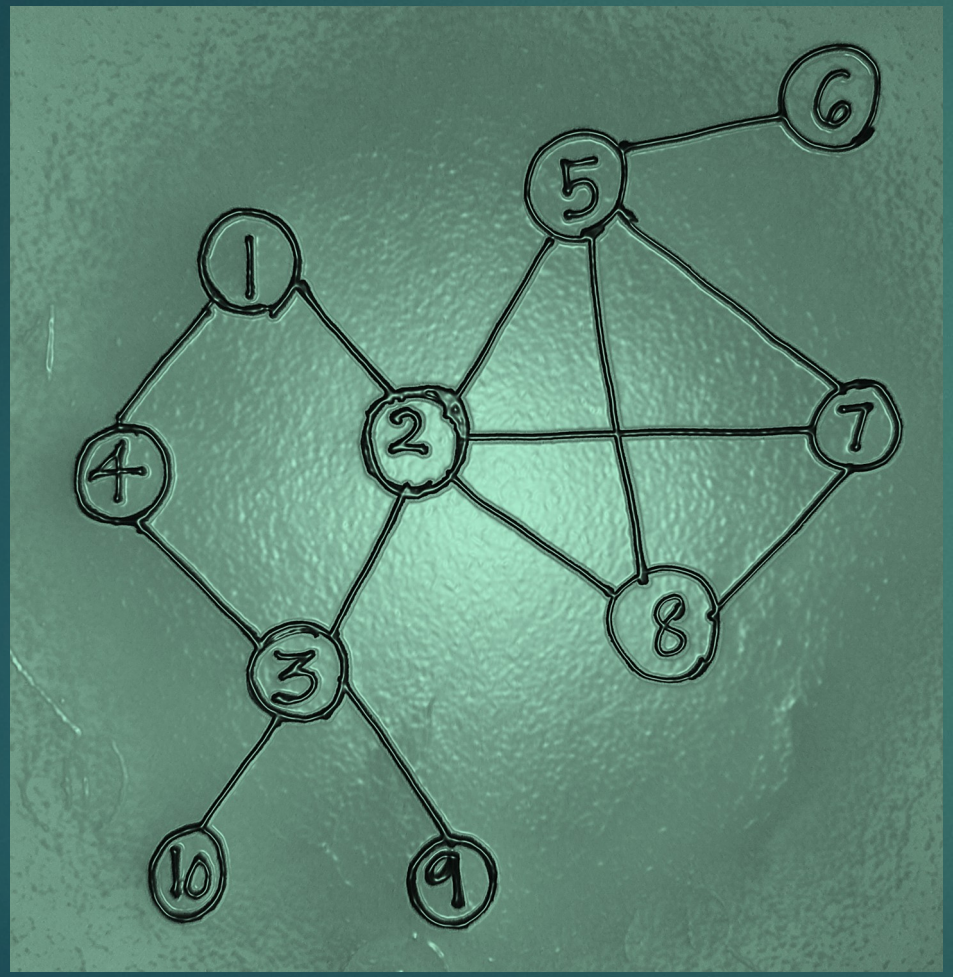
Visiting Vertex
Exploration of Vertex

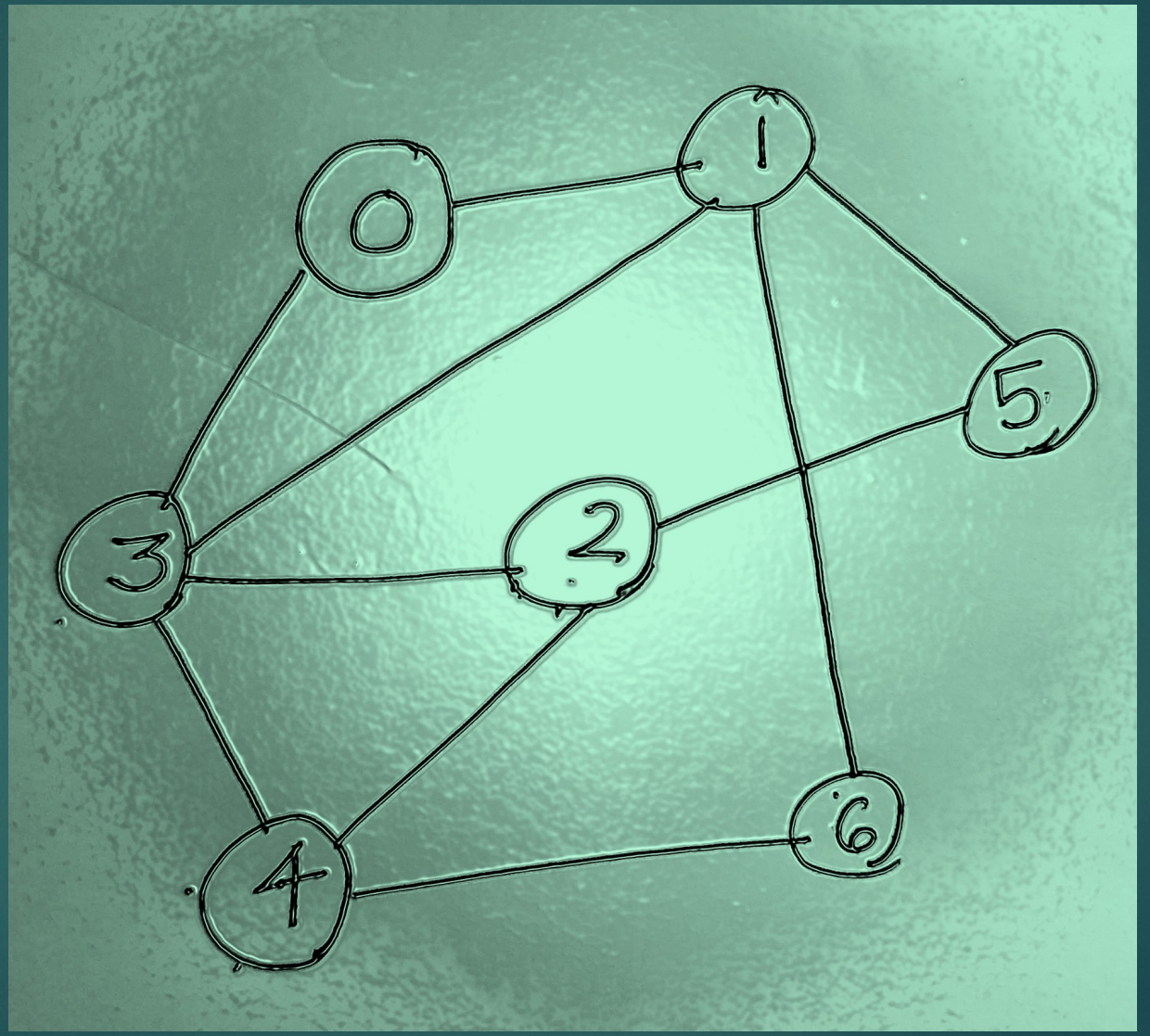


DEPTH-FIRST-SEARCH (DFS)

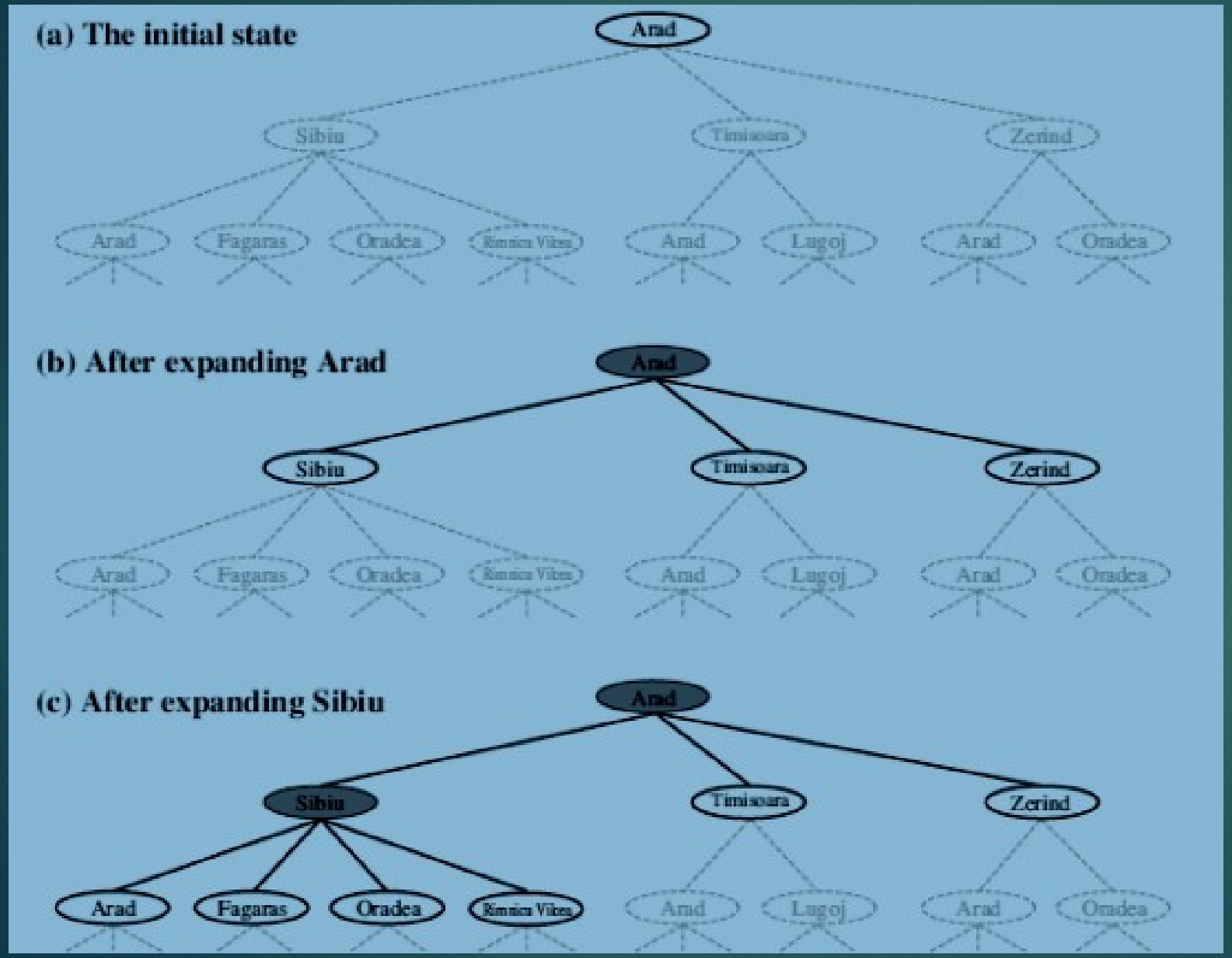


Perform BFS and DFS for the graph given below




0
1
2
3
4
5
6

Search Tree for finding Route



- 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.

An informal description of the general tree-search and graph-search algorithms.

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

---

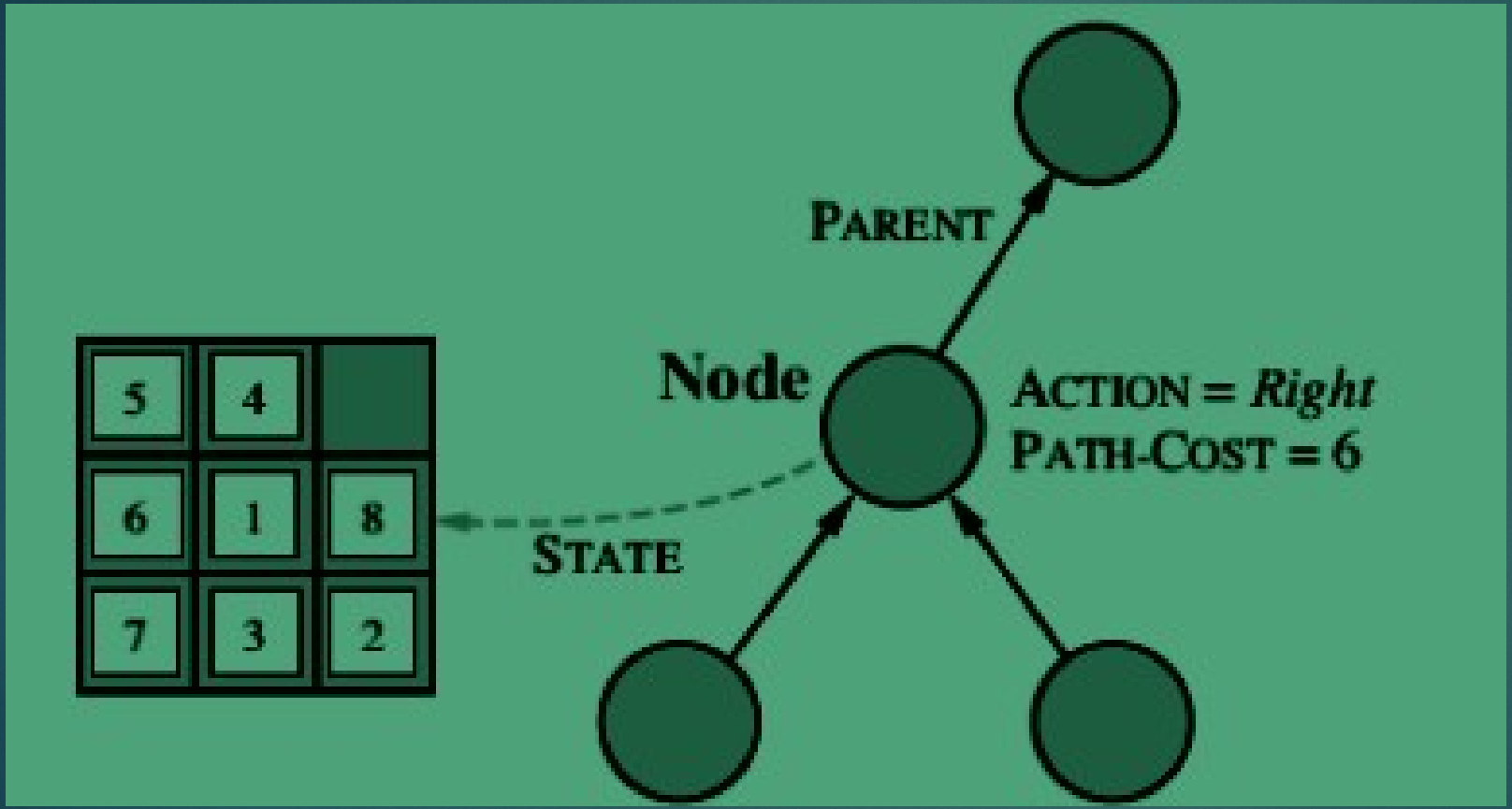
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the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

Infrastructure for search algorithms

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

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

Infrastructure for search algorithms (Cont'd)



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

The function CHILD-NODE takes a parent node and an action and returns the resulting child node.

```
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)
```

QUEUE

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a queue.

The operations on a queue are as follows:

- **EMPTY?(queue):** returns true only if there are no more elements in the queue.
- **POP(queue) :** removes the first element of the queue and returns it.
- **INSERT(element, queue):** inserts an element and returns the resulting queue.

Measuring problem-solving performance

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

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

OPTIMALITY: Does the strategy find the optimal solution?

TIME COMPLEXITY: How long does it take to find a solution?

SPACE COMPLEXITY: How much memory is needed to perform the search?

Search Strategies

1. Uninformed search (Blind search)
2. Informed search (Heuristic search)

1. Uninformed search (Blind search)

- **The strategies have no additional information about states beyond that provided in the problem definition.**
- **All they can do is generate successors and distinguish a goal state from a non-goal state.**
- **All search strategies are distinguished by the *order* in which nodes are expanded.**

E.g. 1: Breadth-first search

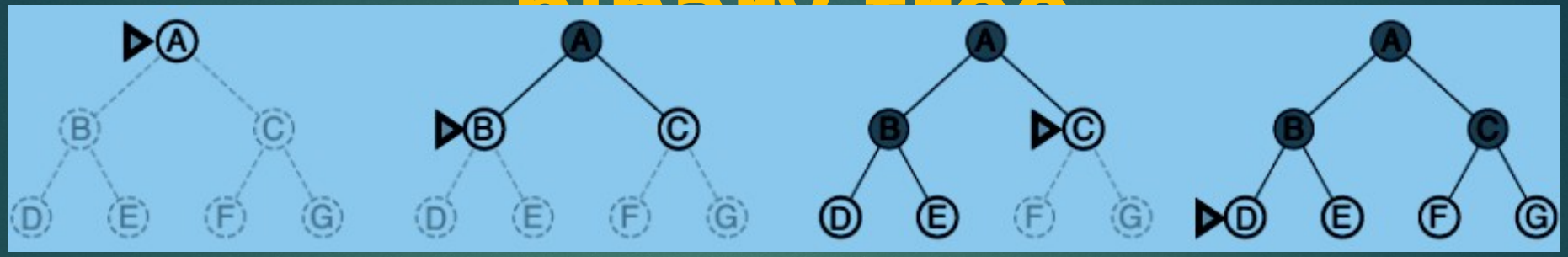
Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.

All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search on a graph

```
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)
```

Breadth-first search on a simple binary tree



- **At each stage, the node to be expanded next is indicated by a marker.**

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* $g(n)$.
- This is done by storing the frontier as a priority queue ordered by g.

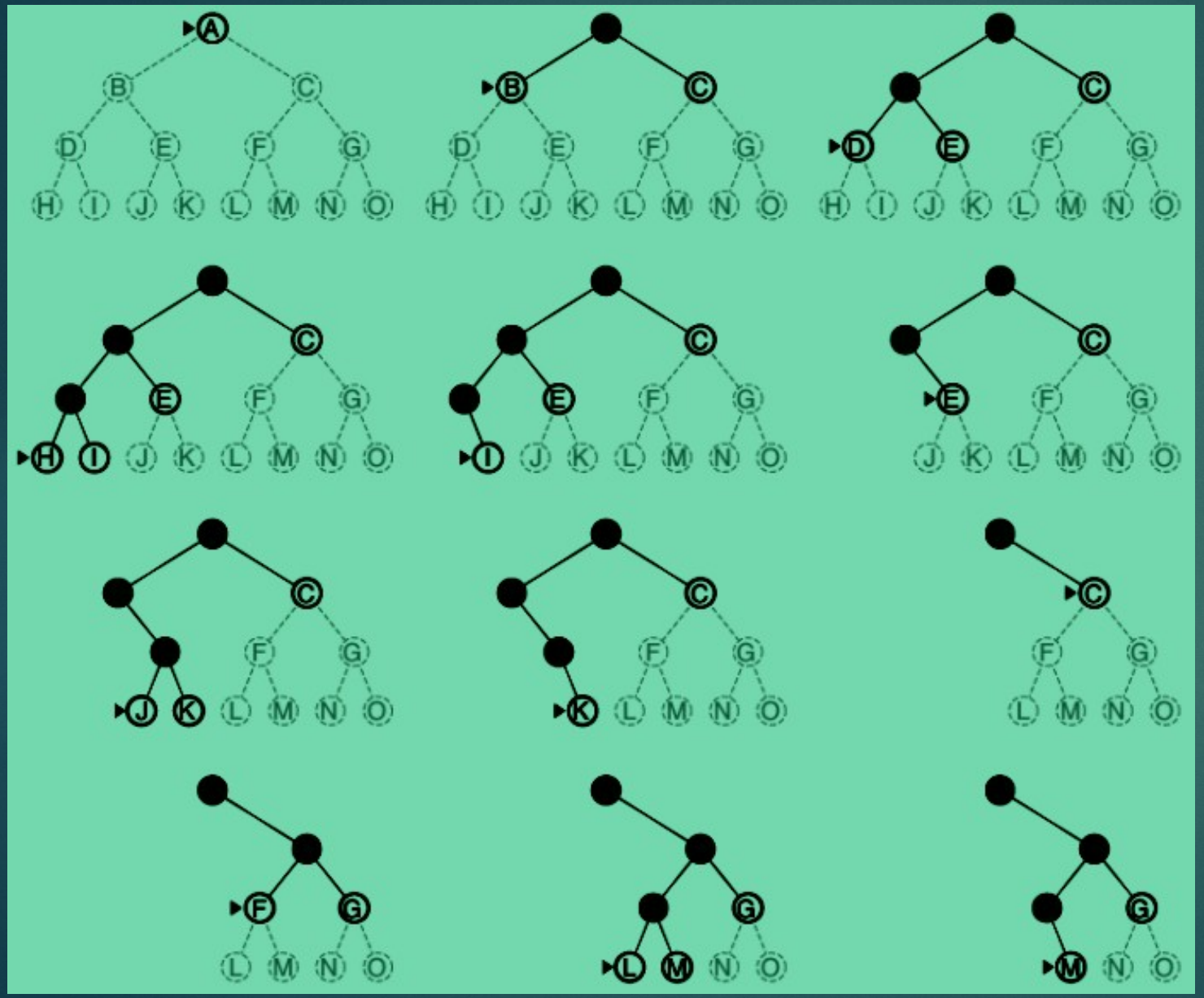
Uniform-cost search (Cont'd)

- Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost.
- Uniform-cost search is guided by path costs rather than depths

E.g. 2: Depth-first search

- **Depth-first search** 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.
- Breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- A LIFO queue means that the most recently generated node is chosen for expansion.
- This must be the deepest unexpanded node because it is one deeper than its parent —which, in turn, was the deepest unexpanded node when it was selected.

Depth-first search on a binary tree



- The unexplored region is shown in light Gray
- Explored nodes with no descendants in the frontier are removed from memory.
- Nodes at depth 3 have no successors and M is the only goal

Depth-limited search

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit.
- Nodes at depth are treated as if they have no successors.
- The depth limit solves the infinite-path problem.
- Depth-limited search will also be nonoptimal if we choose $l > d$.

Informed search (Heuristic search)

- Strategies that know whether one non-goal state is “more promising” than another are called **informed search** or **heuristic search** strategies.
- one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.

Best-first Search

- The general approach we consider is called **best-first search**.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm
- 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 implementation of best-first graph search is identical to that for uniform-cost search, except for the use of *f* instead of *g* to order the priority queue.
- 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

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, $f(n) = h(n)$.

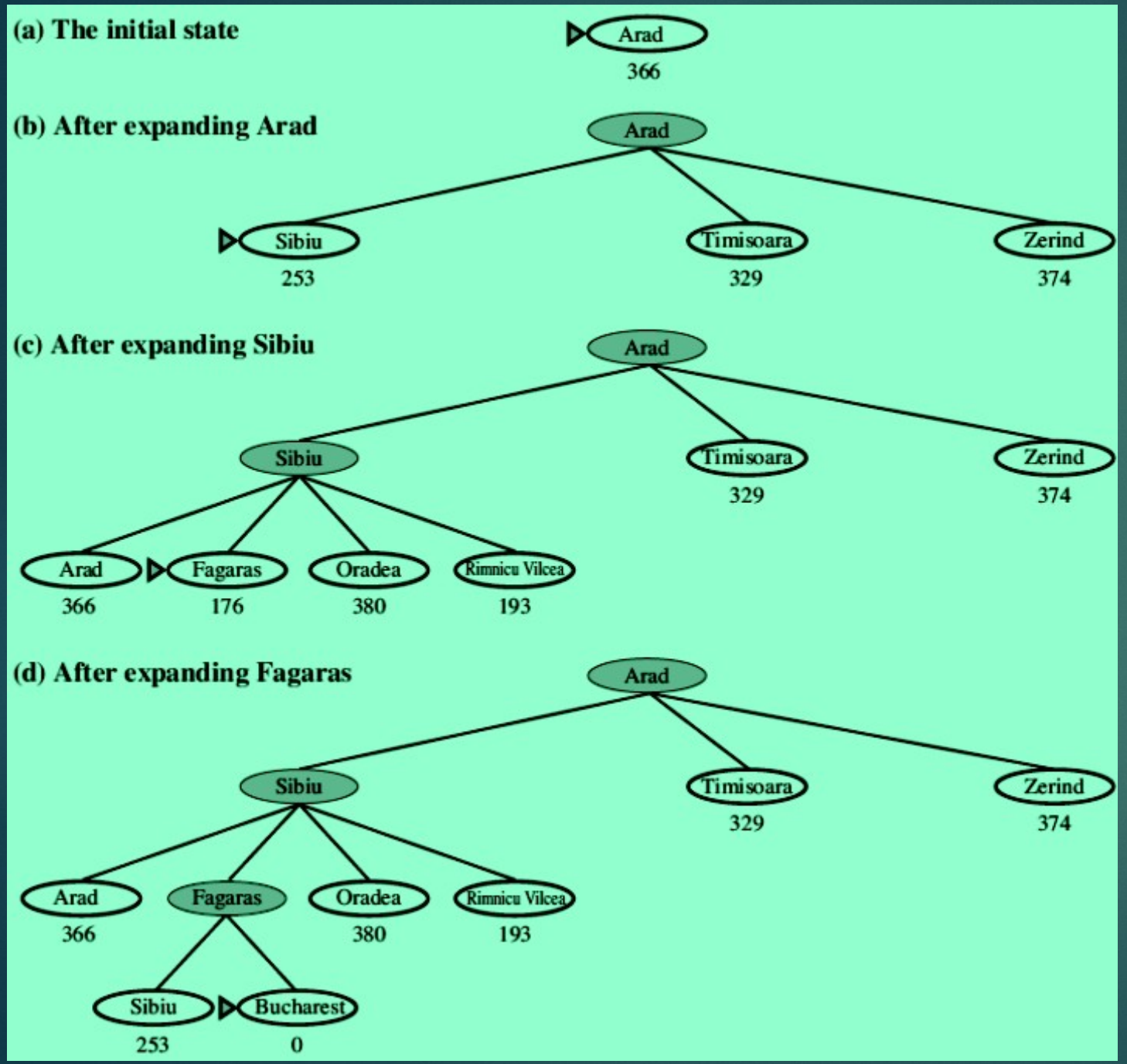
STRAIGHT- LINE DISTANCE HEURISTIC

- Route-finding problems in Romania - can use the straight line distance heuristic, h_{SLD}.
- If the goal is Bucharest, we need to know the straight-line distances to Bucharest. For example, h_{SLD}(In(Arad)) = 366.

Greedy best-first search using h_{SLD} to find a path from Arad to Bucharest

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

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



Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic hSLD.

- Nodes are labelled with their h-values.
- Figure shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras because it is closest.
- Fagaras in turn generates Bucharest, which is the goal.
- For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on

A* search: Minimizing the total estimated solution cost

- The most widely known form of best-first search is called **A * search.**
- 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:

$$f(n) = g(n) + h(n)$$

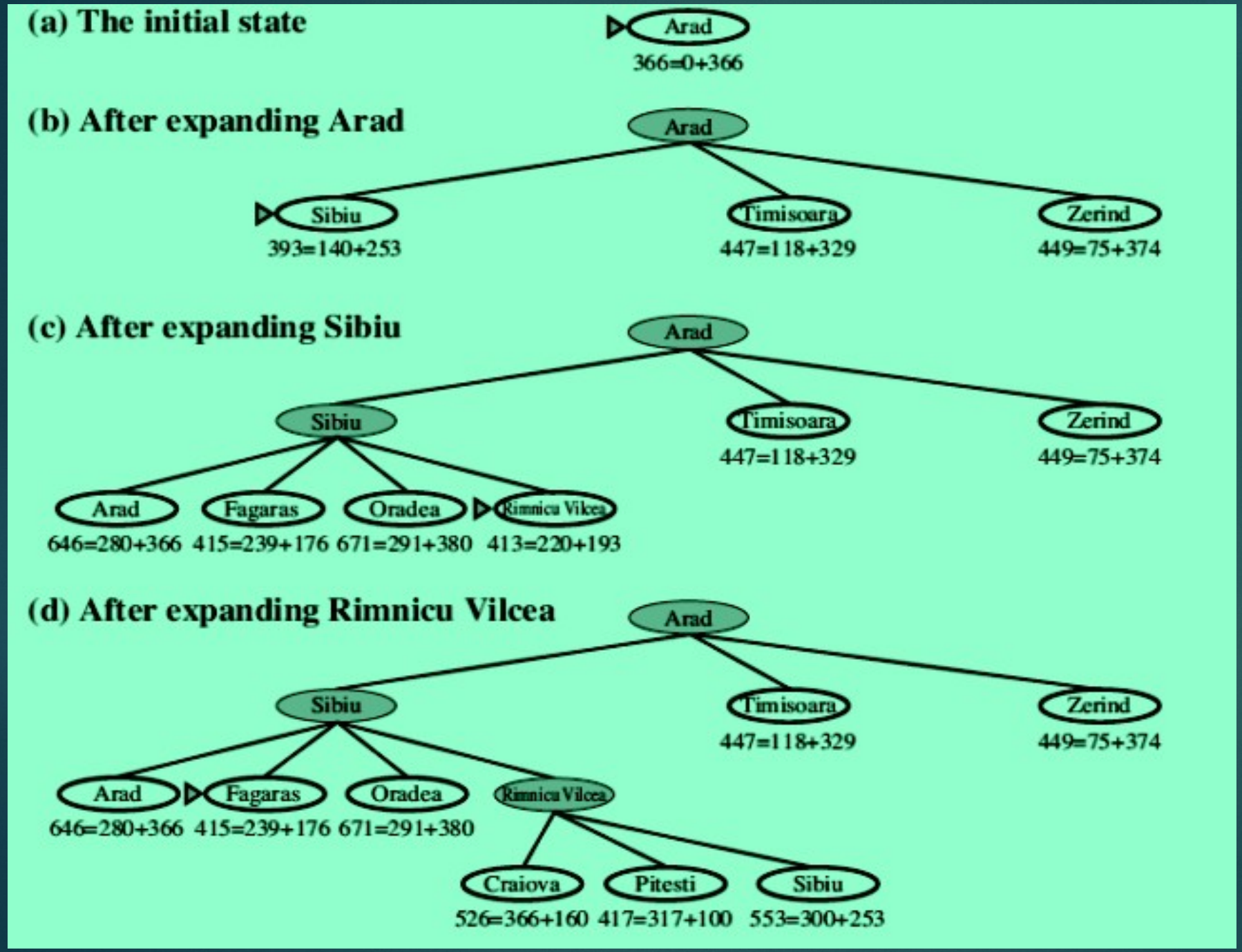
g(n) gives the path cost from the start node to node *n*,

h(n) is the estimated cost of the cheapest path from *n* to the goal,

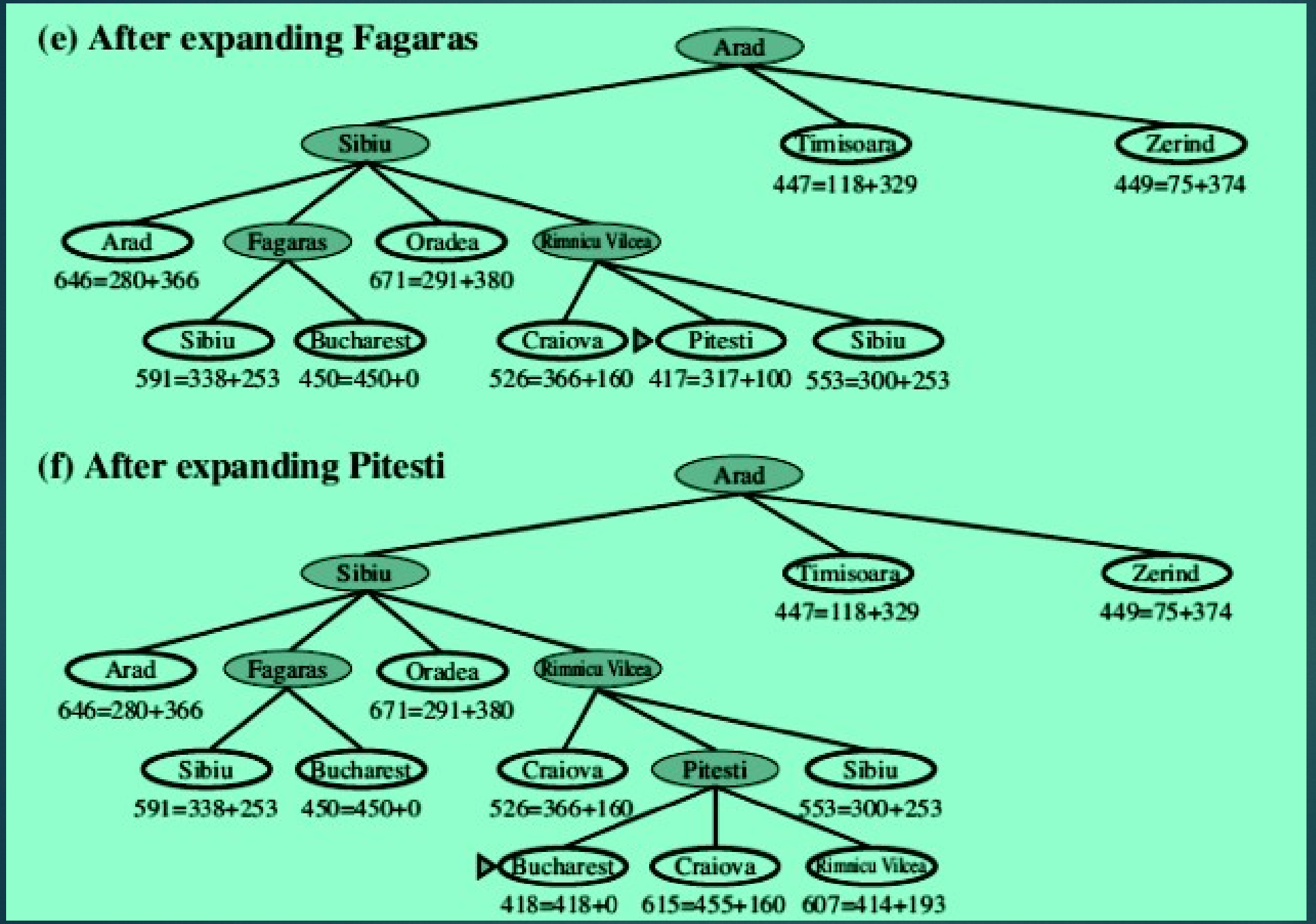
f(n) = estimated cost of the cheapest solution through *n*

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of *g(n)* + *h(n)*.
- It turns out that this strategy is more than just reasonable: provided that the heuristic function *h(n)* satisfies certain conditions

A * search is **both complete and optimal.**

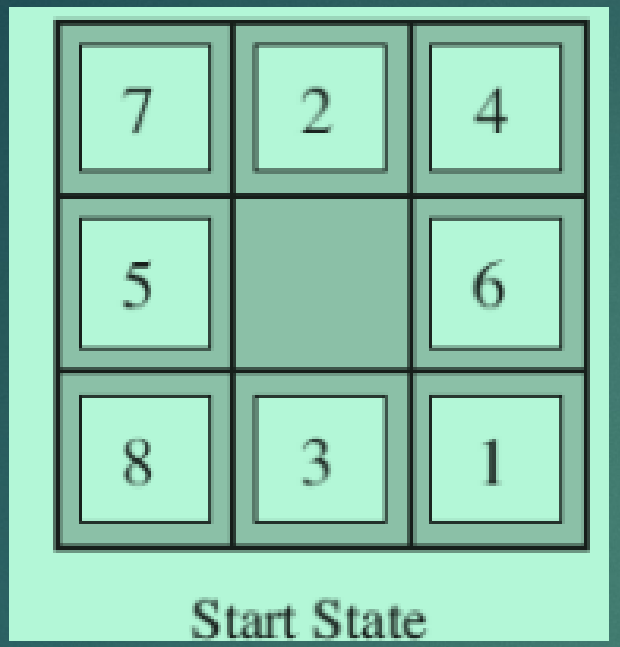


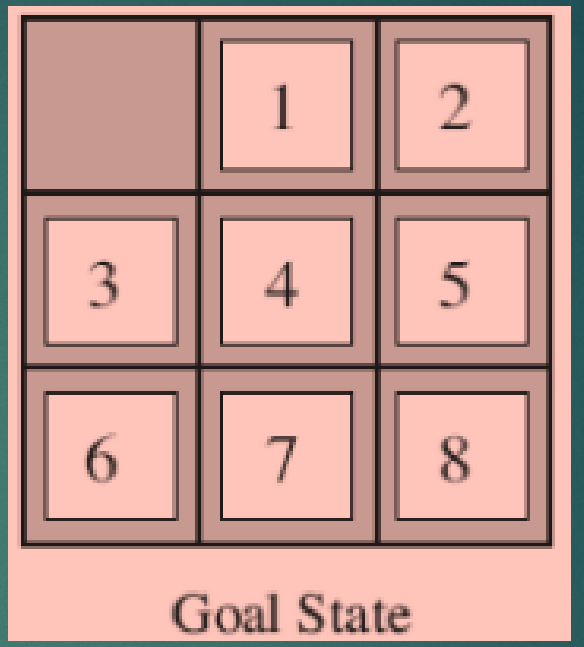
- Stages in an A* search for Bucharest.
- Nodes are labeled with $f = g + h$.
- The h values are the straight-line distances to Bucharest



- Stages in an A* search for Bucharest.
- Nodes are labeled with $f = g + h$.
- The h values are the straight-line distances to Bucharest

Heuristic functions





A typical instance of the 8-puzzle. The solution is 26 steps long.

Heuristics for the 8-puzzle

- Object of the puzzle - slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.

Heuristic functions

- Two commonly used candidates (heuristics) for the 8-puzzle
  - **h_1 = the number of misplaced tiles.**

For Figure given in above all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

- **h_2 = the sum of the distances of the tiles from their goal positions.**

Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**.

h_2 is also admissible because all any move can do is move one tile one step closer to the goal.

Tiles 1 to 8 in the start state give a Manhattan distance of

$h_2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18.$

As expected, neither of these overestimates the true solution cost, which is 26.

References:

Stuart Russell and Peter Norvig, "Artificial intelligence-a modern approach 3rd ed." (2016).

ARTIFICIAL INTELLIGENCE

INTERIM SEMESTER 2021-22 BPL

CSE3007-LT-AB306

FACULTY: SIMI V.R.

Adversarial search

- **Competitive** environments
- which the agents' goals are in conflict
- giving rise to **adversarial search** problems—often known as **games**.
- Games, are interesting *because* they are too hard to solve.
- Optimal move and an algorithm for finding it
- **Pruning** allows us to ignore 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.

Games

A game can be formally defined as a kind of search problem with the following elements:

- 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 . Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.

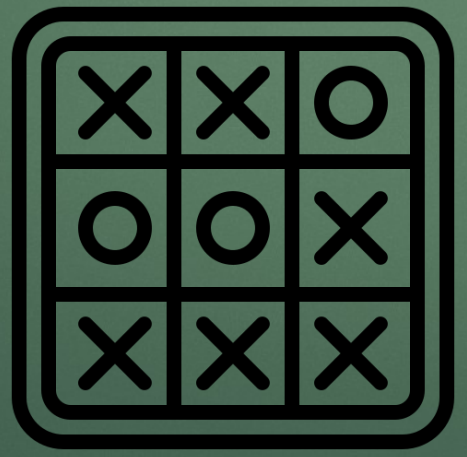
Game tree

- A **zero-sum game** is defined as one 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 + .

Game Tree

- The initial state, **ACTIONS** function, and **RESULT** function define the **game tree** for the game.
- The nodes are game states and the edges are moves.

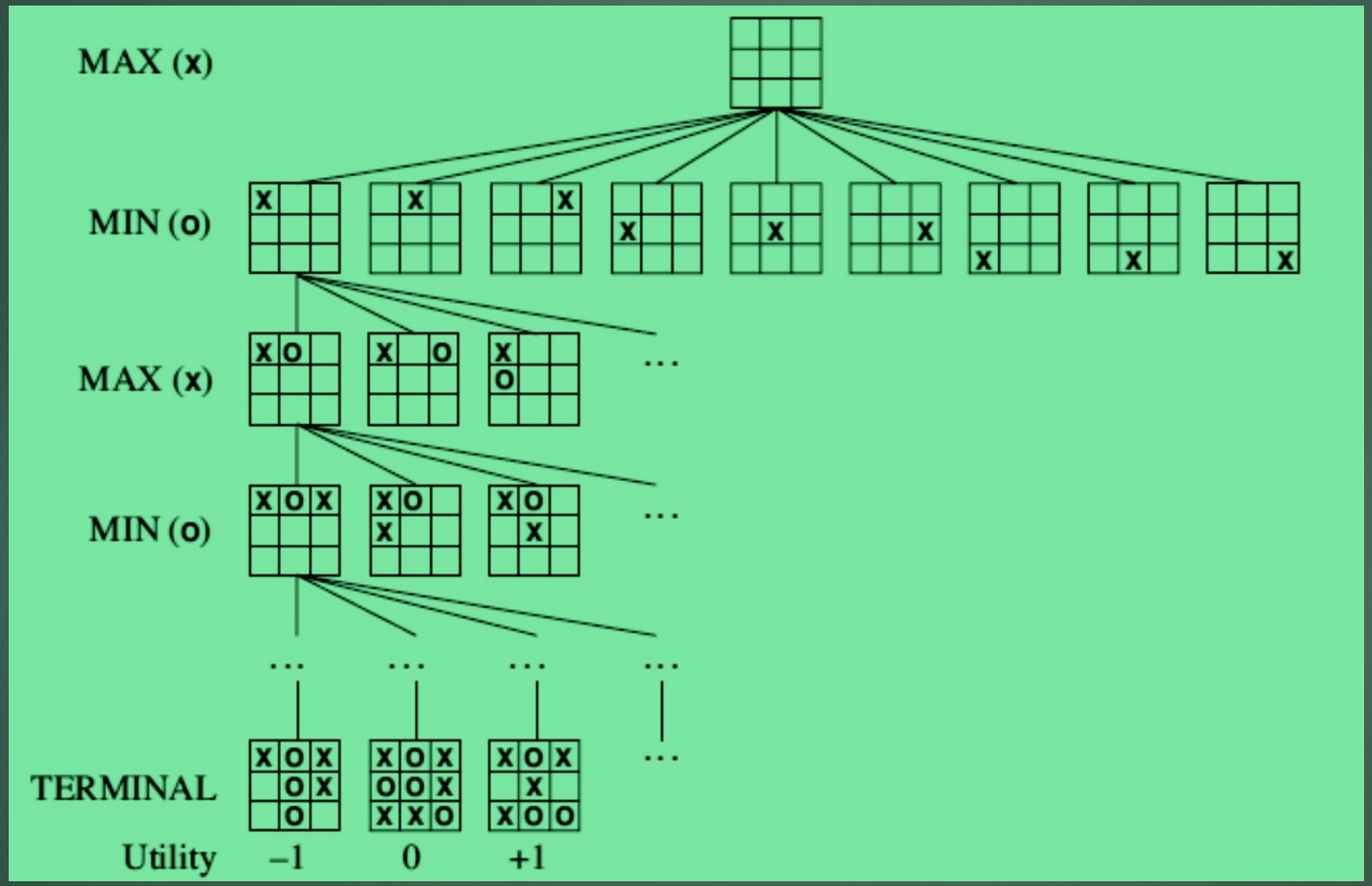
Game tree for tic-tac-toe (noughts and crosses)



Game tree for tic-tac-toe (Cont'd)

- From the initial state, MAX has nine possible moves.
- Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).
- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.

A (partial) game tree for the game of tic-tac-toe.



- The top node is the initial state, and MAX moves first, placing an X in an empty square.
- We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Optimal decisions in games

Two-ply game tree

- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.
- The possible moves for MAX at the root node are labelled *a1*, *a2*, and *a3*. The possible replies to *a1* for MIN are *b1*, *b2*, *b3*, and so on.
- This particular game ends after one move each by MAX and MIN.
- The utilities of PLY the terminal states in this game range from 2 to 14.

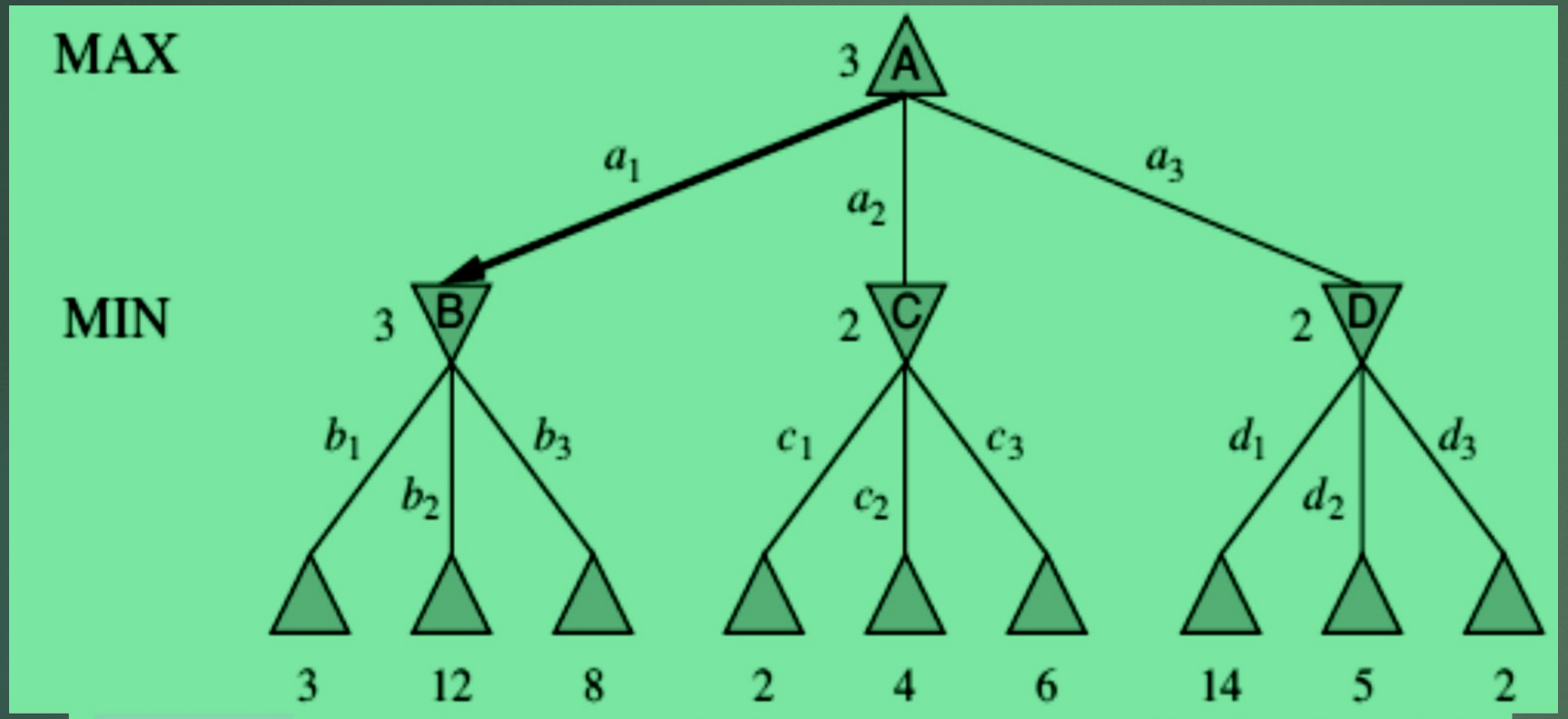
PLY MINIMAX VALUE

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(*n*).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.
- The minimax value of a terminal state is just its utility.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

- MINIMAX algorithm – Back tracking algorithm
- Best move strategy

A two-ply game tree



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 terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labelled B, has three successor states with values 3, 12, and 8, so its minimax value is 3.
- Similarly, the other two MIN nodes have minimax value 2.
- The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.
- MINIMAX DECISION at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

The minimax algorithm

- The **minimax algorithm** computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- 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.
- For example, in Figure given above, the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively.
- Then it takes the minimum of these values, 3, and returns it as the backed up value of node B.
- A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.
- The minimax algorithm performs a complete depth-first exploration of the game tree.

The minimax algorithm (cont'd)

```
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
```

An algorithm for calculating minimax decisions.

- It returns the action corresponding
- to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility.
- The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

$\text{argmax}_{a \in S} f(a)$

The notation computes the element a of set S that has the maximum value of *f(a)*.

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.
- Unfortunately, we can’t eliminate the exponent, but it turns out we can effectively cut it in half.
- **Pruning** eliminate large parts of the tree from consideration.

ALPHA–BETA

- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

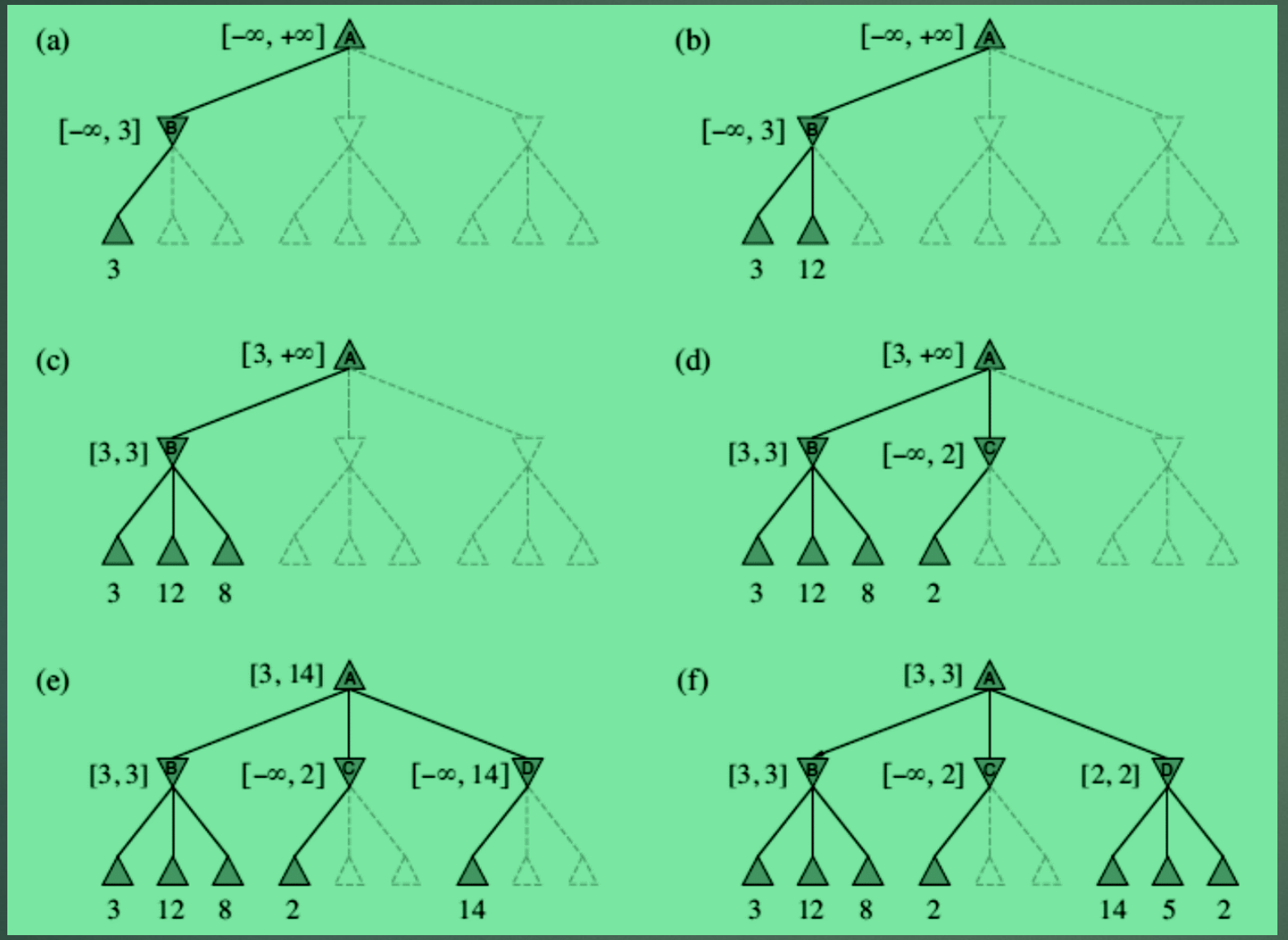


Figure 1: Stages in the calculation of the optimal decision for the game tree

At each point, we show the range of possible values for each node.

(a)The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3.

(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.

(c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3.

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning.

(e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

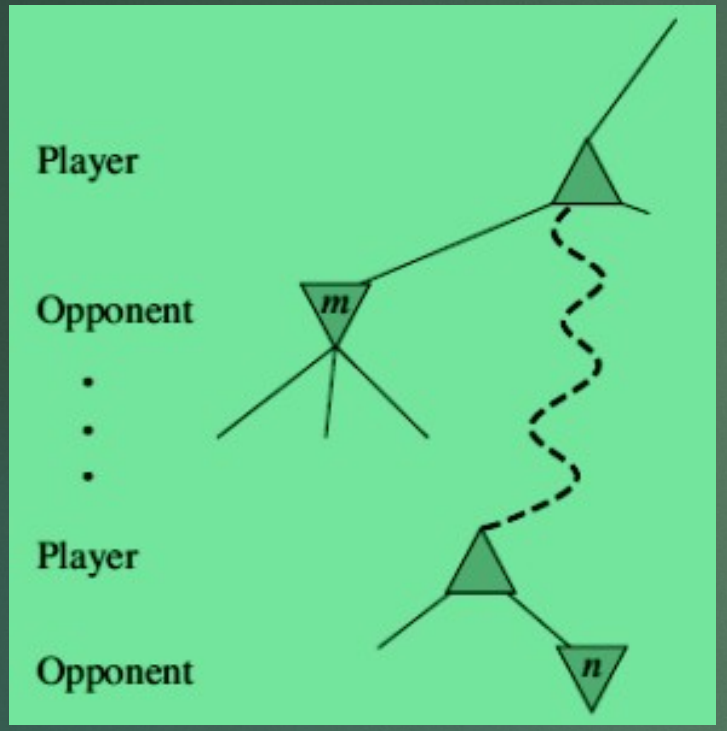
(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

The value of the root node is given by

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

The value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

General Case for alpha-beta pruning



If m is better than n for Player, we will never get to n in play.

Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively

```
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
```

The alpha–beta search algorithm.

Notice that these routines are the same as the MINIMAX functions except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Knowledge Representation

KNOWLEDGE BASE

- The central component of a knowledge-based agent is its **knowledge base**, or KB.
- A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.)

KNOWLEDGE REPRESENTATION LANGUAGE

- Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.
- Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

INFERENCE

- There must be a way to add new sentences to the knowledge base and a way to query what is known.
- The standard names for these operations are TELL and ASK, respectively.
- Both operations may involve **inference**—that is, deriving new sentences from old.
- Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told to the knowledge base previously.

A generic knowledge-based agent

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

The details of the representation language are hidden inside **three functions** that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other.

- **MAKE-PERCEPT-SENTENCE** constructs a sentence asserting that the agent perceived the given percept at the given time.
- **MAKE-ACTION-QUERY** constructs a sentence that asks what action should be done at the current time.
- **MAKE-ACTION-SENTENCE** constructs a sentence asserting that the chosen action was executed.

Logic

- Logic is used to represent simple facts.
- Logic defines the ways of putting symbols together to form sentences that represent facts.
- Sentences are either true or false but not both are called propositions.

Examples :

Sentence	Truth value	Is it a Proposition ?
"Grass is green"	"true"	Yes
"2 + 5 = 5"	"false"	Yes
"Close the door"	-	No
"Is it hot out side?"	-	No
"x > 2"	-	No (since x is not defined)
"x = x"	-	No (don't know what is "x" and "=" mean; "3 = 3" or say "air is equal to air" or "Water is equal to water" has no meaning)

Propositional Logic (PL)

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} &\rightarrow \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} &\rightarrow (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
&\mid \neg\, \mathit{Sentence} \\
&\mid \mathit{Sentence} \wedge \mathit{Sentence} \\
&\mid \mathit{Sentence} \vee \mathit{Sentence} \\
&\mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
&\mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Propositional Logic (PL) (Cont'd)

- **A proposition is a statement - which in English is a declarative sentence and Logic defines the ways of putting symbols together to form sentences that represent facts.**
- **Every proposition is either true or false.**
- **Propositional logic is also called Boolean algebra.**

Examples: (a) The sky is blue., (b) Snow is cold. , (c) 12 * 12=144

Propositional logic : It is fundamental to all logic.

- Propositions are "Sentences"; either true or false but not both.
- A sentence is smallest unit in propositional logic
- f proposition is true, then truth value is "true"; else "false"

Example: Sentence "Grass is green";

Truth value " true";

Proposition "yes"

Statement, Variables and Symbols

Statement : A simple statement is one that does not contain any other statement as a part. A compound statement is one that has two or more simple statements as parts called components.

Operator or connective : Joins simple statements into compounds, and joins compounds into larger compounds.

Symbols for connectives

assertion	P					"p is true"
nagation	¬p	~	!		NOT	"p is false"
conjunction	p ∧ q	·	&&	&	AND	"both p and q are true"
disjunction	P v q	||	|		OR	"either p is true, or q is true, or both "
implication	p → q	⊃	⇒		if . . then	"if p is true, then q is true" " p implies q "
equivalence	↔	≡	⇔		if and only if	"p and q are either both true or both false"

Truth table

- **Is a convenient way of showing relationship between several propositions.**

The truth table for negation, conjunction, disjunction, implication and equivalence are shown below.

p	q	¬p	¬q	p ∧ q	p v q	p→q	p ↔ q	q→p
T	T	F	F	T	T	T	T	T
T	F	F	T	F	T	F	F	T
F	T	T	F	F	T	T	F	F
F	F	T	T	F	F	T	T	T

Tautology

A Tautology is proposition formed by combining other propositions (p, q, r, . . .) which is true regardless of truth or falsehood of p, q, r, ...

The important tautologies are :

$(p \rightarrow q) \leftrightarrow \neg\, [p \wedge (\neg q)]$ and $(p \rightarrow q) \leftrightarrow (\neg p) \vee q$

A proof of these tautologies, using the truth tables are given below.

Tautologies $(p \rightarrow q) \leftrightarrow \neg\, [p \wedge (\neg q)]$ and $(p \rightarrow q) \leftrightarrow (\neg p) \vee q$

Table Proof of Tautologies

p	q	p→q	¬q	p ∧ (¬q)	¬ [p ∧ (¬q)]	¬p	(¬p) ∨ q
T	T	T	F	F	T	F	T
T	F	F	T	T	F	F	F
F	T	T	F	F	T	T	T
F	F	T	T	F	T	T	T

Predicate Logic

- **Predicate Logic/ Calculus extends the syntax of propositional calculus with predicates and quantifiers:**
- **P(X) – P is a predicate.**
- **First Order Predicate Logic allows predicates to apply to objects or terms, but not functions or predicates.**

- ∀ - For all:
  - ∀xP(x) is read "For all x'es, P (x) is true".
- ∃ - There Exists:
  - ∃x P(x) is read "there exists an x such that P(x) is true".
- Relationship between the quantifiers:
  - ∃x P(x) ≡ ¬(∀x)¬P(x)
  - "If There exists an x for which P holds, then it is not true that for all x P does not hold".

References:

Stuart Russell and Peter Norvig, "Artificial intelligence-a modern approach 3rd ed." (2016).

S. Rajasekaran and G.A. Vijayalaksmi Pai ,Neural Network, Fuzzy Logic, and Genetic Algorithms - Synthesis and Applications, (2005), Prentice Hall.

Artificial Intelligence

INTERIM SEMESTER 2021-22 BPL

CSE3007-LT-AB306

FACULTY: SIMI V.R.

Machine Learning

Machine Learning Problems

- **Spam Detection**: Given email in an inbox, identify those email messages that are spam and those that are not. Having a model of this problem would allow a program to leave non-spam emails in the inbox and move spam emails to a spam folder.
- **Credit Card Fraud Detection**: Given credit card transactions for a customer in a month, identify those transactions that were made by the customer and those that were not. A program with a model of this decision could refund those transactions that were fraudulent.
- **Digit Recognition:** Given a zip codes hand written on envelops, identify the digit for each hand written character. A model of this problem would allow a computer program to read and understand handwritten zip codes and sort envelops by geographic region.
- **Speech Understanding:** Given an utterance from a user, identify the specific request made by the user. A model of this problem would allow a program to understand and make an attempt to fulfil that request. The iPhone with Siri has this capability.
- **Face Detection:** Given a digital photo album of many hundreds of digital photographs, identify those photos that include a given person. A model of this decision process would allow a program to organize photos by person. Some cameras and software like iPhoto has this capability.

Machine Learning Problems

- **Product Recommendation**: Given a purchase history for a customer and a large inventory of products, identify those products in which that customer will be interested and likely to purchase. A model of this decision process would allow a program to make recommendations to a customer and motivate product purchases. Amazon has this capability. Also think of Facebook, Google Plus and LinkedIn that recommend users to connect with you after you sign-up.
- **Medical Diagnosis**: Given the symptoms exhibited in a patient and a database of anonymized patient records, predict whether the patient is likely to have an illness. A model of this decision problem could be used by a program to provide decision support to medical professionals.
- **Stock Trading**: Given the current and past price movements for a stock, determine whether the stock should be bought, held or sold. A model of this decision problem could provide decision support to financial analysts.
- **Customer Segmentation**: Given the pattern of behaviour by a user during a trial period and the past behaviours of all users, identify those users that will convert to the paid version of the product and those that will not. A model of this decision problem would allow a program to trigger customer interventions to persuade the customer to covert early or better engage in the trial.
- **Shape Detection**: Given a user hand drawing a shape on a touch screen and a database of known shapes, determine which shape the user was trying to draw. A model of this decision would allow a program to show the platonic version of that shape the user drew to make crisp diagrams.

Types of Machine Learning Problems

There are common classes of problem in Machine Learning. The problem classes below are archetypes for most of the problems we refer to when we are *doing* Machine Learning.

- **Classification**: Data is labelled meaning it is assigned a class, for example spam/non-spam or fraud/non-fraud. The decision being modelled is to assign labels to new unlabelled pieces of data. This can be thought of as a discrimination problem, modelling the differences or similarities between groups.
- **Regression**: Data is labelled with a real value (think floating point) rather then a label. Examples that are easy to understand are time series data like the price of a stock over time, The decision being modelled is what value to predict for new unpredicted data.
- **Clustering**: Data is not labelled, but can be divided into groups based on similarity and other measures of natural structure in the data. An example from the above list would be organising pictures by faces without names, where the human user has to assign names to groups, like iPhoto on the Mac.
- **Rule Extraction**: Data is used as the basis for the extraction of propositional rules (antecedent/consequent like *if-then*). Such rules may, but are typically not directed, meaning that the methods discover statistically supportable relationships between attributes in the data, not necessarily involving something that is being predicted.

Handwritten Digit Recognition

Handwriting recognition (HWR), also known as Handwritten Text Recognition (HTR), is the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens and other devices.

Important steps in the Algorithm

1. Preparation of dataset

-Ask different persons for writing digits from 0 to 9 in a paper.

- From each person 100 data can be collected.

- If 100 people is involving 10000 data can be prepared.

2. Scanning and digitizing of data

3. Pre-processing of data

- Improving the quality of images
- That may include denoising, contrast improvement and sharpening

5. Training phase CNN with input images and Target values

- 80% of total data will be used for training

6. Testing phase

- 20% of data can be used for testing.

Artificial Intelligence-C11

Fuzzy Reasoning

Interim Semester 2021-22 BPL

CSE3007-LT-AB306

Faculty: SIMI V.R.

Fuzzy Sets: Introduction

- Proposed by Lotfi A. Zadeh
- Generalization of classical set theory
- Uncertainty (Incomplete knowledge, generality, vagueness, ambiguity)
- Effective solving of uncertainty in the problem.
- A classical set is a set with a crisp boundary.
- An element : Belongs to / not belongs to a set (0 or 1)

  Eg.: C classical set A of real numbers greater than 6

  $$A = \{x \mid x>6\}$$

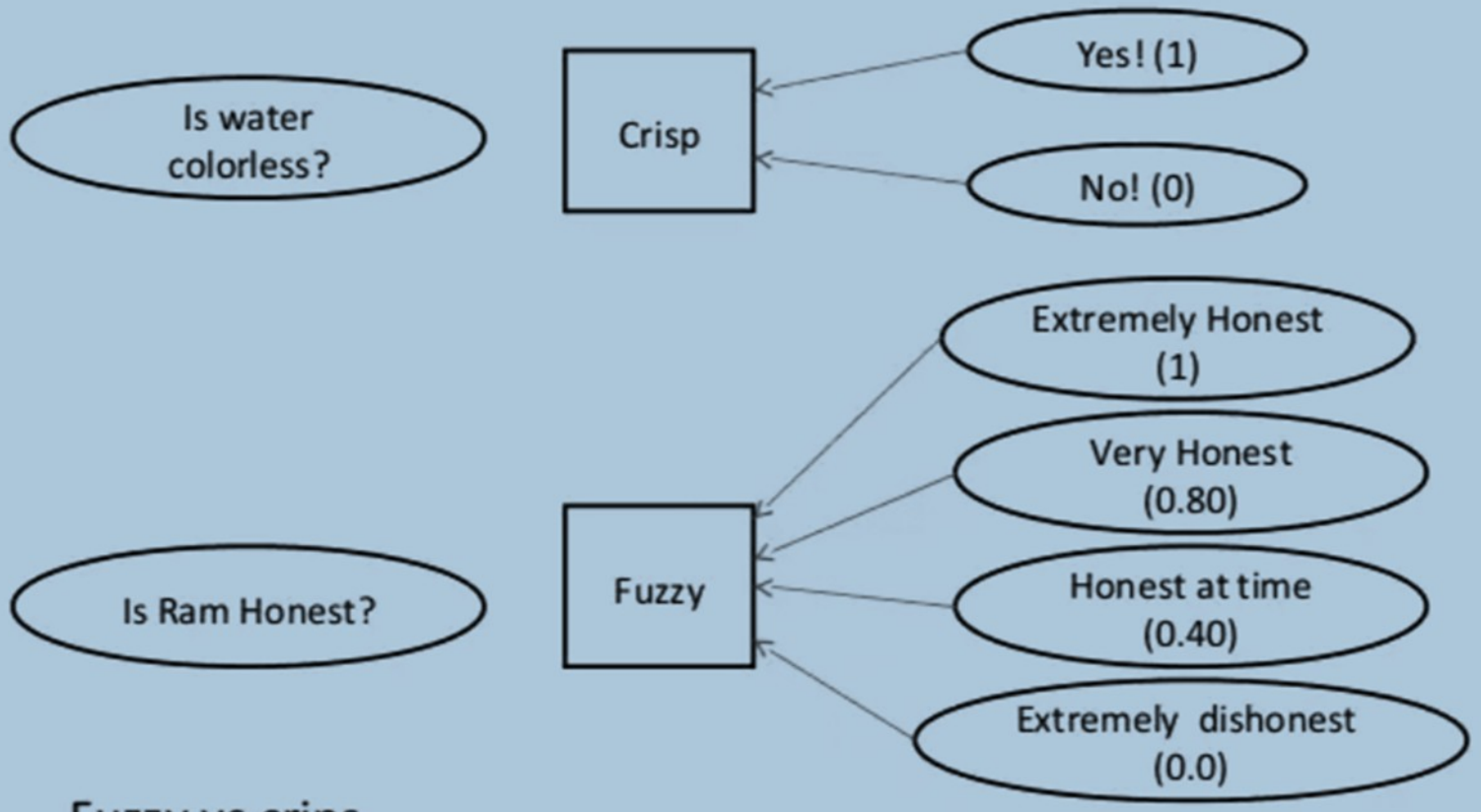
- Fuzzy set many degrees of membership (0 & 1)
- Membership function $\mu_A(x)$ – associated with a fuzzy set A such that the function maps every element of the universe of discourse X to the interval [0,1]

Classical set theory	Fuzzy set theory
• Classes of objects with sharp boundaries.	• Classes of objects with un-sharp boundaries.
• A classical set is defined by crisp(exact) boundaries, i.e., there is no uncertainty about the location of the set boundaries.	• A fuzzy set is defined by its ambiguous boundaries, i.e., there exists uncertainty about the location of the set boundaries.
• Widely used in digital system design	• Used in fuzzy controllers.

The areas of potential fuzzy implementation are numerous and not just for control:

- **Speech recognition**
- **fault analysis**
- **decision making**
- **image analysis**
- **scheduling**

Example

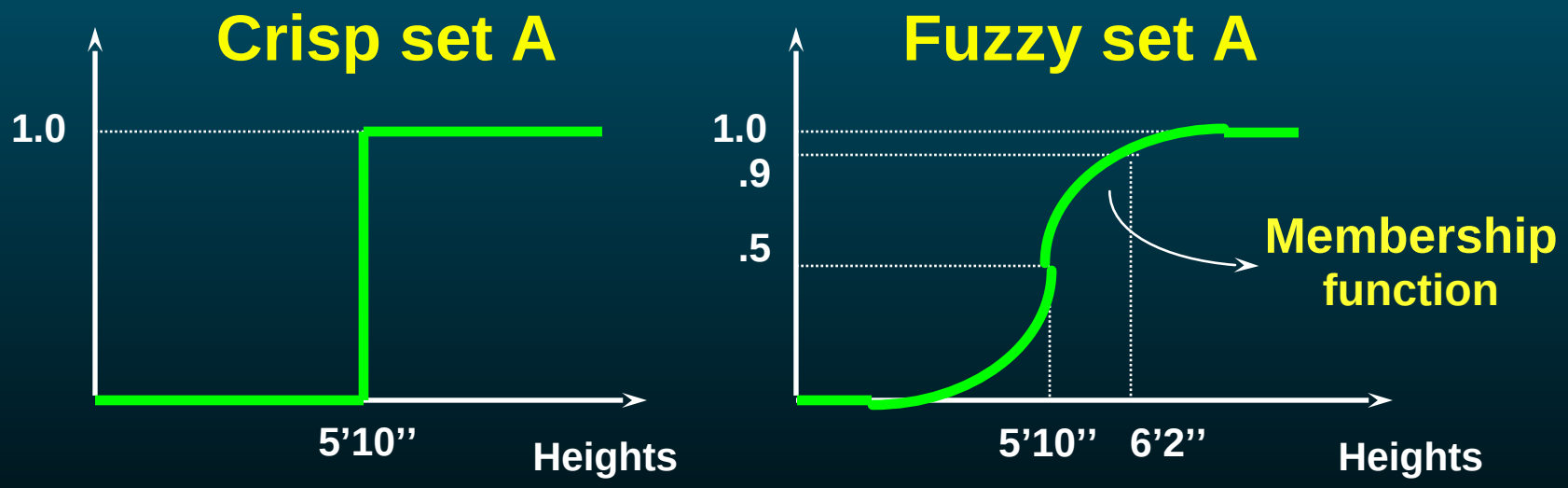


Fuzzy vs crips

Fuzzy Sets

Sets with fuzzy boundaries

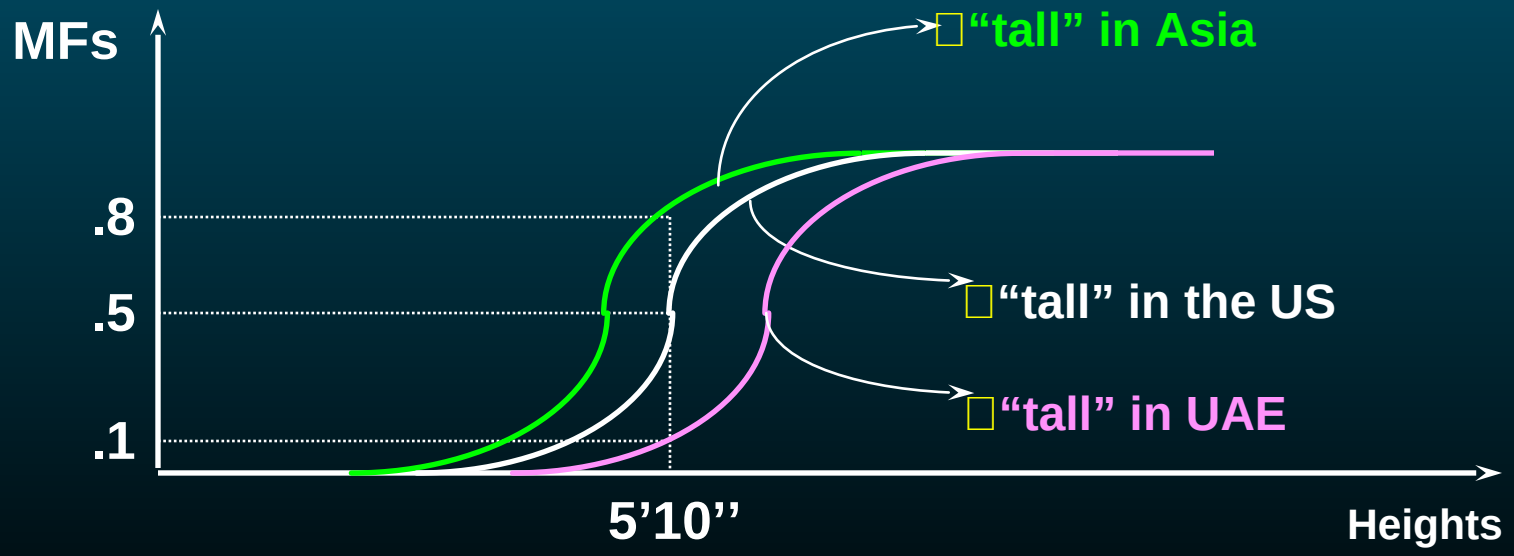
A = Set of tall people



Membership Functions (MFs)

Characteristics of MFs:

- Subjective measures
- Not probability functions



Fuzzy Sets

Formal definition:

A fuzzy set *A* in *X* is expressed as a set of ordered pairs:

$$A = \{(x, \mu_A(x)) \mid x \in X\}$$

Fuzzy set

Membership function (MF)

Universe or universe of discourse

A fuzzy set is totally characterized by a membership function (MF).

Alternative Notation

A fuzzy set A can be alternatively denoted as follows:

X is discrete

$$A = \sum_{x_i \in X} \mu_A(x_i) / x_i$$

X is continuous

$$A = \int_X \mu_A(x) / x$$

- **Note that Σ and integral signs stand for the union of membership grades; "/" stands for a marker and does not imply division.**
- **Membership Function (MF) – Maps each element of X to a membership grade (membership value) between 0 and 1**

Fuzzy Sets with Discrete Universes

Fuzzy set C = "desirable city to live in"

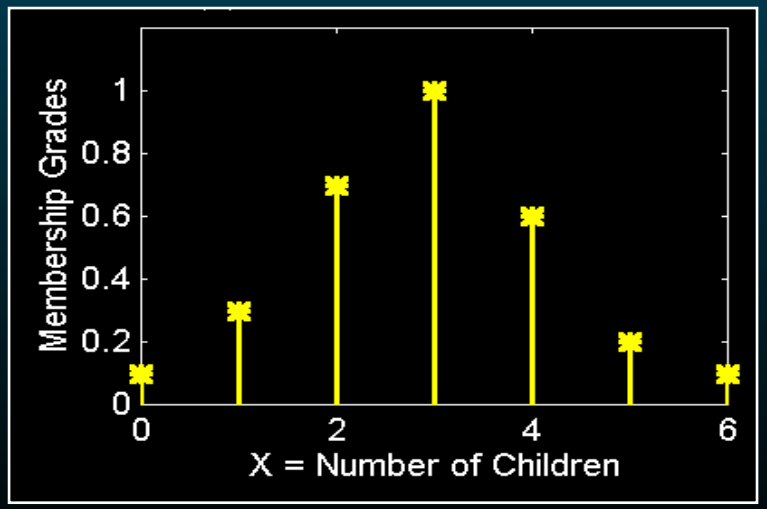
X = {SF, Boston, LA} (discrete and nonordered)

C = {(SF, 0.9), (Boston, 0.8), (LA, 0.6)}

Fuzzy set A = "sensible number of children"

X = {0, 1, 2, 3, 4, 5, 6} (discrete universe)

A = {(0, .1), (1, .3), (2, .7), (3, 1), (4, .6), (5, .2), (6, .1)}



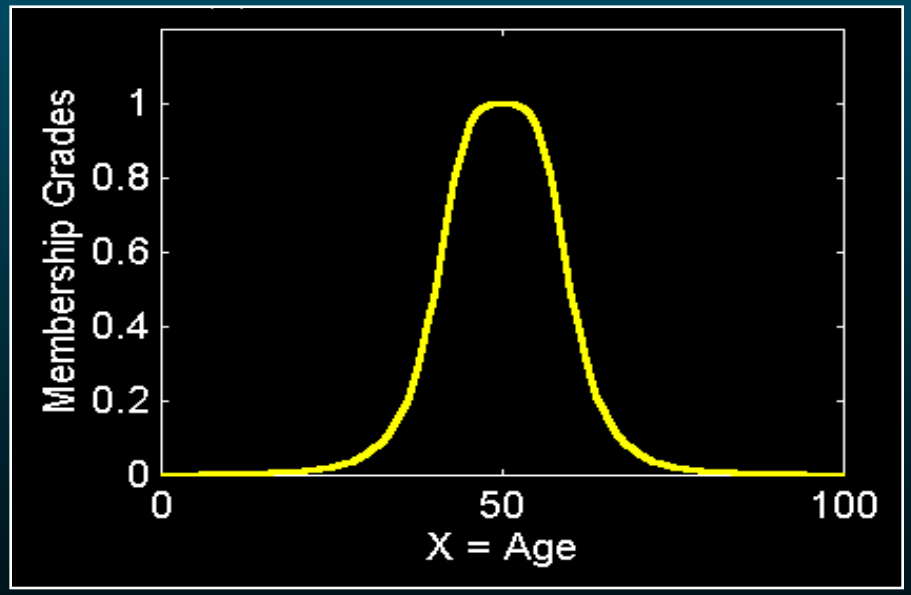
Fuzzy Sets with Cont. Universes

Fuzzy set B = "about 50 years old"

X = Set of positive real numbers (continuous)

B = {(x, μ_B(x)) | x in X}

$$\mu_B(x) = \frac{1}{1 + \left(\frac{x - 50}{10} \right)^2}$$

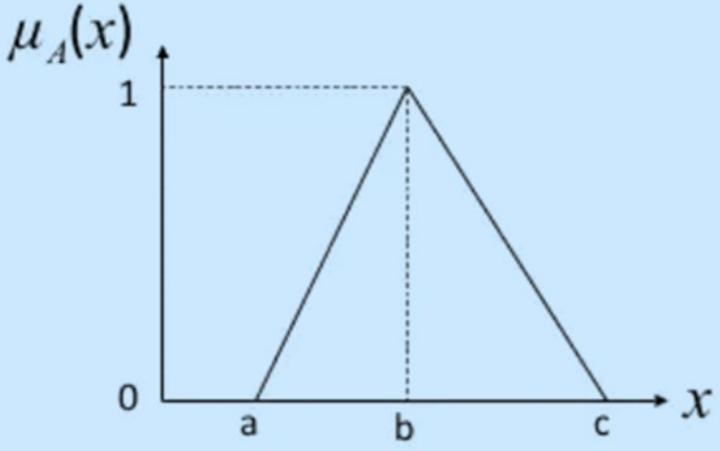


- **Triangular membership function**

  A *triangular* membership function is specified by three parameters {a, b, c}

  a, b and c represent the *x* coordinates of the three vertices of $\mu_A(x)$ in a fuzzy set A (a: lower boundary and c: upper boundary where membership degree is zero, b: the centre where membership degree is 1)

$$\mu_A(x) = \begin{cases} 0 & if\ x \le a \\ \dfrac{x-a}{b-a} & if\ a \le x \le b \\ \dfrac{c-x}{c-b} & if\ b \le x \le c \\ 0 & if\ x \ge c \end{cases}$$



- **Trapezoid membership function**
- A *trapezoidal* membership function is specified by four parameters {a, b, c, d} as follows:

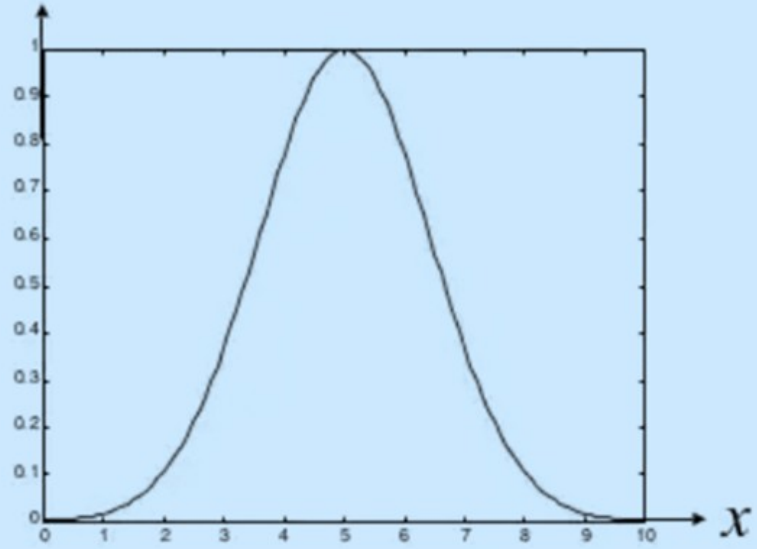
$$\mu_A(x) = \begin{cases} 0 & \text{if } x \le a \\ \dfrac{x-a}{b-a} & \text{if } a \le x \le b \\ 1 & \text{if } b \le x \le c \\ \dfrac{d-x}{d-c} & \text{if } c \le x \le d \\ 0 & \text{if } d \le x \end{cases}$$

- **Gaussian membership function**

$$\mu_A(x,c,s,m) = \exp\left[-\frac{1}{2}\left|\frac{x-c}{s}\right|^m\right]$$

  - c: centre
  - s: width
  - m: fuzzification factor (e.g., m=2)

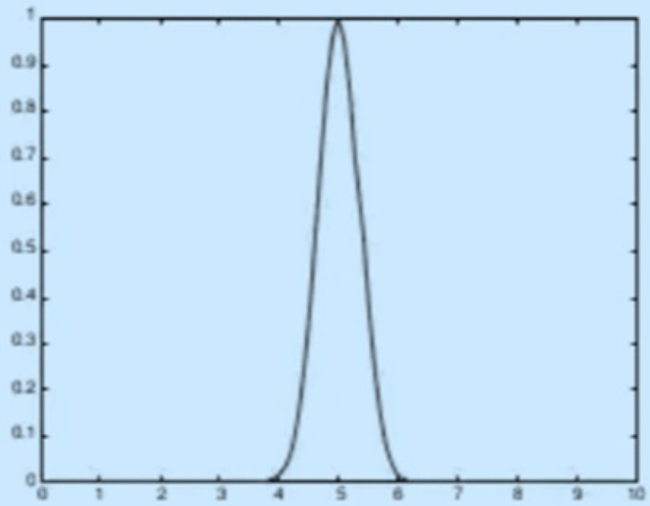
$\mu_A(x)$



c=5

s=2

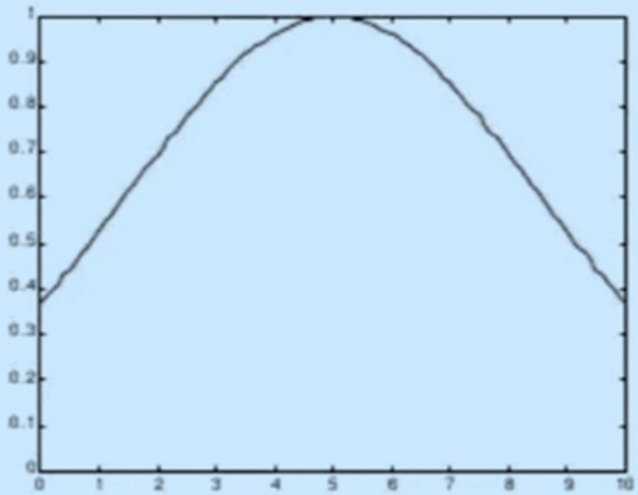
m=2



$c=5$

$s=0.5$

$m=2$



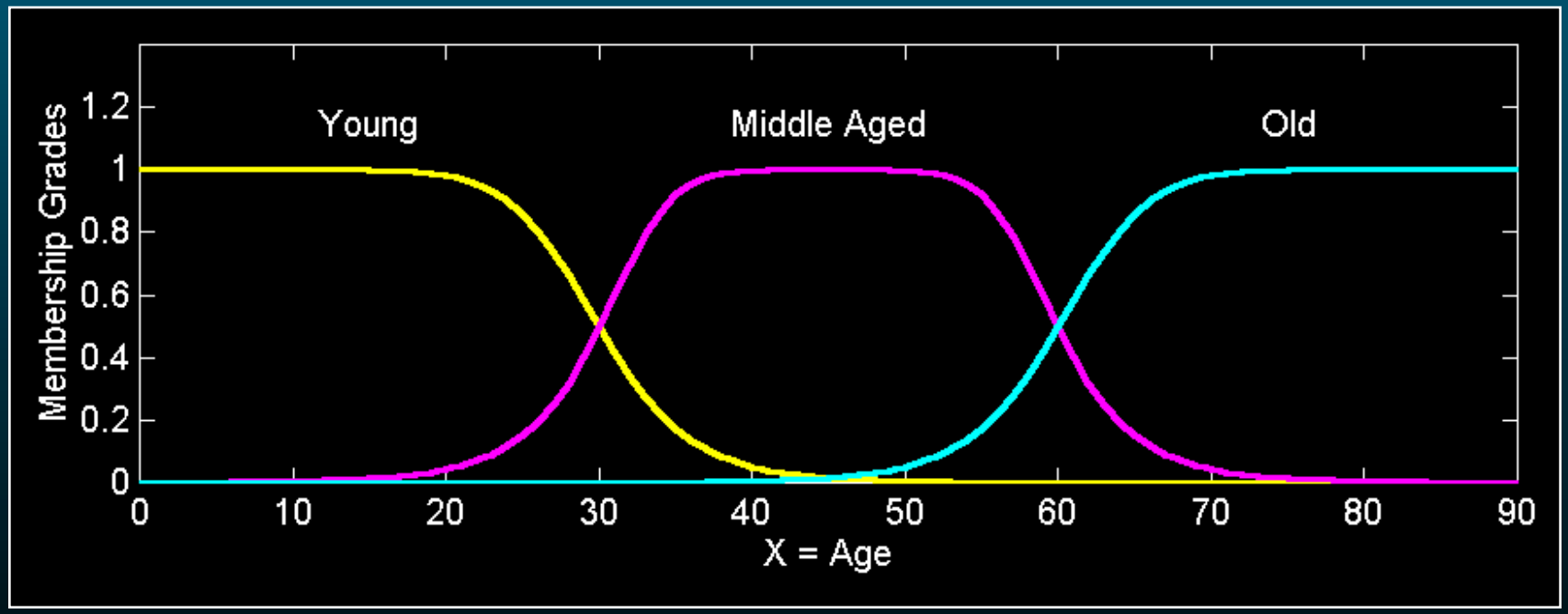
$c=5$

$s=5$

$m=2$

Fuzzy Partition

Fuzzy partitions formed by the linguistic values "young", "middle aged", and "old":



More Definitions

- Fuzzy set is uniquely specified by its MF
- To define membership functions more specifically

Support

Core

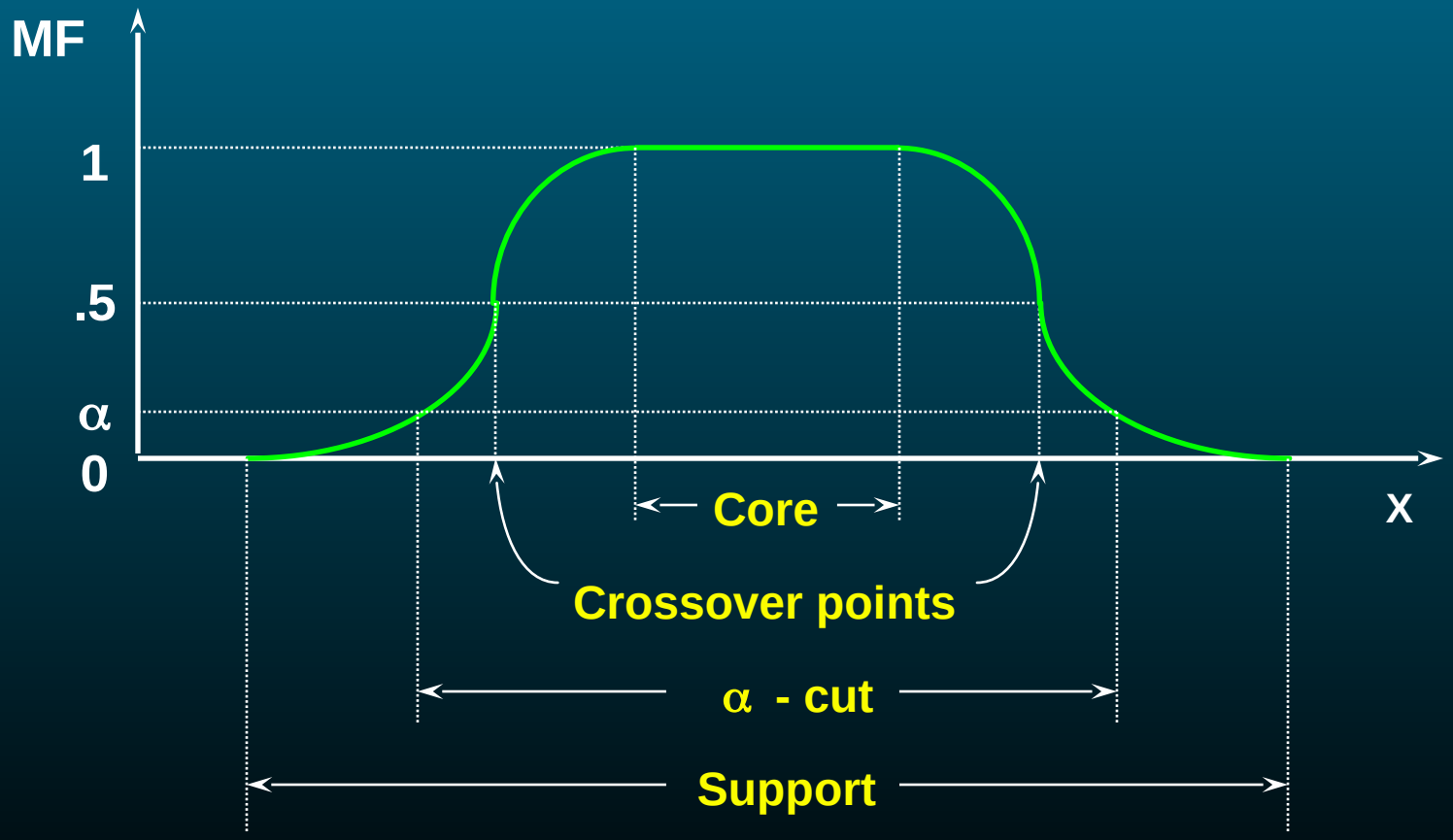
Normality

Crossover points

Fuzzy singleton

α-cut, strong α-cut

MF Terminology



MF Terminology (Cont'd)

The **support** of a fuzzy set A is the set of all points x in X such that $\mu_A(x) > 0$:

$$\text{support}(A) = \{x | \mu_A(x) > 0\}.$$

The **core** of a fuzzy set A is the set of all points x in X such that $\mu_A(x) = 1$:

$$\text{core}(A) = \{x | \mu_A(x) = 1\}.$$

A fuzzy set A is **normal** if its core is nonempty. In other words, we can always find a point $x \in X$ such that $\mu_A(x) = 1$.

A **crossover point** of a fuzzy set A is a point $x \in X$ at which $\mu_A(x) = 0.5$:

$$\text{crossover}(A) = \{x | \mu_A(x) = 0.5\}.$$

A fuzzy set whose support is a single point in X with $\mu_A(x) = 1$ is called a **fuzzy singleton**.

MF Terminology (Cont'd)

The **α-cut** or **α-level set** of a fuzzy set A is a crisp set defined by

$$A_\alpha = \{x | \mu_A(x) \geq \alpha\}.$$

Strong α-cut or **strong α-level set** are defined similarly:

$$A'_\alpha = \{x | \mu_A(x) > \alpha\}.$$

Let A be a fuzzy set.

$A = \{(x_1, 0.1), (x_2, 0.5), (x_3, 0.8), (x_4, 1.0), (x_5, 0.7), (x_6, 0.2)\}$

$support(A) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$core = \{x_4\}$

The α-cuts of the fuzzy set A are:

$A_{0.1} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$A_{0.2} = \{x_2, x_3, x_4, x_5, x_6\}$

$A_{0.5} = \{x_2, x_3, x_4, x_5\}$

$A_{0.7} = \{x_3, x_4, x_5\}$

$A_{0.8} = \{x_3, x_4\}$

$A_{1.0} = \{x_4\}$

The Strong α-cuts of the fuzzy set A are:

$A_{0.1} = \{ x_2, x_3, x_4, x_5, x_6\}$

$A_{0.2} = \{x_2, x_3, x_4, x_5\}$

$A_{0.5} = \{ x_3, x_4, x_5\}$

$A_{0.7} = \{x_3, x_4\}$

$A_{0.8} = \{x_4\}$

Set-Theoretic Operations

Subset:

$$A \subseteq B \Leftrightarrow \mu_A \leq \mu_B$$

Complement:

$$\overline{A} = X - A \Leftrightarrow \mu_{\overline{A}}(x) = 1 - \mu_A(x)$$

Union:

$$C = A \cup B \Leftrightarrow \mu_c(x) = \max(\mu_A(x), \mu_B(x)) = \mu_A(x) \vee \mu_B(x)$$

Intersection:

$$C = A \cap B \Leftrightarrow \mu_c(x) = \min(\mu_A(x), \mu_B(x)) = \mu_A(x) \wedge \mu_B(x)$$

Let A and B be two fuzzy sets.

A = {(1, 0.1), (2, 0.5), (3, 0.8), (4, 1)} and

B = {(1, 1), (2, 0.8), (3, 0.4), (4, 0.1)}

Perform union and intersection operations on those fuzzy sets.

Answer:

$$A \cup B = \{(1,\max(0.1, 1)), (2,\max(0.5, 0.8)), (3,\max(0.8, 0.4)), (4,\max(1, 0.1))\}$$

$$= \{(1, 1), (2, 0.8), (3, 0.8), (4, 1)\}$$

$$A \cap B = \{(1,\min(0.1, 1)), (2,\min(0.5, 0.8)), (3,\min(0.8, 0.4)), (4,\min(1, 0.1))\}$$

$$= \{(1, 0.1), (2, 0.5), (3, 0.4), (4, 0.1)\}$$

Fuzzy System

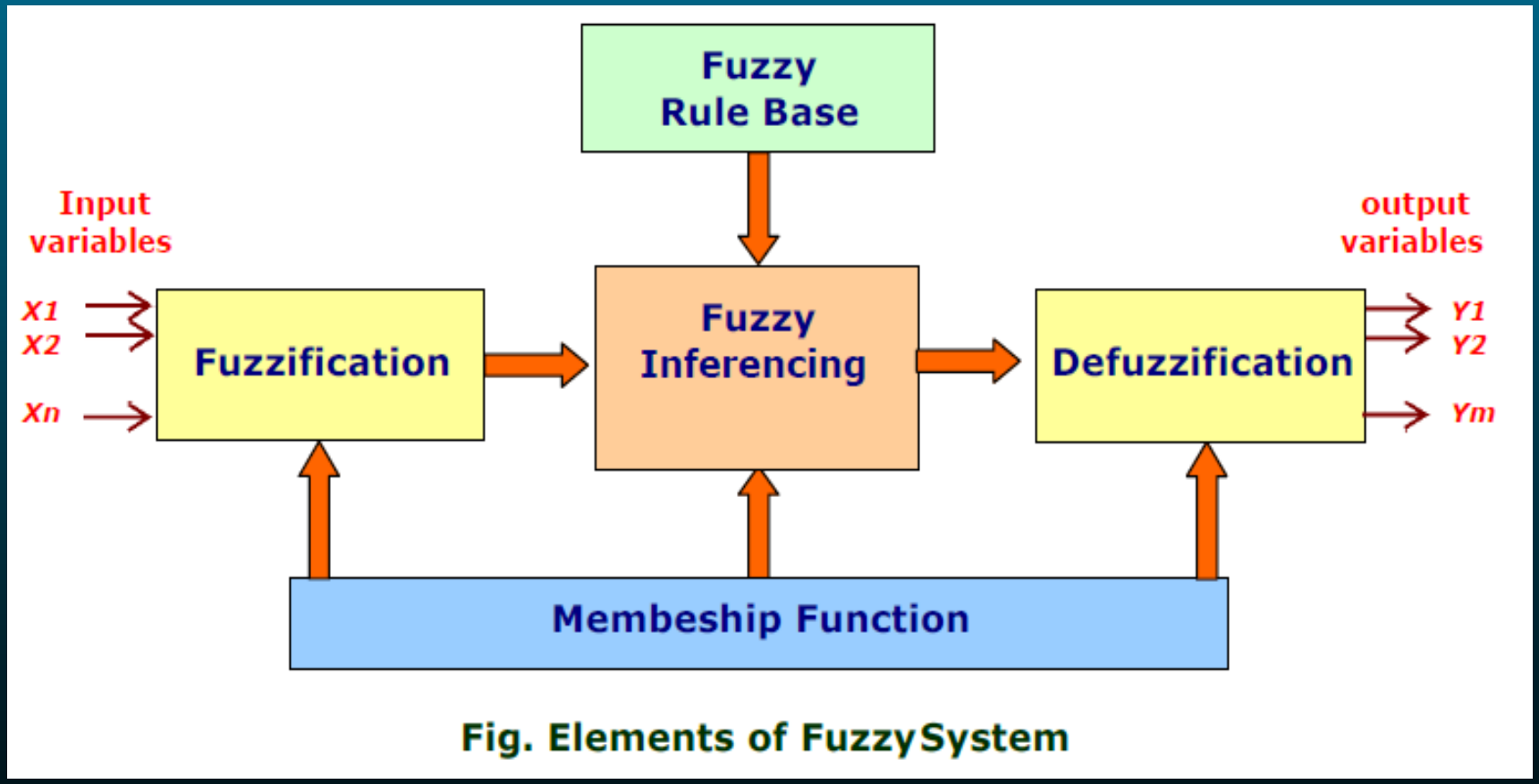


Fig. Elements of FuzzySystem

Fuzzy System elements

- **Input Vector** : $X = [x_1 , x_2, \ldots x_n]^T$ are crisp values, which are transformed into fuzzy sets in the fuzzification block.
- **Output Vector** : $Y = [y_1 , y_2, \ldots y_m]^T$ comes out from the defuzzification block, which transforms an output fuzzy set back to a crisp value.
- **Fuzzification** : a process of transforming crisp values into grades of membership for linguistic terms, "far", "near", "small" of fuzzy sets.
- **Fuzzy Rule base** : a collection of propositions containing linguistic variables; the rules are expressed in the form:

  If (x is A) AND (y is B) . . . . . THEN (z is C)

  where **x, y** and **z** represent variables (e.g. distance, size) and **A, B** and **Z** are linguistic variables (e.g. `far', `near', `small').
- **Membership function** : provides a measure of the degree of similarity of elements in the universe of discourse **U** to fuzzy set.
- **Fuzzy Inferencing** : combines the facts obtained from the Fuzzification with the rule base and conducts the Fuzzy reasoning process.
- **Defuzzyfication:** Translate results back to the real world values.

References:

J-S R Jang and C-T Sun, Neuro-Fuzzy and Soft Computing, Prentice Hall, 1997

S. Rajasekaran and G.A. Vijayalaksmi Pai ,Neural Network, Fuzzy Logic, and Genetic Algorithms - Synthesis and Applications, (2005), Prentice Hall.