UNIT - 2
PROBLEM SOLVING AND REPRESENTATION OF KNOWLEDGE

By Vishvajit Bakrola

OUTLINE

- Problem Solving Agents
- Searching for Solutions
- Search Techniques – Uninformed and Informed
- Heuristic Functions
- Local Search and Optimization
- Knowledge based Agents
- Logics

PROBLEM SOLVING AGENTS

PROBLEM SOLVING AGENTS

- They are Goal based or Goal directed agents.
- We know that, an intelligent agent are supposed to maximize their performance measures.
- Goal formation is the first step of any problem solving agent.
- A goal can be a state or a path (*i.e. sequential collection of states*).
- The process of looking for a sequence of actions that leads an agent to the goal is known as Search.
- Search algorithm takes a problem as an input and returns solution having the sequence of actions.

WELL-DEFINED PROBLEMS AND SOLUTIONS

- A problem can be systematically defined using following five components:

Optimal solution should have lowest path cost

Initial State → Actions → Transition Model → Goal Test → Path Cost

- **Initial State**: The starting state
- **Actions**: ACTIONS(Cs)
- **Transition Model**: It is the description of action results
- **Goal Test**: IsGoal(Cs)?
- **Path Cost**: The function returns the numeric value. Used to estimate the performance measures

REAL-WORLD EXAMPLE

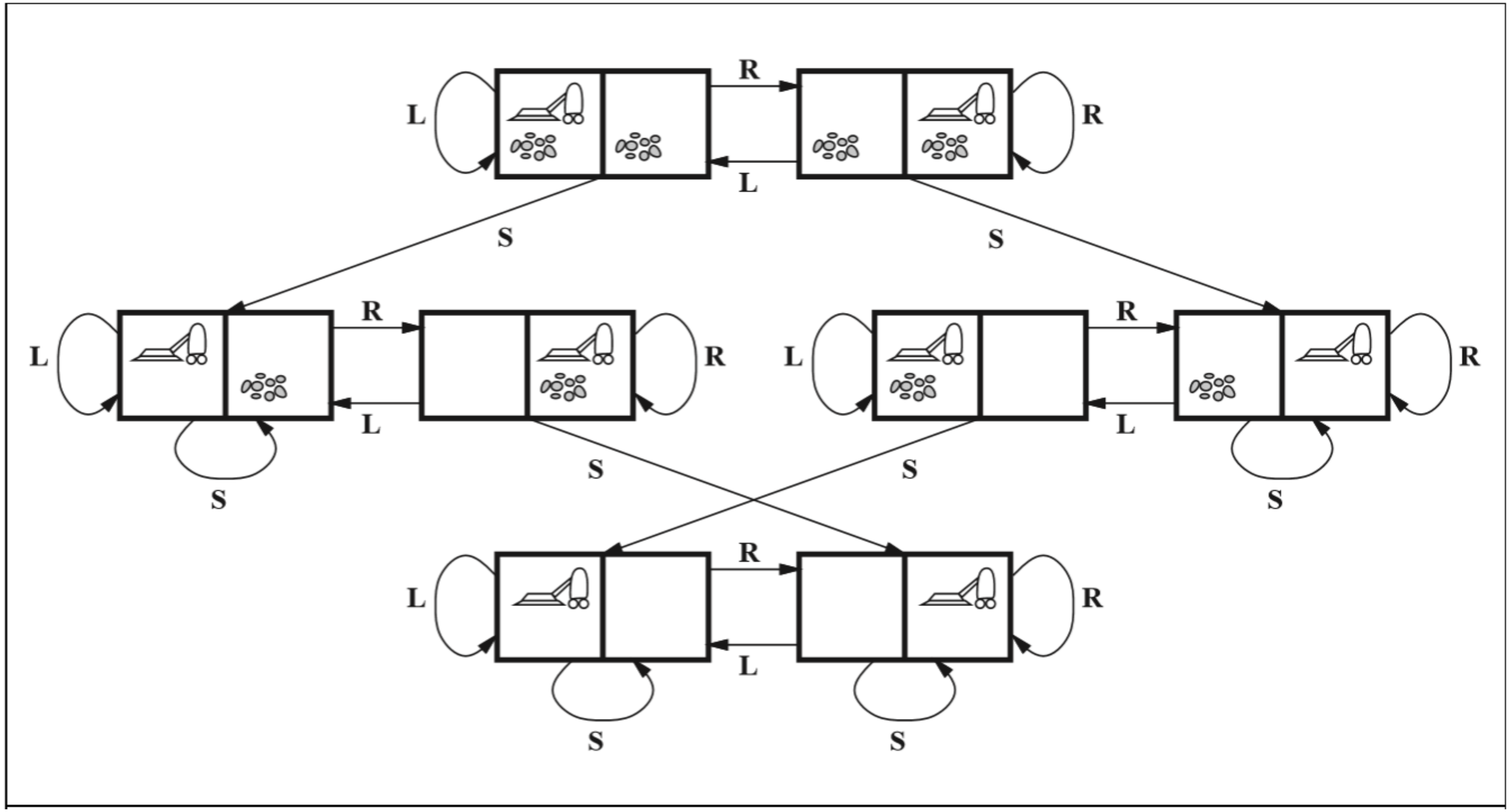


Figure 1: The state space of vacuum cleaner world

PROBLEM FORMULATION

- **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 $n \times 2^n$ 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 *Sucking* in a clean square have no effect.

CONTINUE...

- **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.

THE 8-PUZZLE EXAMPLE

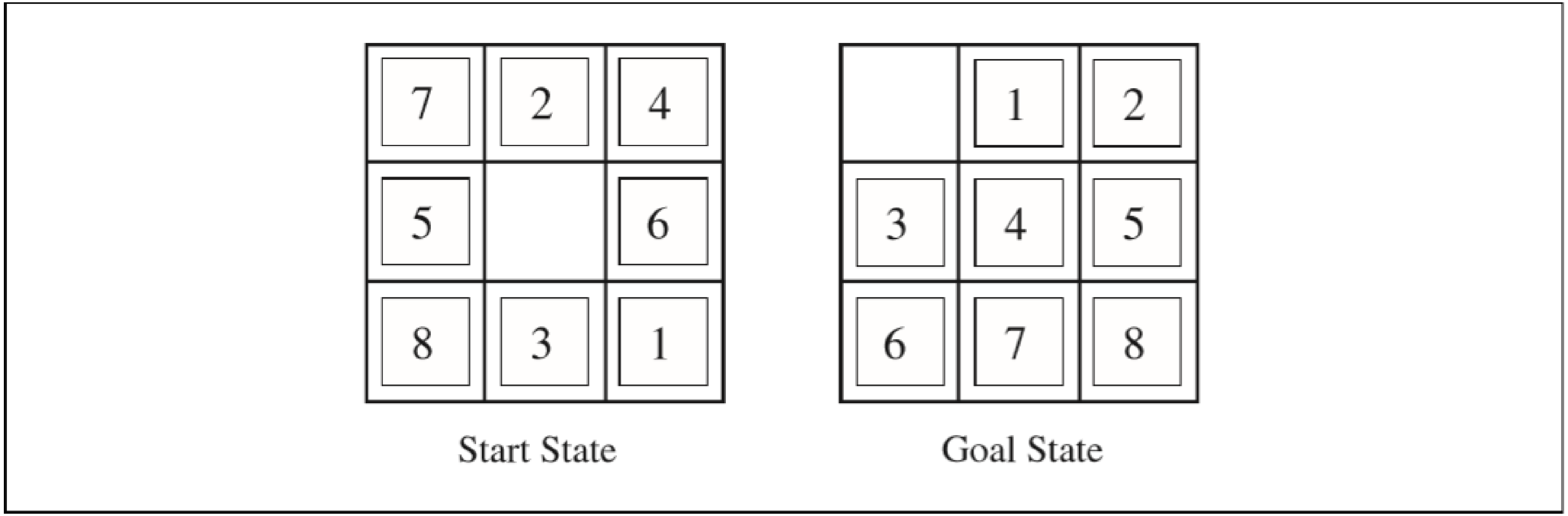


Figure 2: A typical instance of 8-puzzle problem

EXERCISE

- State??
- Initial state??
- Actions??
- Transition model??
- Goal test??
- Path cost??

CONTINUE...

- Other examples can be,
  - ✓ Sliding block puzzle
  - ✓ 8-queen problem

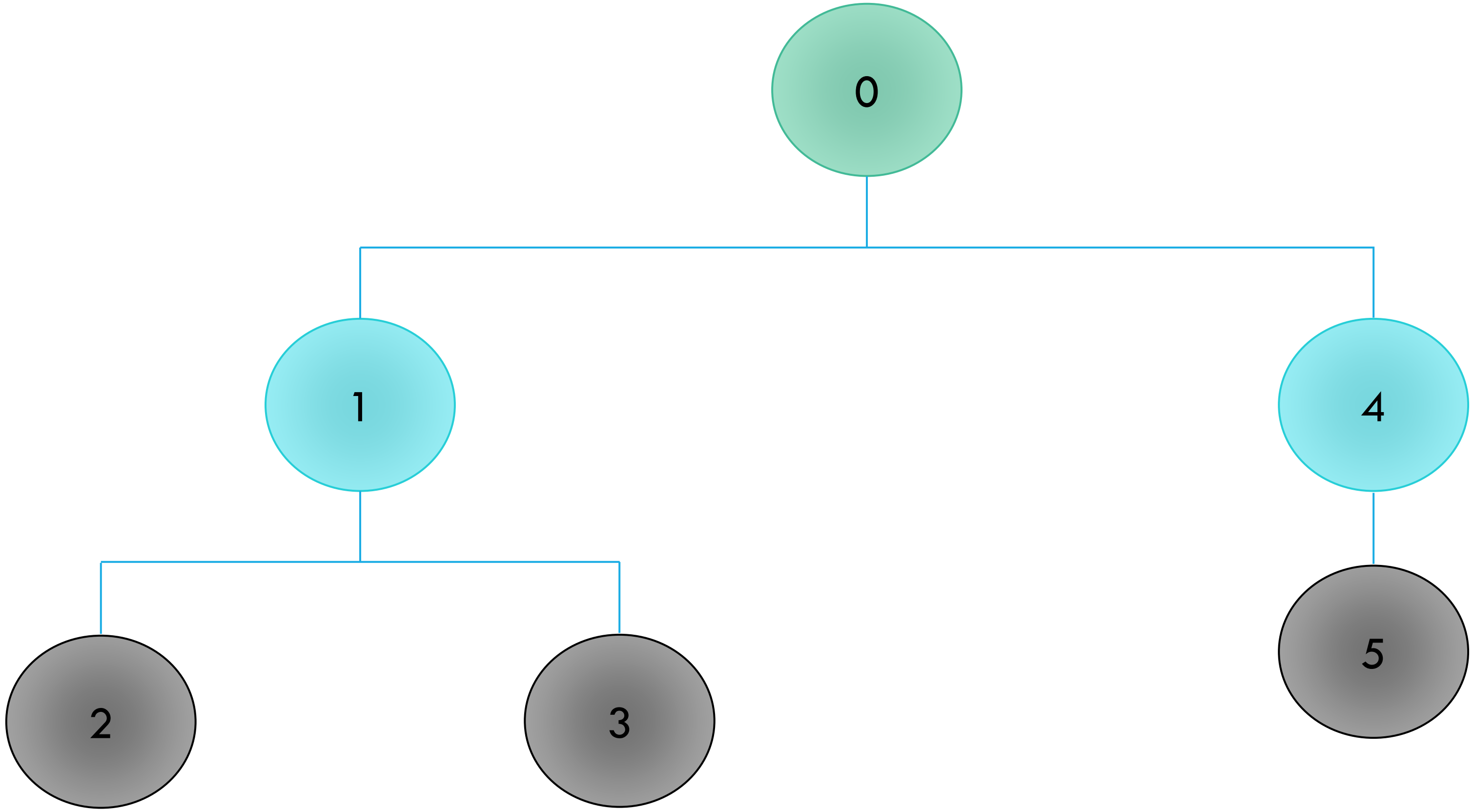
- Some real-world problems
  - ✓ Rout-finding problems
  - ✓ Touring problems
  - ✓ TSP
  - ✓ VLSI layout designing
  - ✓ Robot navigation
  - ✓ Assembly sequencing

SEARCHING FOR THE SOLUTION

SEARCHING OF THE SOLUTION

- As we know, solution is a sequence of actions.
- Select the initial node (*if possibilities are many*)
- Then we apply actions to current node, with possible legal actions and generate new set of states.
- If there is no goal state from newly generate set of states, then the same process will be repeated until we reach goal state.
- In last, the search algorithm will return goal state/solution.

CONTINUE...



Root Node/Current State
Check IsGoal()
Action = *Expand(0)*

Nodes/States at next level
Check IsGoal()
Action = *Expand(0)*

Nodes/States at next level
Check IsGoal()
If goal found, return goal
Terminate the search

Figure 3: Search tree expansion

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:
  1. **n.STATE**: the state in the state space to which the node corresponds.
  2. **n.PARENT**: the node in the search tree that generated this node.
  3. **n.ACTION**: the action that was applied to the parent to generate the node.
  4. **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.

MEASURING PROBLEM SOLVING PERFORMANCE

- We can evaluate an algorithm's performance in four ways:
  1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
  2. **Optimality:** Does the strategy find the optimal solution?
  3. **Time complexity:** How long does it take to find a solution?
  4. **Space complexity:** How much memory is needed to perform the search?

CONTINUE…

- Time and space complexity are always considered with respect to some measure of the problem difficulty.
- The typical measure is the size of the state space graph, $|V| + |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).
- For these reasons, complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node; d, the **depth** of the shallowest goal node (i.e., the number of steps along the path from the root); and m, the maximum length of any path in the state space.
- To assess the effectiveness of a search algorithm, we can consider just the **search cost**— which typically depends on the time complexity but can also include a term for memory usage—or we can use the **total cost**, which combines the search cost and the path cost of the solution found.

UNINFORMED SEARCH STRATEGIES

SOME IMPORTANT TERMS

- **Search space** → possible conditions and solutions.
- **Initial state** → state where the searching process started.
- **Goal state** → the ultimate aim of searching process.
- **Problem space** → "what to solve"
- **Searching strategy** →strategy for controlling the search.
- **Search tree** → tree representation of search space, showing possible solutions from initial state.

SEARCHING STRATEGIES

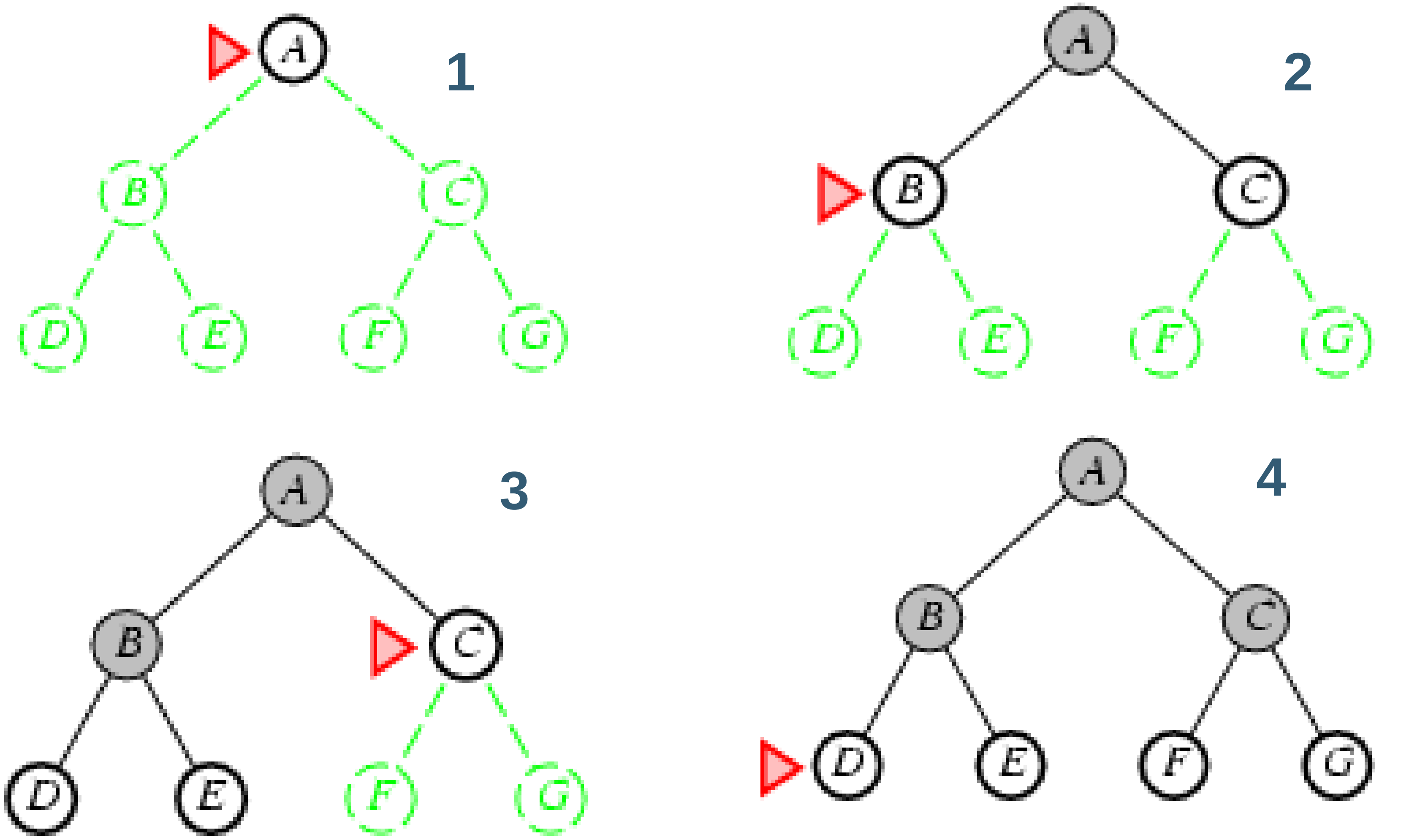
- **Blind search →** traversing the search space until the goal nodes is found (might be doing exhaustive search).
- Techniques : **Breadth First Uniform Cost ,Depth first, Interactive Deepening search.**
- Guarantees solution.

- **Heuristic search →** search process takes place by traversing search space with applied rules (information).
- Techniques: **Greedy Best First Search, A* Algorithm**
- There is no guarantee that solution is found.

BREADTH FIRST SEARCH (BFS)

- **Strategy:** Search all the nodes expanded at given depth before any node at next level.
- **Concept** : First In First Out (FIFO) queue.
- **Complete ?:** Yes with finite *b* (branch).
- **Space** : Keep nodes in every memory
- **Optimal ?** = Yes (if cost =1)

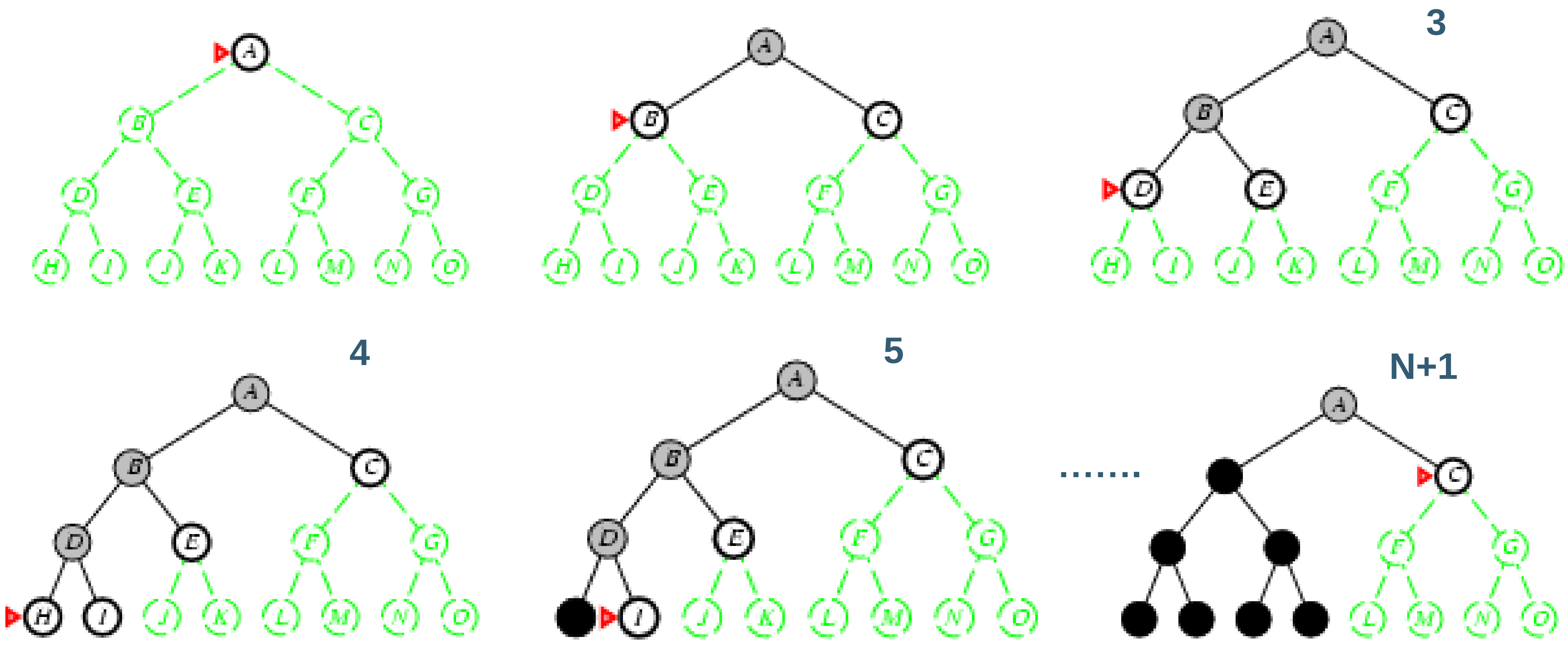
BFS - EXAMPLE



DEPTH FIRST SEARCH (DFS)

- **Strategy:** Search all the nodes expanded in deepest path.
- **Concept:** Last In First Out
- **Complete ?:** No
- **Complexity:** $O(b^m)$
- **Space :** O*(bm) – b ; branching factor, m ; max. depth*
- **Optimality ? :** No

DFS - EXAMPLE



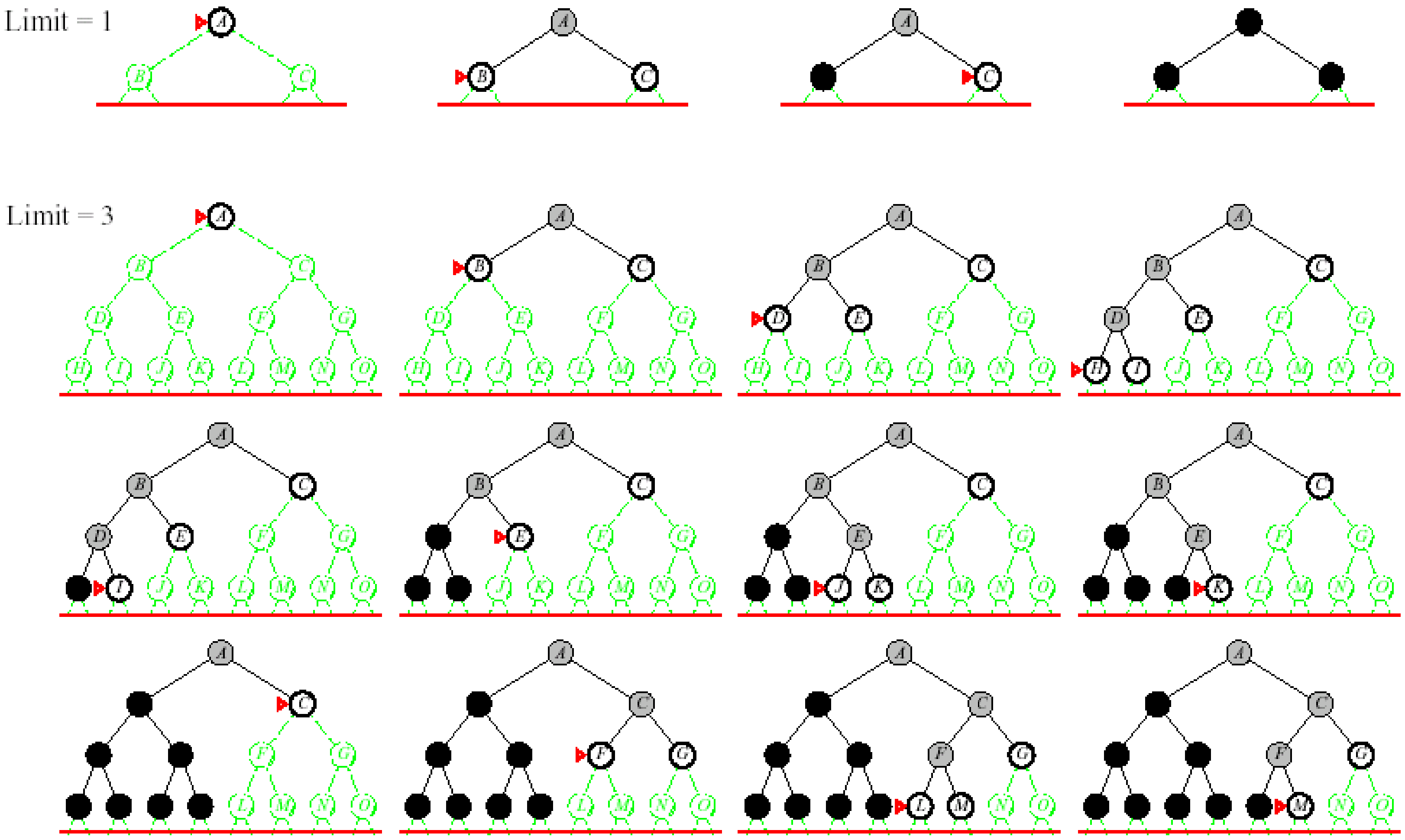
DEPTH-LIMITED SEARCH (DLS)

- DLS is DFS with a depth limit, L
  - ✓ Nodes at depth limit are treated as leaves
  - ✓ Depth limits can come from domain knowledge
    - E.g., L = <u>diameter</u> of state space
- Implemented same as DFS, but with depth limit
- Performance
  - ✓ Incomplete if L < d
  - ✓ Nonoptimal if L > d
  - ✓ Time and space complexity <= DFS

ITERATIVE DEEPENING DFS (ID-DFS)

- IDDFS performs DLS one or more times, each time with a larger depth limit, l
  - ✓ Start with l = 0, perform DLS
  - ✓ If goal not found, repeat DLS with l = 1, and so on
- Performance
  - ✓ Combines benefits of BFS (completeness, optimality) with benefits of DFS (relatively low space complexity)
- Not as inefficient as it looks
  - ✓ Repeating low-depth levels: these levels do not have many nodes

ID-DFS - EXAMPLE



BIDIRECTIONAL SEARCH

- Perform two searches simultaneously
  - ✓ One search from the initial state to a goal state
  - ✓ Another search from a goal state to the initial state
  - ✓ Stop searching when either search reaches a state that is in the fringe of the other state (i.e., when they “meet in the middle”)
- Performance
  - ✓ Complete and optimal if both searches are BFS

CONTINUE…

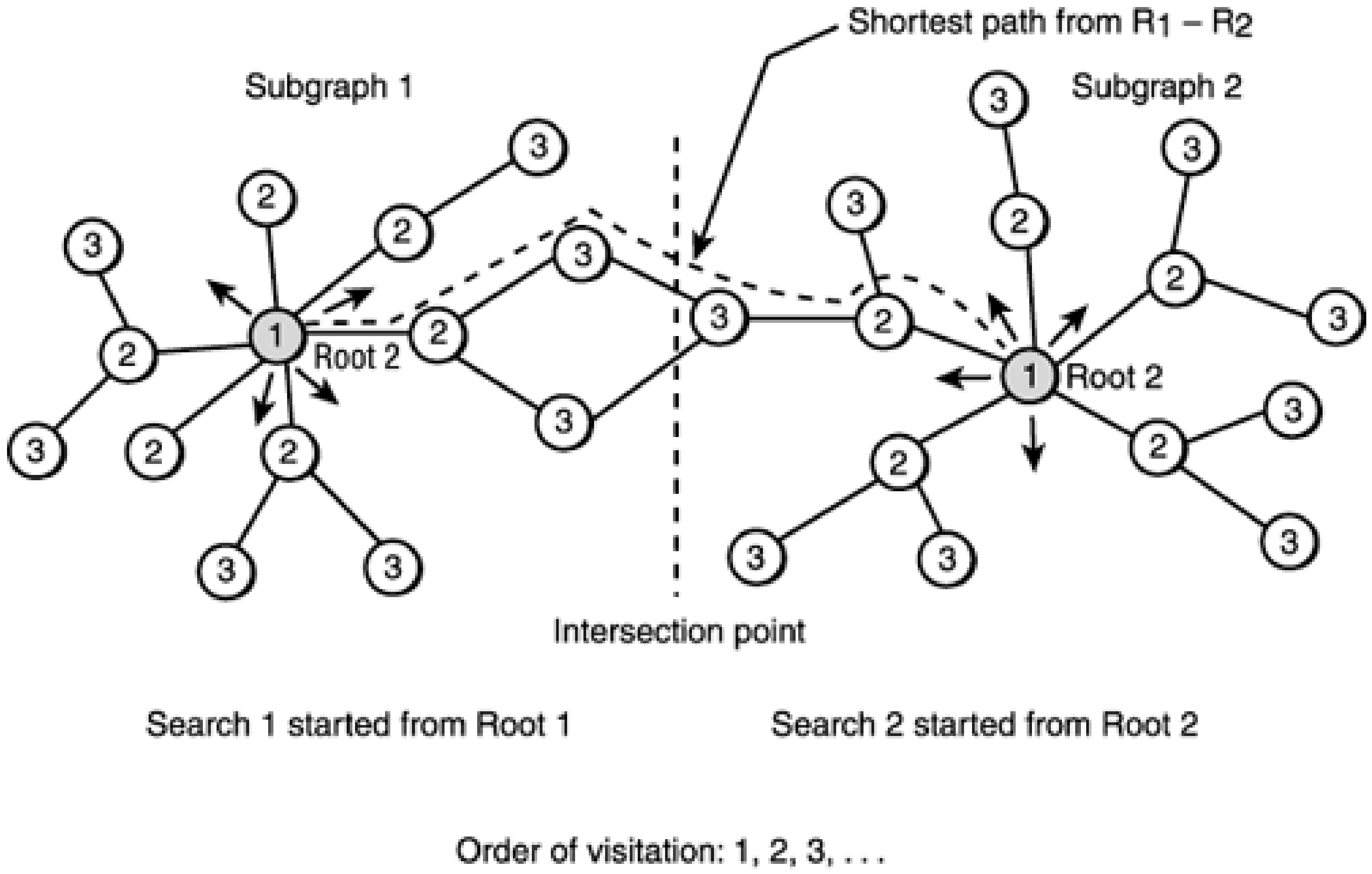


Figure 4: Bidirectional Search Example

CONTINUE…

- Reaching a node with a repeated state means that there are at least two paths to the same state
  - ✓ Can lead to infinite loops
- Keep a list of all nodes expanded so far (called the closed list)

INFORMED SEARCH AND EXPLORATION

INFORMED SEARCH AND EXPLORATION

- Review of Associated Functions
- Greedy Best-First Search
- A* Search
- Heuristic Functions

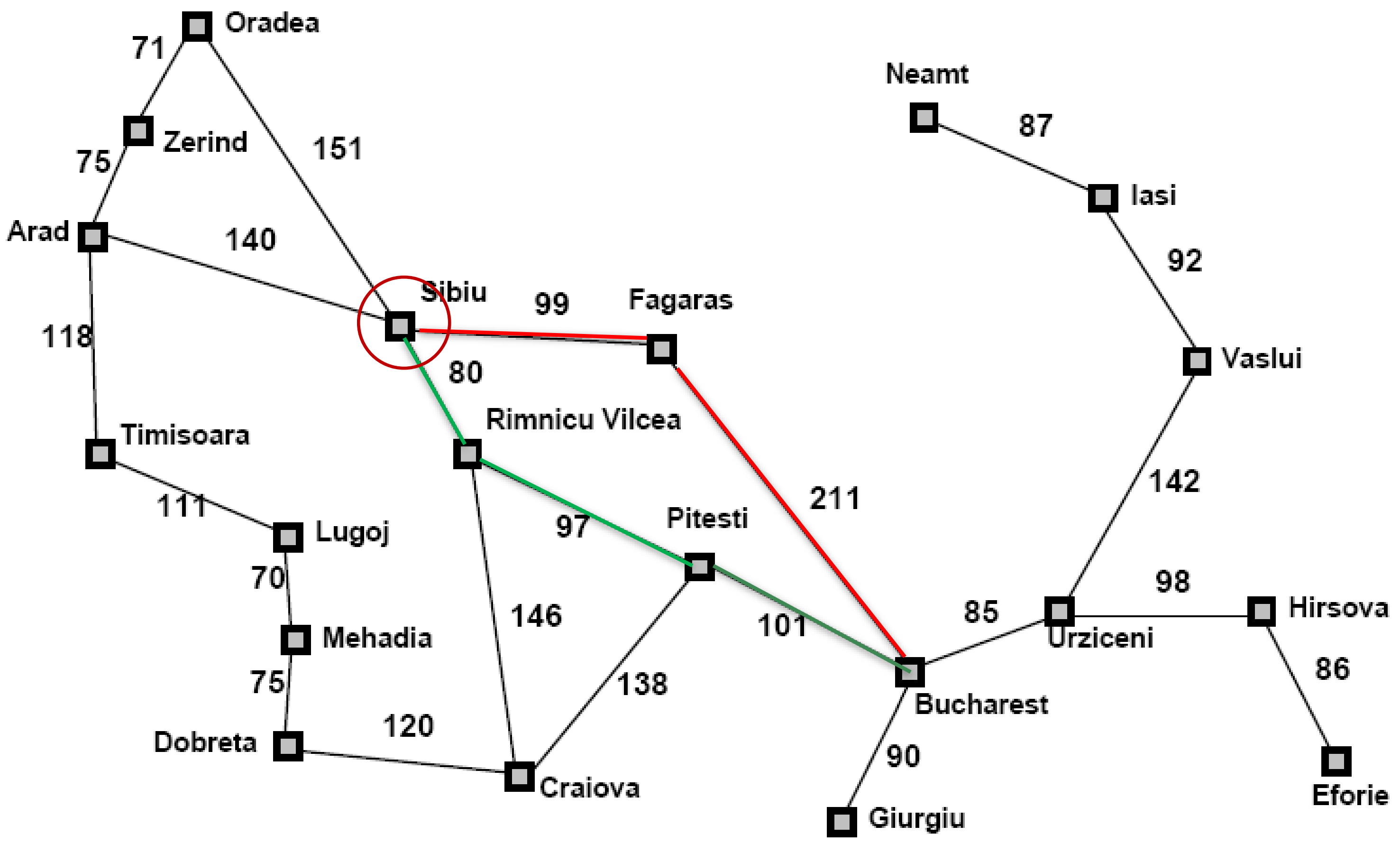
REVIEW OF ASSOCIATED FUNCTIONS

- g(n) = cost from the initial state to the current state n
- h(n) = estimated cost of the cheapest path from node n to a goal node
- f(n) = evaluation function to select a node for expansion (usually the lowest cost node)

GREEDY BEST-FIRST SEARCH

- Greedy Best-First search tries to expand the node that is **closest to the goal** assuming it will lead to a solution quickly
  - ✓ $f(n) = h(n)$
  - ✓ a.k.a. "Greedy Search"
- Implementation
  - ✓ expand the "most desirable" node into the fringe queue
  - ✓ sort the queue in decreasing order of desirability
- Example: consider the straight-line distance heuristic h_{SLD}
  - ✓ Expand the node that appears to be closest to the goal

CONTINUE...



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 5: Greedy BFS Example

CONTINUE…

- Notice that the values of h_{SLD} cannot be computed from the problem itself
- It takes some experience to know that h_{SLD} is correlated with actual road distances
  - ✓ Therefore a useful heuristic

CONTINUE…

- **Complete**
  - ✓ No, GBFS can get stuck in loops (e.g. bouncing back and forth between cities)
- **Time**
  - ✓ O(bm) but a good heuristic can have dramatic improvement
- **Space**
  - ✓ O(bm) – keeps all the nodes in memory
- **Optimal**
  - ✓ No!

HEURISTIC FUNCTIONS

- Heuristic Function - function applied to a state in a search space to indicate a likelihood of success if that state is selected:
  - ✓ Heuristic search methods are known as "weak methods" because they are dependent on domain specific knowledge.
- The heuristic tells us, how far the state is from the goal state.
- **h(n)**

A* SEARCH

- A* (A star) is the most widely known form of Best-First search
  - ✓It evaluates nodes by combining g(n) and h(n)
  - ✓**f(n) = g(n) + h(n)**

Where,

g(n) = cost so far to reach n

h(n) = estimated cost to goal from n

f(n) = estimated total cost of path through n

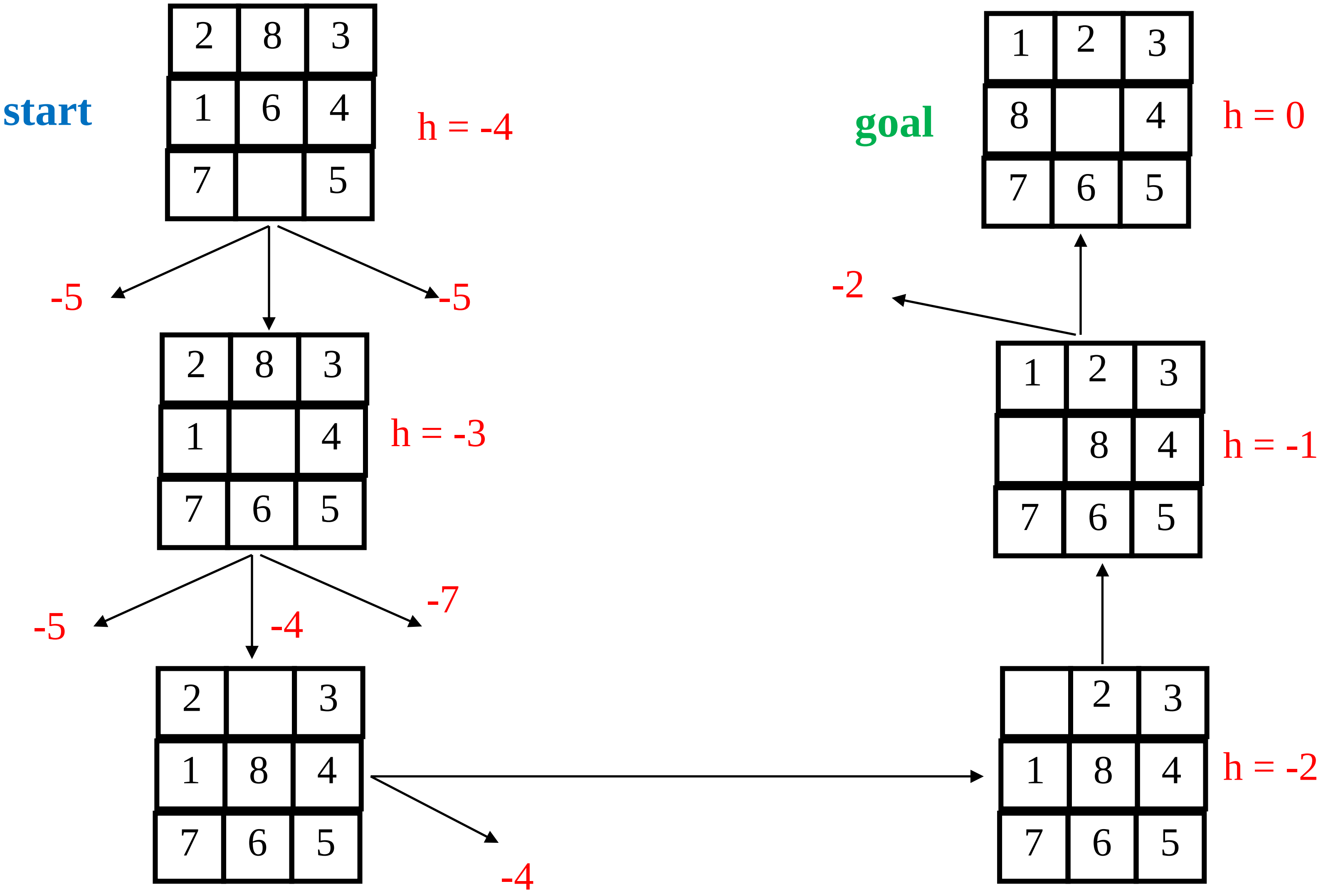
CONTINUE...

- When h(n) = actual cost to goal
  - ✓Only nodes in the correct path are expanded
  - ✓Optimal solution is found
- When h(n) < actual cost to goal
  - ✓Additional nodes are expanded
  - ✓Optimal solution is found
- When h(n) > actual cost to goal
  - ✓Optimal solution can be overlooked

LOCAL SEARCH ALGORITHMS

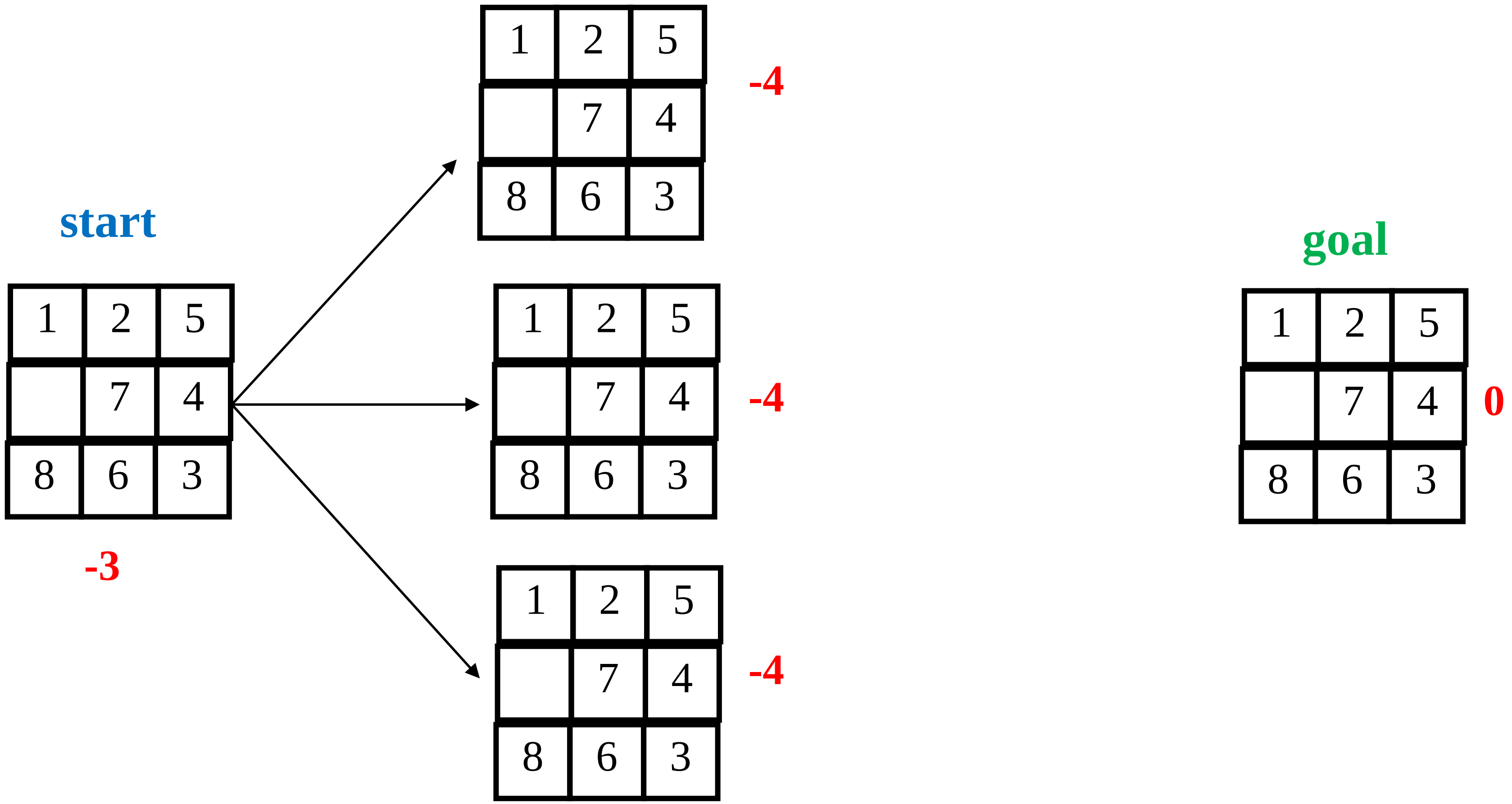
- The concept we have adopted so far is systematic exploration of search space.
- Some problems are path irrelevant type of problems.
- Local search techniques are used here.
- Use single current state and move to neighboring states.
- Advantages:
  - ✓ Use very little memory
  - ✓ Find often reasonable solutions in large or infinite state spaces
- Are also useful for pure optimization problems.
  - ✓ Find best state according to some objective function

HILL CLIMBING EXAMPLE



f(n) = -(number of tiles out of place)

EXAMPLE OF A LOCAL MAXIMUM



LEARNING HEURISTIC FROM EXPERIENCE

- h(n) is supposed to estimate the cost of a solution beginning from the state at node n.
- How could an agent construct such a function?
- One of possible solution is learn from the Experiences.
- From each optimal solution of a problem, h(n) can be trained.
- Each example consists of a state from the solution path and the actual cost of the solution from that point.
- From these examples, a learning algorithm can be used to construct a function h(n) that can predict solution costs for other states that arise during the searching process.

CONTINUE…

- Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state's value, rather than with just the raw state description.
- For example, the feature "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal.
- Let's call this feature x1(n).
- We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when X1(n) is 5.
- Given these data, the value of X1 can be used to predict h(n).
- We can even take some other feature, say X2(n). That might be "number of pairs of adjacent tiles that are not adjacent in goal state".
- To combine more that one feature to form a common h(n), linear combination can be used: h(n) = C1X1(n) + C2X2(n).

HILL-CLIMBING SEARCH

- The hill-climbing algorithm is simply a loop that continuously moves in the direction of increasing value – that is Uphill move.
- It terminates the process when it reaches to peak, where no neighbour has higher value.
- Hill-climbing does not look ahead beyond the immediate neighbours of the current state.
- Hill-climbing can be considered as Greedy local search, because it grabs a good neighbour state without thinking about next situations.

CONTINUE…

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*

Figure 6: The Hill-Climbing search algorithm

CONTINUE…

- Unfortunately, hill-climbing often gets stuck for the following reasons:
  1. Local Maxima
  2. Ridges
  3. Plateau/Shoulder

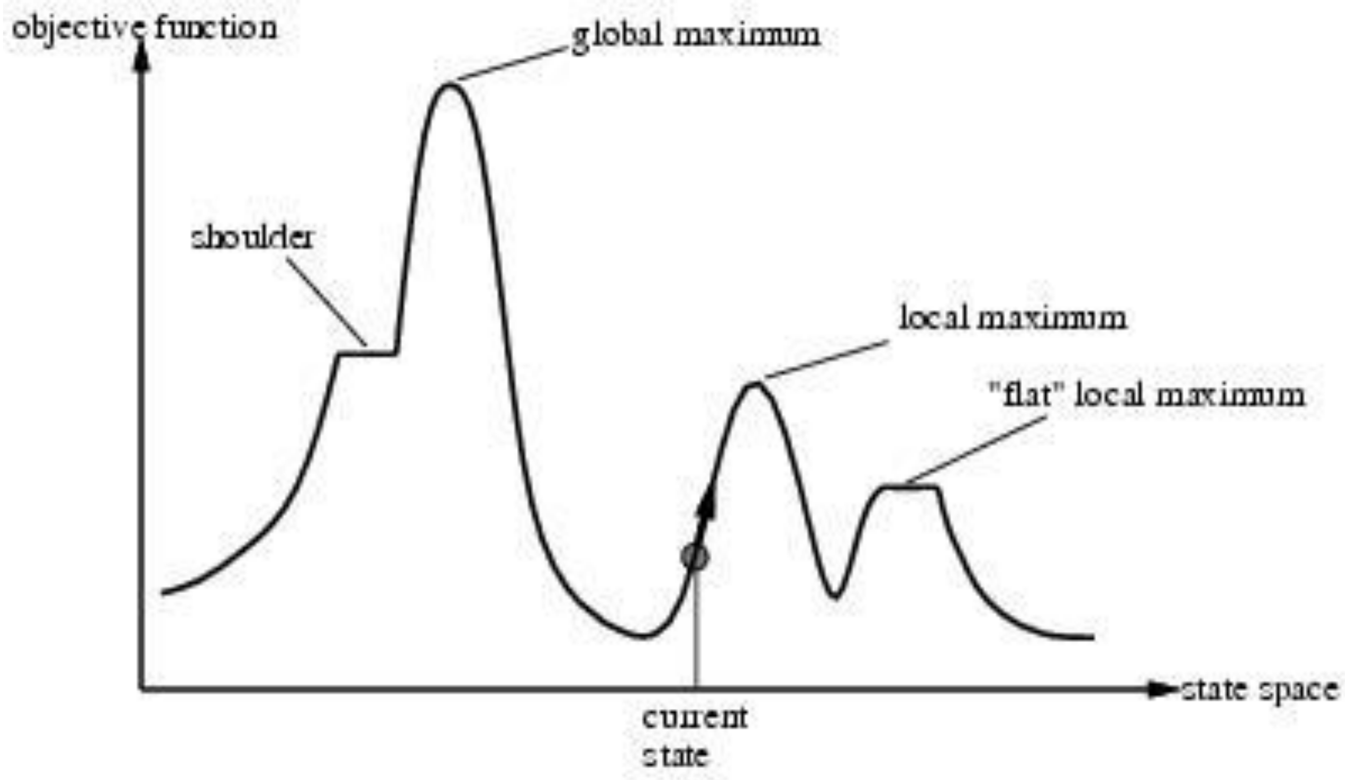


Figure 7: Issues with Hill-climbing algorithm

STIMULATED ANNEALING

- Hill-climbing moves towards goal with lower value or higher value is guaranteed to be incomplete, as it stuck with local maxima.
- Other side the random walks are complete but extremely inefficient.
- Stimulated annealing is hybrid formation that achieve efficiency of hill-climbing with completeness of random walks.
- Instead of picking the best move, it picks the random move first. And if the move improves the solution algorithm will accept that move and go further with the hill-climbing approach.
- This was first applied to solve VLSI layout problems in the 1980s.

LOCAL BEAM SEARCH

- The local beam search keeps track of all required states in the memory.
- It begins with randomly generated states. At each step, all possible successors of all the states are generated.
- If any one is a goal state the algorithm halts, otherwise it selects the best successors from the complete list and repeats.
- In its simple form the algorithm become concentrated on a very small region of the state space, which is very expensive in nature like hill-climbing.
  - The solution is upgrade algorithm – Stochastic Beam Search

GENETIC ALGORITHM (GA)

- The GA is a variant of beam searching technique, in which successor states are generated by combining two parent states rather than by modifying a single state.
- Similar to beam search, GA begins with a set of randomly generated states, called the **population**.
- Each state is called **individual**, and is represented as a string of digits or alphabets.
- For production of the new states with combination of current state, GA uses **fitness function**. It should returns higher values for better state.

CONTINUE...

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

CONTINUE...

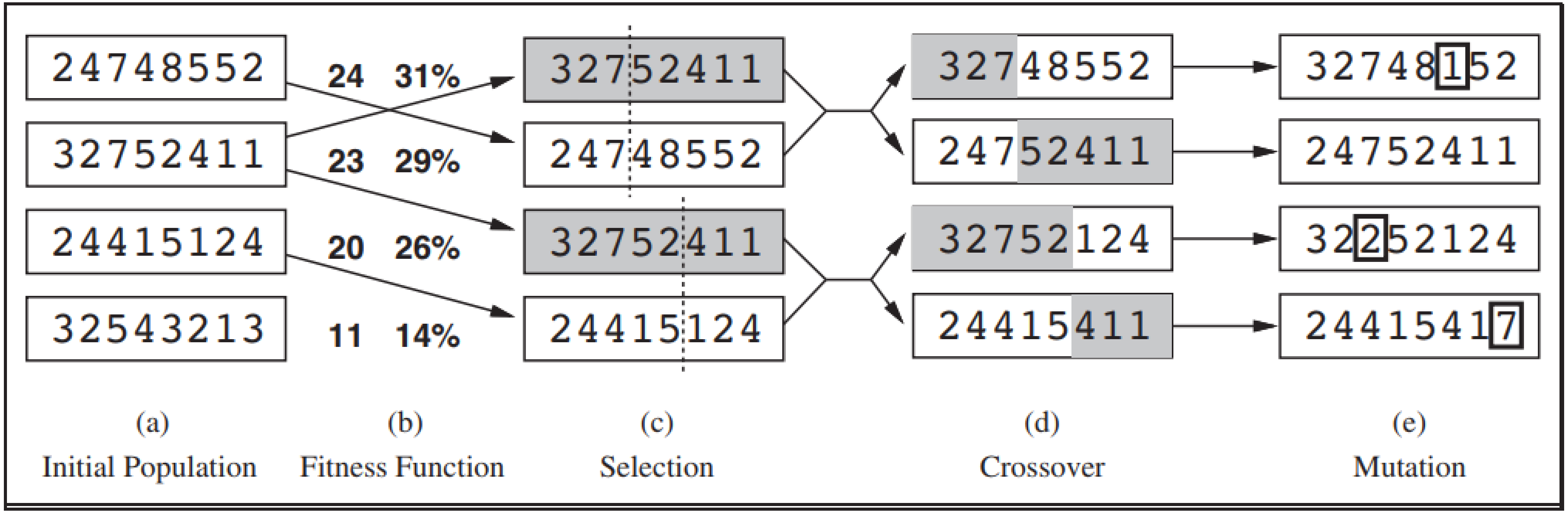


Figure 8: GA explanation with randomly selected 8-digit strings

CONTINUE…

- Applications of GAs,
  - ✓ Optimization problems
  - ✓ Circuit layout
  - ✓ Job scheduling

KNOWLEDGE-BASED AGENTS

KNOWLEDGE-BASED AGENTS

- The human way.
- Agents that can form representations of a complex world using inference to derive new representations about world, and use these new representations to deduce what to do in the task environment, are known as **Logical agents.**

CONTINUE…

- We learn by the process of reasoning that operates on internal representation of knowledge.
- This approach of human way intelligence is incorporated in AI by **Knowledge-based agents.**
- **Knowledge base** or **KB** is the central component.
- **KB** is a set of sentences. Each sentence is represented in a language called **knowledge representation language.**
- The **TELL** and **ASK** operations for inference – that derives new sentences from old.

AGENT PROGRAM

- Similar to ordinary agents, they are too having percept as input and they returns an action.
- The agent contains a knowledge base, KB, which may initially have some background knowledge.
- Upon calling agent program does three things.
  1. **TELL** – the KB what it perceives
  2. **ASK** – the KB what actions it should perform
  3. **TELL** – the KB which actions was chosen, and then the agent executes the action

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*))
  $t \leftarrow t + 1$
  return *action*

CONTINUE…

- **Knowledge level –** we specify only what the agent knows and what its goal are, to fix agent's overall behavior.
- Any knowledge-based agent can be built by simply TELLing it what the agent need to know, in order to achieve the solution in the given task environment.
- It starts with the empty knowledge base, we can TELL sentences one by one until agent knows how to operate in given environment. This is call **declarative approach** to system building.
- The contrast one is **procedural approach**, that encodes the desired behavior directly into agent's program.
- **Declarative vs procedural???**

THE WUMPUS WORLD PROBLEM

- PEAS description.
- Performance measure:
  - ✓**+1000 points** for picking up the gold — this is the goal of the agent.
  - ✓**−1000 points** for dying i.e. entering a square containing a pit or a live Wumpus monster.
  - ✓**−1** point for each action taken.
  - ✓**−10** points for using the arrow trying to kill the Wumpus to avoid performing unnecessary actions.

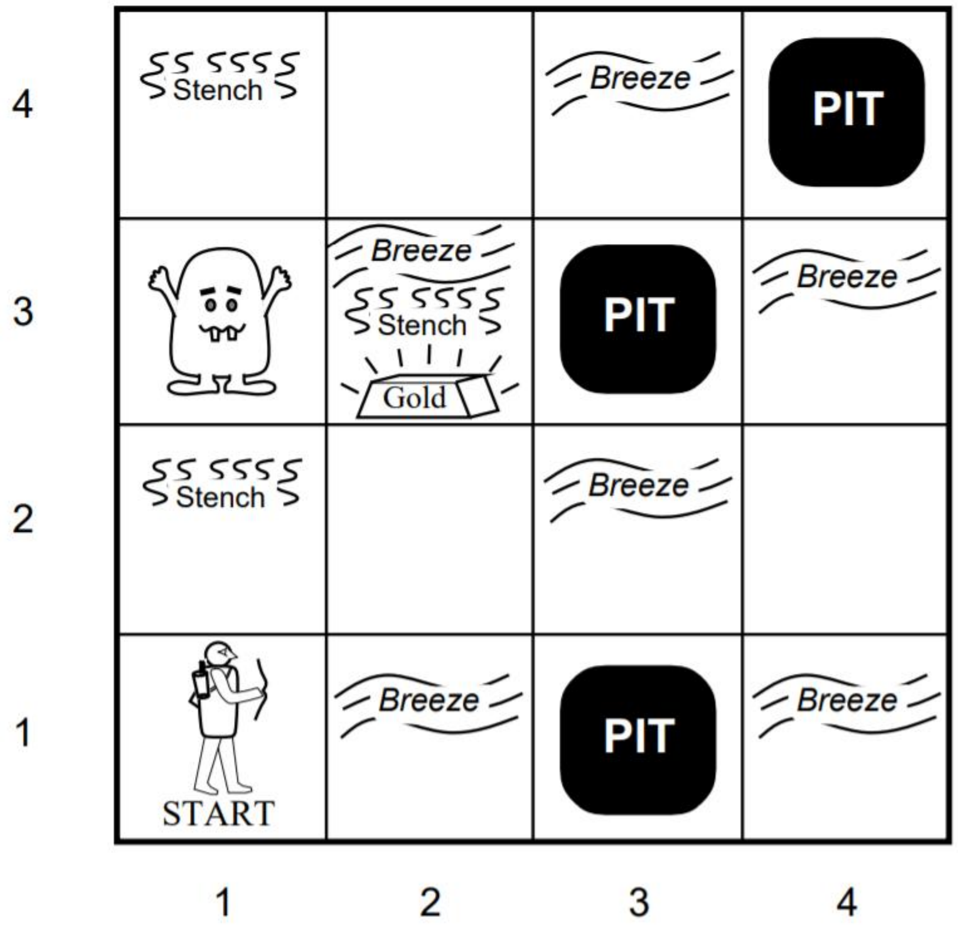
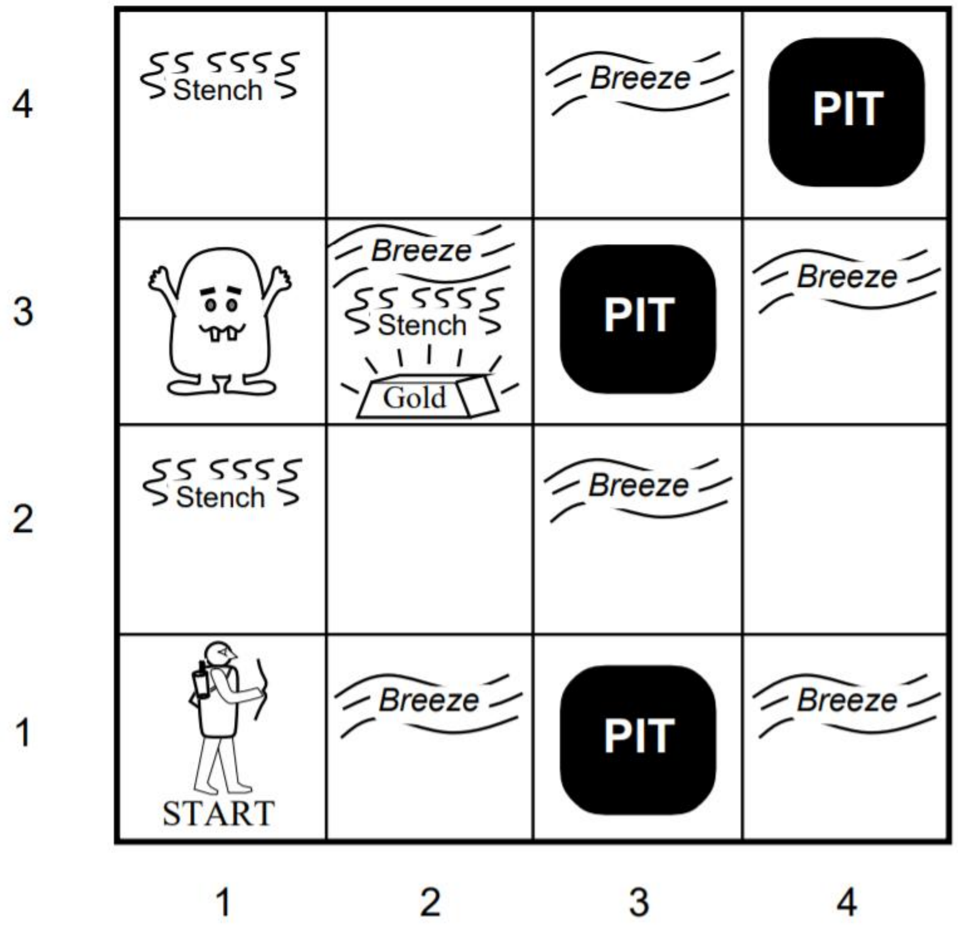


Figure 11: The Wumpus world

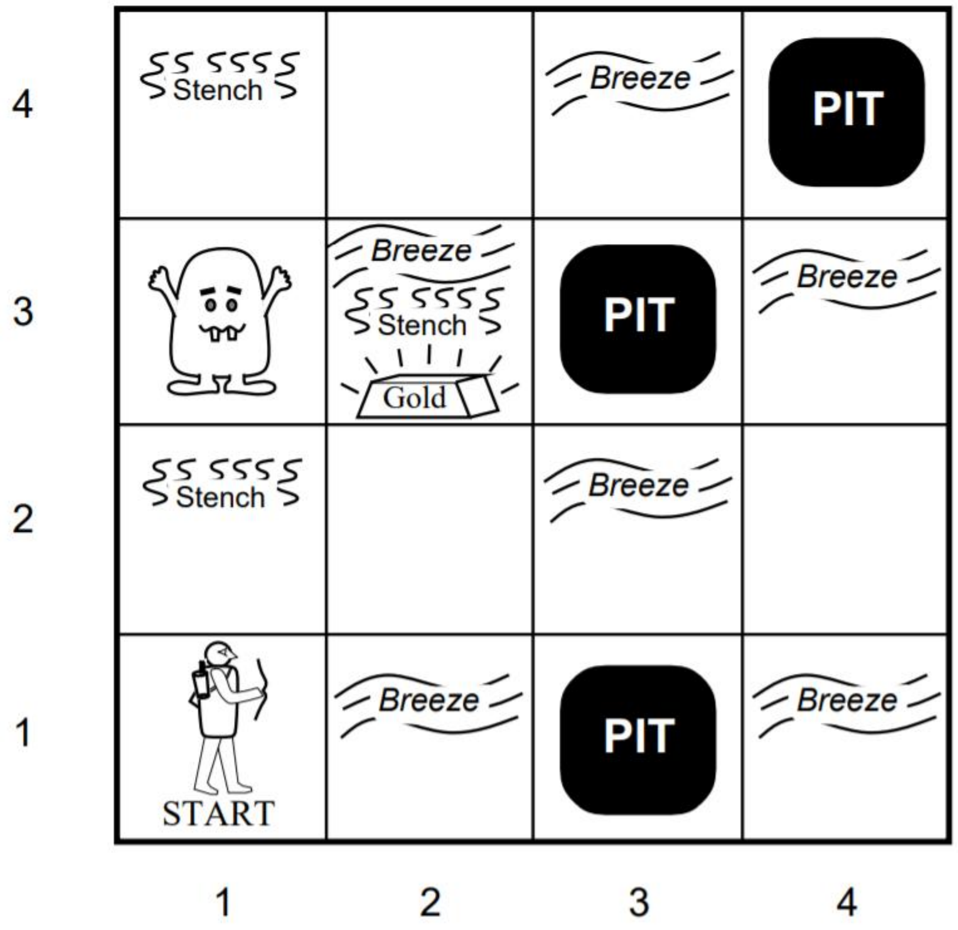
CONTINUE…

- Environment has A 4 × 4 grid of squares with,
  - ✓ The agent starting from square [1, 1] facing right.
  - ✓ The gold in one square.
  - ✓ The initially live Wumpus in one square, from which it never moves.
  - ✓ May be pits in some squares.
  - ✓ The starting square [1, 1] has no Wumpus, no pit, and no gold — so the agent neither dies nor succeeds straight away.



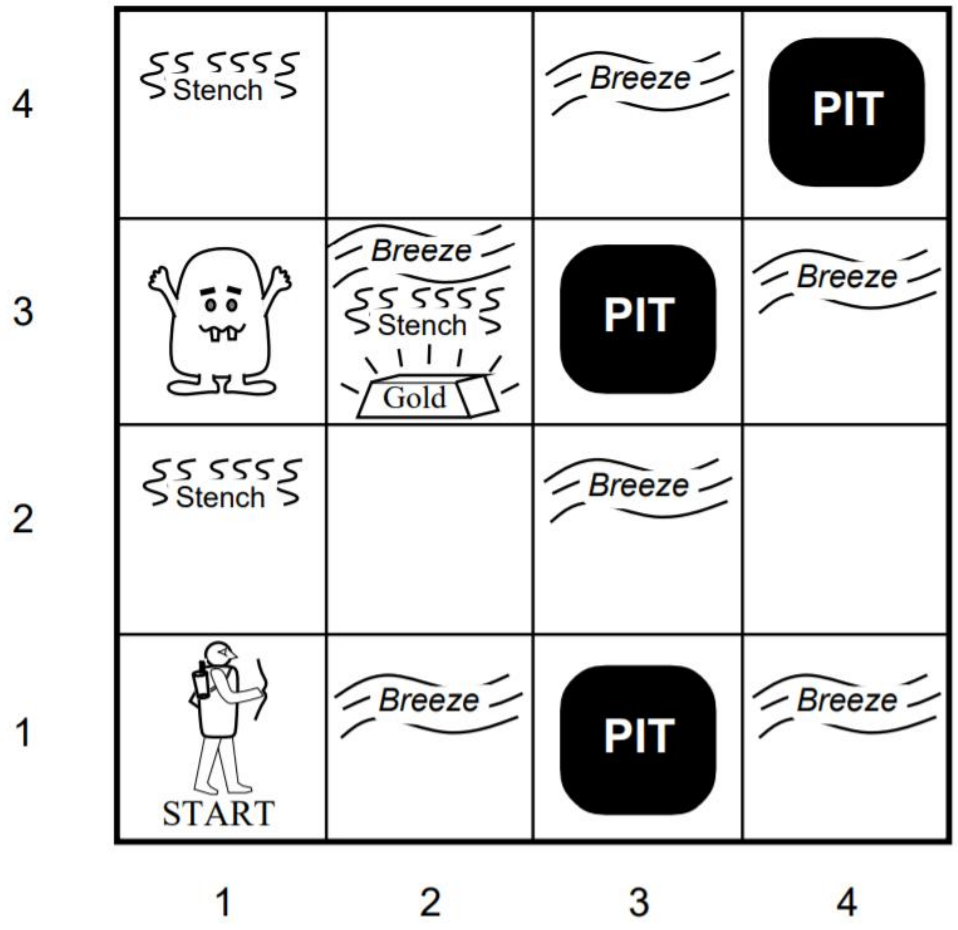
CONTINUE…

- Actuators:
  - ✓**Turn** 90◦ left or right.
  - ✓**Walk** one square forward in the current direction.
  - ✓**Grab** an object in this square.
  - ✓**Shoot** the single arrow in the current direction, which flies in a straight line until it hits a wall or the Wumpus.



CONTINUE…

- Sensors: Agent has 5 true/false sensors which reports,
  - ✓**Stench** when the Wumpus is in an adjacent square — directly, not diagonally.
  - ✓**Breeze** when an adjacent square has a pit.
  - ✓**Glitter** when the agent perceives the glitter of the gold in the current square.
  - ✓**Bump** when the agent walks into an enclosing wall (and then the action had no effect).
  - ✓**Scream** when the arrow hits the Wumpus, killing it.



- **Top left:** Agent A starts at [1, 1] facing right.
- [1,1] is **OK** as the state is not deadly.
- Agent A gets the percept "**S**tench = **B**reeze = **G**litter = **B**ump = **S**cream = false".
- Agent A infers from this percept and **β** that its both neighbouring squares [1, 2] and [2, 1] are also OK.

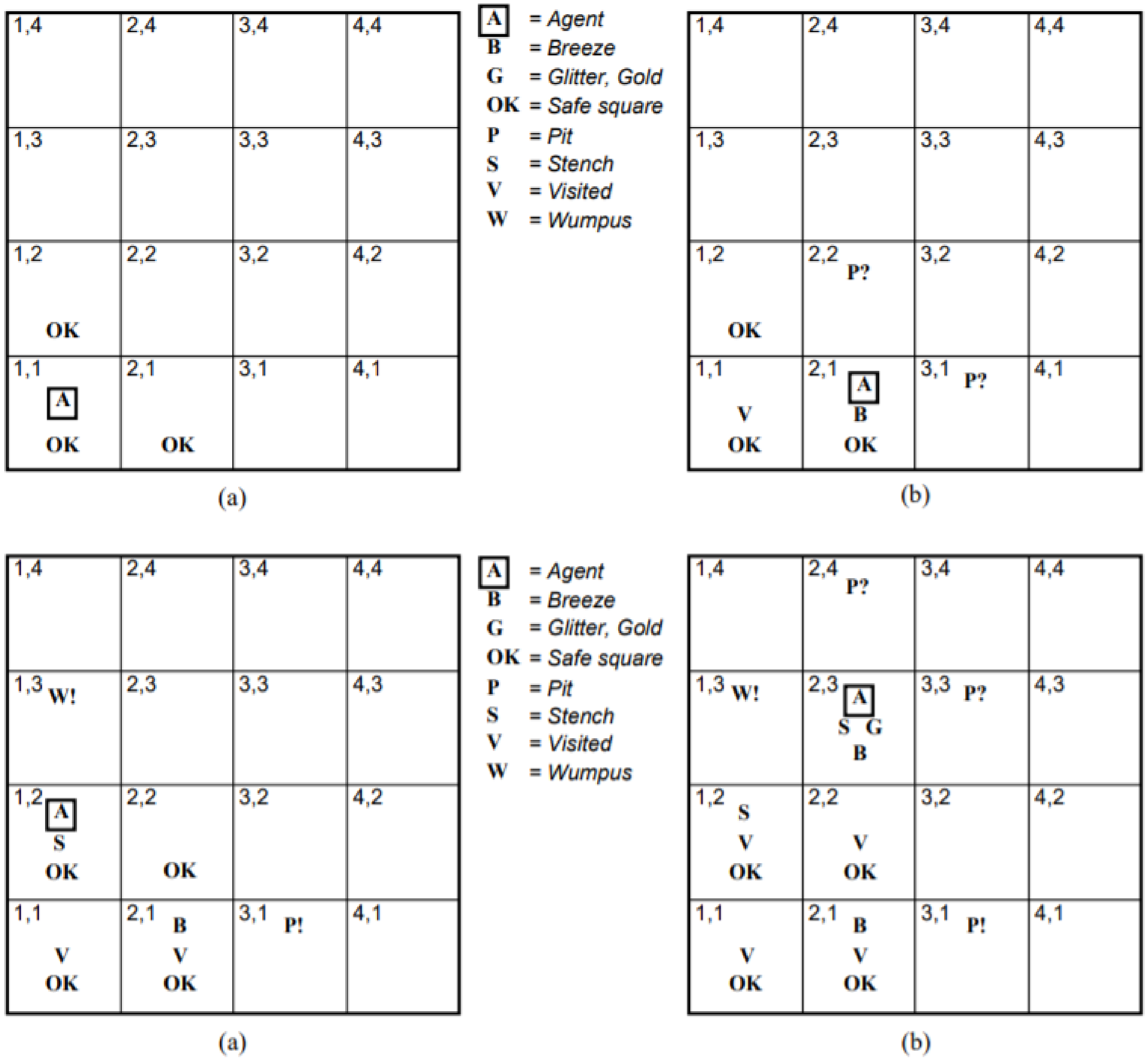
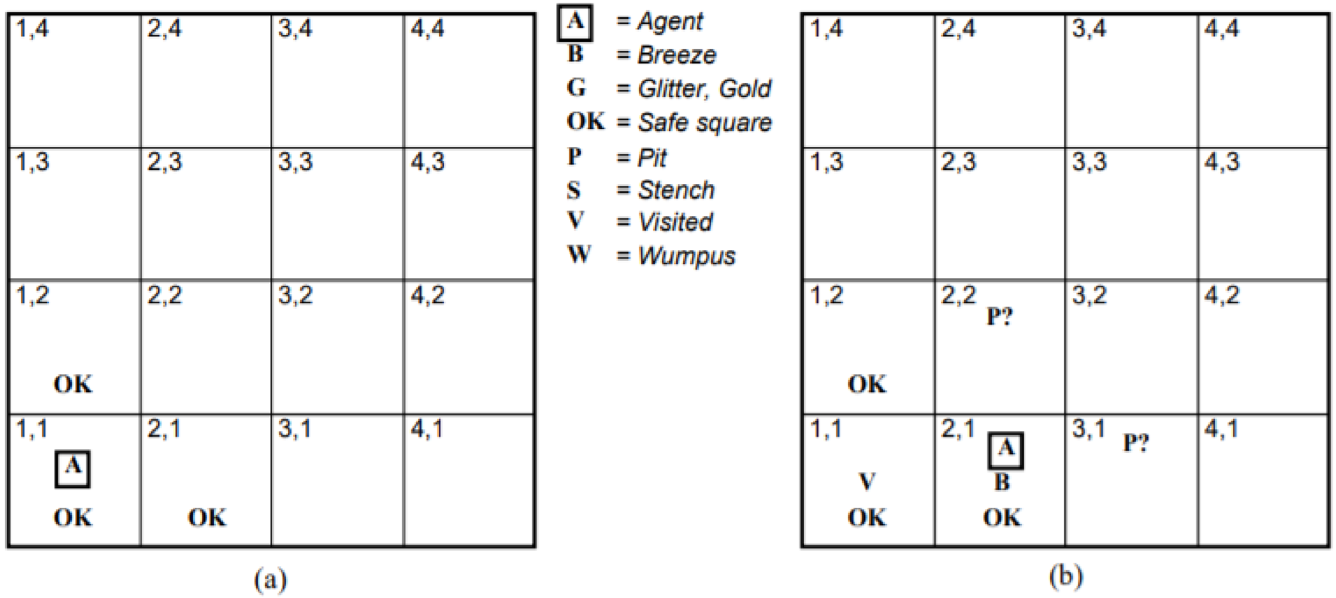
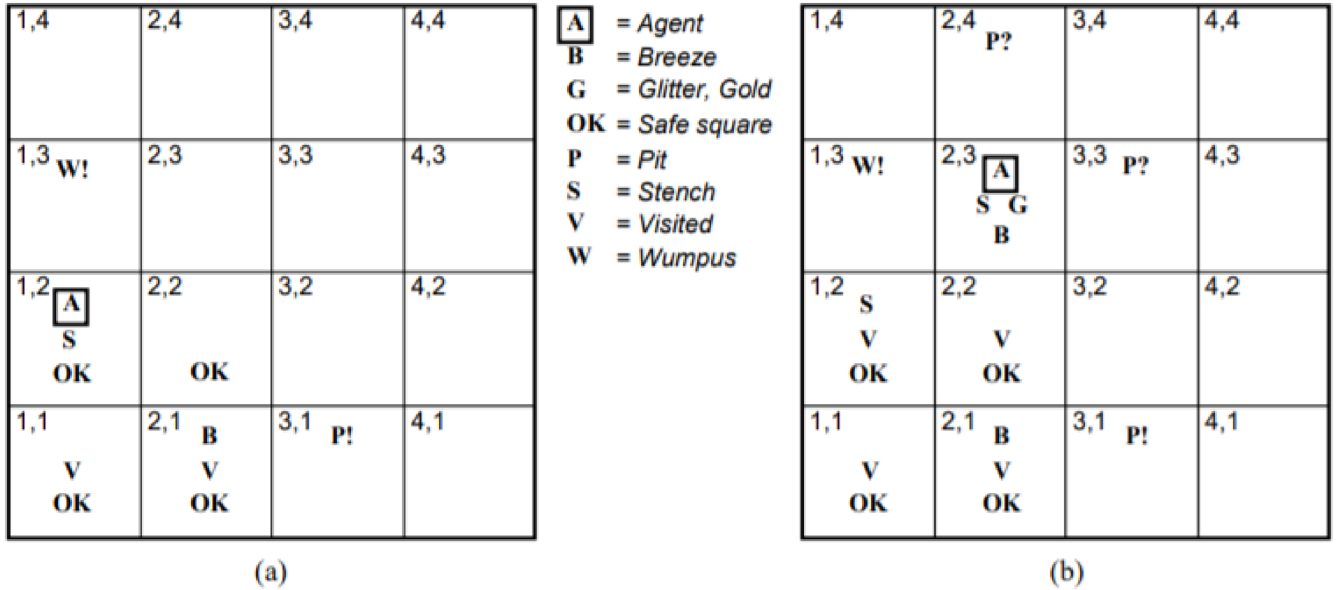


Figure 12: The Wumpus adventure

- **Top Right:** Agent A will only move to **OK** squares.
- Agent A walks into [2, 1], because it is **OK**, and in the direction where agent A is facing, so it is cheaper than the other choice [1, 2]. Agent A also marks [1, 1] Visited.
- Agent A perceives a **Breeze** but nothing else.
- Agent A infers: "At least one of the adjacent squares [1, 1], [2, 2] and [3, 1] must contain a Pit. There is no Pit in [1, 1] by background knowledge **β**. Hence [2, 2] or [3, 1] or both must contain a Pit."
- Hence agent A cannot be certain of either [2, 2] or [3, 1], so [2, 1] is a dead end for a Agent A.

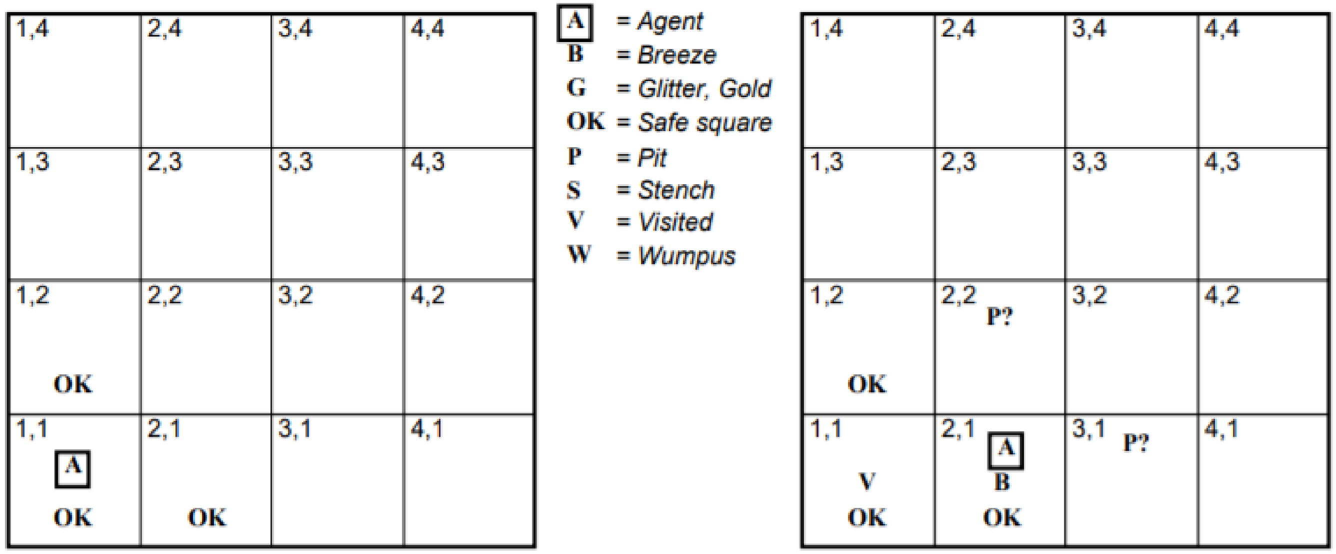


(a) (b)

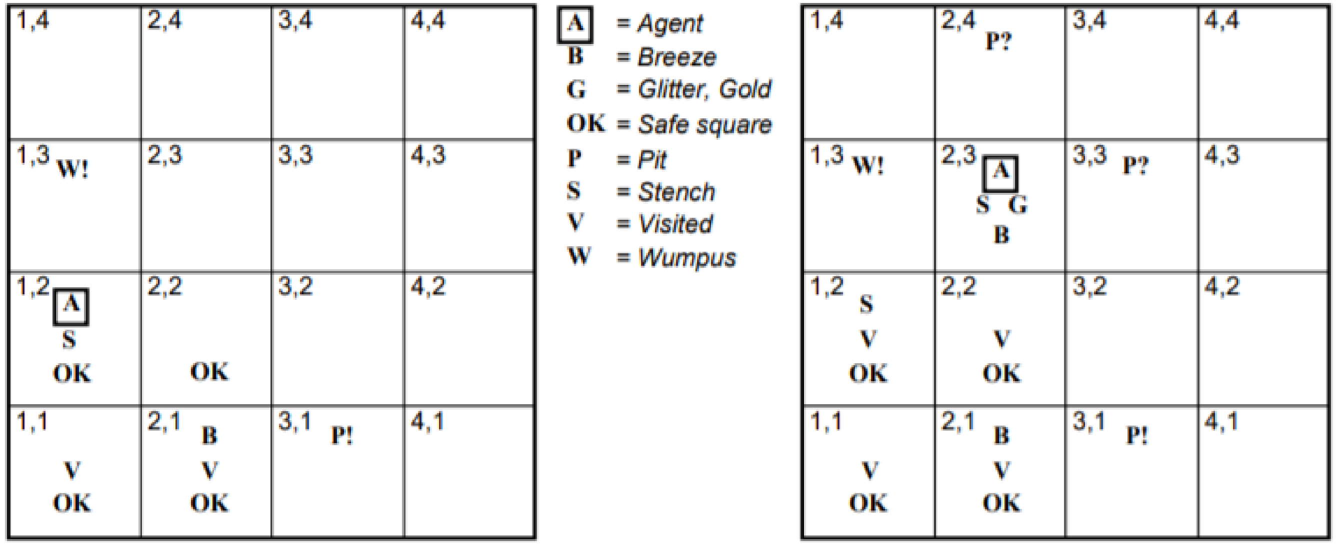


(a) (b)

- **Bottom Left:** Agent A has turned back from the dead end [2, 1] and walked to examine the other OK choice [1, 2] instead.
- Agent A perceives a Stench but nothing else.
- Agent infers that the Wumpus is in an adjacent square. It was not in [1,1], not in [2,1] either, else it would have sensed the stench in [2,1]. Hence, it is in [1,3].
- There is not breeze here in [1,2] so there is no pit in any adjacent square, i.e. [2,2].
- It finally infers [2,2] as OK.

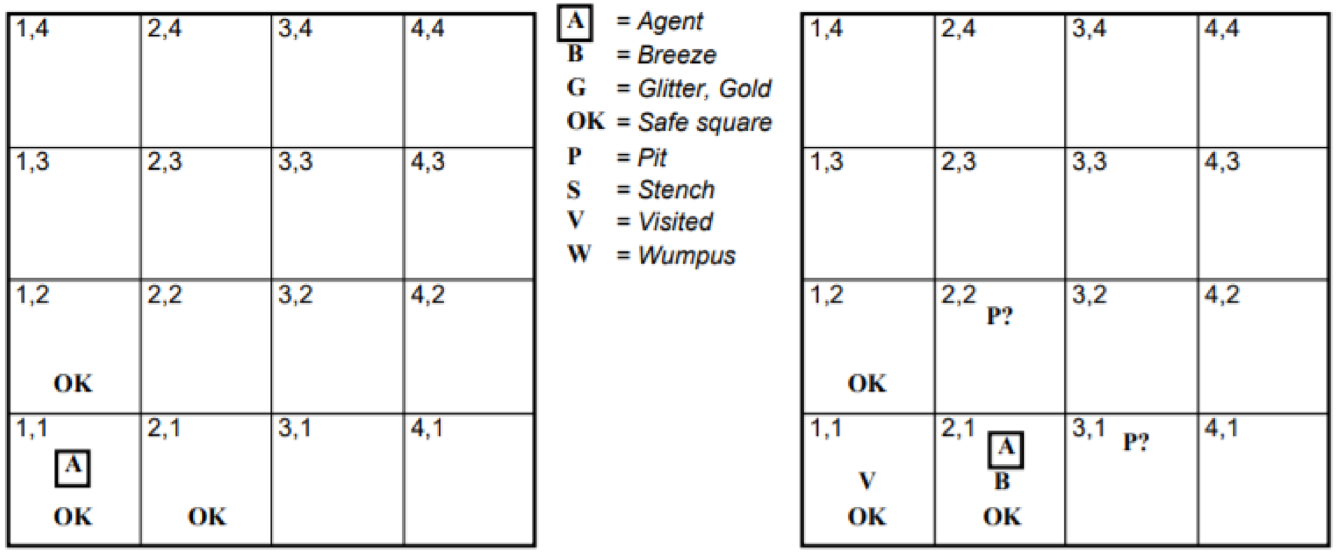


(a) (b)

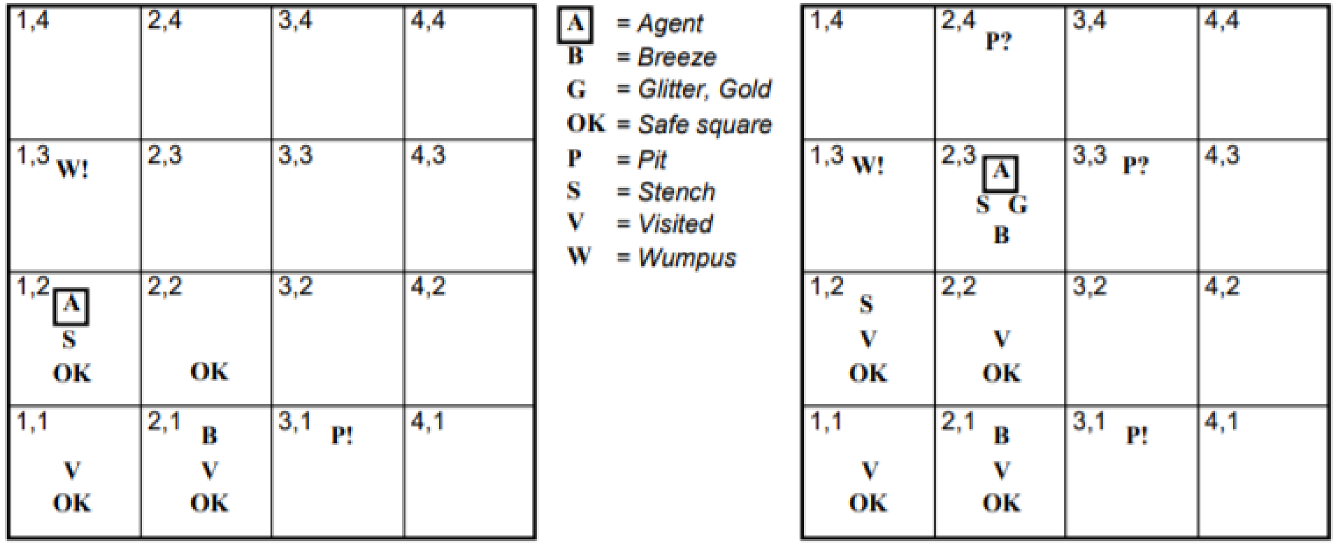


(a) (b)

- **Bottom Right:** Agent A walks to the only unvisited OK choice [2, 2].
- There is no Breeze here, and since the square of the Wumpus is now known too, [2, 3] and [3, 2] are OK too.
- **Agent A walks into [2, 3] and senses the Glitter there, so he grabs the gold and succeeds.**



(a) (b)



(a) (b)

LOGICS

LOGIC AND INFERENCE

- For any agent to act in a meaningful manner, it must have some kind of symbolic representations.
- By intelligent activity we means not just optimization or survival, but something that involves awareness of goals, awareness of situation and informed decision making.
- **Formal logic:**
  - It is primary machinery for realizing the reasoning.
  - Given a set of completely true statements, the machinery determines what other sentences are argued to be true.

CONTINUE…

- Formal logic example:

  FROM: “All kings are brave.”

  AND: “Krishna is a king.”

  INFER: “Krishna is brave. ”

  FROM: “All detectives are rich.”

  AND: “Byomkesh is detective.”

  INFER: “Byomkesh is rich. ”

Sounds falsy inference

ENTAILMENT

- In some way all humans have instinct curiosity, a desire to know the truth about something or the other.
- To a large extent we rely upon our senses.
- One another source that we unknowingly utilize is **LOGIC**.
- In logic we are focused to deal with only true statements.
- So, given a collection of true statements, also called **premises**, we are interested in knowing "what other/new statements are logically entailed/made true with the help of premises."

We ignore the rumours and other speculations that sounds unreliable.

PROOFS

- We know, entailment is concerned with true statements.
- To determine, whether the statement is true or not true is not straightforward.
- Logic uses concept of proofs. Which is made up of a sequence of inference steps.
- Each inference step allow us to add one more new sentence to the existing set of sentences.
- Each inference step is based on *rule of inference*.
- Example:

  FROM: "ALL x's are y's"
  AND: "k is x"
  INFER: "k is y"

SOUNDNESS AND COMPLETENESS

- A logic is said to be sound if it produces only true statements or in other words if it does not produce false statements.
- A logic is complete if it produces all true statements.
- Formally,
  - If P, provable statement is a subset of T, true statement then logic is SOUND.
  - If T is subset of p, then logic is COMPLETE.

PRINCIPLES BEHIND LOGICS

- The actual syntax in which the logical statements are written as formulas is just a "print-out" of the underlying data structure.
- This underlying data structure is the parse tree of the formula.
- From a programming viewpoint, implementations of logical inference create from existing such structures new ones in ways which respect their **meaning.**
- This **meaning** or **semantics** of a statement defines whether it is true or false in the given possible world.
- In our agent viewpoint, possible world = possible state of the environment.

CONTINUE...

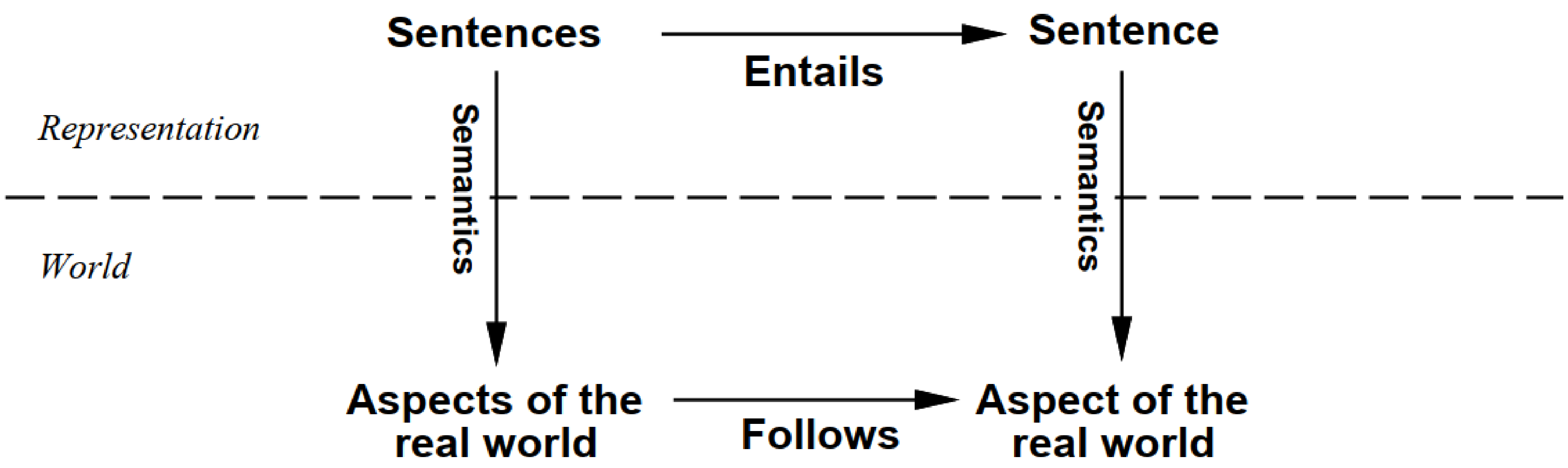


Figure 13: The world of representation

PROPOSITIONAL LOGIC

- The smallest "addressable unit" of propositional logic is a whole sentence which can be either true or false.
- That is, any expression η such that
  - ✓ "Is it the case that η?"
- Asking this about η = "Manoj is devotee of Krishna"
- On the other hand, e.g. neither "Is it the case that devotee?" nor "Is it the case that Krishna?" is a meaningful question.
- Note that "Is it the case that Manoj devotee?" is another meaningful question — but this is a different sentence.

CONTINUE...

- The **syntax** of propositional logic defines the allowable sentences.
- The **atomic sentences** consist of a single **proposition symbol.**
- Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W1,3 and North.
- The names are arbitrary but are often chosen to have some mnemonic value—we use W1,3 to stand for the proposition that the Wumpus is in [1,3].

CONTINUE...

- **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives.**
- There are five common connectives to be used:

1. $\neg$ (not). A sentence such as $\neg W1,3$ is called the **negation** of $W1,3$.
2. $\wedge$ (and). A sentence whose main connective is $\wedge$, such as $W1,3 \wedge P3,1$, is called a **conjunction.**
3. $\vee$ (or). A sentence using $\vee$, such as $(W1,3 \wedge P3,1) \vee W2,2$, is a **disjunction** of the **disjuncts** $(W1,3 \wedge P3,1)$ and $W2,2$.
4. $\Rightarrow$ (implies). A sentence such as $(W1,3 \wedge P3,1) \Rightarrow \neg W2,2$ is called an **implication**
5. $\Leftrightarrow$ (if and only if). The sentence $W1,3 \Leftrightarrow \neg W2,2$ is a **biconditional.**

CONTINUE...

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \mid ComplexSentence \\
AtomicSentence & \rightarrow & True \mid False \mid P \mid Q \mid R \mid \ldots \\
ComplexSentence & \rightarrow & (\, Sentence\,) \mid [\, Sentence\,] \\
& \mid & \neg\, Sentence \\
& \mid & Sentence \land Sentence \\
& \mid & Sentence \lor Sentence \\
& \mid & Sentence \;\Rightarrow\; Sentence \\
& \mid & Sentence \;\Leftrightarrow\; Sentence \\
\\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 14: Formal grammar of proposition logic with BNF (Backus-Naur Form) notations

BNF GRAMMAR

- The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways.
- To eliminate the ambiguity we define a precedence for each operator.
- The "not" operator ($\neg$) has the highest precedence, which means that in the sentence $\neg A \land B$ the $\neg$ binds most tightly, giving us the equivalent of $(\neg A) \land B$ rather than $\neg(A \land B)$.

PROPOSITIONAL THEOREM PROVING

- Applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models, called theorem proving.
- If the number of models is large but the length of the proof is short, then theorem proving can be more efficient.
- **Logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. i.e. $\alpha \equiv \beta$.
- Any two sentences α and β are equivalent only if each of them entails the other:
  - ✓ $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$
- **Validity**: A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true.

CONTINUE...

- **Satisfiability**: A sentence is satisfiable if it is true in, or satisfied by, *some* model.

REFERENCES

[1] Artificial Intelligence – A Modern Approach, 3rd Edition, Stuart Russell and Peter Norvig, Pearson Publication

[2] https://www.humanbrainfacts.org/human-brain-memory.php

[3] https://operativeneurosurgery.com/doku.php?id=hippocampus