Introduction to Artificial Intelligence

LIAW, RUNG-TZUO

Department of Computer Science and Information Engineering

Fu Jen Catholic University, Taiwan

Chapter 3
Solving Problems by Searching

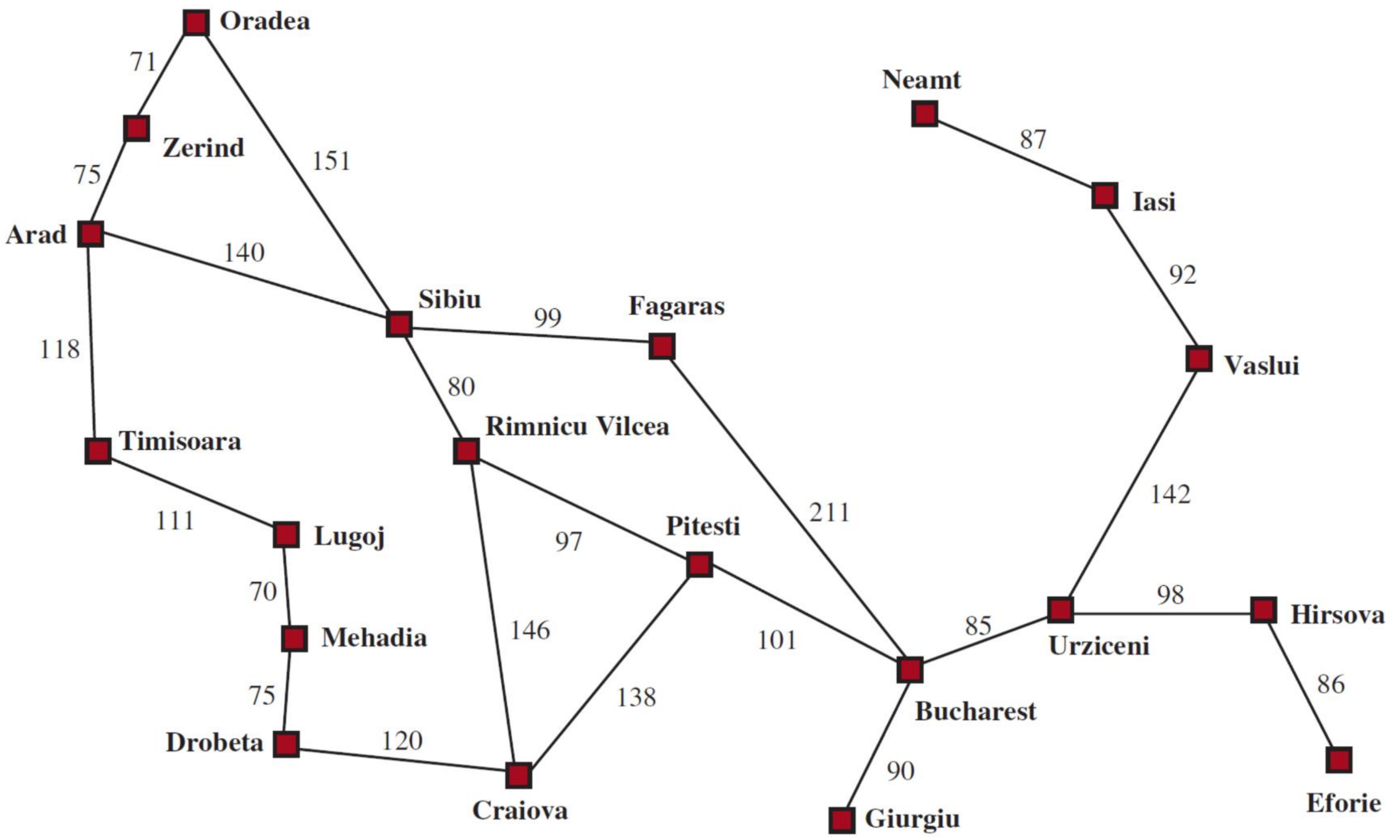
Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Example: Romania



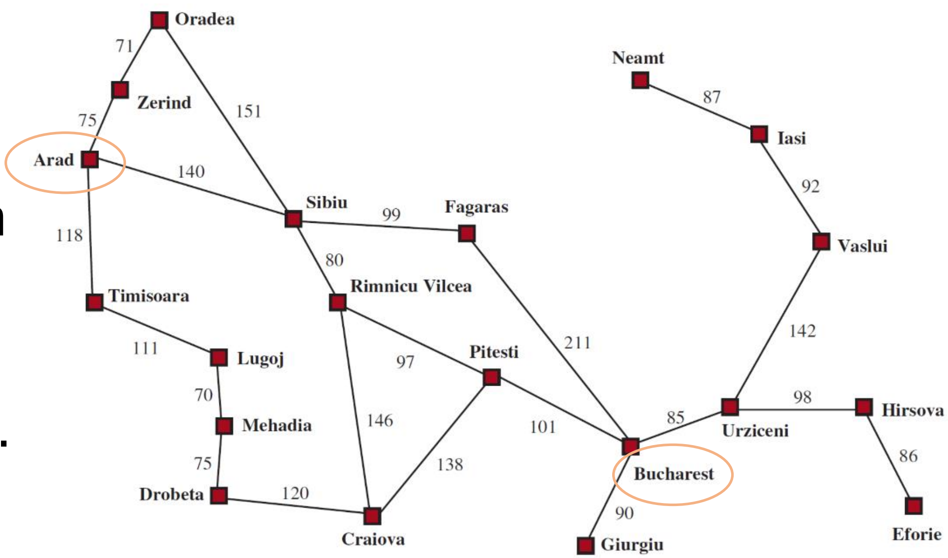
A simplified road map of part of Romania, with road distances in miles.

Example: Romania

- **On holiday in Romania**

  Currently in Arad; flight leaves from Bucharest tomorrow

  - Formulate goal
    - **Be in Bucharest**
  - Formulate problem
    - States: various cities
    - Actions: drive between cities
  - Find solution
    - Sequence of cities e.g. Arad, Sibiu, Fagaras, Bucharest, …



Problem-Solving Agent

- **A kind of goal-based agent**
  - Goal: solve the problem
- **Four general steps in problem solving:**
  - Goal formulation
    - What are the successful world states
  - Problem formulation
    - What actions and states to consider given the goal
  - **Search**
    - Determine the possible *sequence of actions* that lead to the states of known values and then choosing the best sequence
  - Execute
    - Give the solution perform the actions

Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*)
returns an action

  static: *seq*, an action sequence
    state, a description of the current world state
    goal, a goal
    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 Types

- **Deterministic, fully observable → single state problem**
  - Agent knows exactly which state it will be in
  - Solution is a sequence

Goal Formulation

- **Determine what success means**
- **Perhaps the most important fundamental question**

Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Problem Formulation

- **A problem is defined by:**
  - Initial state, e.g. Arad
  - Successor function S(X) = set of <action,state> pairs
    - e.g. S(Arad)={<Arad → Zerind, Zerind>, < , >, …}
    - Initial state + successor function = state space
  - Goal test, can be
    - Explicit, e.g. x='at Bucharest'
    - Implicit, e.g. checkmate(x)
  - Path cost (additive)
    - e.g. sum of distances, number of actions executed, …
    - c(x,a,y) is the step cost, assumed to be >= 0

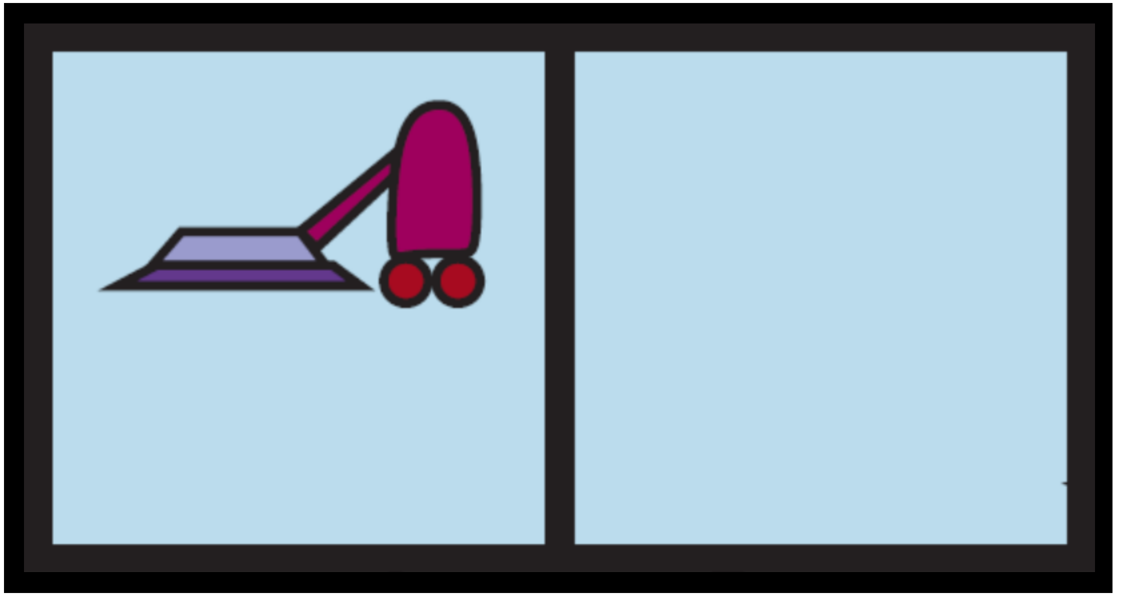
- A **solution** is a sequence of actions from initial to goal state.
- **Optimal solution** has the lowest path cost.

State Space

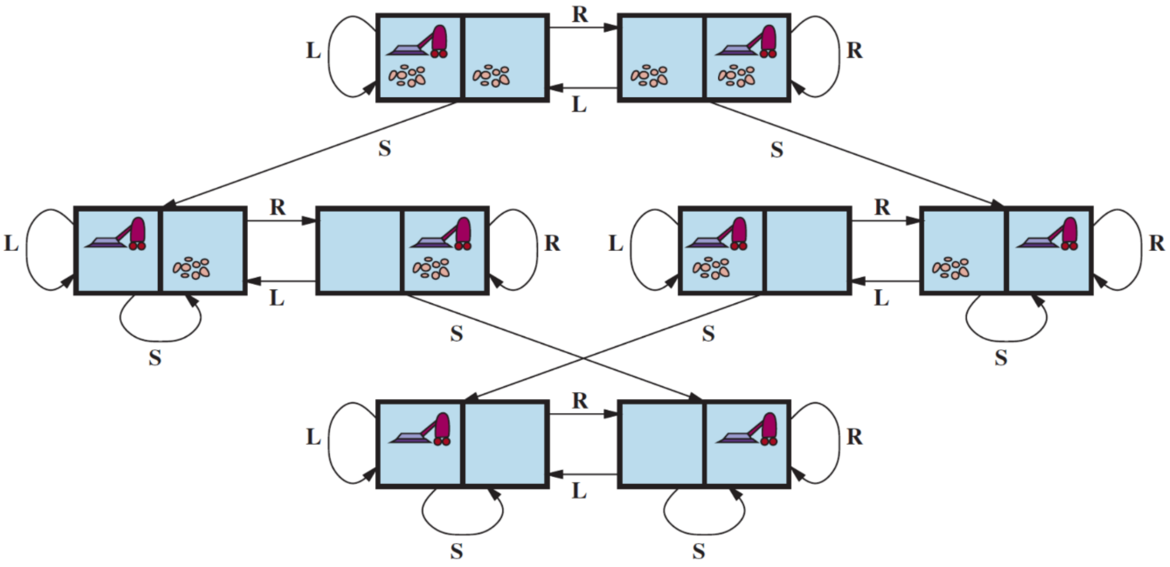
- **Real world is absurdly complex**
  - State space must be abstracted for problem solving
  - (Abstract) state = set of real states
  - (Abstract) action = complex combination of real actions
    - e.g. Arad → Zerind represents a complex set of possible routes, detours, rest stops, etc.
    - The abstraction is valid if the path between two states is reflected in the real world
    - Each abstract action should be “easier” than the real problem
  - (Abstract) solution = set of real paths that are solutions in the real world

Example: Vacuum World

- Consider an environment with two cells and one vacuum

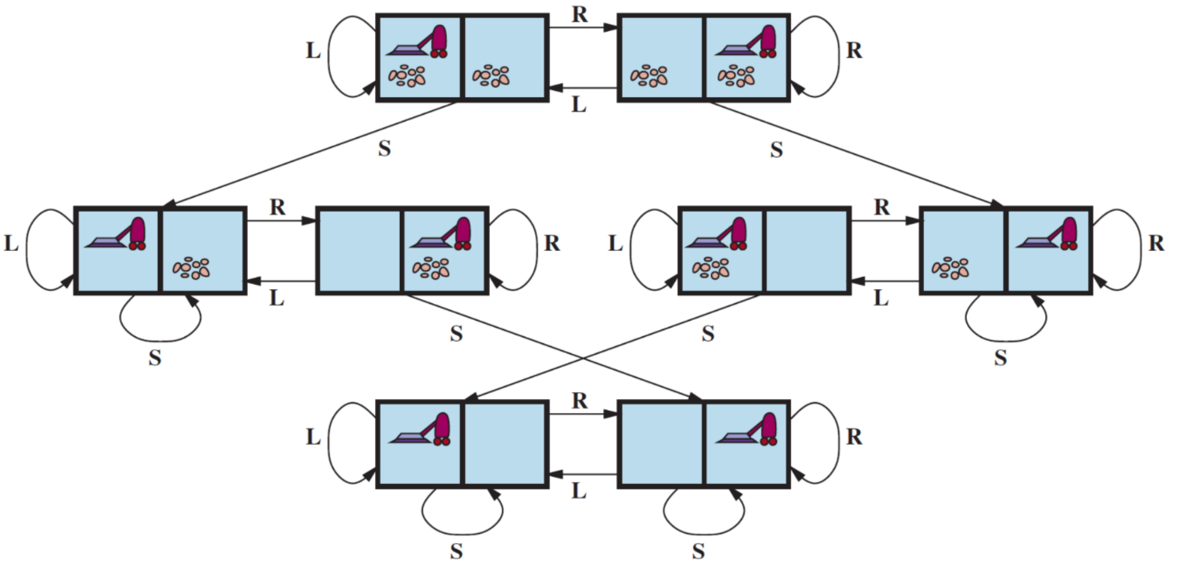


Example: Vacuum World



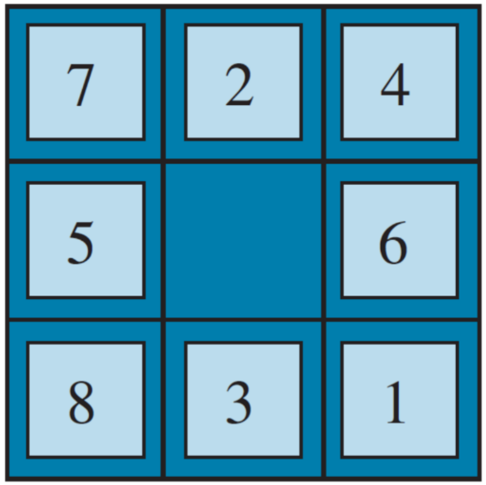
- **Problem Formulation**
  - States?
  - Initial state?
  - Actions?
  - Goal test?
  - Path cost?

Example: Vacuum World

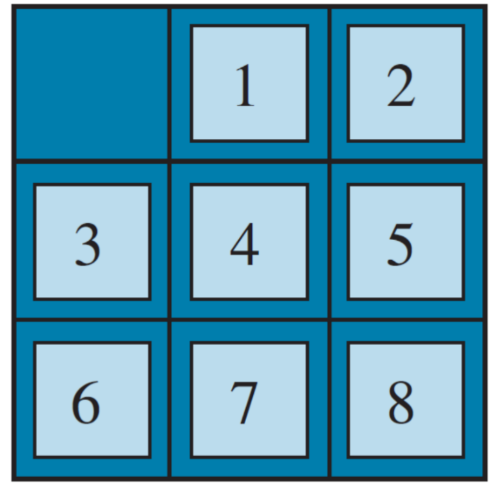


- **Problem Formulation**
  - States? (vacuum, dust)
  - Initial state? Any
  - Actions? L = *Left*, R = *Right*, S = *Suck*.
  - Goal test? Check if no dust exists.
  - Path cost? Number of actions to reach goal.

Example: 8-Puzzle



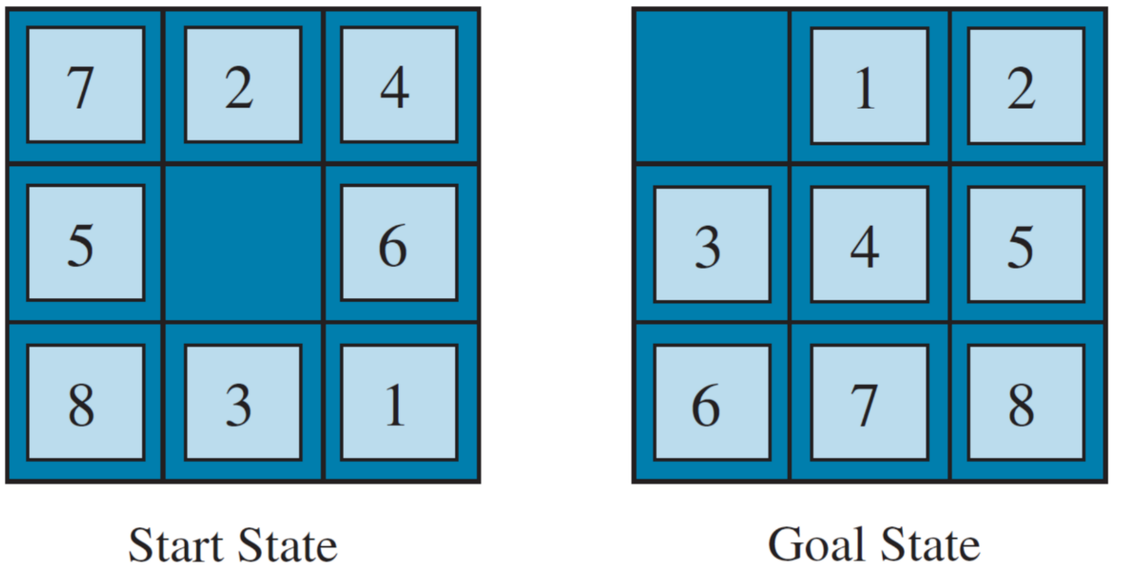
Start State



Goal State

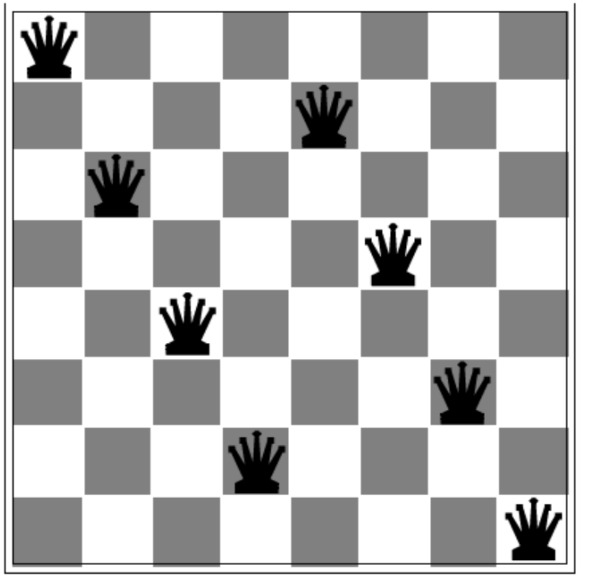
- **Problem Formulation**
  - States?
  - Initial state?
  - Actions?
  - Goal test?
  - Path cost?

Example: 8-Puzzle



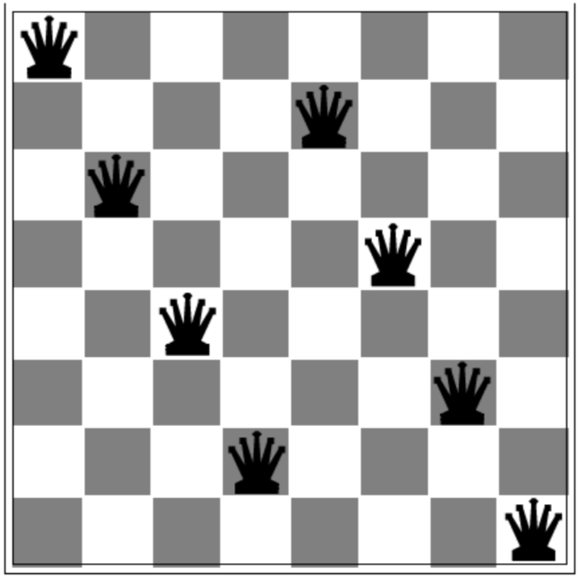
- **Problem Formulation**
  - States? Integer location of each tile. How many of them?
  - Initial state? Any state
  - Actions? (tile, direction)
    where direction is one of {*Left*, *Right*, *Up*, *Down*}
  - Goal test? Check whether goal configuration is reached
  - Path cost? Number of actions to reach goal

Example: 8 queens problem



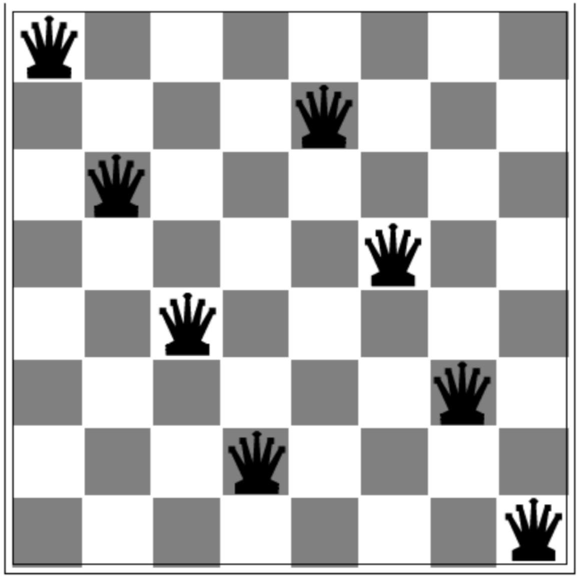
- Incremental vs. complete state formulation
  - Incremental formulation starts with an empty state and involves operators that augment the state description
  - A complete state formulation starts with all 8 queens on the board and moves them around

8 queens problem: representation is key



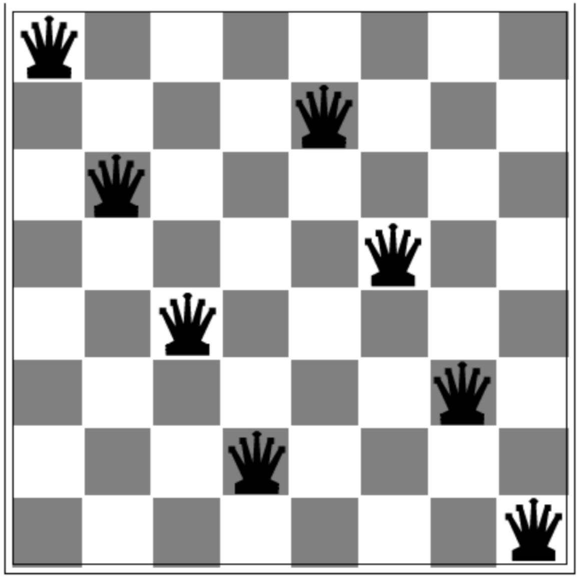
- **Incremental Formulation**
  - States? Any arrangement of 0 to 8 queens
  - Initial state? No queens
  - Actions? Add queen in empty square
  - Goal test? 8 queens on board and none attacked
  - Path cost? None but 64*63*…*57 ~ 3E14 states

8 queens problem: a better representation is key



- **Another Incremental Formulation**
  - States? n queens on the board n left-most columns
  - Initial state? No queens
  - Actions? Add queen in leftmost empty column
  - Goal test? 8 queens on board and none attacked
  - Path cost? None but 2057 states

8 queens problem: a better representation is key



- Complete-state **Formulation**
  - States? Position of 8 queens
  - Initial state? Any possible positions
  - Actions? Move a queen to another place
  - Goal test? 8 queens on board and none attacked
  - Path cost? #movements

n queens problem

- A solution is a goal node, not a path to this node (typical of design problem)
- Number of states in state space:
  - 8-queens → 2,057
  - 100-queens → 10^{52}
- But techniques exist to solve n-queens problems efficiently for large values of n

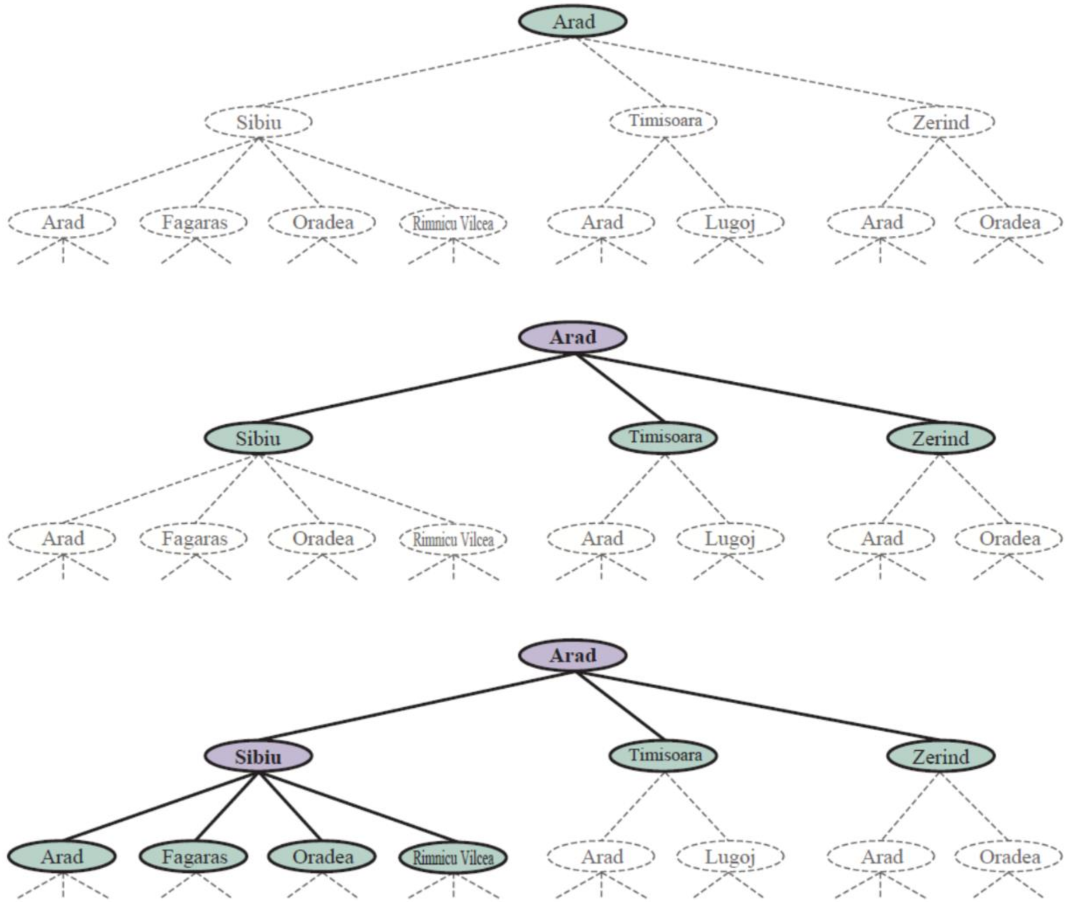
Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Basic Search Algorithms

- **How do we find the solutions of previous problems?**
  - **Search** the state space
  - Remember complexity of space depends on state representation
  - Here: search through **explicit tree** generation
    - ROOT= initial state
    - **Expand** current state
      - Generate new set of states (nodes and leafs) by applying *successor function* to current state
      - **Search strategy** chooses which state to expand
  - More generally, search generates a *graph*
    - Same state through multiple paths

Simple Tree Search Example



Simple Tree Search Example

function TREE-SEARCH(*problem*, *strategy*)
returns a solution or failure

initialize search tree using the initial state

loop do

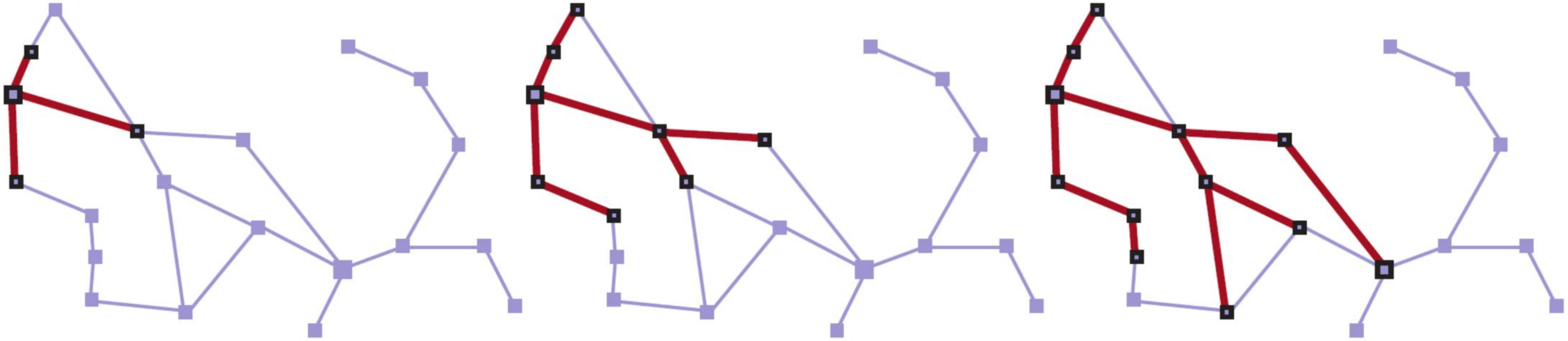
if no candidates for expansion **return** failure

choose leaf node for expansion as per *strategy*

if node contains goal state **then return** solution

else expand the node and add resulting nodes to the search tree

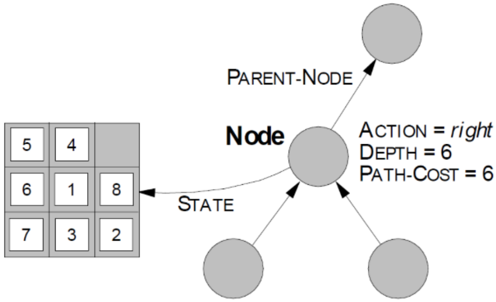
Simple Tree Search Example



A sequence of search trees generated by a graph search on the Romania problem

State Space vs. Search Tree

- **A state is a (representation of) a physical configuration**
- **A node is a data structure belong to a search tree**
  - A node has a parent, children, … and includes path cost, depth, …
  - Here node = <state, parent-node, action, path-cost, depth>
  - FRINGE = contains generated nodes which are not yet expanded



Tree Search Algorithm

function TREE-SEARCH(problem, fringe) **returns** a solution or failure

fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do

if EMPTY(fringe) **then return** failure

node ← REMOVE-FIRST(fringe)

if GOAL-TEST[problem] applied to STATE[node] succeeds

then return SOLUTION(node)

fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

Tree Search Algorithm.

function EXPAND(*node*, *problem*) **returns** a set of nodes

  successors ← the empty set

  for each <*action*, *result*> **in** SUCCESSORFN[*problem*](STATE[*node*]) **do**

    s ← a new NODE

    STATE[*s*] ← *result*

    PARENT-NODE[*s*] ← *node*

    ACTION[*s*] ← *action*

    PATH-COST[*s*] ← PATH-COST[*node*] +
      STEP-COST(STATE[*node*], *action*, *result*)

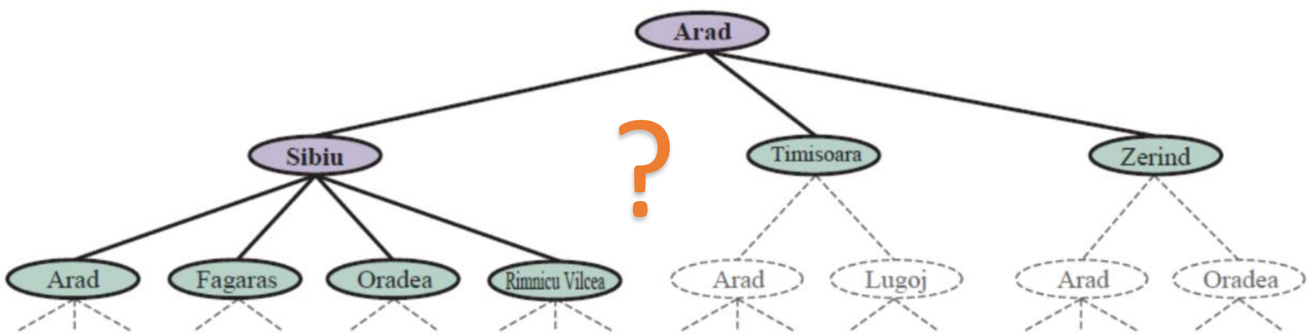
    DEPTH[*s*] ← DEPTH[*node*] + 1

    add *s* to *successors*

return *successors*

Search Strategies

- **A strategy is defined by picking the order of node expansion**
  - Should consider performance



Search Strategies (cont'd)

- **Problem-solving performance is measured in four ways:**
  - **Completeness**
    - Does it always find a solution if one exists?
  - **Optimality**
    - Does it always find the least-cost (optimal) solution?
    - (Is what it found always the least-cost (optimal) solution?)
  - **Time Complexity**
    - Number of nodes generated/expanded?
  - **Space Complexity**
    - Number of nodes stored in memory during search?

Search Strategies.

- **Time and space complexity are measured in terms of problem difficulty defined by:**
  - b: branching factor or maximum #successors of any node
  - d: depth of the shallowest goal node
  - m: maximum depth of the state space (may be ∞)

Outline

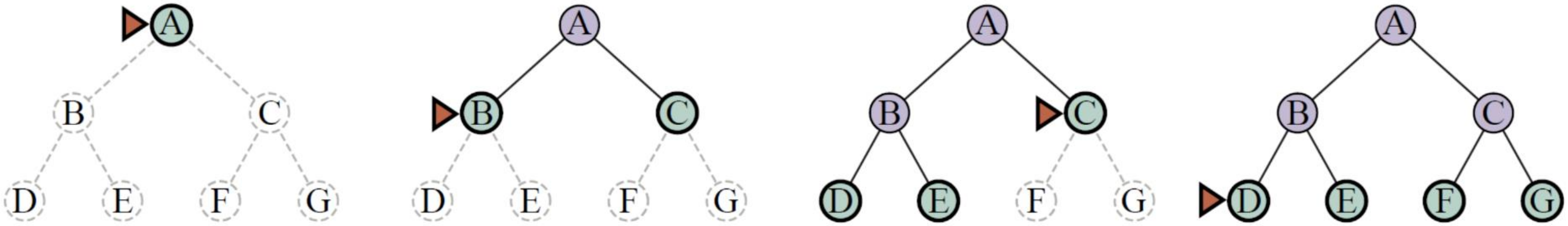
- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Uninformed Search Strategies

- **Uninformed search**
  - a.k.a. blind search
  - Uses only information available in problem definition
    - When strategies can determine whether one non-goal state is *better than* another → informed or heuristic search
  - Categories defined by expansion algorithm:
    - Breadth-first search
    - Uniform-cost search
    - Depth-first search
    - Depth-limited search
    - Iterative deepening search
    - Bidirectional search

Breadth-First Search (BFS)

- **Expand shallowest unexpanded node**
  - Implementation: fringe is a FIFO queue
  - E.g. breadth-first search on a simple binary tree
    - At each stage, the node to be expanded next is indicated by the triangular marker



Breadth-First Search (BFS)

- **Completeness:**
  - Does it always find a solution if one exists?
  - YES
    - If shallowest goal node is at some finite depth d
    - Condition: If b is finite (maximum number of successive nodes is finite)
- **Optimality:**
  - Does it always find the least-cost (optimal) solution?
  - In general, YES
    - unless actions have different cost (i.e. path cost is a decreasing function)

Breadth-First Search (BFS)

- **Time complexity:**
  - Assume a state space where every state has b successors
    - Root has b successors
      - each node at the next level has again b successors (total b^2)
      - –…
    - Assume solution is at depth d
    - Worst case: expand all but the last node (goal) at depth d
    - Total #nodes generated: $b + b^2 + b^3 + \cdots + b^d + \left(b^{d+1} - b\right) = O(b^{d+1})$
- **Space complexity:**
  - Idem $O(b^{d+1})$ if each node is retained in memory

Breadth-First Search (BFS)

- **Two lessons**
  - Memory requirements are a bigger problem than its execution time
  - Exponential complexity search problems cannot be solved by uninformed search for any but the smallest instances
    - e.g. b=10; 10,000 nodes/s; 1000 bytes/node

DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Uniform-Cost Search (UCS)

- **Extension of BFS**
  - Expand node with lowest path cost
  - UCS is the same as BFS when all step-costs are equal
  - Implementation: Fringe = queue ordered by path cost

Uniform-Cost Search (UCS)

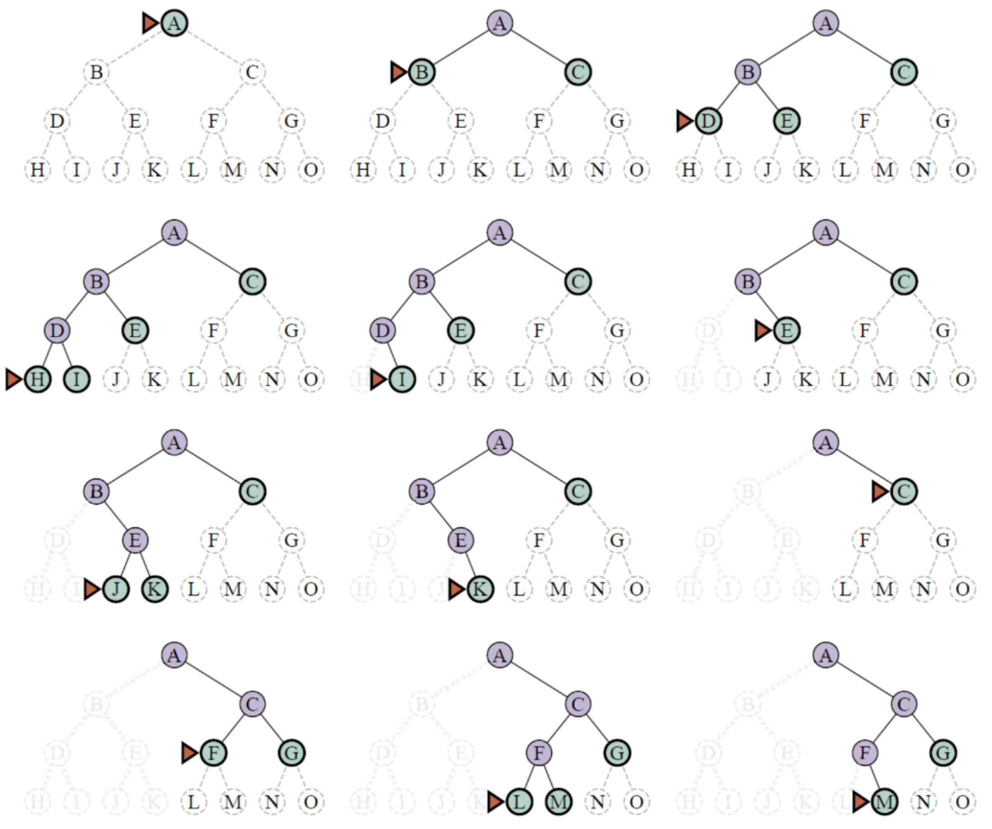
- **Completeness:**
  - Does it always find a solution if one exists?
  - YES, if step-cost > ε (small positive constant) (why?)
    - The search will get stuck in a infinite loop if a node has a zero-cost action leading back to the same state
- **Optimality:**
  - Does it always find the least-cost (optimal) solution?
  - YES, if complete
    - Nodes expanded in order of increasing path cost

Uniform-Cost Search (UCS)

- **Time complexity:**
  - Assume C^* the cost of the optimal solution
  - Assume that every action costs at least ε
  - Worst-case: $O(b^{1+C^*/\varepsilon})$
- **Space complexity:**
  - Idem to time complexity

Depth-First Search (DFS)

- **Expand deepest unexpanded node**
  - Implementation: fringe is a LIFO queue (=stack)
  - e.g.



A dozen steps in the progress of a DFS on a binary tree from start state A to goal M

Depth-First Search (DFS)

- **Completeness:**
  - Does it always find a solution if one exists?
  - **No**
    - e.g. the left subtree is unbounded depth but no solution
    - DFS is complete only if search space is finite and no loops
- **Optimality:**
  - Does it always find the least-cost (optimal) solution?
  - **No**
    - e.g. Assume nodes J and C contain goal states (But only C is optimal)

Depth-First Search (DFS)

- **Time complexity:** $O(b^m)$
  - Terrible if m (max. depth) is much larger than d (depth of optimal solution)
  - But if many solutions, then faster than BFS
- **Space complexity:** $O(bm)$
  - Backtracking search uses even less memory
  - One successor instead of all b
  - DFS' advantage over BFS: $O(b^{d+1})$

Depth-Limited Search

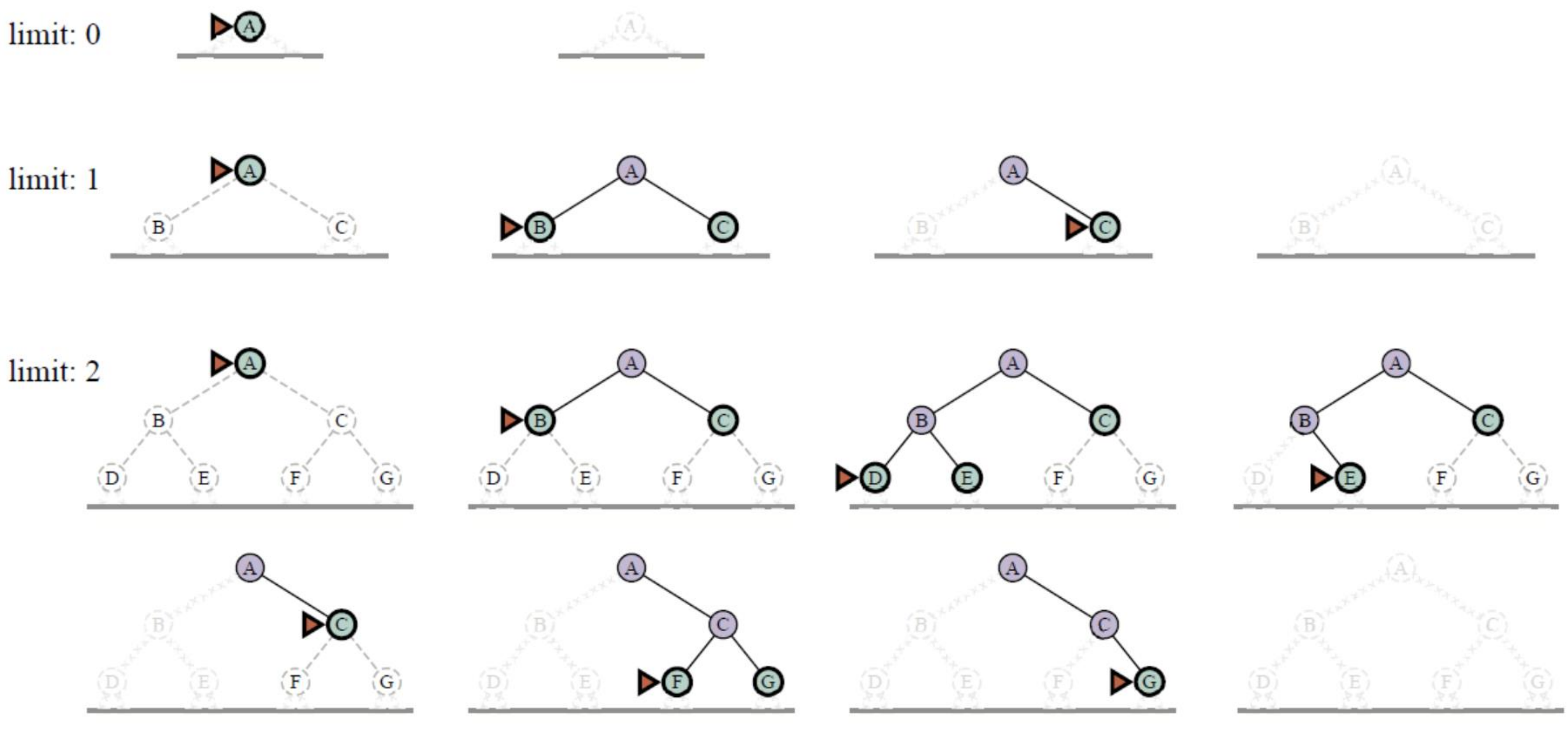
- **Limits the search depth of DFS to l**
  - Solves the infinite-path problem
  - i.e. nodes at depth l have no successors
    - If $l < d$ then incompleteness results
    - If $l > d$ still can be not optimal (why?)
  - Problem knowledge can be used
- **Time complexity:** $O(b^l)$
- **Space complexity:** $O(bl)$

Iterative Deepening Search (IDS)

- **A general strategy to find best depth limit *l***
  - Goals is found at depth d, the shallowest goal-node depth
  - Often used in combination with DFS
- **Combines benefits of DFS and BFS**

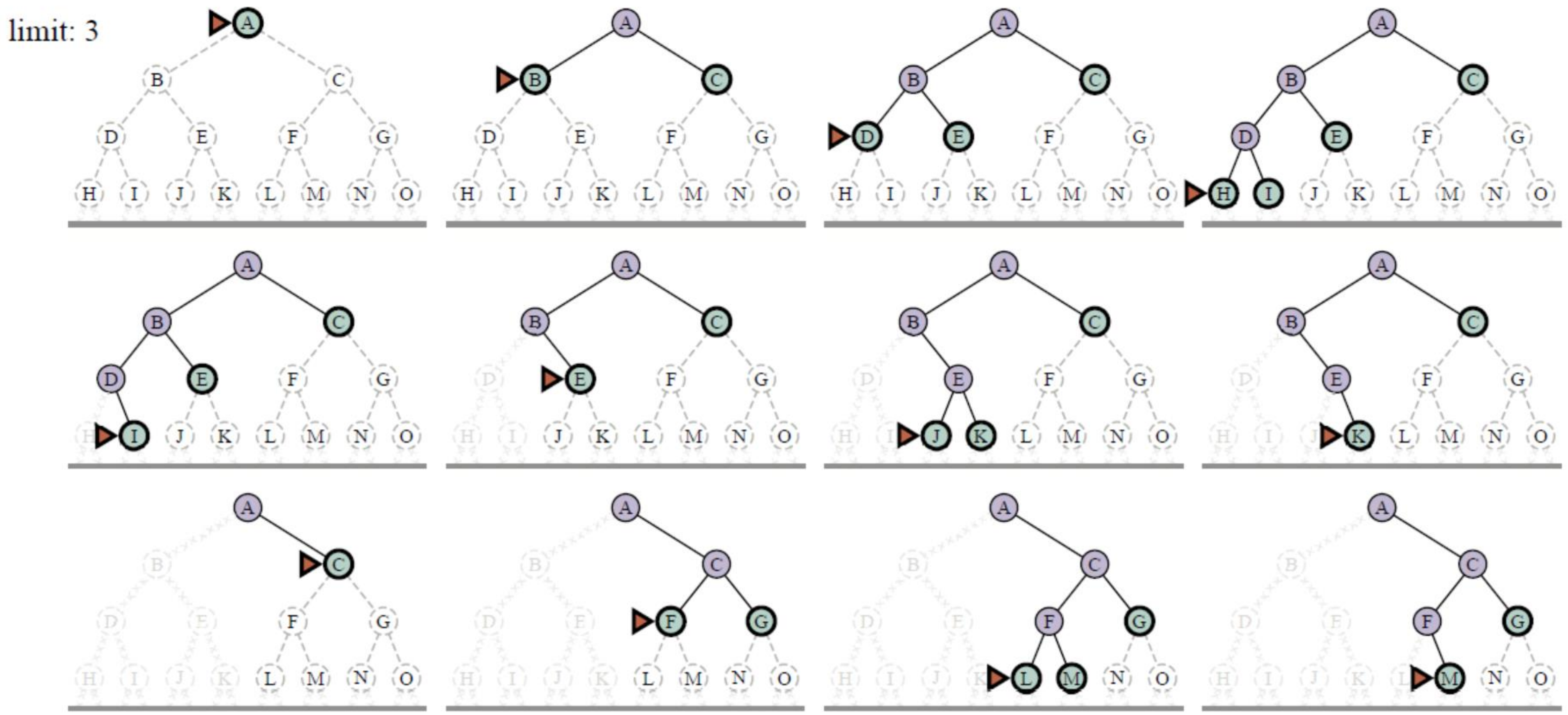
function ITERATIVE_DEEPENING_SEARCH(*problem*)
return a solution or failure
   inputs: *problem*, a problem
   for *depth* ← 0 **to** ∞ **do**
      result ← DEPTH-LIMITED_SEARCH(*problem*, *depth*)
      if *result* ≠ cuttoff **then return** *result*

Iterative Deepening Search (IDS)



Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3.

Iterative Deepening Search (IDS)



Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3.

Iterative Deepening Search (IDS)

- **Completeness:**
  - YES (no, if infinite paths)
- **Optimality:**
  - YES
  - Can be extended to iterative lengthening search
    - Same idea as uniform-cost search
    - Increases overhead

Iterative Deepening Search (IDS)

- **Time complexity:** $\boldsymbol{O(b^d)}$
  - Algorithm seems costly due to repeated generation of certain states
  - Node generation:
    - Level 1: d
    - Level 2: d – 1
    - …
    - Level d-1: 2
    - Level d: 1

$$N(IDS) = db + (d-1)b^2 + .. + b^d$$
$$N(BFS) = b + b^2 + .. + b^d + (b^{d+1} - b)$$

  - e.g. b=10 and d=5

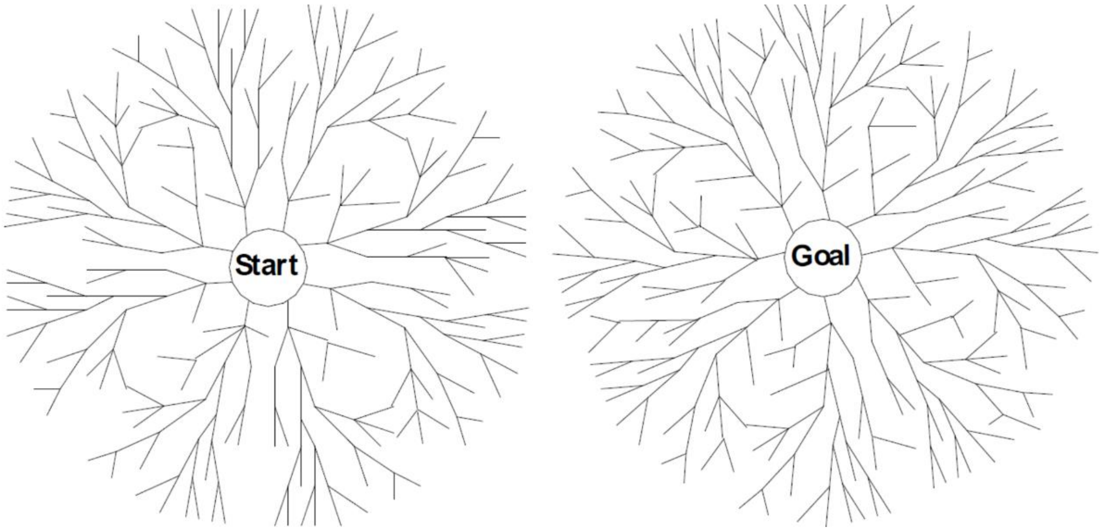
$$N(IDS) = 5 * 10 + 4 * 10^2 + 3 * 10^3 + 4 * 10^4 + 1 * 10^5 = 123450$$
$$N(BFS) = 10 + 10^2 + 10^3 + 10^4 + 10^5 + (10^6 - 10) = 1111100$$

Iterative Deepening Search (IDS)

- **Space complexity:** $O(bd)$
  - cf. DFS: $O(bm)$

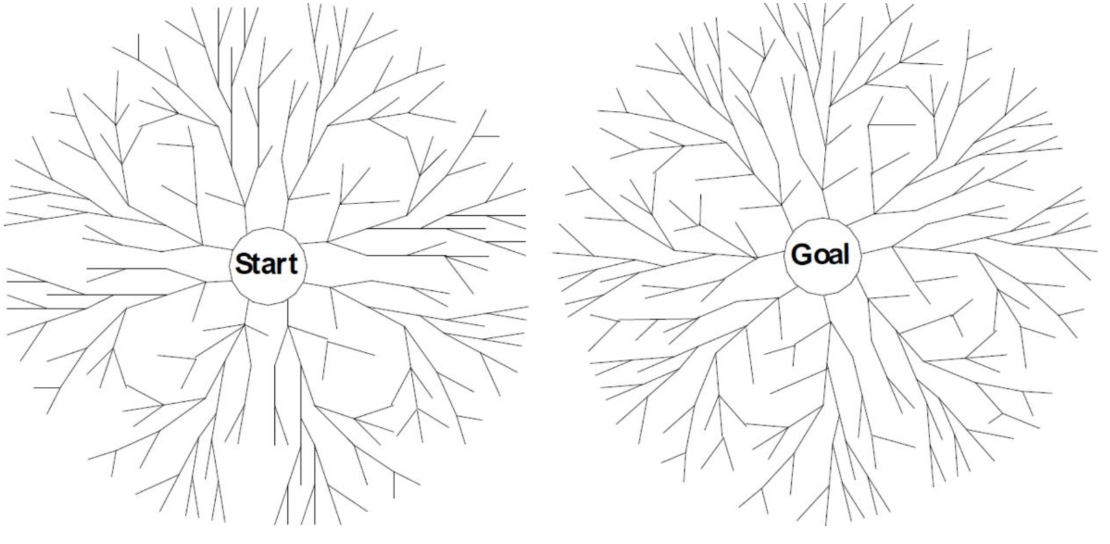
Bidirectional Search

- **Two simultaneous searches from start and goal**
  - Motivation: $b^{d/2} + b^{d/2} < b^d$ (BFS)
  - Check whether the node belongs to the other fringe before expansion



Bidirectional Search

- **The predecessor of each node should be efficiently computable**
  - When actions are easily reversible
  - Space complexity is the most significant weakness
  - Complete and optimal if both searches are BFS



Summary of Algorithms

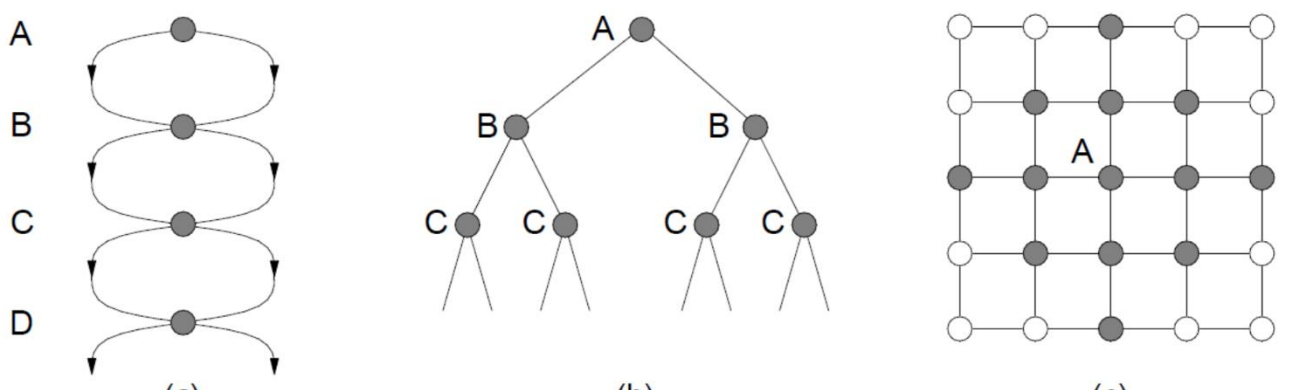
Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional
Complete	**YES***	**YES***	**NO**	**YES,** if $l \geq d$	**YES**	**YES***
Time	b^{d+1}	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal	**YES***	**YES***	**NO**	**NO**	**YES**	**YES**

Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Repeated States

- **Failure to detect repeated states can turn a solvable problems into unsolvable ones**



(a) (b) (c)

Graph-Search

- **Closed list stores all expanded nodes**
-

function GRAPH-SEARCH(*problem*, *fringe*) **return** a solution or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if EMPTY(*fringe*) **then return** failure
    node ← REMOVE-FIRST(*fringe*)
    if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds **then**
      return SOLUTION(*node*)
    if STATE[*node*] is not in ***closed*** **then**
      add STATE[*node*] to *closed*
    fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

Graph-Search

- **Optimality:**
  - When detecting a repeated state, i.e. matching a node on closed list, GRAPH-SEARCH discard *newly discovered* paths
    - This may result in a sub-optimal solution
      - If the new one is shorter
      - Won’t happen when using uniform-cost search or BFS with constant step cost

Graph-Search

- **Time and space complexity:**
  - Proportional to the size of the state space
    - May be much smaller than O(bd)
    - On problem with many repeated states, GRAPH-SEARCH is much more efficient than TREE-SEARCH
  - DFS and IDS with closed list require more than linear space
    - Since all nodes are stored in closed list!

Fundamental tradeoff between **time** and **space**:
Algorithms that forget their history are doomed to repeat it!

Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Search with Partial Information

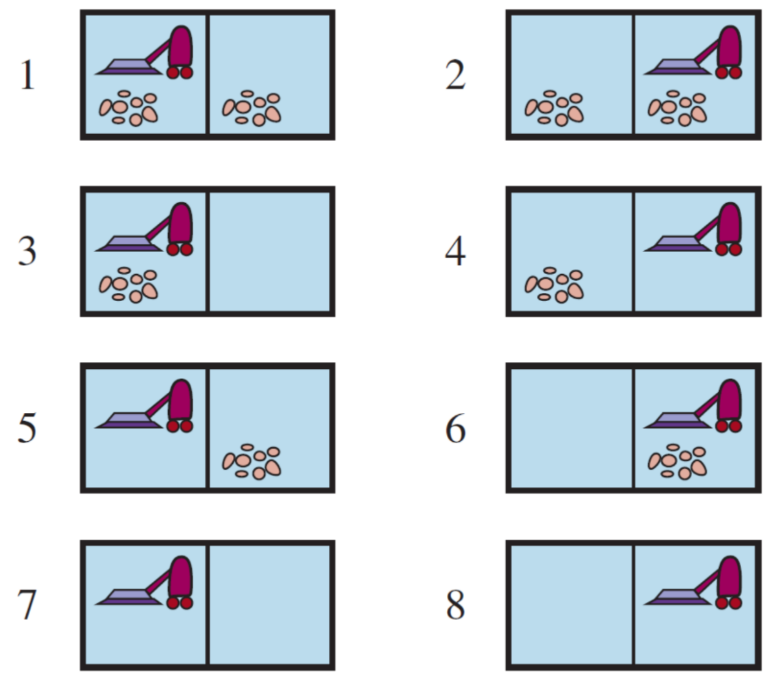
- **Previous assumption:**
  - Environment is *fully observable*
  - Environment is *deterministic*
  - Agent *knows* the effects of its actions
- **What if knowledge of states or actions is incomplete or uncertain?**

Search with Partial Information

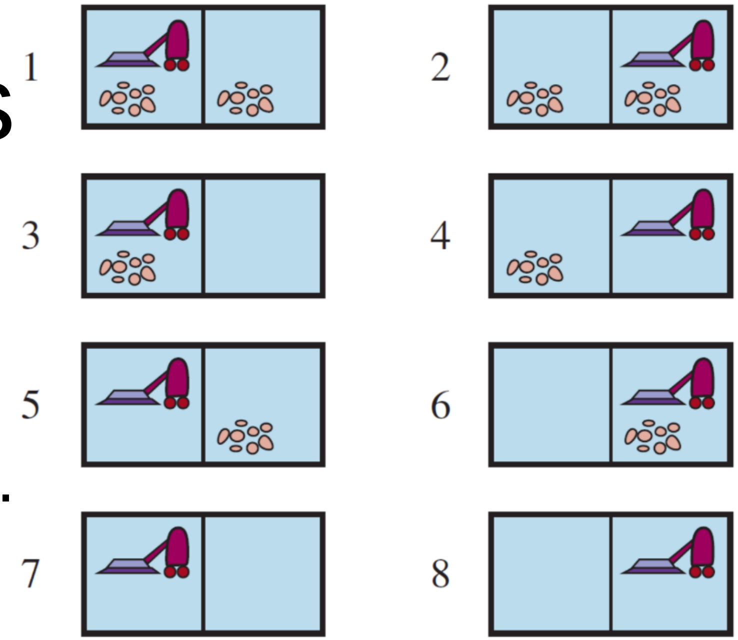
- **Partial knowledge of states and actions**
  - Sensorless or conformant problem
    - Non-observable
    - Agent may have no idea where it is; solution (if any) is a sequence
  - Contingency problem
    - Nondeterministic and/or partially observable
    - Percepts provide new information about current state; solution is a tree or policy; often interleave search and execution
    - If uncertainty is caused by actions of **another** agent → adversarial problem
  - Exploration problem
    - Unknown state space
    - When states and actions of the environment are unknown

Search with Partial Information

- **Single-state problem:**
  - Q: Single state, starting in #5. Solution?
  - A: [Right, Suck]
- **Sensorless problem:**
  - Q: Start in {1,2,3,4,5,6,7,8}
    e.g. Right goes to {2,4,6,8}.
    Solution?
- **Contingency:**
  - Q: Start in {1,3}.
    Assume Murphy’s law, Suck can dirty a clean carpet and local sensing: [*location*,*dirt*] only. Solution?



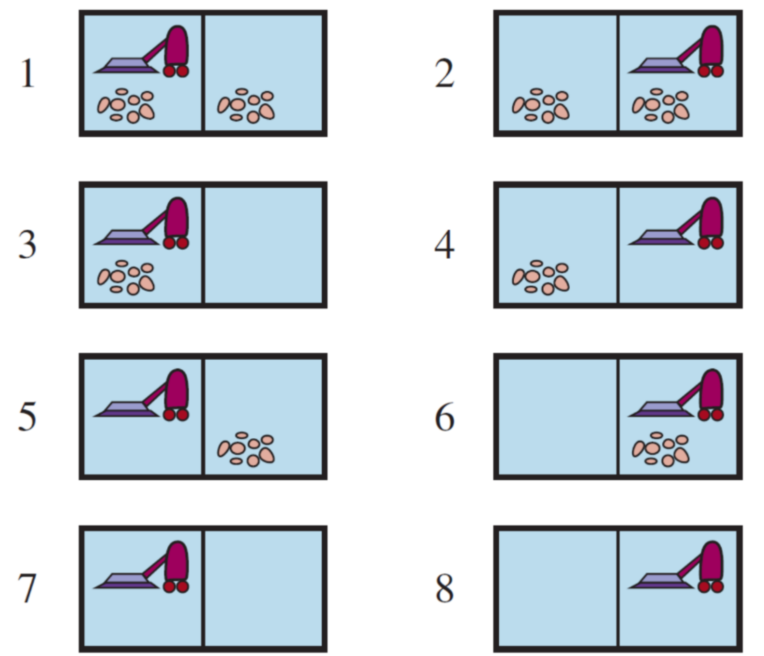
Sensorless Problems



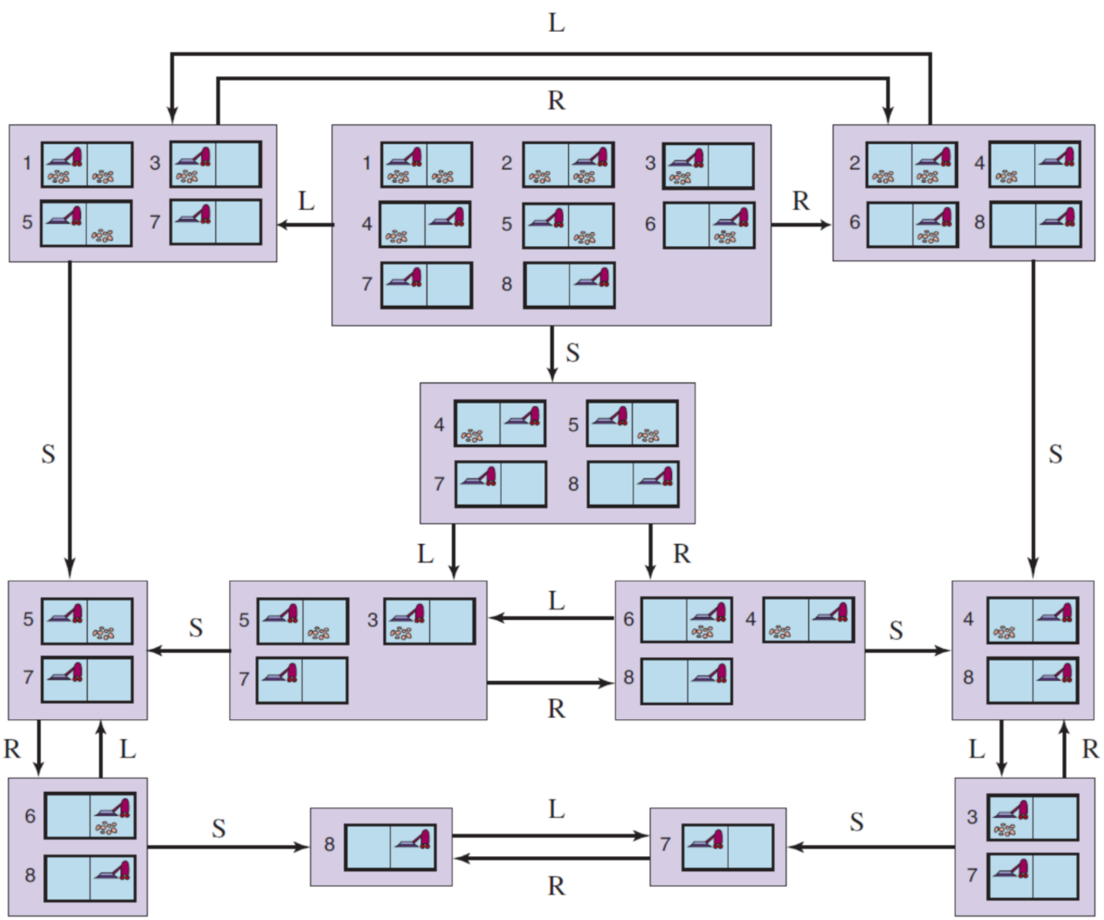
- **e.g. Vacuum world**
  - Q: Start in {1,2,3,4,5,6,7,8}
    e.g. Right goes to {2,4,6,8}.
    Solution?
  - A: [Right, Suck, Left, Suck]
- **When the world is not fully observable:**
  - reason about a set of states that **might be reached** → belief state
    - If fully observable, each belief state contains only one physical state

Sensorless Problems

- **Search space of belief states**
  - Search the space of belief states rather than physical states
  - Solution = a path to a belief state, *all* of whose members are goal states

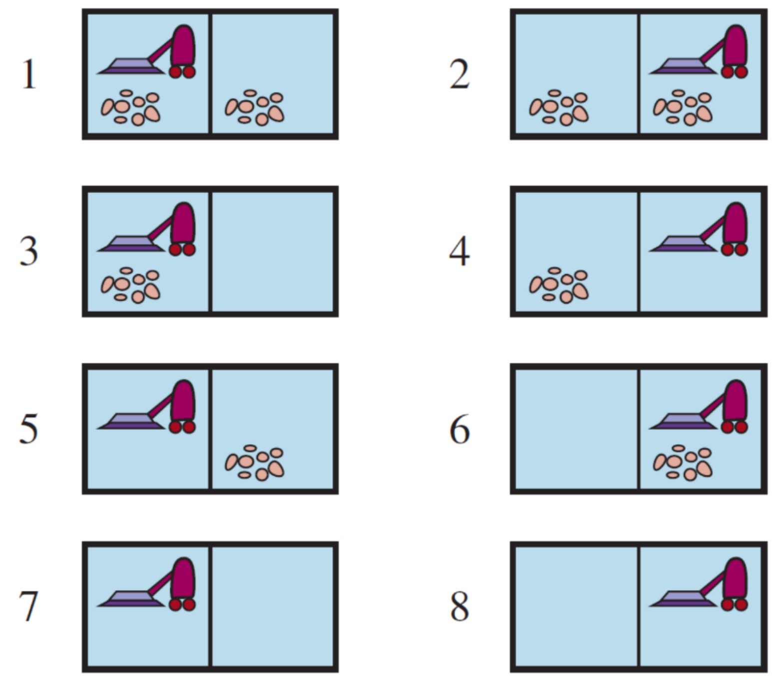


Belief-state of Vacuum World



Contingency Problems

- **e.g. Vacuum world**
  - Start in {1,3},
    Murphy's law: Suck can dirty a clean carpet
    Local sensing: dirt, location
    - Percept = [L,Dirty] ={1,3}
    - [Suck] = {5,7}
    - [Right] ={6,8}
    - [Suck] in {6}={8} (Success)
    - BUT [Suck] in {8} may return {6} (Murphy's law)
  - Solution??
    - Belief-state: no fixed action sequence guarantees solution



Contingency Problems

- **Relax requirement**
  - If don’t insist on fixed action sequence, then a solution:

    [Suck, Right, if [R,dirty] then Suck]
  - Select actions based on contingencies arising during execution
  - Most real-world problems are contingency problems, because exact prediction is impossible

Outline

- **Problem-solving agents**
- **Problem formulation**
- **Search for solutions**
- **Uninformed search strategies**
- **Avoiding repeated states**
- **Search with partial information**
- **Problem difficulty**

Reasons of Difficult Problems

- **Huge** search space
  - Exhaustive search for the best answer is forbidden
- **Complicated**
  - Result from the simplified models is useless
- **Dynamic**
  - The evaluation function is noisy or varies with time
    - Thereby needs not just a single solution but a series of ones
- **Constrained**
  - Even on e feasible answer is difficult, let alone the optimum
- **Multiple** objectives
  - Conflict between objectives

Size of Search Space

- **Satisfiability problem (SAT)**
  - Given a logic formula in CNF, the problem is to determine if the formula is satisfiable
    - E.g. 100-variable SAT:
      $f(\vec{x}) = (x_1 \vee x_5 \vee \neg x_3) \wedge (x_6) \wedge (\neg x_4 \vee x_2)$
  - For a n-variable SAT, the size of search space is 2^n
    - E.g. $|S| = 2^{100} \approx 10^{30}$
      For a computer that can test 1000 strings/sec, from Big Bang to now (15 billion years), we'd have examined < 1%
  - Evaluation function is another issue
    - Evaluation function guides us the quality of solution
    - What if SAT?

Size of Search Space

- **Traveling salesman problem (TSP)**
  - Find the shortest path to visit each city exactly
    - Symmetric and asymmetric
  - The size of search space is $n!/2n = (n-1)!/2$
    - E.g. 50-city TSP: $|S| = 49!/2 \approx 10^{62}$
    - 100-city TSP: $|S| \approx 4.67 \times 10^{155}$
    - 500-city TSP: $|S| \approx 1.22 \times 10^{1131}$
    - 3000-city TSP: $|S| \approx 6.91 \times 10^{9126}$

TSP instance with 24,978 cities

Size of Search Space

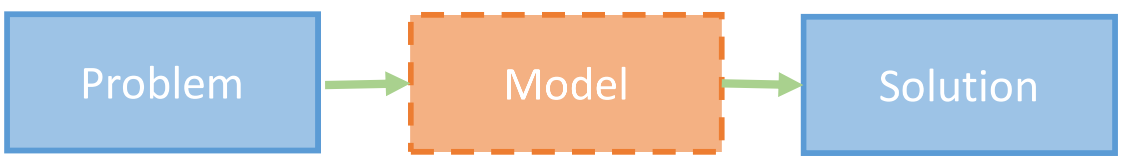
- **Nonlinear problem (NLP)**
  - Find the global maximum of function G2

    $$G_2(\vec{x}) = \left|\frac{\sum_{i=1}^{n}\cos^4(x_i) - 2\prod_{i=1}^{n}\cos^2(x_i)}{\sqrt{\sum_{i=1}^{n} i x_i^2}}\right|$$

    s.t. $\prod_{i=1}^{n} x_i \geq 0.75, \sum_{i=1}^{n} x_i \leq 7.5n$
    - Nonlinear
    - Constraints: 1 non-linear, 1 linear
  - Search space
    - Infinitely large due to real
    - Depends on resolution
      - E.g. precision guaranteed 6 digits
        each dimension has 10^7 values
        search space = 10^{7n}

Modeling the Problem

- **Solving a problem means we are in reality only find the solution to a model of the problem**
- **Process**
  - Create a model of the problem
  - Use the model to generate a solution



Modeling the Problem

- **E.g. SAT, TSP, NLP are canonical models**
  - SAT: model checking
  - TSP: die bonding, factory scheduling
  - NLP: function optimization
- **However, there is always a compromise on precision between model and solution**

Modeling the Problem

- Example
  - Suppose a company has *n* warehouses that store paper supplies in reams
  - These supplies are to be delivered to *k* distribution centers
  - The transportation cost between warehouse *i* and distribution center *j* is f_{ij} :

$$f_{ij}(x) = \begin{cases} 0 & x = 0 \\ 4 + 3.33x & 0 < x \leq 3 \\ 19.5 & 3 < x \leq 6 \\ 0.5 + 10\sqrt{x} & 6 < x \end{cases}$$

  - The problem is to minimize the transportation cost of all:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{k} f_{ij}(x_i) \quad s.t. \sum_{j=1}^{k} x_{ij} \leq warehouse(i), \sum_{i=1}^{n} x_{ij} \geq distribution(j), x_{ij} \geq 0$$

Modeling the Problem

- **Example** (cont'd)
  - The transportation cost serves as an evaluation function
    - Discontinuous → cannot use gradient-based methods
  - Two options
    - Simplify the model so that traditional optimizers might return better answers:
      Problem → $\text{Model}_{\mathbf{a}}$ → $\text{Solution}_{\mathbf{p}}(\text{Model}_{a})$
    - Keep the model but use non-traditional approach to find a near-optimum solution:
      Problem → $\text{Model}_{\mathbf{p}}$ → $\text{Solution}_{a}(\text{Model}_{\mathbf{p}})$
  - Difficult to obtain precise solution since we have to approximate either the *model* or the *solution*

Change over Time

- **Problem (environment) changes**
  - Before you model it
  - While you are deriving a solution
  - After you execute your solution
- **E.g.**
  - TSP: the cost changes over time
  - Game: your competitors are trying to defeat your solution

Constraints

- During search, we need to move from one solution to the next:
  - Feasible?
  - Improved?
- **Hard** constraints
  - Must be absolutely satisfied in order to have a feasible solution
- **Soft** constraints
  - We hope to accomplish but are NOT mandatory

Any solution that meets the hard constraints is feasible,
but not necessary optimal in light of soft constraints.

Constraints

- **Deal with infeasible solutions**
  - Discard
    - Diversity
  - Repair
    - Diversity
    - Computational cost
  - Prevent
    - Devise variation operators that never corrupt a feasible solution into an infeasible
    - Not always achievable

Multiple Objectives

- **Presence of multiple conflicting objectives**
  - Buying a car: speed vs. price vs. reliability
  - Engineering design: lightness vs. strength
  - Factors are conflicting and the final decision bears a compromise, a trade-off
- **Two part problem**
  - Finding set of good solutions
  - Choice of the best for particular application

Multiple Objectives

- **Methods**
  - Aggregation (scalarization)
    - Use weights to integrate into a single objective
    - Disadvantages?
  - Evolutionary algorithms
    - Good at solving MOP
    - Returns a set of solutions
    - NSGA2, PEAS, MOEA, etc.