Problem Solving & Searching

Lectures on AI & Soft Computing

References:

- AI: A modern approach by Russell & Norvig
- AI: Rich & Knight

Problem solving agents

- Problem-solving agents: find sequence of actions that achieve goals.
- Problem-Solving Steps:
  - *Goal transformation*: where a goal is set of acceptable states.
  - *Problem formation*: choose the operators and state space.
  - *search*
  - *execute solution*

Formulating Problems

- Problem types:
  - *Single state problems*: state is always known with certainty.
  - *Multi state problems*: know which states might be in.
  - *Contingency problems*: constructed plans with conditional parts based on sensors.
  - *Exploration problems*: agent must learn the effect of actions.

- Formal definition of a problem:
  - Initial state (or set of states)
  - set of operators
  - goal test on states
  - path cost

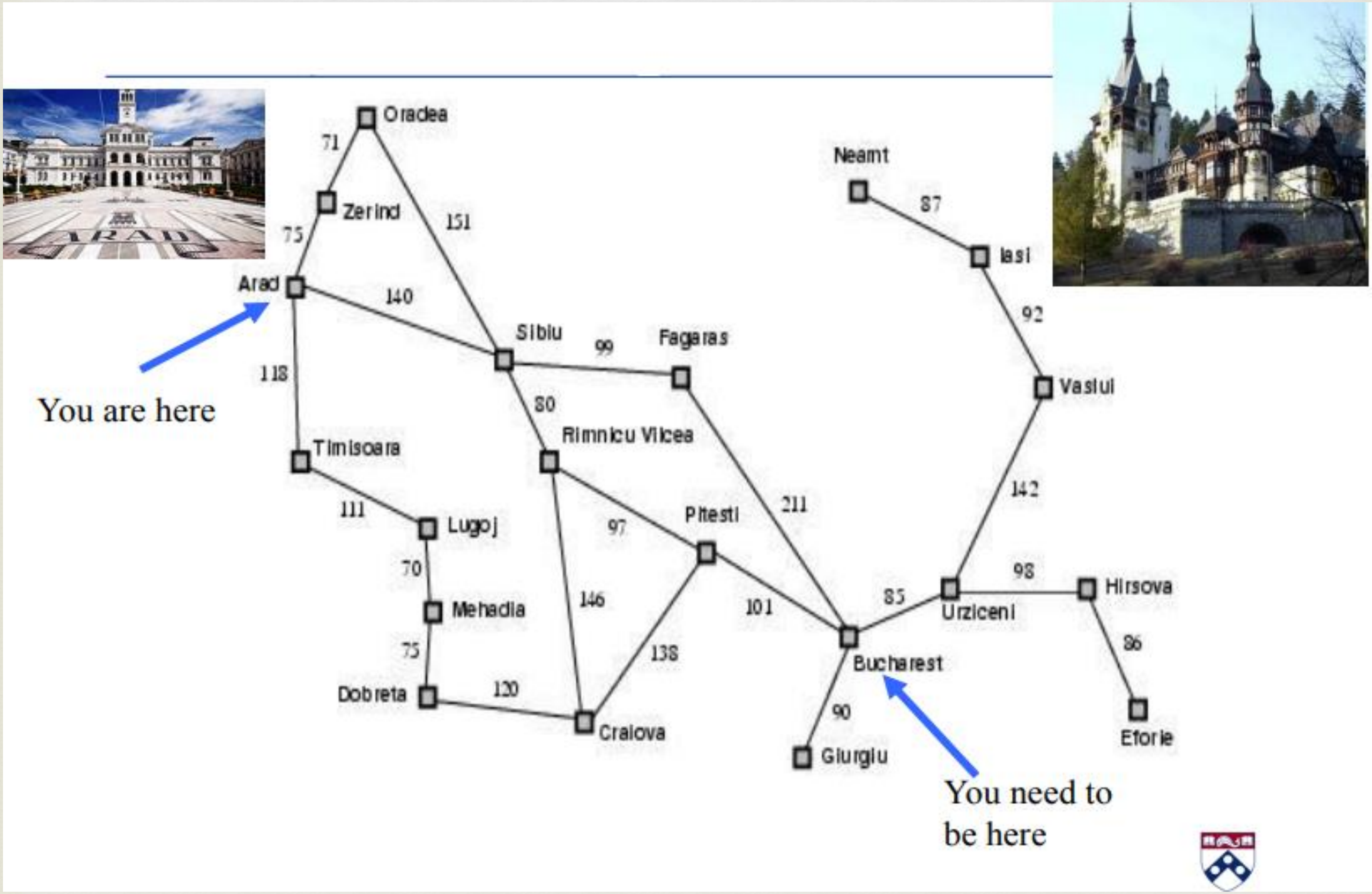
Formulating Problems

- Measuring performance:
  - Does it find a solution?
  - What is the **search cost**?
  - What is the **total cost**?

  (total cost = path cost + search cost)
  - Performance measure: minimize total moves
  - Finding solution:

  Sequence of pieces moved/moves made: 3,1,6,3,1,…
- Choosing states and actions:
  - Abstraction: remove unnecessary information from representation; makes it cheaper to find a solution.

Example of Search Problem: holiday in Romania



Holiday in Romania

- **On holiday in Romania; currently in Arad**
  - Flight leaves tomorrow from Bucharest
- Formulate **goal**
  - Be in Bucharest
- Formulate **search problem**
  - States: various cities
  - Actions: drive between cities
  - Performance measure: minimize distance
- Find **solution**
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest,

Formal definition of a problem

1. A set of *states* $\boldsymbol{S}$
2. An *initial state* $s_i \in S$
3. *A set of actions* $\boldsymbol{A}$
   — $\forall s$ *Actions(s)* = *the set of actions that can be executed in* s, *that are* applicable in s.
4. *Transition Model:* $\forall s \forall a \in Actions(s)\ Result(s, a) \rightarrow s_r$
   — s_r *is called a* successor *of* s
   — $\{s_i\} \cup Successors(s_i)^*$ = *state space*
5. *Goal test* ***Goal(s)***
   — Can be implicit, e.g. ***checkmate(x)***
   — s is a *goal state* if ***Goal(s)*** is *true*
6. *Path cost* (additive)
   — e.g. sum of distances, number of actions executed, …
   — $c(x,a,y)$ is the step cost, assumed ≥ 0
     - (where action a goes from state x to state y)

Solution

- A **solution** is a sequence of actions from the **initial state** to a **goal state**.

- **Optimal Solution**:

  A solution is **optimal** if no solution has a lower path cost.

Design for a simple problem solving agent

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

      state <- UPDATE-STATE(state, percept)
      if s is empty then
            g <- FORMULATE-GOAL(state)
            problem <- FORMULATE-PROBLEM(state)
            s <- SEARCH(problem)

      action <- RECOMMENDATION(s, state)
      s <- REMAINDER(s)
      return action
```

Example Problems

- Toy problems
  - 8-puzzle
  - 8-queen/n-queen
  - cryptarithmetic
  - vacuum world
  - missionaries and cannibals
- Real World
  - Traveling Salesperson (NP hard)
  - VLSI layout
  - robot navigation
  - assembly sequencing

Formulating a Search Problem

- **Which properties matter & how to represent**
  - *Initial State, Goal State, Possible Intermediate States*
- **Which actions are possible & how to represent**
  - *Operator Set: Actions and Transition Model*
- **Which action is next**
  - *Path Cost Function*

Formulation greatly affects combinatorics of search space and therefore speed of search

Example: Missionaries & Cannibals

Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten.

How shall they cross the river?

Example: Missionaries & Cannibals

States: (CL, ML, BL)

Initial 331 **Goal** 000

Actions:

Travel Across	*Travel Back*
-101	101
-201	201
-011	011
-021	021
-111	111

Missionaries & Cannibals: assumptions

- Number of cannibals should lesser than the missionaries on either side.
- Only one boat is available to travel.
- Only one or maximum of two people can go in the boat at a time.
- All the six have to cross the river from bank.
- There is no restriction on the number of trips that can be made to reach of the goal.
- Both the missionaries and cannibals can row the boat.

Missionaries & Cannibals: Production rules

Production Rules for Missionaries and Cannibals Problem.

Rule No	Production Rule and Action
1	(i,j) : Two missionaries can go only when i-2 >=j or i-2=0 in one bank and i+2>=j in the other bank.
2	(i,j) : Two cannibals can cross the river only when j-2, <=i or i=0 in one bank and j+2 <=i or I+0 or i=0 in the other.
3	(i,j) : One missionary and one cannibal can go in a goat only when i-1>= j-1 or i=0 in one bank and i+1 >= j+1 or i=0 in the other.
4	(i,j) : one missionary can cross the river only when i-1>=j or i=0 in one bank and i+1 >=j in the other bank.
5	(i,j) : One cannibal can cross the river only when j-1 < i or i=0 in one bank and j+1<=i or j=0 in the other bank of the river.

Fig:- Production rules for the missionaries and cannibals problem.

For this problem, there are several sequences of operators that will solve the problem . The next figure is one of the solutions.

PRODUCTION SYSTEMS

A production system consists of rules and factors. Knowledge is encoded in a declarative from which comprises of a set of rules of the form

Missionaries & Cannibals

- **Possible Moves**

A move is characterized by the number of missionaries and the number of cannibals taken in the boat at one time. Since the boat can carry no more than two people at once, the only feasible combinations are:

Carry (2, 0).

Carry (1, 0).

Carry (1, 1).

Carry (0, 1).

Carry(0, 2).

Where Carry (M, C) means the boat will carry M missionaries and C cannibals on one trip.

Missionaries & Cannibals

- **Feasible Moves**

Once we have found a possible move, we have to confirm that it is feasible. It is not a feasible to move more missionaries or more cannibals than that are present on one bank.

When the state is state(M1, C1, left) and we try carry (M,C) then

M <= M1 and C <= C1

must be true.

When the state is state(M1, C1, right) and we try carry(M, C) then

M + M1 <= 3 and C + C1 <= 3

must be true.

Missionaries & Cannibals

- **Legal Moves**

  Once we have found a feasible move, we must check that is legal i.e. no missionaries must be eaten.

  - ✓ Legal(X, X).
  - ✓ Legal(3, X).
  - ✓ Legal(0, X).

The only safe combinations are when there are equal numbers of missionaries and cannibals or all the missionaries are on one side.

Missionaries & Cannibals: a sample solution

Situation ------------ Action

Bank 1	Boat	Bank 2	Production rule applied
(3,3)	(0,2)	(0,0)	
(3,1)	(0,1)	(0,2)	------------ 2
(3,2)	(0,2)	(0,1)	------------ 5
(3,0)	(0,1)	(0,3)	------------ 2
(3,1)	(2,0)	(0,2)	------------ 5
(1,1)	(1,1)	(2,2)	------------ 1
(2,2)	(2,0)	(1,1)	------------ 3
(0,2)	(0,1)	(3,1)	-------------1
(0,3)	(0,2)	(3,0)	------------ 5
(0,1)	(0,1)	(3,2)	-------------2
(0,2)	(0,2)	(3,1)	------------ 5
(0,0)		(3,3)	------------ 2

Fig:- One solution to missionaries and cannibals problem

Example: Water Jug Problem

- **Problem:** We are given two jugs, a 4-gallon one and 3-gallon one. Neither has any measuring marked on it. There is a pump, which can be used to fill the jugs with water. How can we get exactly 2 gallons of water into 4-gallon jug?
- **State space:** set of ordered pairs of integers (X, Y) such that X = 0, 1, 2, 3 or 4 and Y = 0, 1, 2 or 3; X is the number of gallons of water in the 4-gallon jug and Y the quantity of water in the 3-gallon jug.
- **Start state** is (0, 0) and the **Goal state** is (2, n) for any value of n, as the problem does not specify how many gallons need to be filled in the 3-gallon jug (0, 1, 2, 3). So the problem has one initial state and many goal states.

The operators to be used to solve the problem can be described as shown in Fig.

1.	(X, Y) if $X < 4 \rightarrow (4, Y)$	Fill the 4-gallon jug
2.	(X, Y) if $Y < 3 \rightarrow (X, 3)$	Fill the 3-gallon jug
3.	(X, Y) if $X = d$ & $d > 0 \rightarrow (X\text{-}d, Y)$	Pour some water out of the 4-gallon jug
4.	(X, Y) if $Y = d$ & $d > 0 \rightarrow (X, Y-d)$	Pour some water out of 3-gallon jug
5.	(X, Y) if $X > 0 \rightarrow (0, Y)$	Empty the 4-gallon jug on the ground
6.	(X, Y) if $Y > 0 \rightarrow (X, 0)$	Empty the 3-gallon jug on the ground
7.	(X, Y) if $X + Y \leq 4$ and $Y > 0 \rightarrow 4, (Y - (4 - X))$	Pour water from the 3-gallon jug into the 4-gallon jug until the gallon jug is full.
8.	(X, Y) if $X + Y \geq 3$ and $X > 0 \rightarrow (X - (3 - Y), 3))$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full.
9.	(X, Y) if $X + Y \leq 4$ and $Y > 0 \rightarrow (X + Y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10.	(X, Y) if $X + Y \leq 3$ and $X > 0 \rightarrow (0, X + Y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11.	$(0, 2) \rightarrow (2, 0)$	Pour the 2-gallons water from 3-gallon jug into the 4;gallon jug
12.	$(2, Y) \rightarrow (0, Y)$	Empty the 2-gallons in the 4-gallon jug on the ground.

Fig. *Production rules (operators) for the water jug problem.*

Water Jug Problem: Assumptions

- We can fill a jug from the pump.
- We can pour water out a jug, onto the ground.
- We can pour water out of one jug into the other.
- No other measuring devices are available.

There are several sequences of operators which will solve the problem, two such sequences are shown in Fig. 2.4:

Water in 4-gallon jug (X)	Water in 3-gallon jug (Y)	Rule applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

Fig. 2.4 (*a*). *A solution to water jug problem.*

X	Y	Rule applied (Control strategy)
0	0	
4	0	I-
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

Fig. 2.4 (*b*). *2nd solution to water jug problem.*

Search fundamentals: Useful Concepts

- ***State space*:** **the set of all states reachable from the initial state by *any* sequence of actions**
  - *When several operators can apply to each state, this gets large very quickly*
  - *Might be a proper subset of the set of configurations*
- ***Path:* *a sequence of actions leading from one state s_j to another state s_k***
- ***Frontier:* *those states that are available for expanding (for applying legal actions to)***
- ***Solution:* *a path from the initial state s_i to a state s_f that satisfies the goal test***

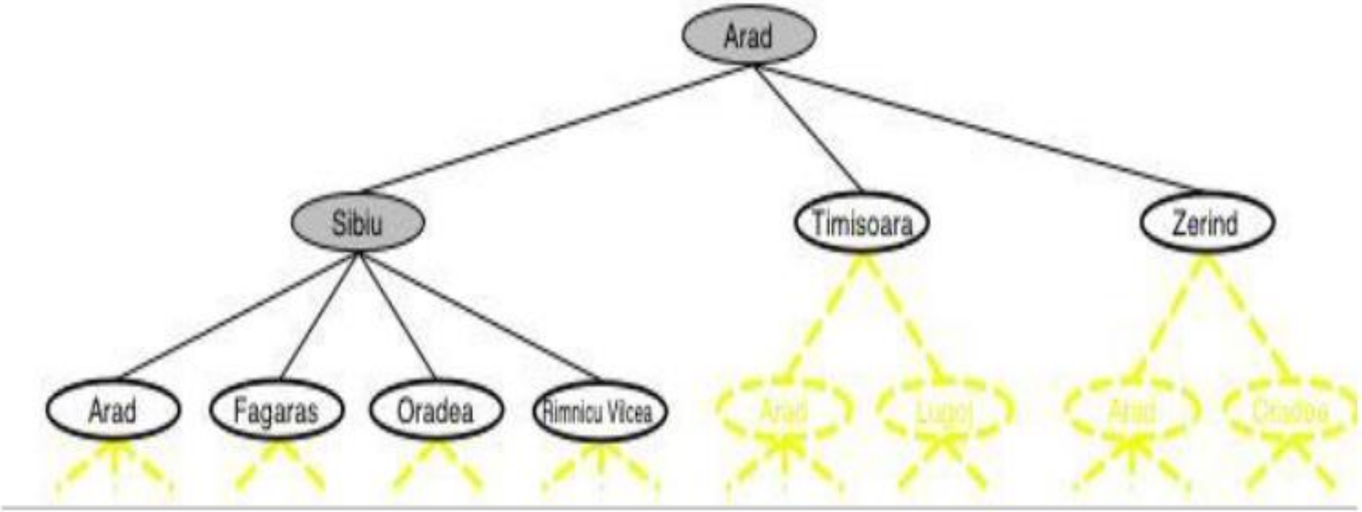
Basic search algorithms: *Tree Search*

- Generalized algorithm to solve search problems
- *Enumerate in some order all possible paths from the initial state*
- Here: search through *explicit tree generation*
  - ROOT= initial state.
  - Nodes in search tree generated through *transition model*
- In general search generates a *graph*(same state through multiple paths), but we'll just look at *trees*
  - Tree search treats different paths to the same node as distinct

Basic search algorithms: *Tree Search*

- Choice of which node to expand is determined by the search strategy.
- Collection of nodes that have been generated but not expanded is the fringe.
- Fringe can be represented as a set of nodes.
- Search strategy needs to look at every element of the set to choose the best one.
- So collection of nodes is implemented as a queue.

Generalized tree search



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

Initialize frontier **to the *initial state* of the *problem***

do

if **the frontier is empty** then return ***failure***

choose leaf node for expansion according to strategy & remove from frontier

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

else **expand the node and add resulting nodes to the frontier**

Determines search *process!!*

General Tree-Search Procedure

```
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)
```

```
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
     STATE[s] ← result
     PARENT-NODE[s] ← node
     ACTION[s] ← action
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
     DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

Make-Node

Queue functions

- **Make-Queue**(element, ..): Create a queue with the given elements.
- **Empty?**(queue): Returns true if the queue is empty.
- **Remove-First**(queue): Removes the element at the head of the queue and returns it.
- **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.

Criteria for Search strategies

- **Completeness**

  Is the strategy guaranteed to find a solution when there is one?
- **Time Complexity**

  How long does it take to find a solution?
- **Space Complexity**

  How much memory does the search require?
- **Optimality**

  Does the strategy find the best solution (with the lowest path cost)?

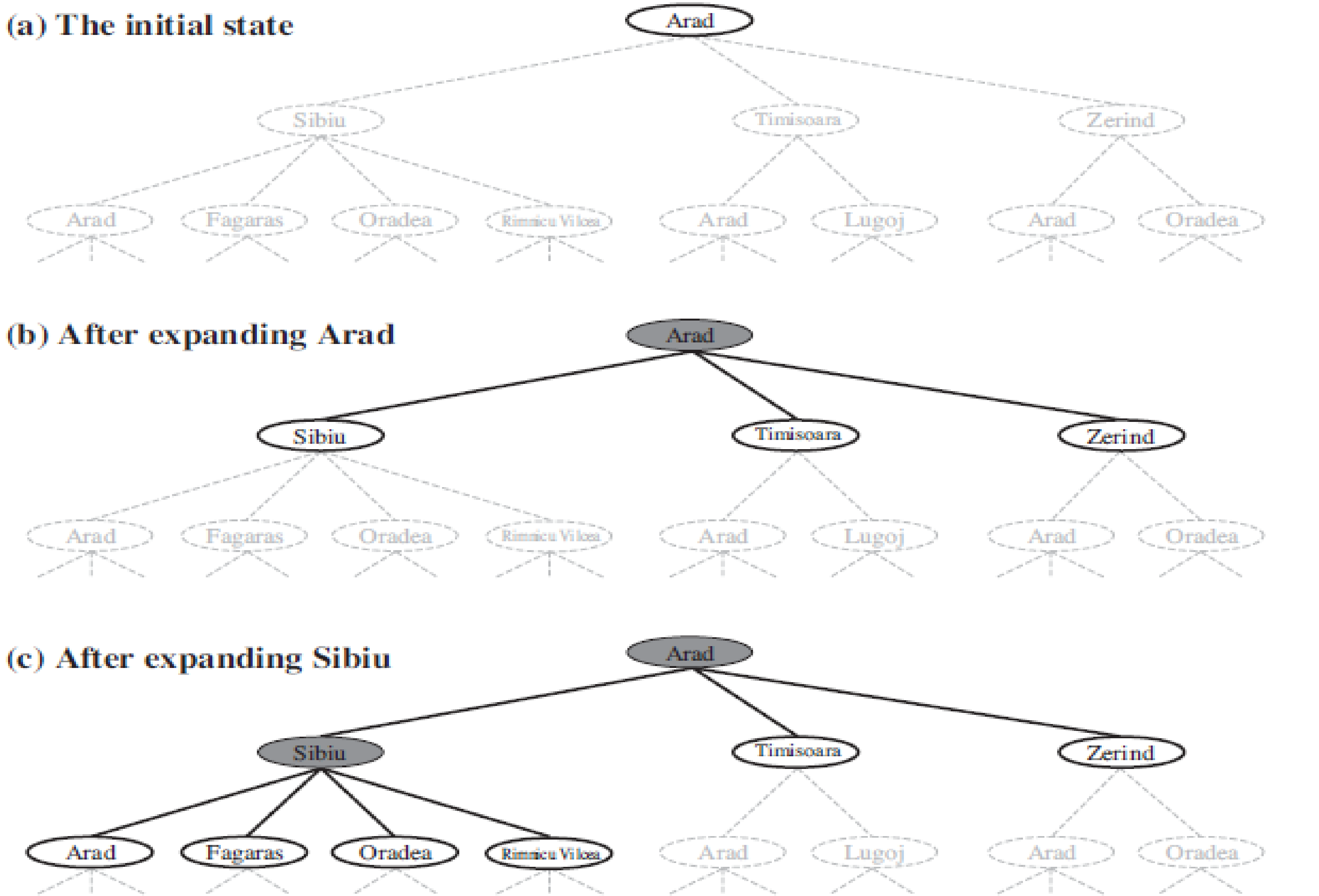
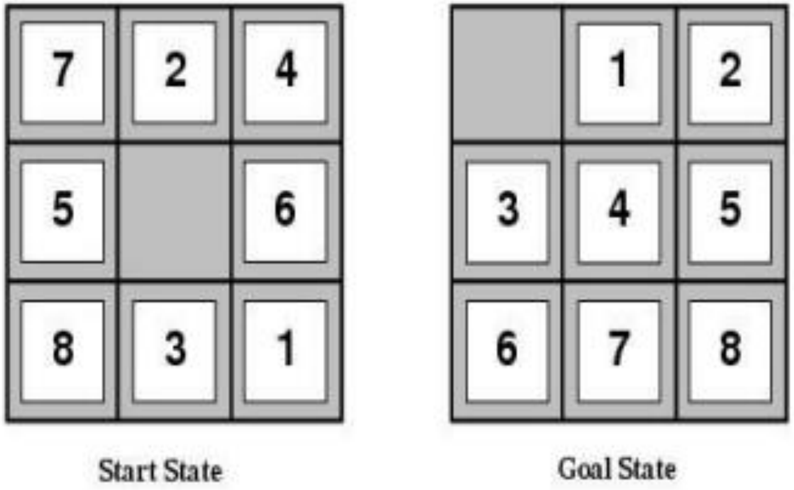


Figure Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

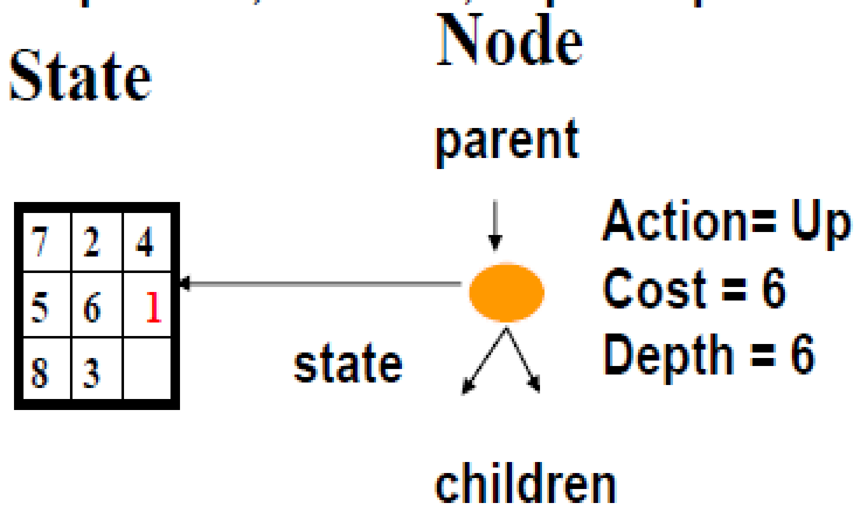
Example: 8-puzzle



- **States??** List of 9 locations- e.g., [7,2,4,5,-,6,8,3,1]
- **Initial state??** [7,2,4,5,-,6,8,3,1]
- **Actions??** *{Left, Right, Up, Down}*
- **Transition Model??** ...
- **Goal test??** Check if goal configuration is reached
- **Path cost??** Number of actions to reach goal

8-Puzzle: States and Nodes

- A *state* is a (representation of a) *physical configuration*
- A *node* is a data structure constituting *part of a search tree*
  - Also includes *parent, children, depth, path cost g(x)*
  - Here *node= <state, parent-node, children, action, path-cost, depth>*
- States do not have parents, children, depth or path cost!

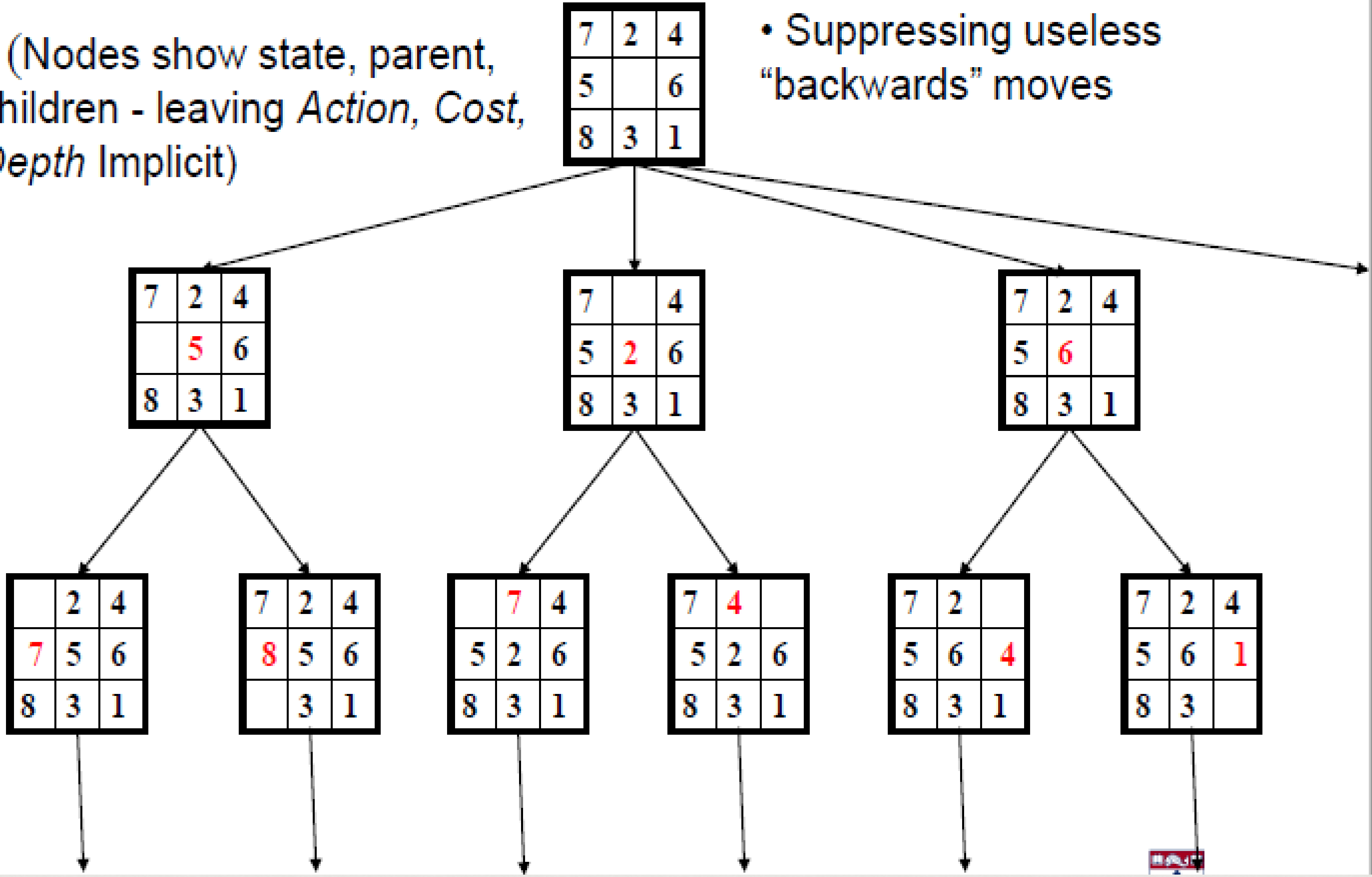


- The EXPAND function
  - uses the Actions and Transition Model to create the corresponding states
    - —creates new nodes,
    - —fills in the various fields

8-Puzzle *Search Tree*

- (Nodes show state, parent, children - leaving *Action*, *Cost*, *Depth* Implicit)

- Suppressing useless "backwards" moves



Uninformed Search Strategies

- Also known as “Blind Search”
- No additional information about the states beyond that provided in problem definition.
- They can only generate successors and distinguish a goal state from a non-goal state.
- “Informed search” or “Heuristic search” strategies help to find if a goal state is more promising than a non-goal state.

Breadth first search strategy

- Nodes are expanded in the order they were produced.
- Fringe is empty.
- Always finds the shallowest goal state first.
- Completeness.
- The solution is optimal, provided the path cost is a non- decreasing function of the depth of the node (e.g., when every action has identical, non-negative costs).

ALGORITHM: BREADTH-FIRST SEARCH

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
   a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
   b. For each way that each rule can match the state described in E do:

i) Apply the rule to generate a new state.
ii) If the new state is the goal state, quit and return this state.
iii) Otherwise add this state to the end of NODE-LIST

Breadth first search: Time & Space requirements

The costs, however, are very high. Let b be the maximal branching factor and d the depth of a solution path. Then the maximal number of nodes expanded is

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) \in O(b^{d+1})$$

Example: $b = 10$, 10,000 nodes/second, 1,000 bytes/node:

Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	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}	3,523 years	1 exabyte

Uniform Cost Search

- Modification of breadth-first search to always expand the node with the lowest-cost g(n).
- At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n.
- The nodes are stored in a priority queue.
- Also known as *Dijkstra's single-source shortest algorithm*.
- The worst case time complexity of uniform-cost search is $O(b^c/m)$, where c is the cost of an optimal solution and m is the minimum edge cost.

Uniform Cost Search (contd.)

- It is guided by path costs (not depths).
- If C* is cost of optimal solution & every action costs at least ϵ then worst case time complexity is:

$$O(b^{\lceil C^*/\epsilon \rceil})$$

- When all step costs are equal above complexity becomes b^d (where **b** is the branching factor and **d** is the depth)

Depth first search strategy

ALGORITHM: DEPTH FIRST SEARCH

1.If the initial state is a goal state, quit and return success.
2.Otherwise, loop until success or failure is signaled.
a) Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.
b) Call Depth-First Search with E as the initial state.
c) If success is returned, signal success. Otherwise continue in this loop.

Depth first search strategy (contd.)

- Always expands deepest node in the current fringe of the search tree.
- Implemented by:
- Tree search with a LIFO structure/stack
- Use a recursive function that calls itself with each successor/child node.
- Upon expanding, a node can be removed from memory after all its descendants have been expanded.
- Storage requirement = bm+1 where b=branching factor and m = max. depth for a given state space.

Depth first search strategy (contd.)

Depth-limited Search

Iterative Deepening DFS

Bidirectional Search

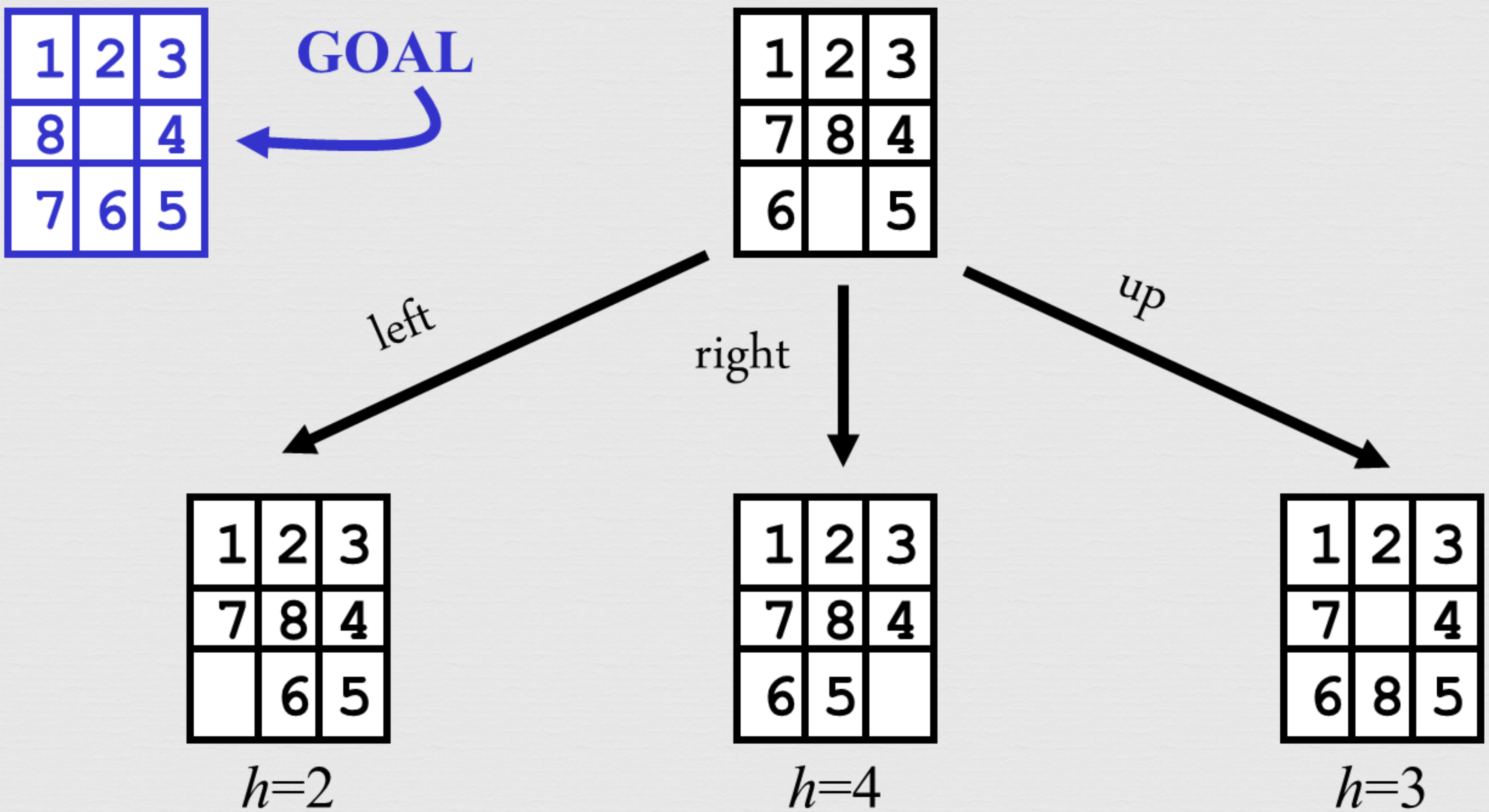
Avoiding repeated states

Searching with partial information

Informed Search

Heuristics

- All informed sea
- Heuristic search techniques make use of domain specific information - a heuristic search is based on problem specific information.
- For the 8-puzzle problem, what heurisitic(s) can we use to decide which 8-puzzle move is “best” (worth considering first).



GOAL
left
right
up
h=2
h=4
h=3

A Simple 8-puzzle heuristic

- ✓ Number of tiles in the correct position.
  - – The higher the number the better.
  - – Easy to compute (fast and takes little memory).
  - – Probably the simplest possible heuristic.

Another approach

- Number of tiles in the incorrect position.
  - This can also be considered a lower bound on the number of moves from a solution!
  - The “best” move is the one with the lowest number returned by the heuristic.
  - Is this heuristic more than a heuristic (is it always correct?).
    - Given any 2 states, does it always order them properly with respect to the minimum number of moves away from a solution?

Heuristic Search

- Generate & Test
- Hill Climbing
- Simulated annealing
- Best-first search
- Means-ends analysis
- Constraint satisfaction

Generate & Test

- Algorithm:

1) Generate a possible solution.
2) Test to see if this is actually a solution.
3) Quit if a solution has been found. Otherwise, return to step 1.

Generate & Test

- Pros:

  Acceptable for simple problems.

- Cons:

  Inefficient for problems with large space.

Generate & Test: variations

1) **Exhaustive** generate-and-test.
2) **Heuristic** generate-and-test: not consider paths that seem unlikely to lead to a solution.
3) **Plan generate-test**:
   - Create a list of candidates.
   - Apply generate-and-test to that list.

Generate & Test

- **Example:** coloured blocks

Arrange four 6-sided cubes in a row, with each side of each cube painted one of four colors, such that on all four sides of the row one block face of each color is showing.

- **Heuristic:**

If there are more red faces than other colors then, when placing a block with several red faces use few of them as possible as outside faces.

Iterative improvement algorithms

- In many optimization problems, the path is irrelevant - the goal state itself is the solution.
- Then the state space can be the set of "complete" configurations.
  – e.g., for 8-queens, a configuration can be any board with 8 queens
- In such cases, use iterative improvement algorithms. Keep a single "current" state, and try to improve it.
  – e.g., for 8-queens, we gradually move some queen to a better place
- The goal would be to find an optimal configuration
  – e.g., for 8-queens, an optimal configuration is where no queen is threatened.
- This takes constant space, and is suitable for online as well as offline search.

Hill Climbing: basics

- Start from some state *s*.
- Move to a neighbour *t* with better score. Repeat.

  [Neighborhood of a state is the set of neighbors. Also called 'move set'.]
- For hill climbing:

  Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.

Hill Climbing

- Algorithm:
  1) Determine successors of current state.
  2) Choose successor of maximum goodness (break ties randomly).
  3) If goodness of best successor is less than current state's goodness, stop.
  4) Otherwise make best successor the current state and go to step 1.

Heuristic function as a way to inject task-specific knowledge into the control process.

Hill Climbing: Examples

- **Example:** coloured blocks

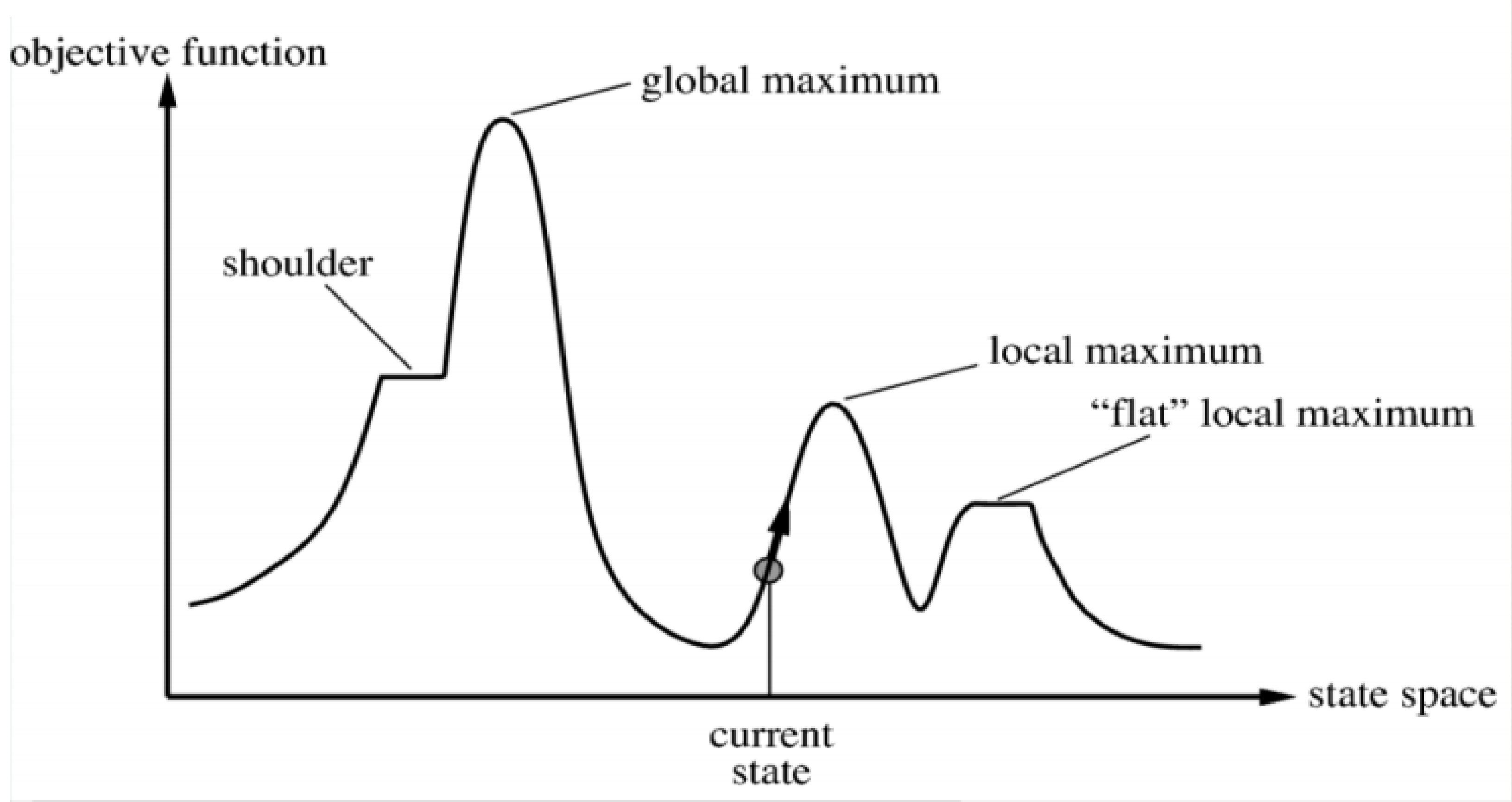
Heuristic function: the sum of the number of different colours on each of the four sides (solution = 16).

- **Example:** Complete state formulation for 8 queen

Successor function: move a single queen to another square in the same column

Cost: number of pairs that are attacking each other.

Hill Climbing: Problems



Hill Climbing: Problems

- **Local maxima** Once the top of a hill is reached the algorithm will halt since every possible step leads down.
- **Plateaux** If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.
- **Ridges** If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search. Orientation of the high region, compared to the set of available moves, makes it impossible to climb up.

[However, two moves executed serially may increase the height.]

- http://www.sdsc.edu/~tbailey/teaching/cse151/lectures/chap03a.html
- http://www.engineeringenotes.com/artificial-intelligence-2/state-space-search/notes-on-water-jug-problem-artificial-intelligence/34582