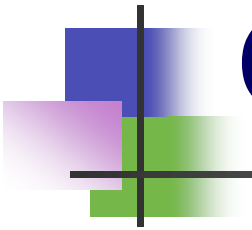
323
ARTIFICIAL INTELLIGENCE & EXPERT SYSTEMS

Solving problems by searching -1

Chapter 3

Dr. Zeynep ORMAN



Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - Uninformed search – they are given no information about the problem other than its definition
    - depth-first, breadth-first
  - Informed search – have some idea of where to look for solutions
    - best-first, hill-climbing

Goal-based Agent

- A goal-based agent needs to achieve certain goals.
- Many problems may be represented as a set of states and a set of rules of how one state is transformed to another.
- The agent must choose a sequence of actions to achieve the desired goal.

Problem-solving agents

- The simplest agent we discussed were the reflex agents.
- These agents had some problems especially when there are many states to generate and store.
- Goal-based agents can succeed this by considering future actions and their outcomes.
- Problem-solving agent is one kind of goal-based agent.

Problem-solving agents

Problem Solving

- Goal formulation, based on the current situation and the agent's performance measure.
- Problem formulation, is the process of deciding what actions and states to consider, given a goal.
- The process of looking for a sequence of actions is called search.
- A search algorithm takes a problem as input and returns a solution in the form of a action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: 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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Problem-solving agents

1. It first formulates a goal and a problem,
2. Searches for a sequence of actions that would solve the problem.
3. Then, executes the actions one at a time.

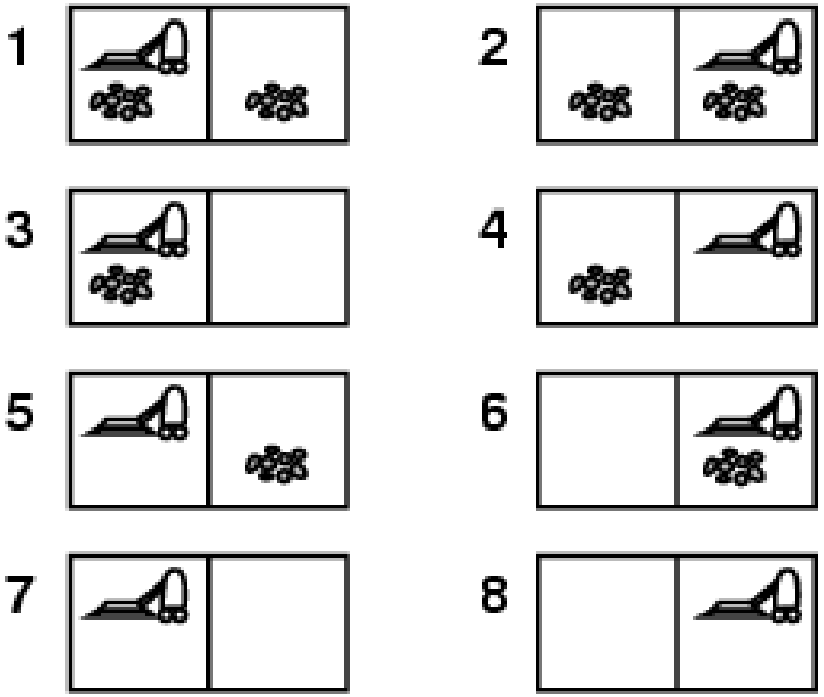
- When this is complete, it formulates another goal and starts over.
  - We first describe the process of problem formulation and then various algorithms for the SEARCH function.
  - We will not discuss the workings of the UPDATE-STATE and FORMULATE-GOAL functions in this lesson.

Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence of actions.
- Non-observable → sensorless problem
  - Agent may have no idea where it is; solution is a sequence of actions.
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search, execution
- Unknown state space → exploration problem
  - states and actions are unkown
  - the agent must act to discover them.
  - an extreme case of contingency problems.

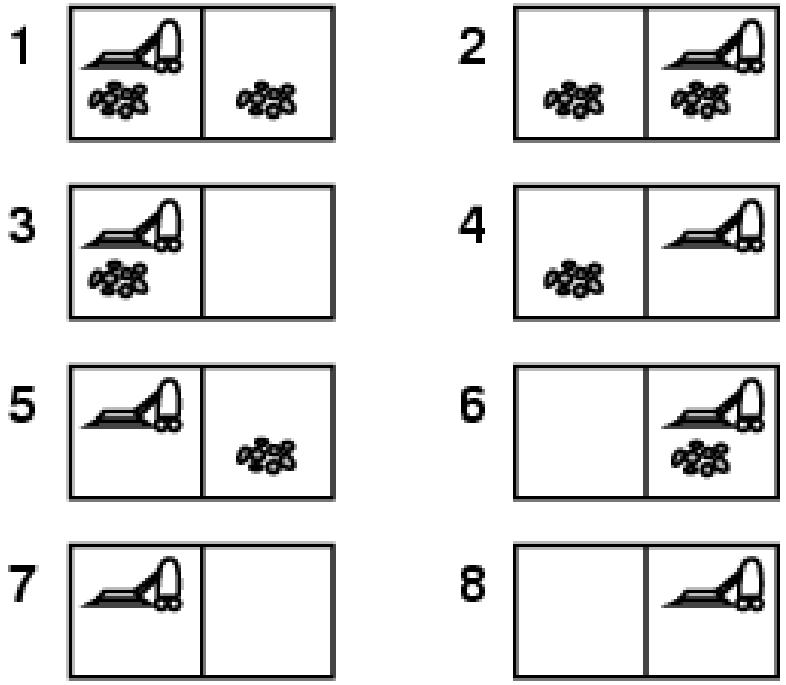
Example: vacuum world

- Single-state, start in #5. Solution?



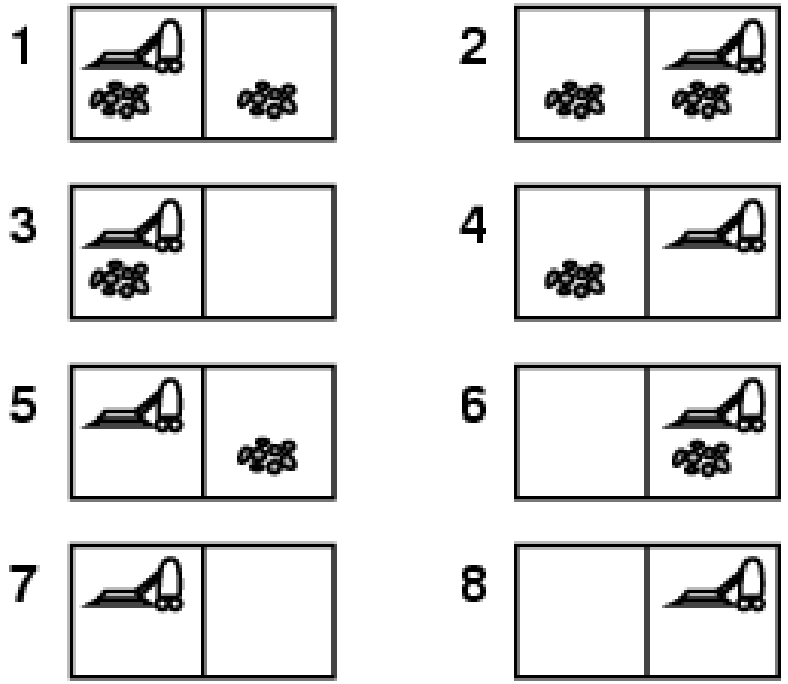
Example: vacuum world

- Single-state, start in #5. Solution? *[Right, Suck]*
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution?



Example: vacuum world

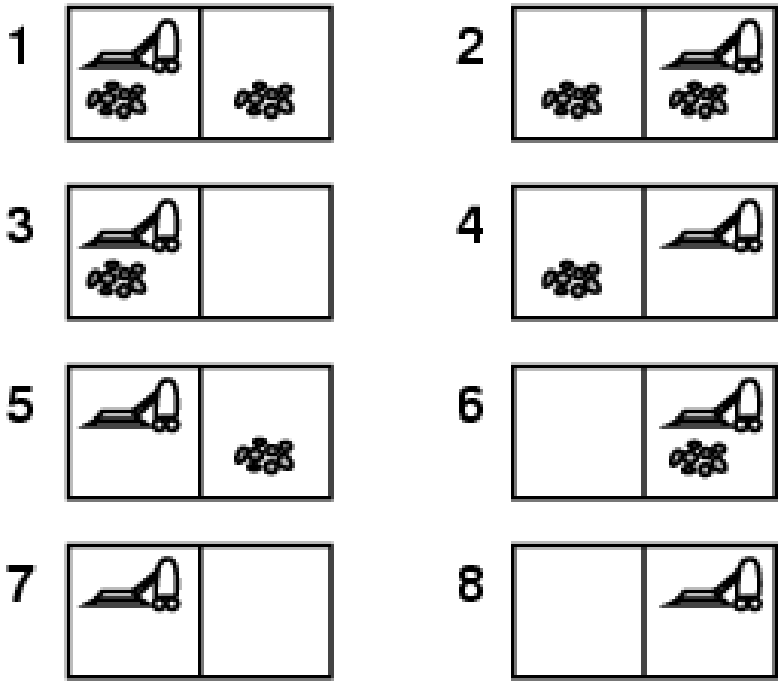
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



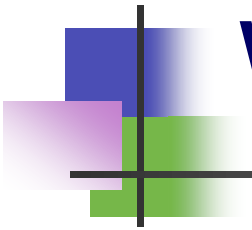
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - i.e., start in #5 or #7
    Solution?

Example: vacuum world

- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*



Why Search?

- Successful
  - Success in game playing programs based on search.
  - Many other AI problems can be successfully solved by search.

Why Search?

- Practical
  - Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them. Search can also be useful in approximation

    (e.g., local search in optimization problems).
  - Often specialized algorithms cannot be easily modified to take advantage of extra knowledge. Heuristics search provides a natural way of utilizing extra knowledge.

Why Search?

- Some critical aspects of intelligent behavior, e.g., planning, can be naturally cast as search.
- Example, a holiday in Jamaica

Things to consider

- Prefer to avoid hurricane season.
- Rules of the road, larger vehicle has right of way (especially trucks).
- Want to climb up to the top of Dunns river falls.
  - But you want to start

  your climb at 8:00 am

  before the crowds arrive!

- Want to swim in the Blue Lagoon

- Want to hike the Cockpit Country

- No roads, need local guide and supplies.

- Easier goal, climb to the top of Blue Mountain
- Near Kingston, organized hikes available.
- Need to arrive on the peak at dawn, before the fog sets in.
- Can get some real Blue Mountain coffee!

How do we plan our holiday?

- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is "hypothetical" reasoning.
  - e.g., if I fly into Kingston and drive a car to Port Antonio, I'll have to drive on the roads at night. How desirable is this?
  - If I'm in Port Antonio and leave at 6:30am, I can arrive to Dunns river falls by 8:00am.

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
  - "what state will I be in after the following sequence of events?"
- From this we can reason about what sequence of events one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Search

- There are many difficult questions that are not resolved by search.
  - In particular, the whole question of how does an intelligent system formulate its problem as a search problem is not addressed by search.
- Search only shows how to solve the problem once we have it correctly formulated.

The formalism

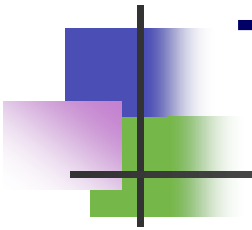
- To formulate a problem as a search problem we need the following components:

1. Formulate a state space over which to search. The state space necessarily involves abstracting the real problem.

2. Formulate actions that allow one to move between different states. The actions are abstractions of actions you could actually perform.

3.Identify the initial state that best represents your current state and the desired condition one wants to achieve.

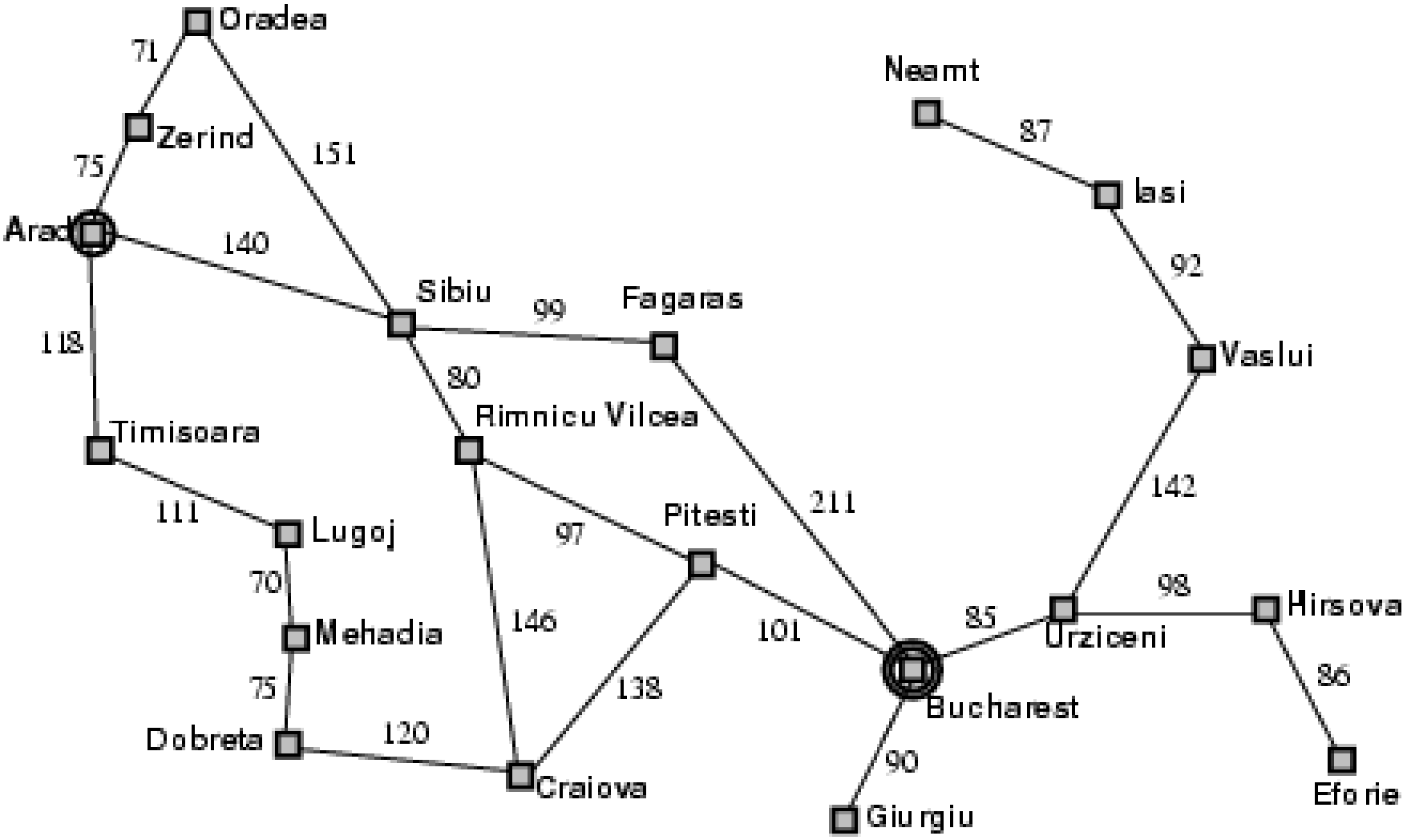
4.Formulate various heuristics to help guide the search process.



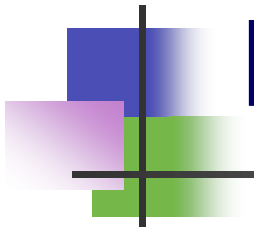
The formalism

- Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.
  - A solution to the problem will be a sequence of actions/moves that can transform your current state into state where your desired condition holds.

Example:Traveling from Arad To Bucharest



Example: Romania



- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest.
- Formulate goal:
  - be in Bucharest.
- Formulate problem:
  - states: various cities
  - actions: drive between neighboring cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Well-defined problems

A problem is defined by **four** items:

1. initial state that the agent starts in e.g., "at Arad"
2. A description of the possible actions available to the agent. It uses a successor function.

   -- Given a particular state x, $SUCCESSOR\text{-}FN(x)$ returns a set of $\langle action, successor \rangle$ ordered pairs.

   - e.g., from the state $In(Arad) = \{$ *<Go(Sibiu),In(Sibiu)>, <Go(Zerind), In(Zerind)>,<Go(Timisoara), In(Timisoara)>* $\}$

   -- A path in the state space is a sequence of states connected by a sequence of actions.

Well-defined problems

3. The goal test which determines whether a given state is a goal state.

   A goal is a description of a set of desirable states of the world.

   Goal states are often specified by a goal test which any goal state must satisfy.

   - explicit, e.g., x = "at Bucharest"
   - implicit, e.g., *NoDirt(x)*

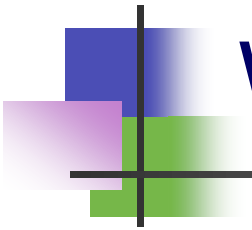
Well-defined problems

4. A path cost function that assigns a numeric cost to each path.

   path → positive number

   usually path cost = sum of step costs

   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost of taking action *a* to go from state *x* to state *y*, assumed to be ≥ 0.



Well-defined problems and solutions

- A solution to a problem is a path from the initial state to a goal state.
  - Solution quality is measured by the path cost function.
  - **An optimal solution** has the lowest path cost among all solutions.

Selecting a state space

- Real world is absurdly complex :
  - the traveling companions
  - what is on the radio
  - the scenery out of the window
  - how far it is to the next rest stop
  - the condition of the road
  - the weather, etc.
- The process of removing detail from a representation is called "abstraction".
- State space must be abstracted for problem solving.

Selecting a state space

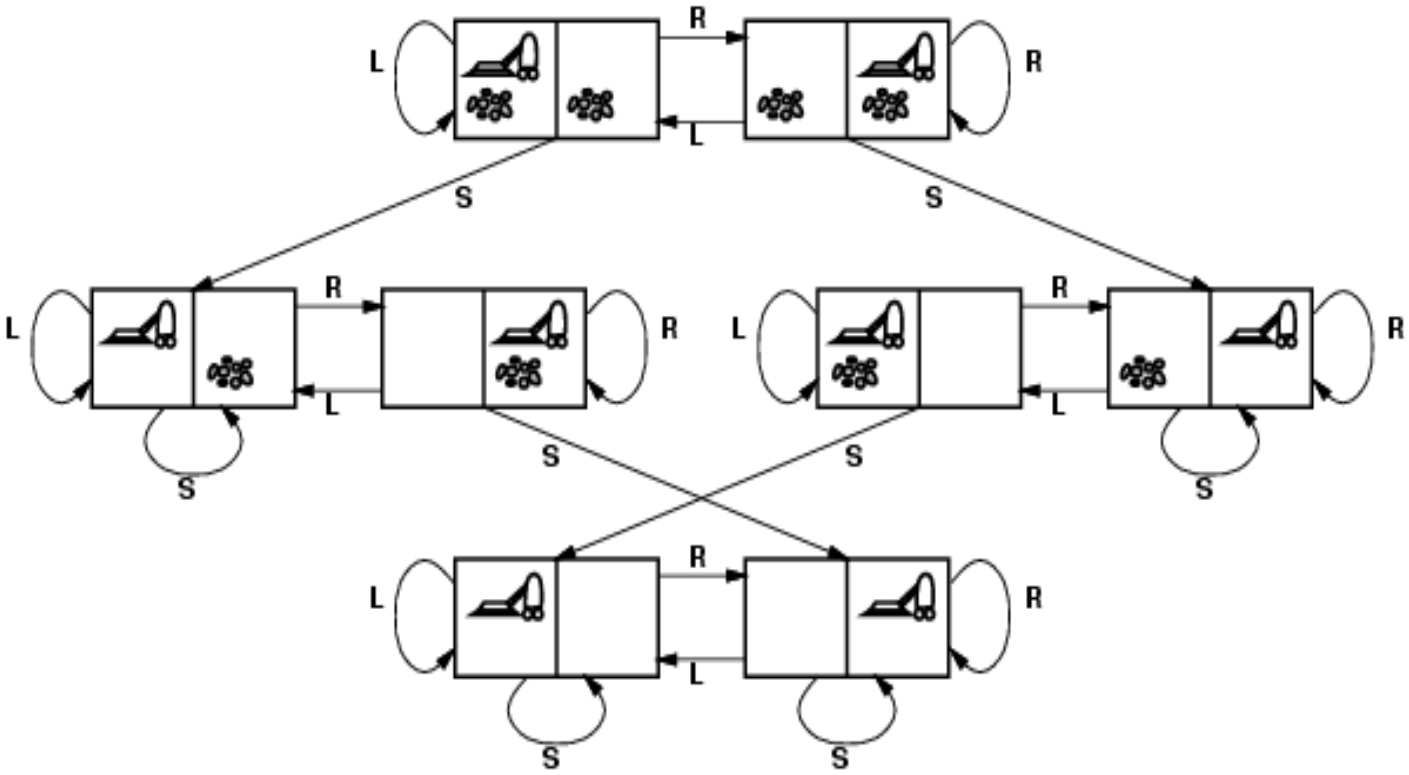
- (Abstract) state = set of real states.
- (Abstract) action = complex combination of real actions.
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world.

Selecting a state space

- Each state is an abstract representation of the agent's environment.
  - It is an abstraction that denotes a configuration of the agent.
- Each abstract action should be "easier" than the original problem.
- A good abstraction involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

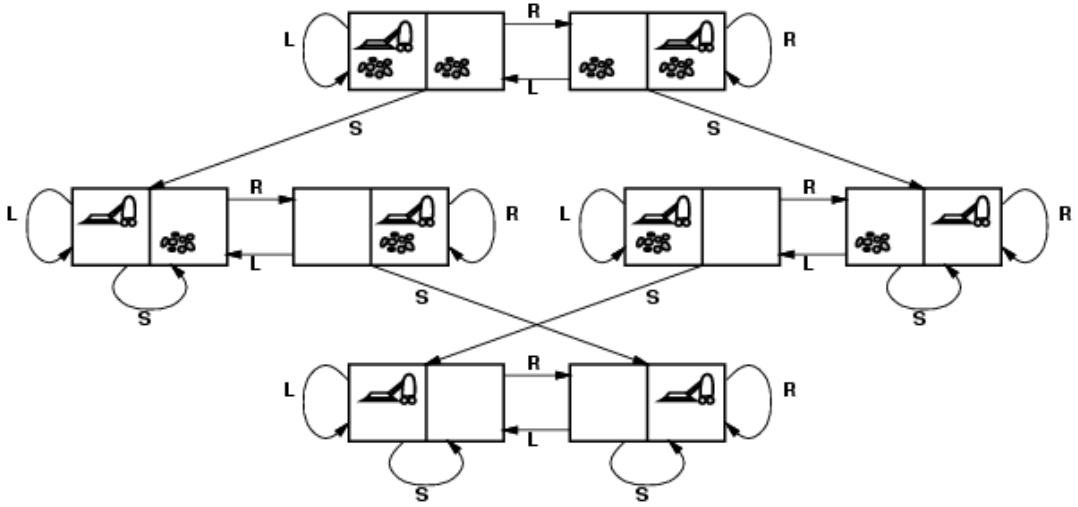
EXAMPLE PROBLEMS

Vacuum world state space graph



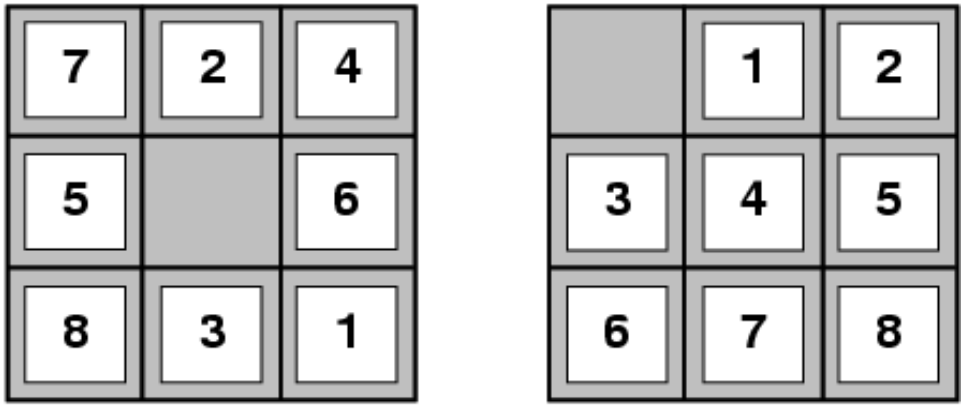
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



- states? agent is in one of the two locations, each of which might or might not contain dirt. $2x2^2=8$ possible world states.
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? each step costs 1, so the path cost is the number of steps in the path. (0 for $NoOp$)

The 8-puzzle



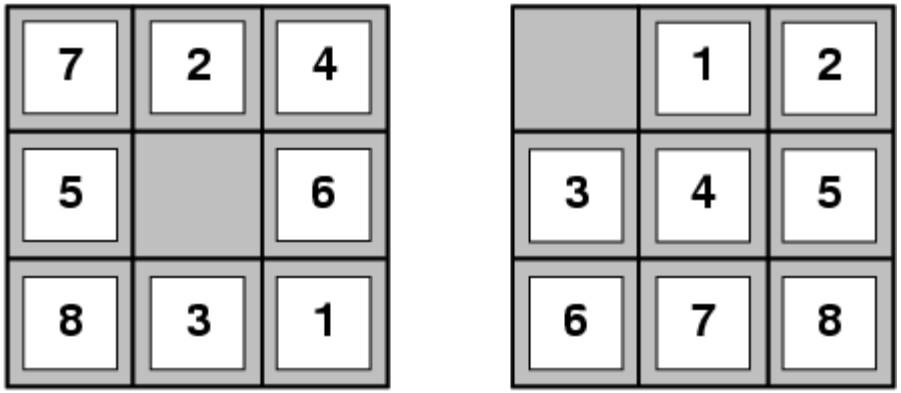
- states?
- actions?
- goal test?
- path cost?

- Consists of a 3x3 board with 8 numbered tiles and a blank space.

- A tile adjacent to the blank space can slide into the space.

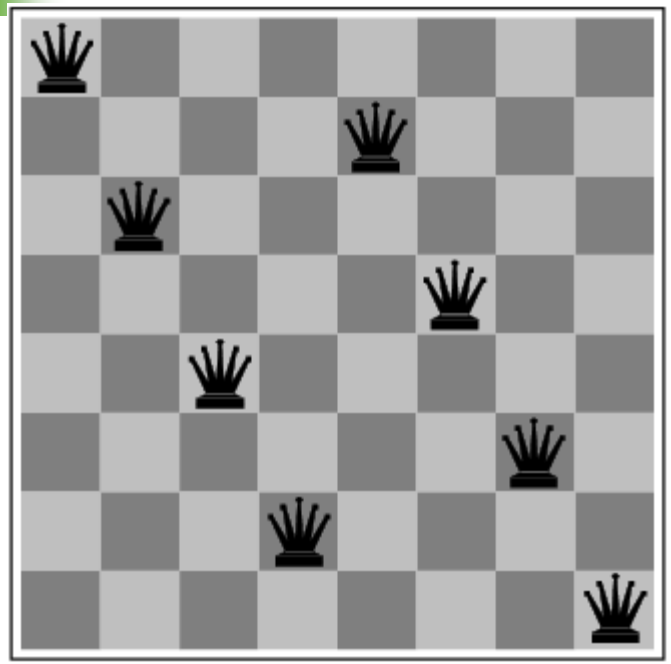
- The object is to reach a specified goal state, such as the one shown on the right.

The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move – number of steps in the path.

8-queens problem



-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.

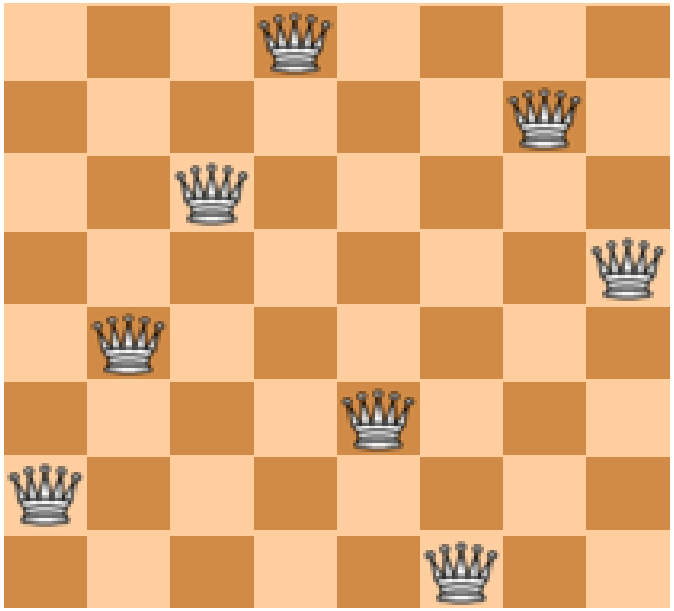
Almost a solution to the 8-queens problem.

states?

actions?

goal test?

8-queens problem



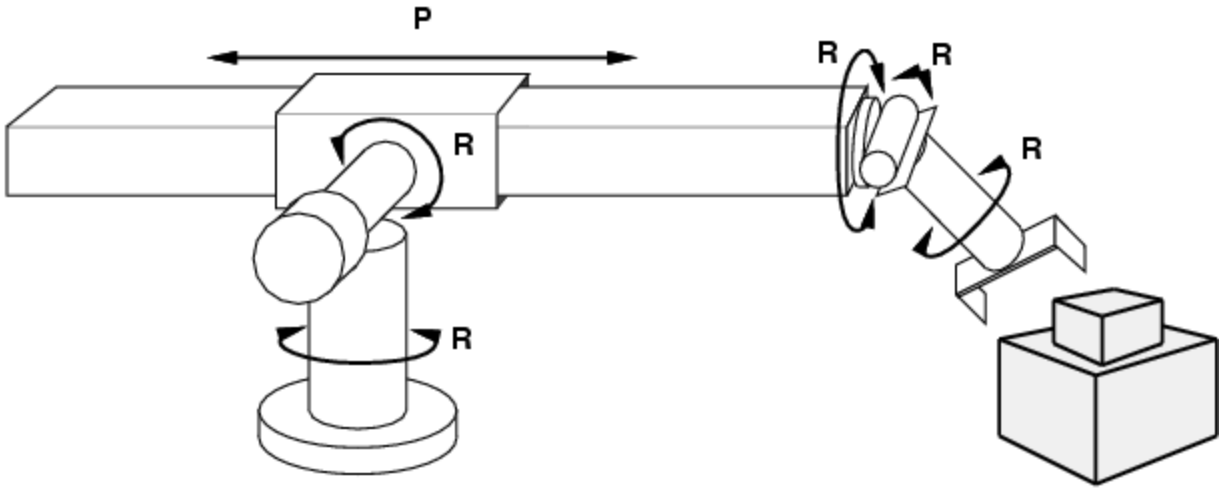
One possible solution to the 8-queens problem.

states? any arrangement of 0 to 8 queens on the board is a state.

actions? add a queen to any empty space.

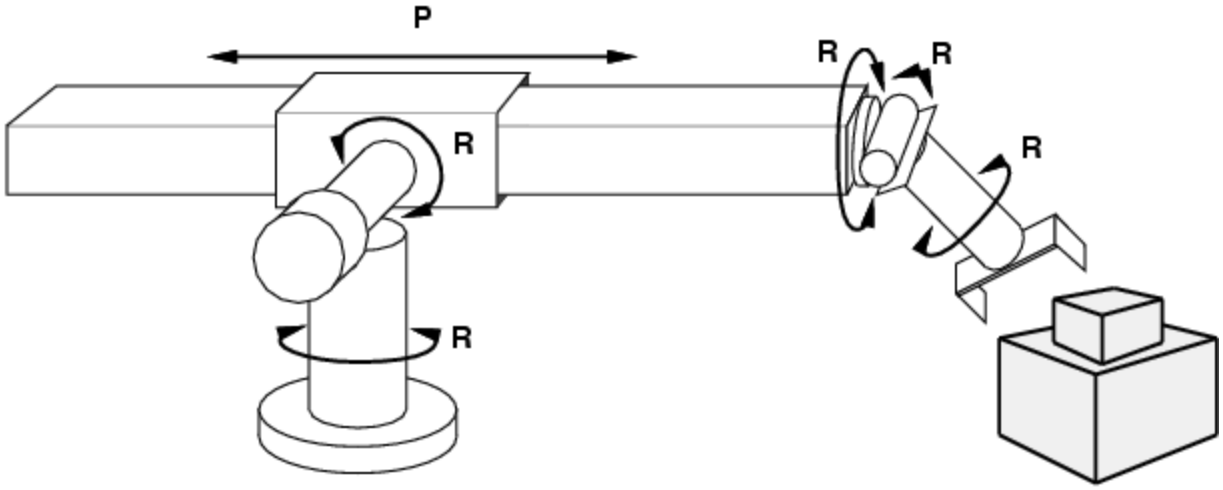
goal test? 8 queens on the board, none attacked.

Real-life example: Robotic Assembly



The aim is to find an order in which to assemble the parts of some object of a robot. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

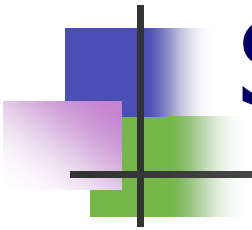
Real-life example: Robotic Assembly



- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

Real-life example: VLSI Layout

- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)
- "optimal way"??
  - minimize surface area
  - minimize number of signal layers
  - minimize number of connections from one layer to another
  - minimize length of some signal lines (e.g., clock line)
  - distribute heat throughout board
  - etc.



SEARCH PROBLEMS

- Problem formulation means choosing a relevant set of states to consider, and a feasible set of actions (operations) for moving from one state to another.
- Search is the process of imagining sequences of operators applied to the initial state and checking which sequence reaches a goal state.

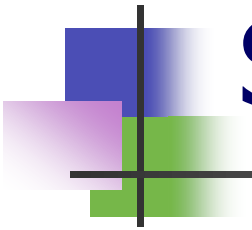
SEARCH PROBLEMS

- S → the full set of states
- s_o → the initial state ($s_o \in S$)
- A : S → S set of actions (operations)
- G: the set of final states
  ($G \subseteq S$)
- Search problem:

Find a sequence of actions which transform the agent from the initial state to a goal state $g \in G$.

SEARCH PROBLEMS

- The search problem consists of finding a solution plan, which is a path from the current state to a goal state.
- Representing search problems:
  - A search problem is represented by using a directed graph.
    - The states are represented as nodes
    - The allowed actions are represented as arcs.

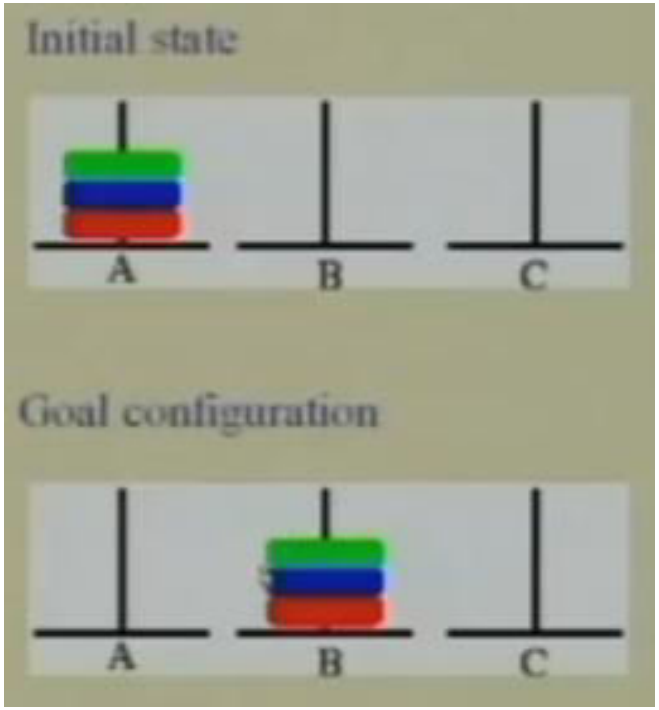


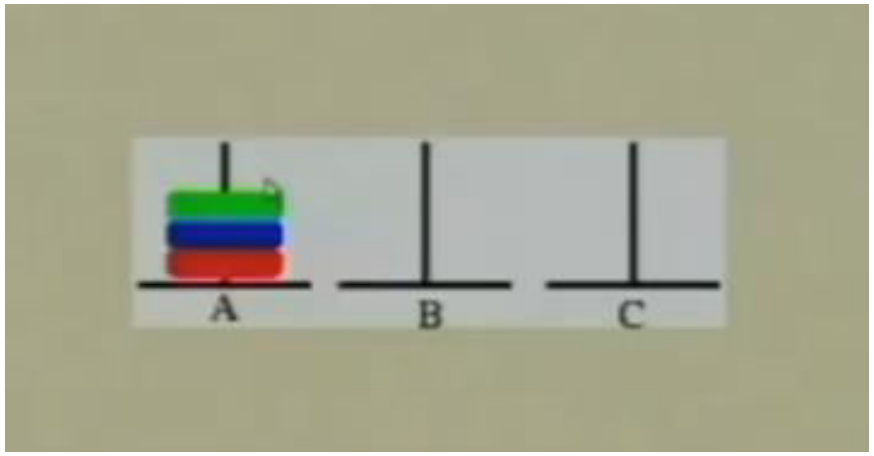
Searching Process

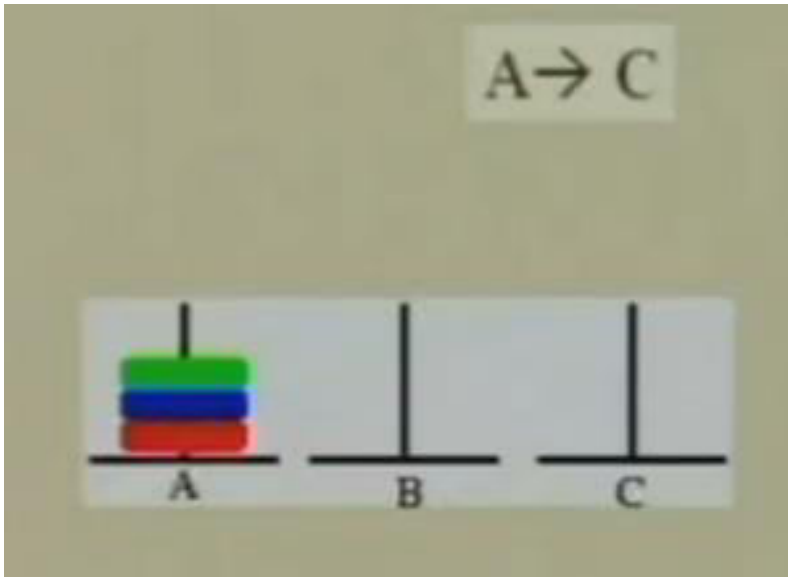
- Check the current state
- Execute allowable actions to move to the next state
- Check if the new state is a solution state
  - if it is not, the new state becomes the current state and the process is repeated until a solution is found or the state space is exhausted.

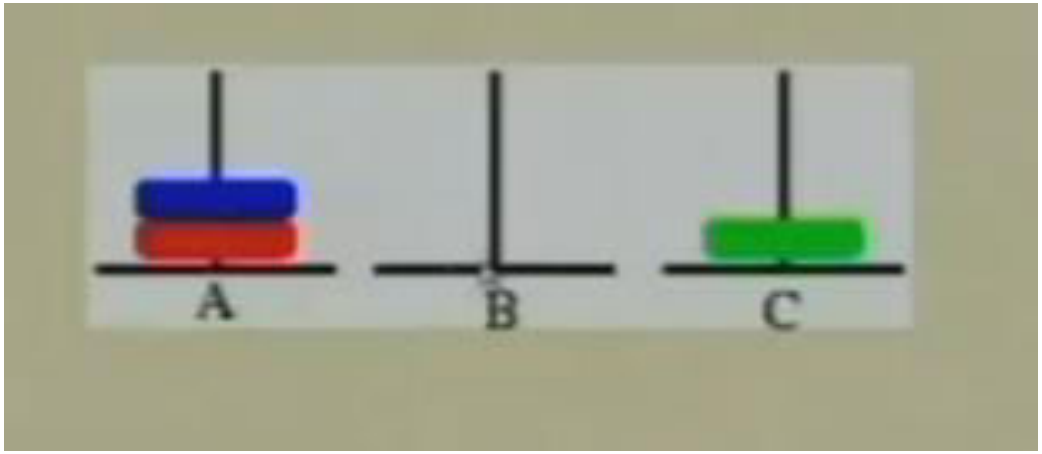
Pegs and Disks

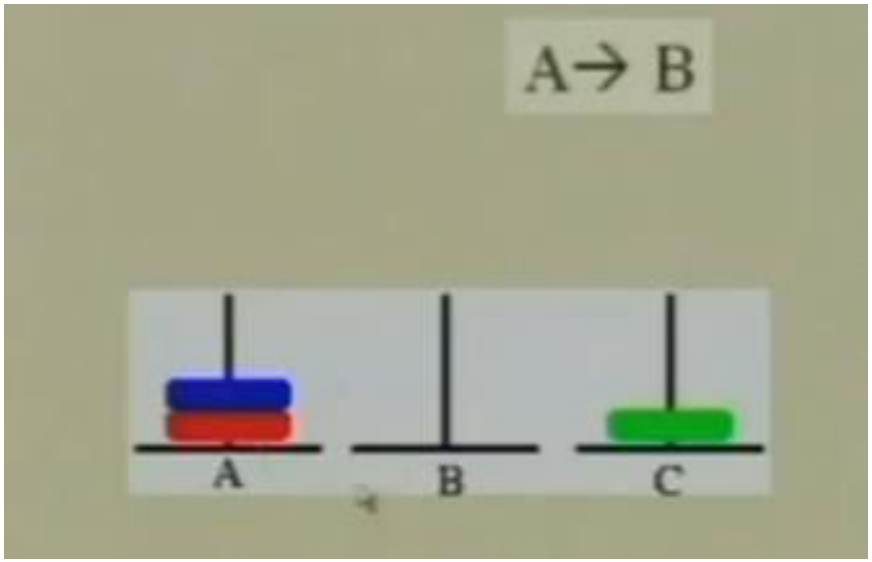
- Consider the following problem. We have 3 pegs and 3 disks.
- Operators: one may move the topmost disk on any needle to the topmost position to any other needle.

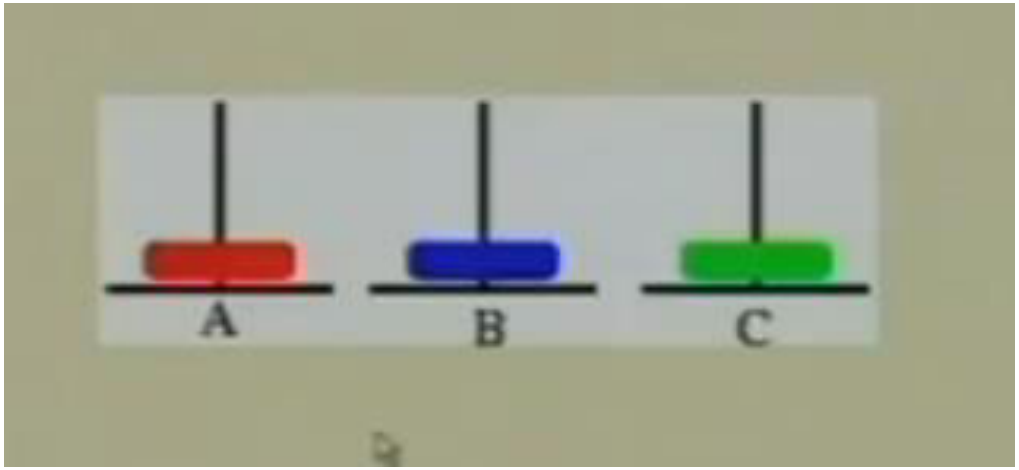


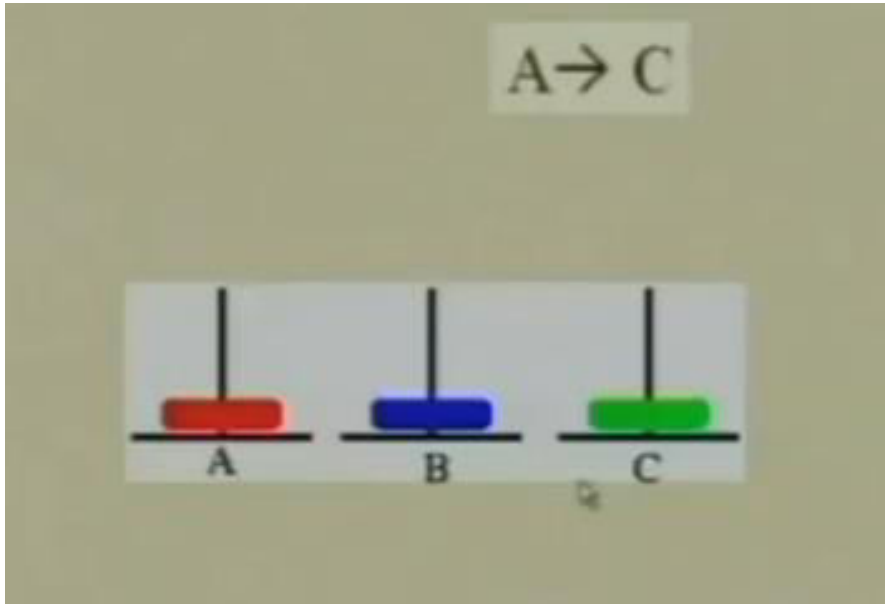

A
B
C

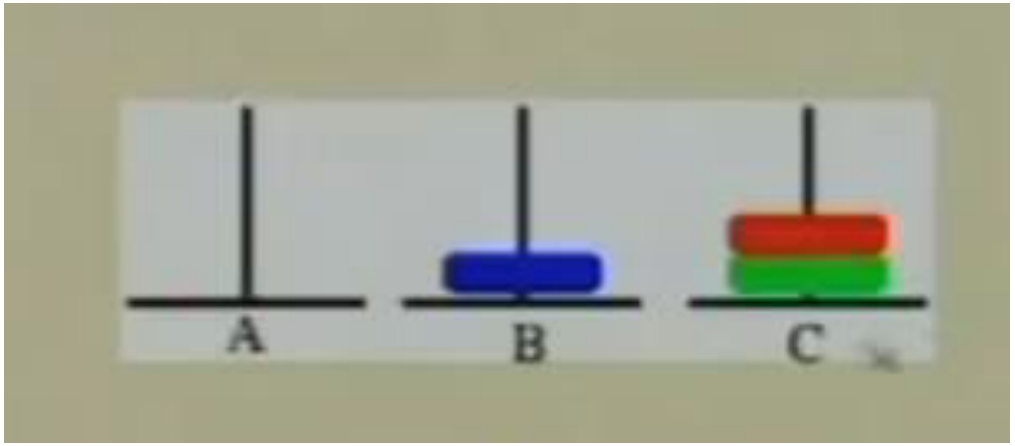

A→ C
A
B
C

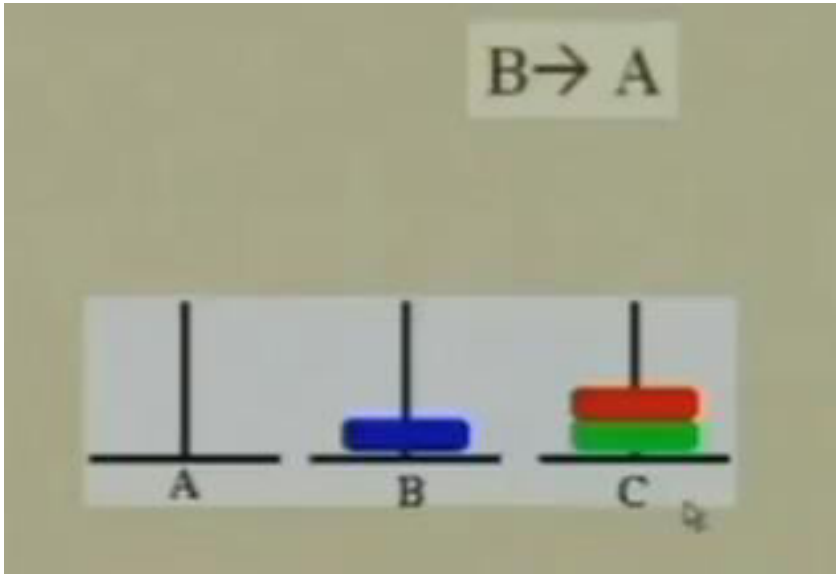

A
B
C

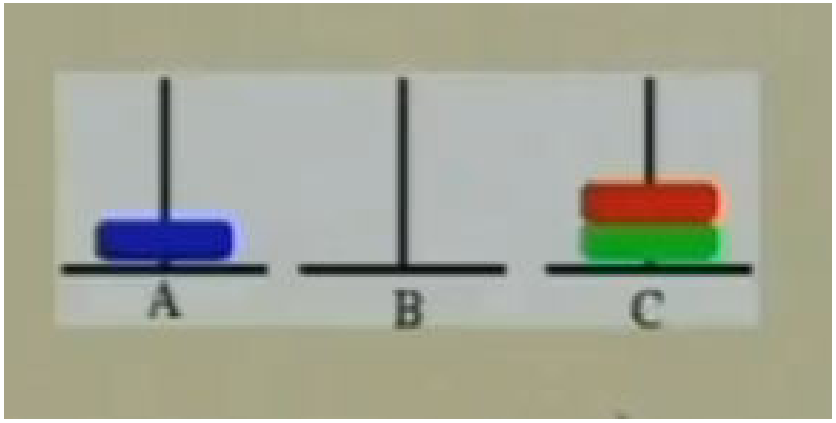

A→ B
A
B
C

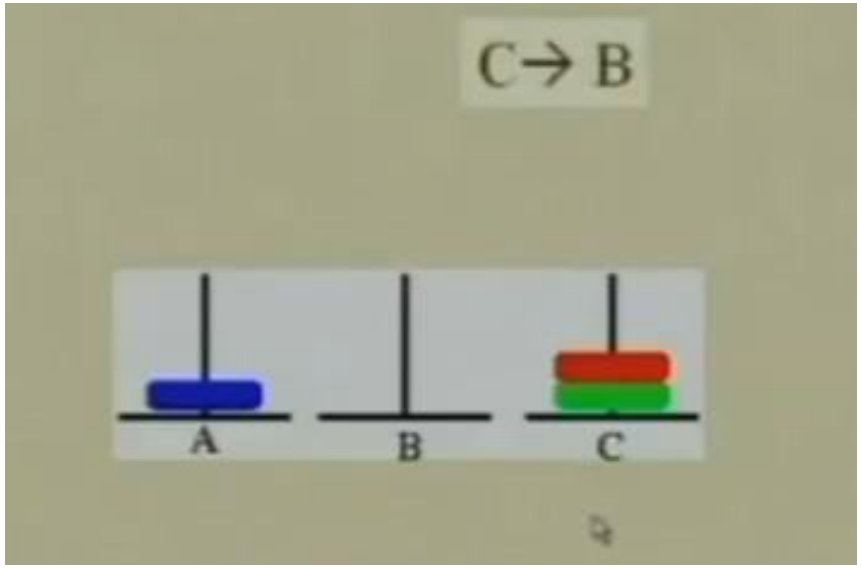

A
B
C

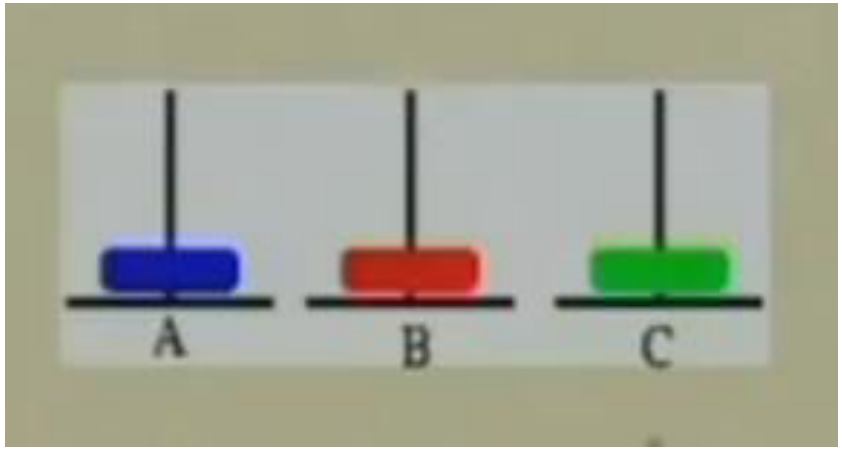

A→ C
A
B
C

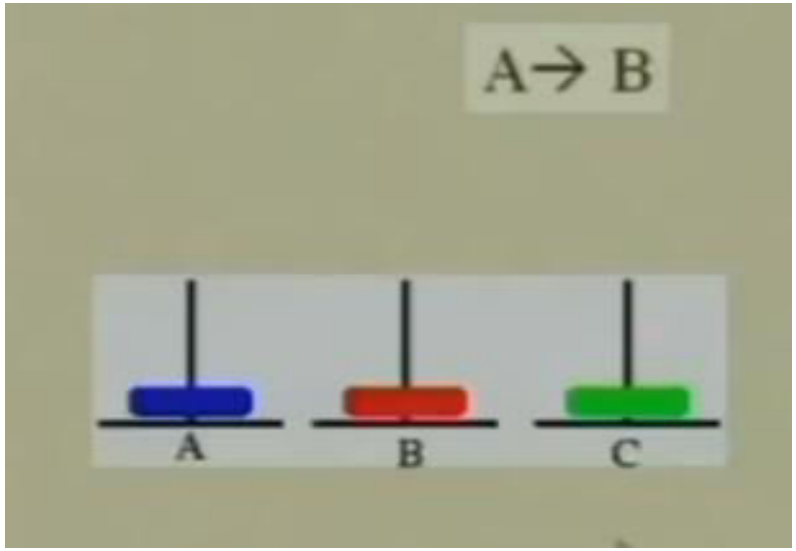

A
B
C

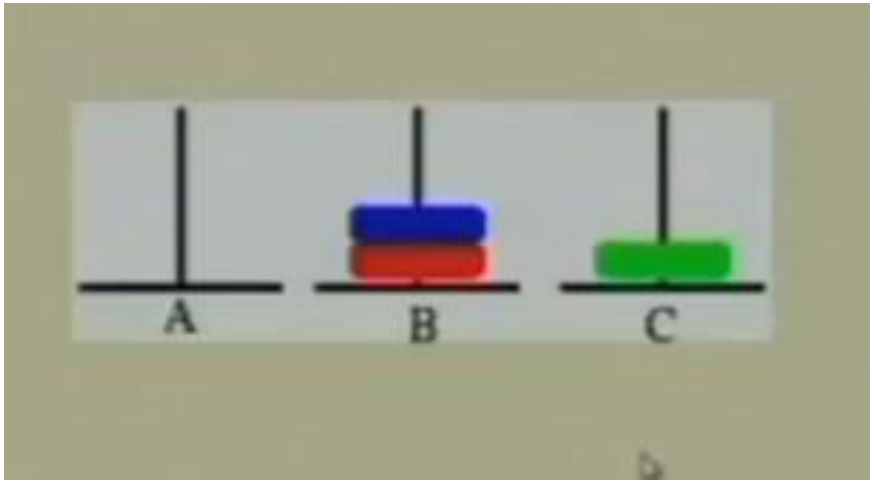

B→ A
A
B
C

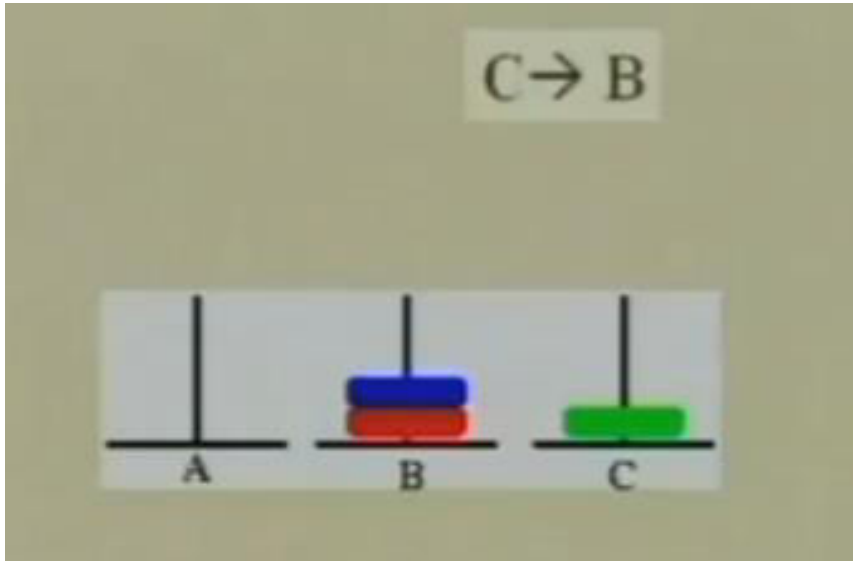

A
B
C

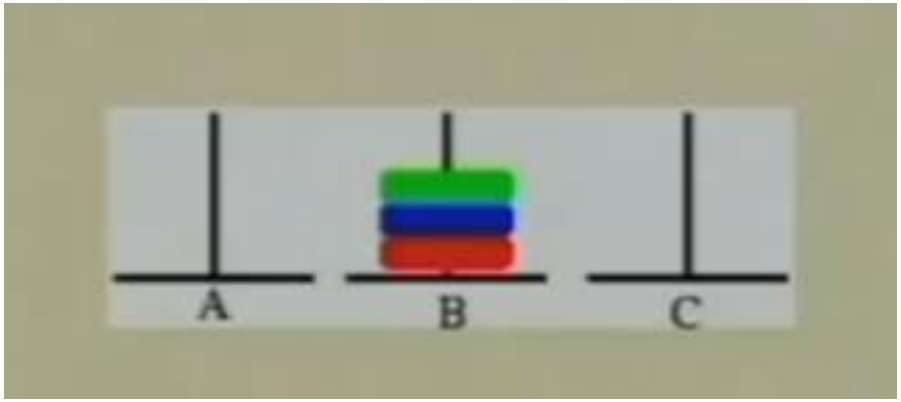

C→ B
A
B
C


A
B
C


A→ B
A
B
C


A
B
C


C→ B
A
B
C


A
B
C

Search Algorithms

- After formulating some problems, we now need to solve them.
- This is done by a search through the state space.
- We'll deal with search techniques that use a search tree that is generated by the initial state and the successor function.
- initial state + successor function = state space
- We may have a search graph rather than a search tree, when the same state can be reached from multiple paths.

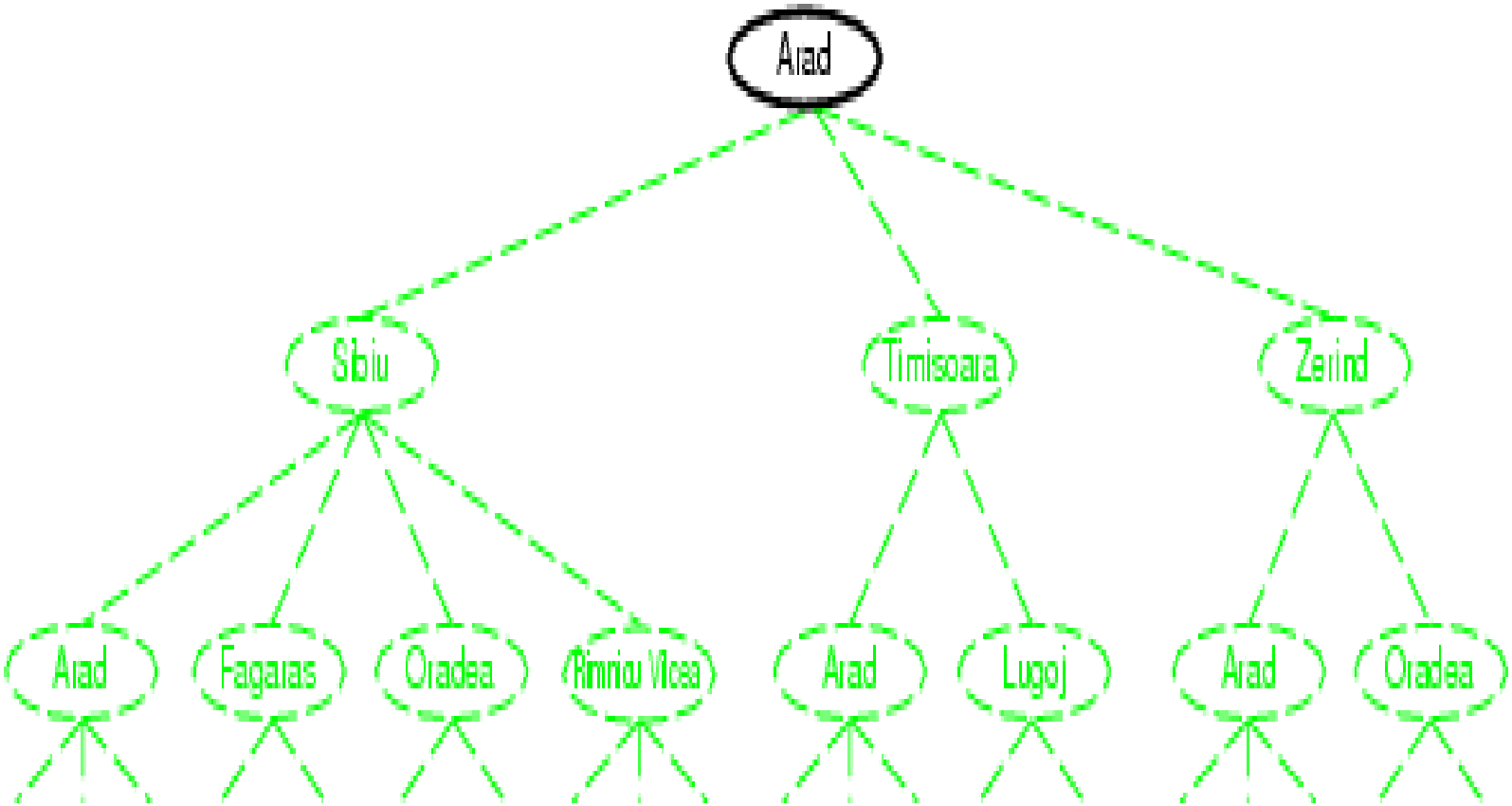
Tree search algorithms

- The root of the search tree is a **search node** corresponding to the initial state. *In(Arad)*
- The first step is to test whether this is a goal state.
- If this is not a goal state, we need to consider some other states. (**expanding** the current state)
- Thereby, **generating** a new set of states.
- The choice of which state to expand is determined by the **search strategy**.

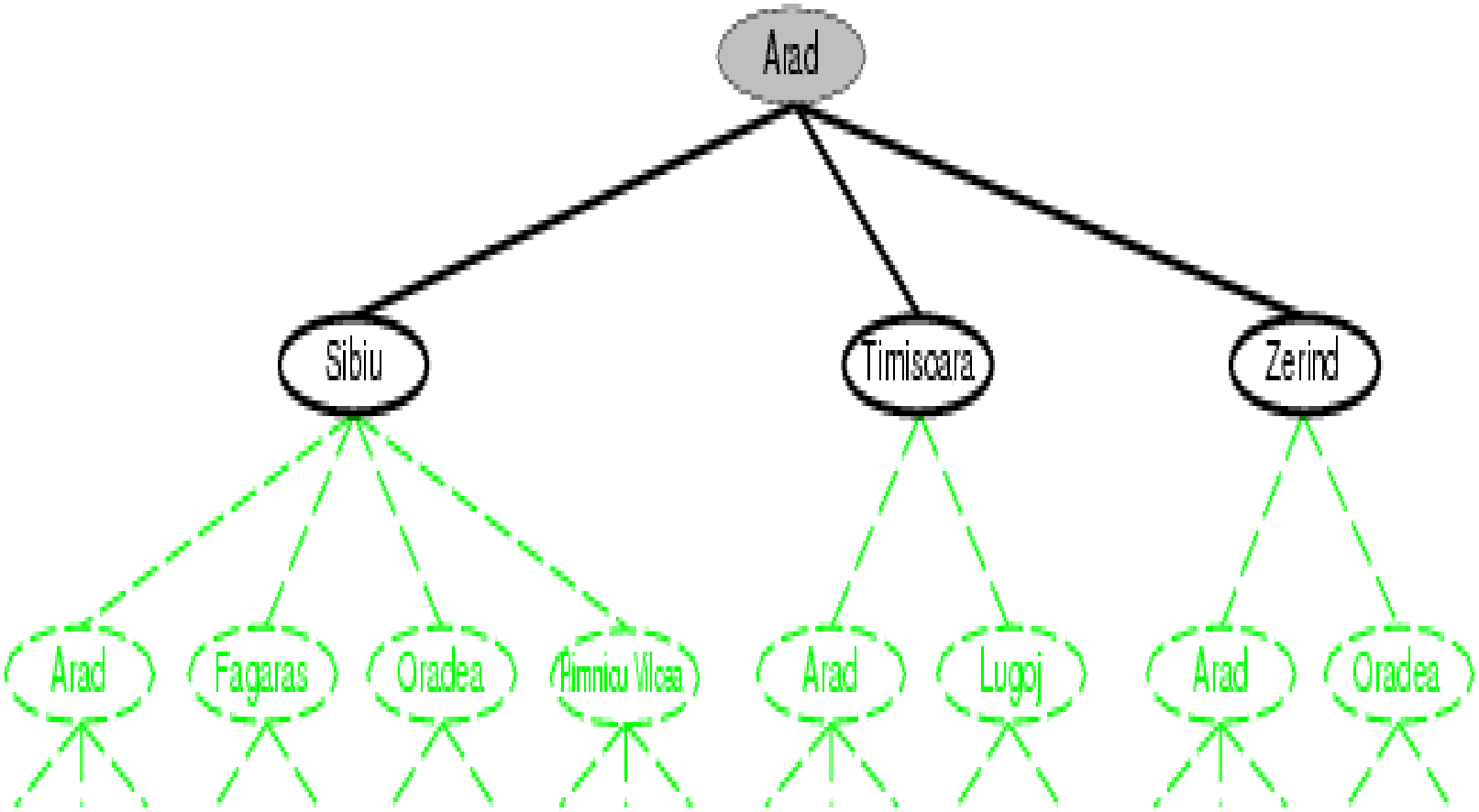
Tree search algorithms

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

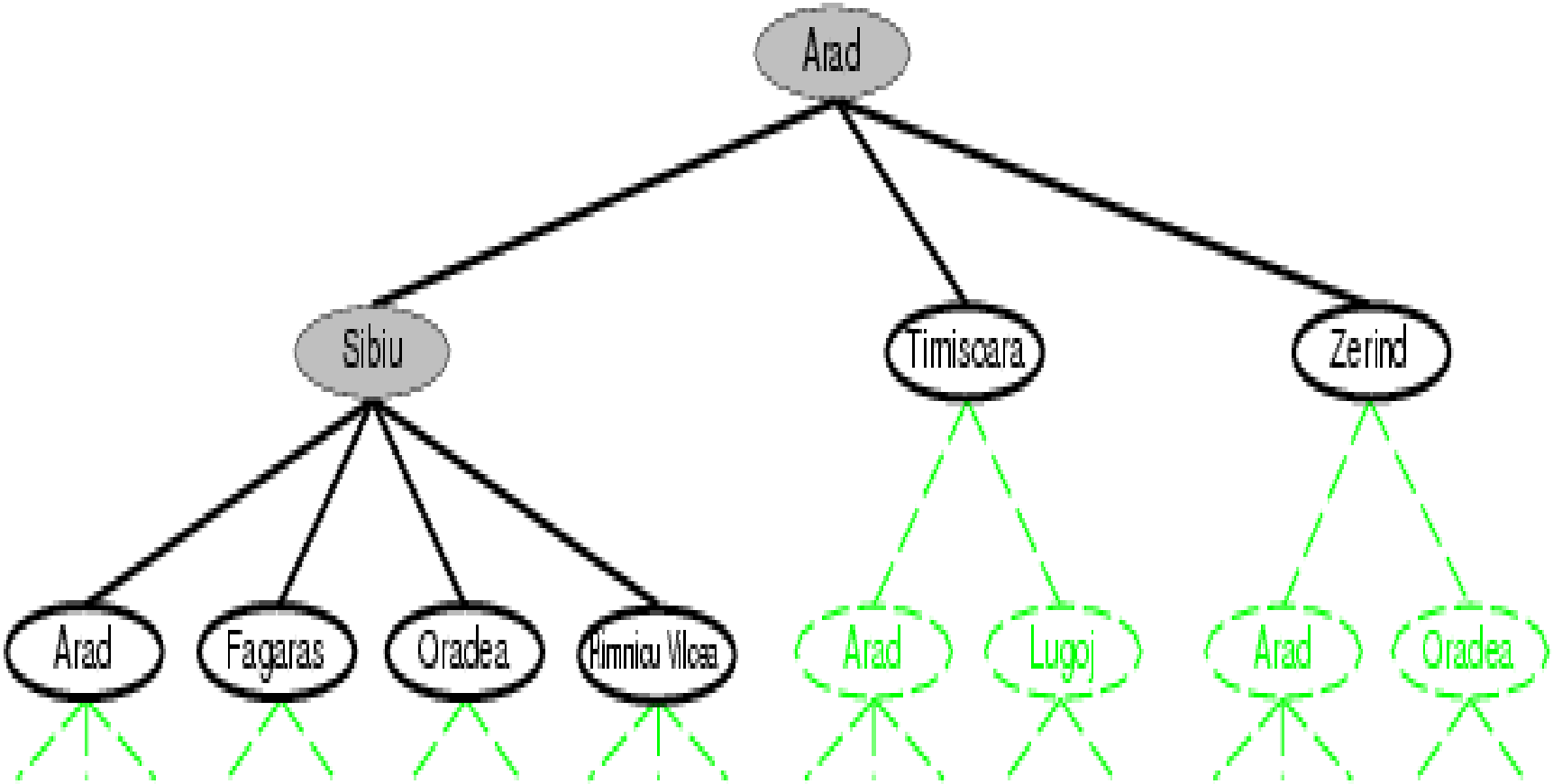
Tree search example



Tree search example



Tree search example



Tree search example

- We continue choosing, testing and expanding until a solution is found or there are no more states to be expanded.
- The collection of nodes that have been generated but not yet expanded is called the **fringe**.
- Each element of the fringe is a **leaf node**.
- We will assume that the collection of nodes is implemented as a **queue**.

Tree search algorithms

The operations on a queue are as follows:

- *MAKE-QUEUE(element,…)* creates a queue with the given element(s).
- *EMPTY?(queue)* returns true only if there are no more elements in the queue.
- *FIRST(queue)* returns the first element of the queue.

Tree search algorithms

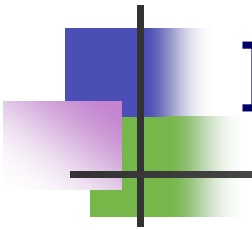
- *REMOVE-FIRST(queue)* returns FIRST(queue) and removes it from the queue.
- *INSERT(element, queue)* inserts an element into the queue and returns the resulting queue.
- *INSERT-ALL(elements, queue)* inserts a set of elements into the queue and returns the resulting queue.

Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

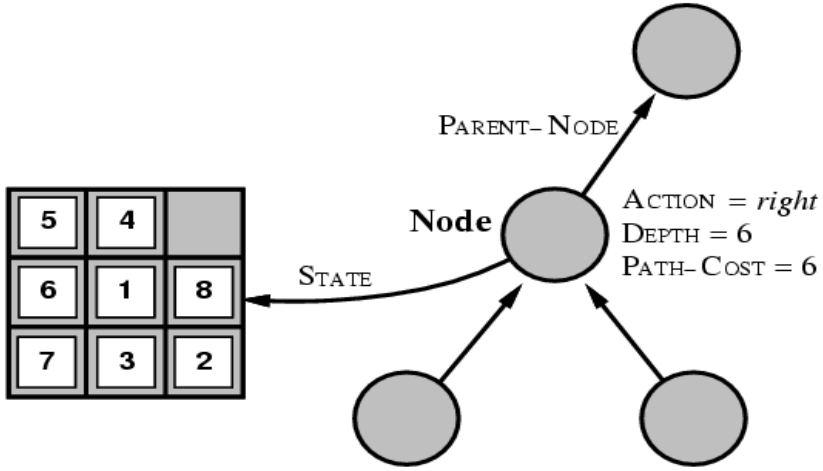
Implementation: general tree search



- The `SOLUTION` function returns the sequence of actions obtained by following parent pointers back to the root.
- The `EXPAND` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration.



- A node is a data structure constituting part of a search

- States do not have parents, children, depth, or path cost!

Implementation: states vs. nodes

- It is important to distiguish the state space and the search tree.
- For the route finding problem, there are only 20 states in the state space.
- But there are infinite number of paths in this state space so the search tree has an infinite number of nodes.

Implementation: states vs. nodes

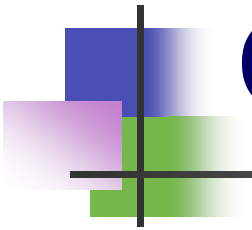
- STATE: the state in the state space to which the node corresponds.
- PARENT-NODE: the node in the search tree that generated this node.
- ACTION: the action that was applied to the parent to generate the node.
- PATH-COST: the cost of the path from the initial state to the node, as indicated by the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

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated/expanded
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

Search strategies

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - C*: path cost of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

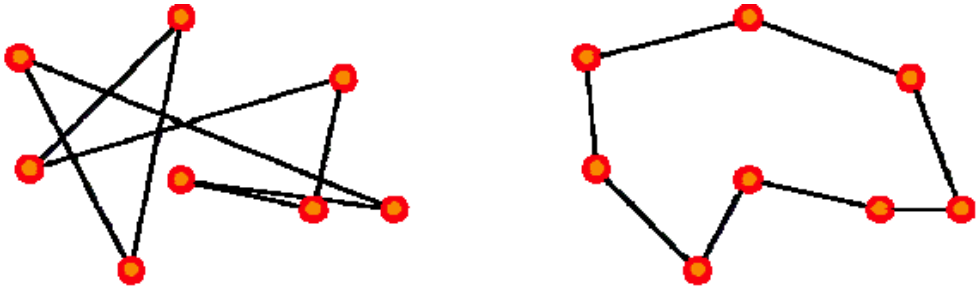


Complexity

- Why worry about complexity of algorithms?
  - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
  - through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size n of a problem when n tends towards infinity

Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length L_{ij} joining city i to city j.
- The salesman wishes to find a way to visit all cities that is optimal in two ways: each city is visited only once, and the total route is as short as possible.



- This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as e^n for n cities.

Why is exponential complexity "hard"?

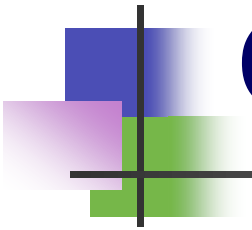
- It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).
- $e^{1} = 2.72$
- $e^{10} = 2.20\ 10^{4}$ (daily salesman trip)
- $e^{100} = 2.69\ 10^{43}$ (monthly salesman planning)
- $e^{500} = 1.40\ 10^{217}$
- $e^{250,000} = 10^{108,573}$ (postal services)
- Fastest computer = 10^{12} operations/second

So…

- In general, exponential-complexity problems *cannot be solved for any but the smallest instances!*

Complexity

- **Polynomial-time (P) problems:** we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.
  - for example: sort n numbers into increasing order: poor algorithms have n^2 complexity, better ones have $n \log(n)$ complexity.
- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?



Complexity

- Yes (until proof of the contrary), for some algorithms, we do not know of any polynomial-time algorithm to solve them. These are referred to as **non-polynomial-time (NP)** algorithms.
  - for example: traveling salesman problem.
- In particular, exponential-time algorithms are believed to be NP.

Note on NP-hard problems

- The formal definition of NP problems is:
  - A problem is nondeterministic polynomial if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.
- (one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)
- In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve the problem.

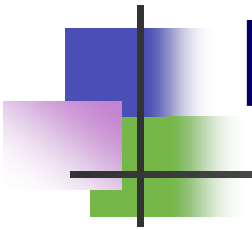
Complexity and the human brain

- Are computers close to human brain power?
- Current computer chip (CPU):
  - 10^3 inputs (pins)
  - 10^7 processing elements (gates)
  - 2 inputs per processing element (fan-in = 2)
  - processing elements compute boolean logic (OR, AND, NOT, etc)
- Typical human brain:
  - 10^7 inputs (sensors)
  - 10^{10} processing elements (neurons)
  - fan-in = 10^3
  - processing elements compute complicated functions

Still a lot of improvement needed for computers, but computer clusters come close!

Search Strategies

- Uninformed search strategies – blind search
  - they have no additional information about states – only the problem definition.
  - all they can do is generate successors and distinguish a goal state from a nongoal state.
- Informed search strategies – heuristic search
  - they know whether a nongoal state is "more promising" than another.
- All search strategies are distinguished by the order in which nodes are expanded.



NEXT WEEK

- **Uninformed search algorithms**
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search