MENG 3065 - MODULE 3

Artificial Intelligence: A Modern Approach – Chapter 3 Solving Problems by Searching

HUMBER

WE ARE HUMBER

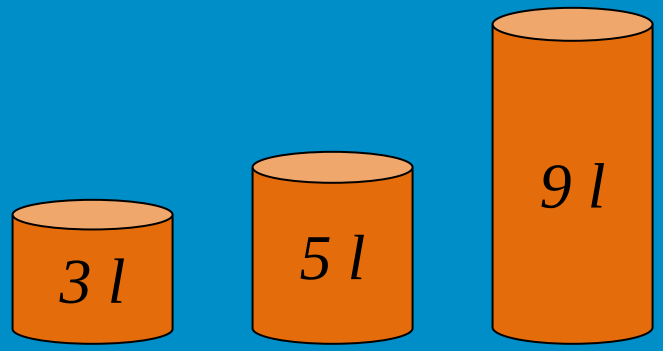
Outline

- Solving Problems By Searching (Chapter 3)
  - **Introduction to Problem Solving**
  - **Un-informed search**
    - Problem formulation
    - Search strategies: depth-first, breadth-first
  - **Informed search**
    - Search strategies: best-first, A*
    - Heuristic functions

Example: Measuring problem!

Problem: Using these three buckets,

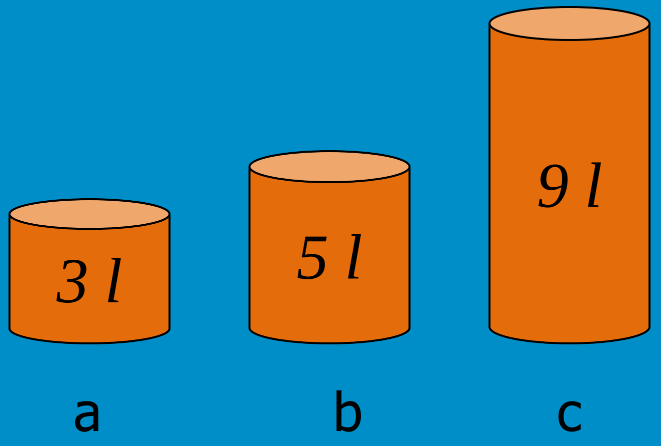
measure 7 liters of water.



Example: Measuring problem!

- **(one possible) Solution:**

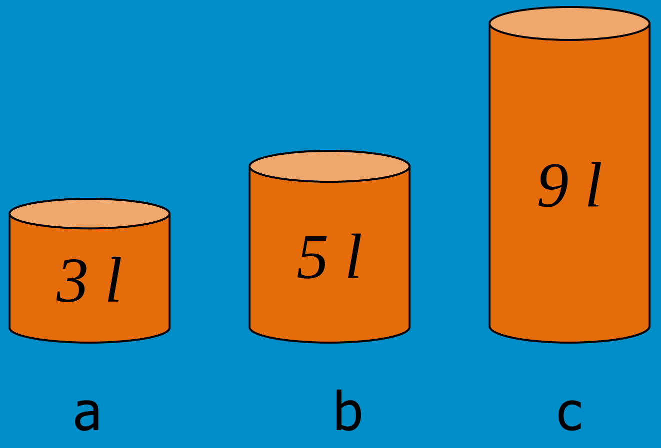
a	b	c	
0	0	0	start



Example: Measuring problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	

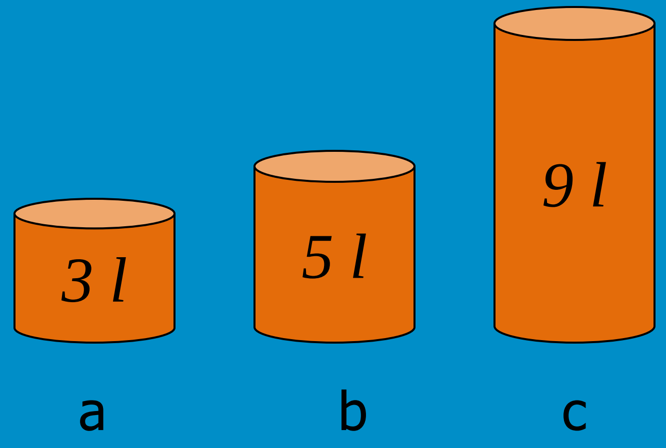


WE ARE HUMBER

Example: Measuring problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	

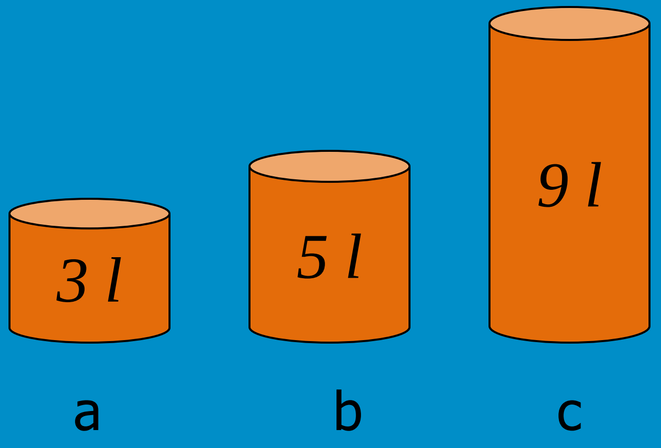


WE ARE HUMBER

Example: Measuring problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	

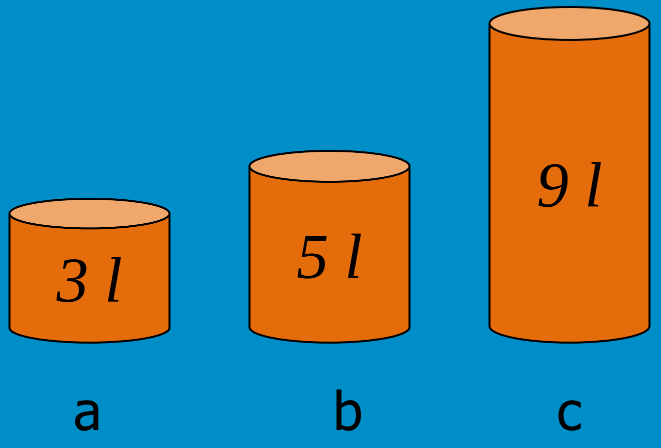


WE ARE HUMBER

Example: Measuring problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	

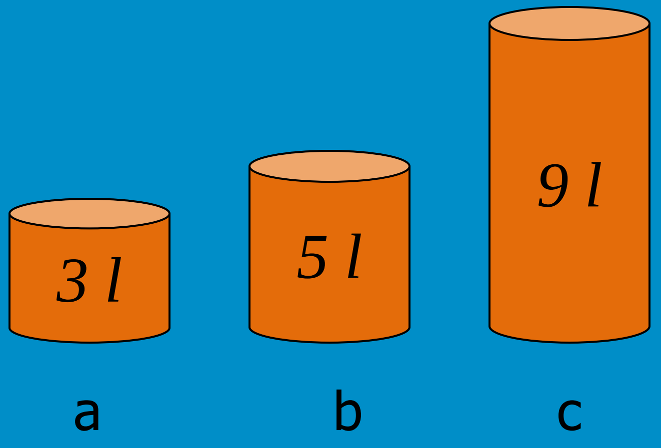


WE ARE HUMBER

Example: Measuring problem!

- **(one possible) Solution:**

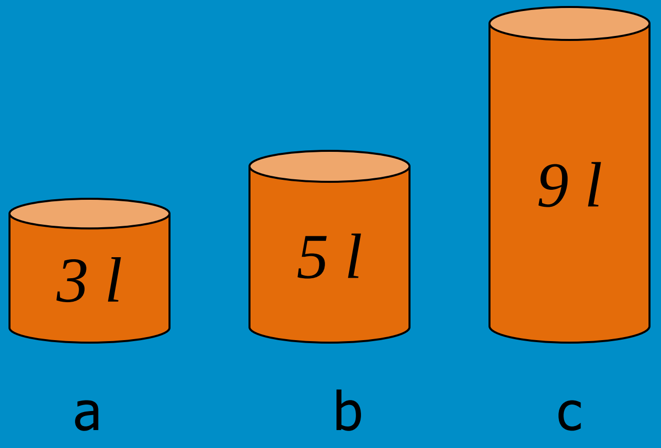
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	



Example: Measuring problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	**7**	**goal**

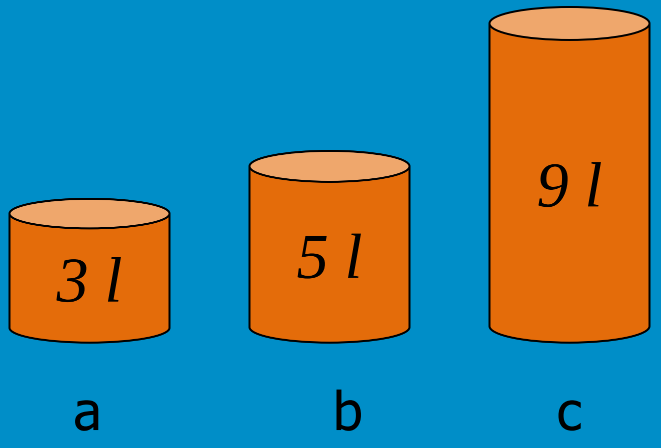


WE ARE HUMBER

Example: Measuring problem!

- **Another Solution:**

a	b	c	
0	0	0	start
0	5	0	

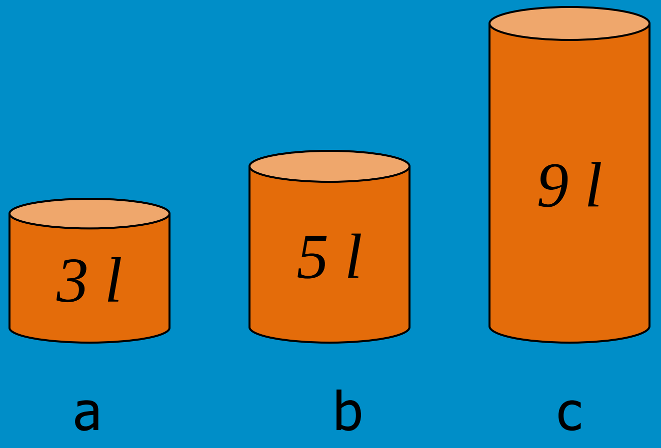


WE ARE HUMBER

Example: Measuring problem!

- **Another Solution:**

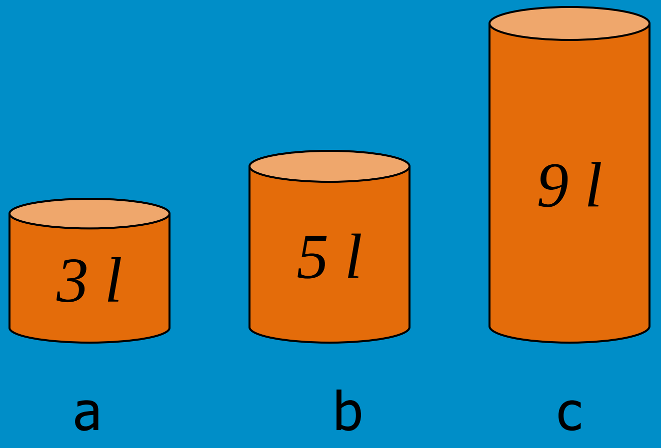
a	b	c	
0	0	0	start
0	5	0	
3	2	0	



Example: Measuring problem!

- **Another Solution:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	

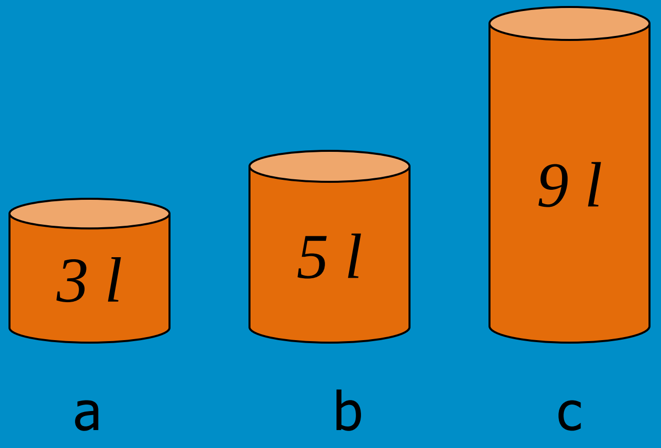


WE ARE HUMBER

Example: Measuring problem!

- **Another Solution:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	**0**	**7**	**goal**



Which solution do we prefer?

- **Solution 1:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	**7**	**goal**

- **Solution 2:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	**0**	**7**	**goal**

WE ARE HUMBER

Problem-solving agents

- Restricted form of general agent:

```
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
        goal ← Formulate-Goal(state)
        problem ← Formulate-Problem(state, goal)
        seq ← Search(problem)
    action ← Recommendation(seq, state)
    seq ← Remainder(seq, state)
    return action
```

- Note: this is offline problem solving; solution executed "eyes closed." Online problem solving involves acting without complete knowledge.

Problem types

- Deterministic, fully observable =⇒ single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable =⇒ conformant problem
  - Agent may have no idea where it is; solution (if any) is a sequence
- Nondeterministic and/or partially observable =⇒ contingency problem
  - percepts provide new information about current state
  - solution is a contingent plan or a policy
  - often interleave search, execution
- Unknown state space =⇒ exploration problem ("online")

Example: Buckets

Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.

- **Formulate goal:** Have 7 liters of water

  in 9-liter bucket

- **Formulate problem:**
  - States: amount of water in the buckets
  - Operators: Fill bucket from source, empty bucket

- **Find solution:** sequence of operators that bring you from current state to the goal state

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

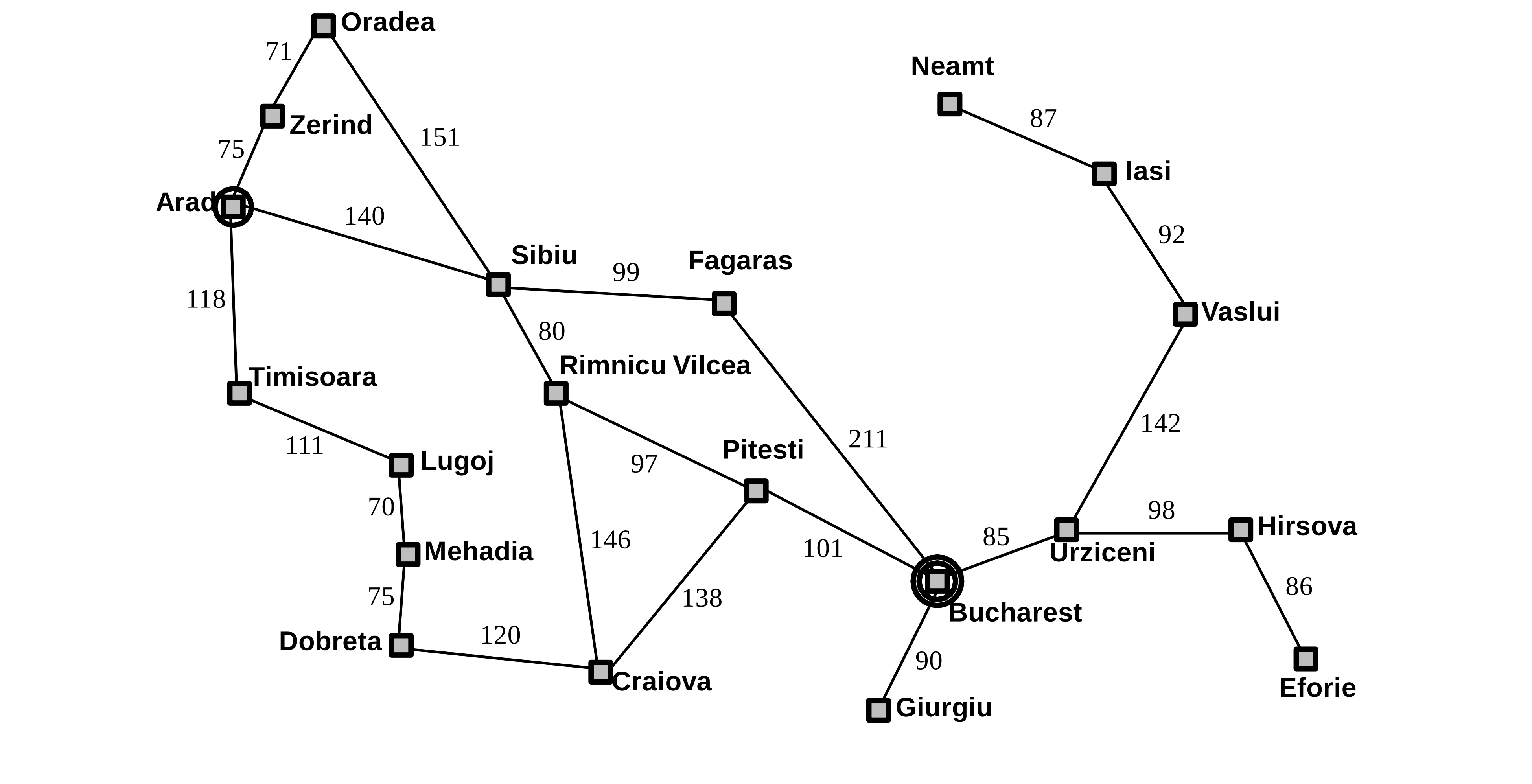
Formulate problem:

states: various cities actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Remember: Environment types

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Operating System	Yes	Yes	No	No	Yes
Virtual Reality	Yes	Yes	Yes/No	No	Yes/No
Office Environment	No	No	No	No	No
Mars	No	Semi	No	Semi	No

- The environment types largely determine the agent design.
- More sensors, more mind

Problem types

- **Single-state problem:** deterministic, accessible
  - *Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.*
- **Multiple-state problem:** deterministic, inaccessible
  - Agent must reason about sequences of actions and states assumed while working towards goal state.
- **Contingency problem:** nondeterministic, inaccessible
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution
- **Exploration problem:** unknown state space
  - Discover and learn about environment while taking actions.

Problem types

- **Single-state problem:** deterministic, accessible
  - Agent knows everything about world (the exact state),
  - Can calculate optimal action sequence to reach goal state.
  - E.g., playing chess. Any action will result in an exact state

WE ARE HUMBER

Problem types

- **Multiple-state problem:** deterministic, inaccessible

  - Agent does not know the exact state (could be in any of the possible states)
    - May not have sensor at all
  - Assume states while working towards goal state.

  - E.g., walking in a dark room
    - If you are at the door, going straight will lead you to the kitchen
    - If you are at the kitchen, turning left leads you to the bedroom
    - …

Problem types

- **Contingency problem:** nondeterministic, inaccessible
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution
  - E.g., a new skater in an arena
    - Sliding problem.
    - Many skaters around

Problem types

- **Exploration problem:** unknown state space

  Discover and learn about environment while taking actions.

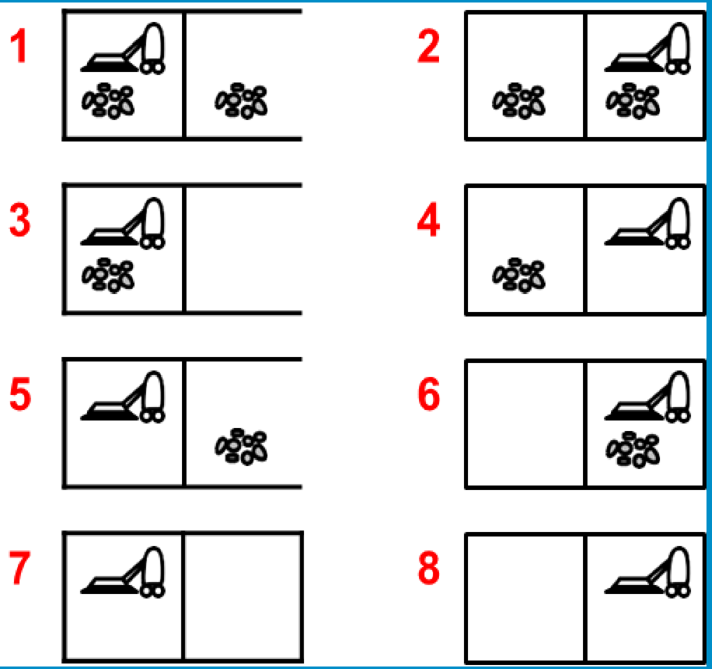
  - *E.g., Maze*

Example: vacuum world

Single-state, start in #5. Solution??
[$Right, Suck$]

Conformant, 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, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
[$Right$, if $dirt$ then $Suck$]



WE ARE HUMBER

Single-state problem formulation

- A problem is defined by four items:
- initial state e.g., "at Arad"
- successor function $S(x)$ = set of action–state pairs
  - e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$
- goal test, can be
  - explicit, e.g., x = "at Bucharest"
  - implicit, e.g., $NoDirt(x)$
- path cost (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - $c(x, a, y)$ is the step cost, assumed to be ≥ 0
- A solution is a sequence of actions leading from the initial state to a goal state

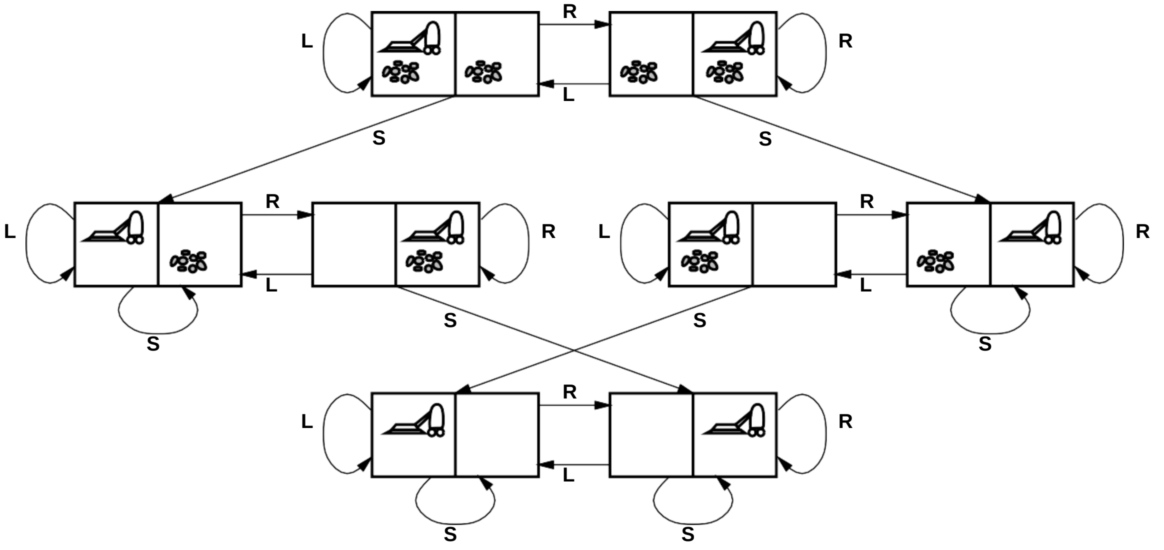
Selecting a state space

- 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.
- 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
- Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??: 1 per action (0 for *NoOp*)

Pearson

WE ARE HUMBER

Example: The 8-puzzle

7	2	4
5		6
8	3	1

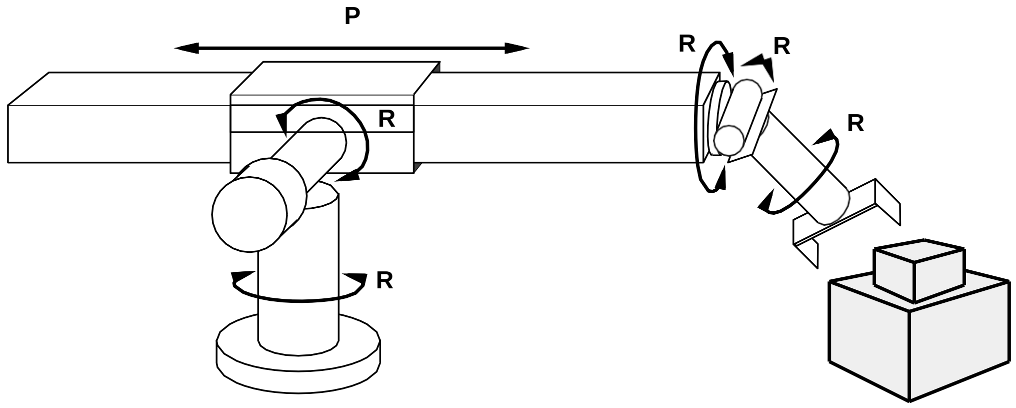
Start State

1	2	3
4	5	6
7	8	

Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

Example: robotic assembly



states??: real-valued coordinates of robot joint angles
parts of the object to be assembled

actions??: continuous motions of robot joints

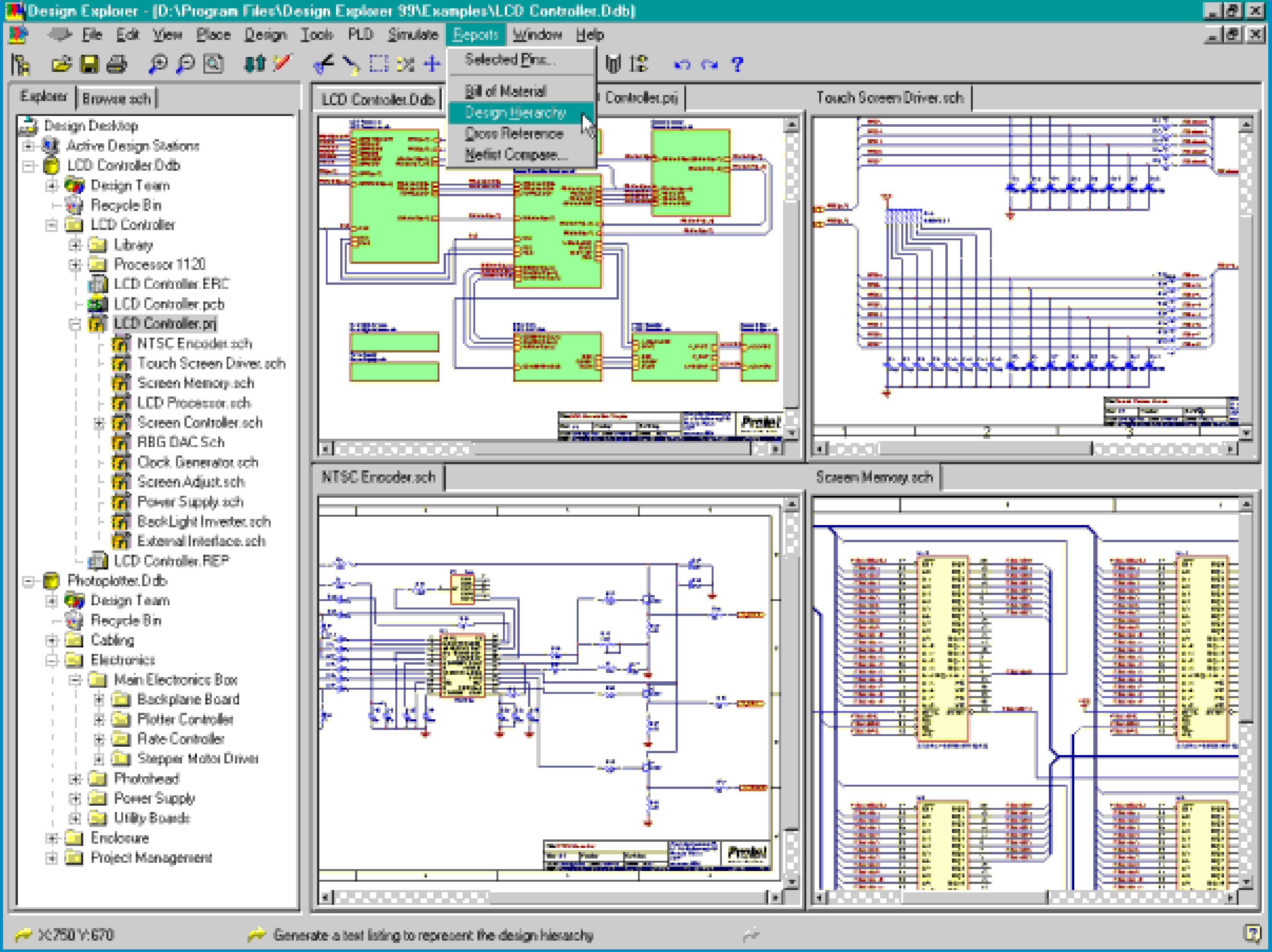
goal test??: complete assembly!

path cost??: time to execute

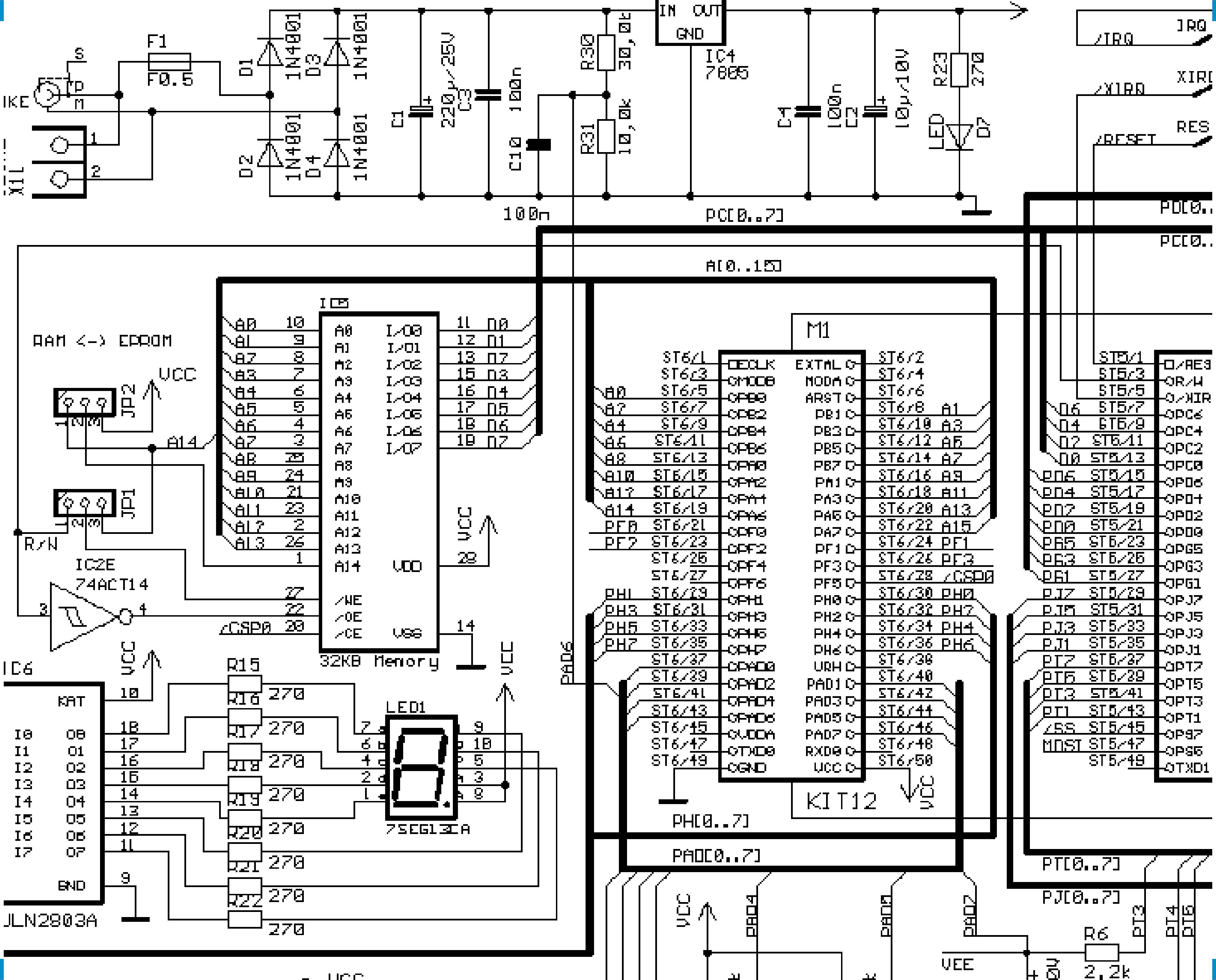
Real-life example: VLSI Layout

- 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 vias (connections from one layer to another)
  - minimize length of some signal lines (e.g., clock line)
  - distribute heat throughout board
  - etc.

Enter schematics;
do not worry about placement & wire crossing

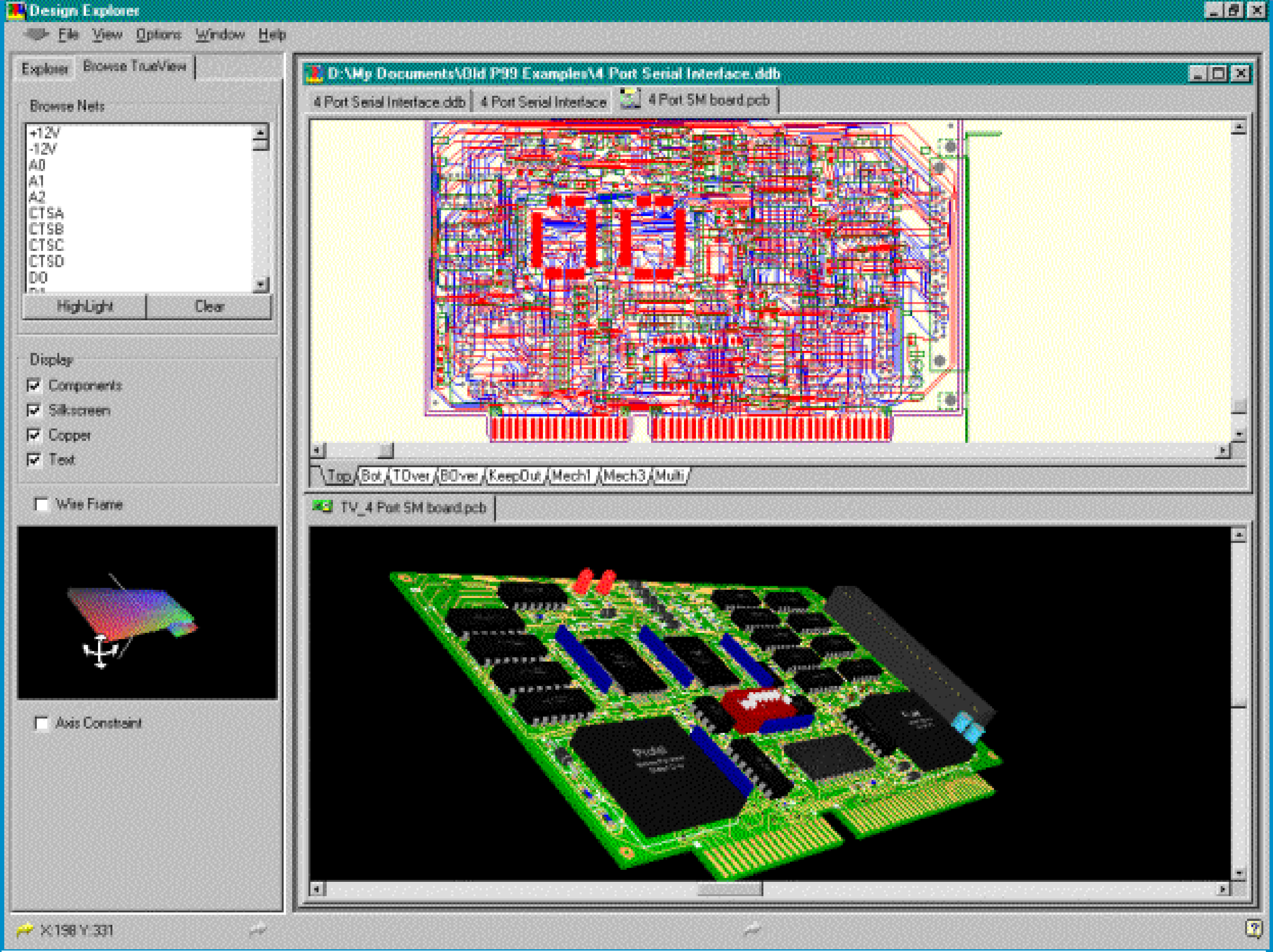


Protel's hierarchical schematic design features let you take a "bottom up" or "top down" approach, depending on your preferred methodology. Protel can automatically generate sub-sheets based on higher-level sheet symbols, or create sheet symbols based on existing sheets.

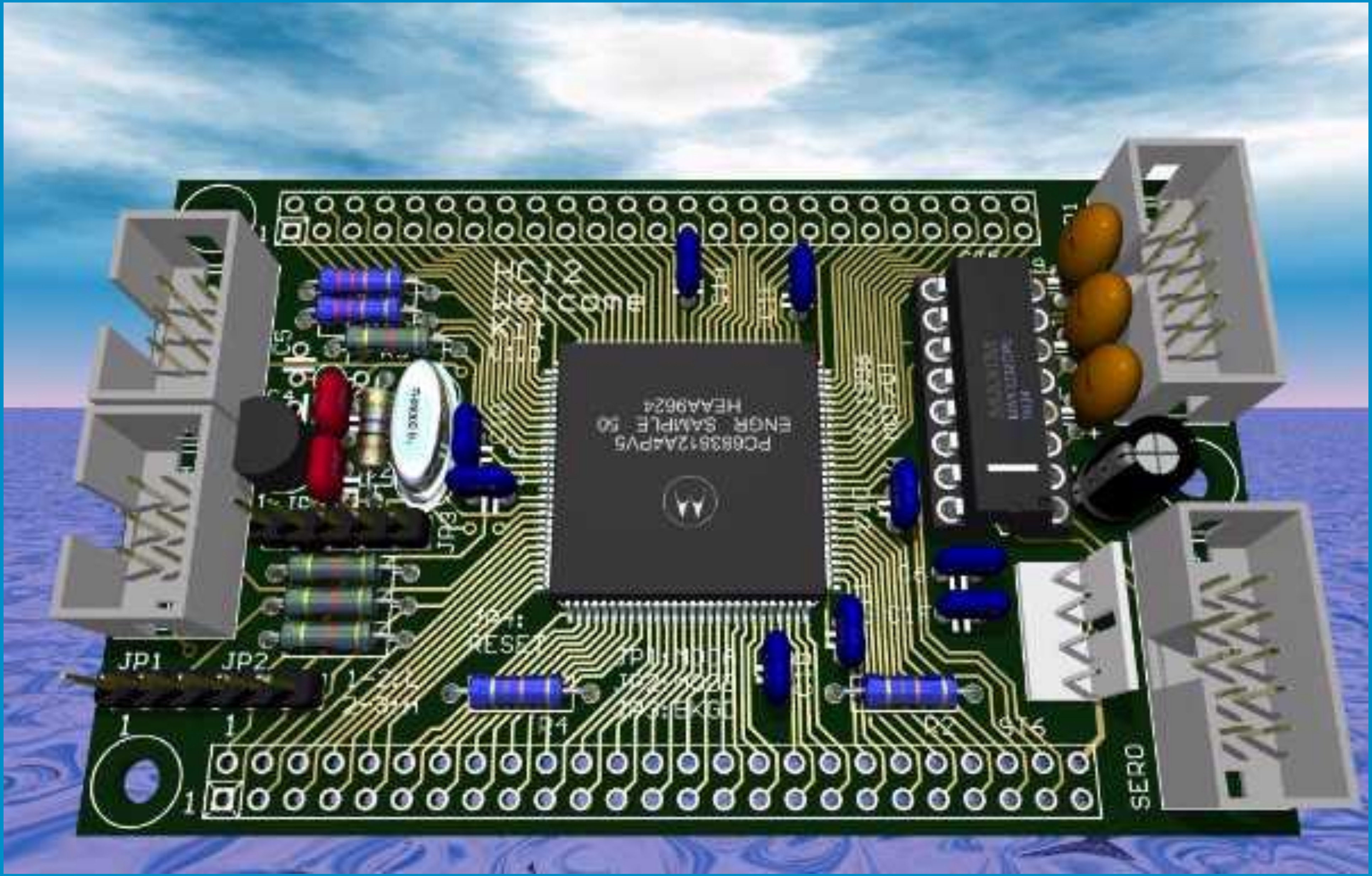


WE ARE HUMBER

Use automated tools to place components and route wiring.



Protel 99 SE's unique 3D visualization feature lets you see your finished board before it leaves your desktop. Sophisticated 3D modeling and extrusion techniques render your board in stunning 3D without the need for additional height information. Rotate and zoom to examine every aspect of your board.


HC12
Welcome
JP1
JP2
SER0

Search algorithms

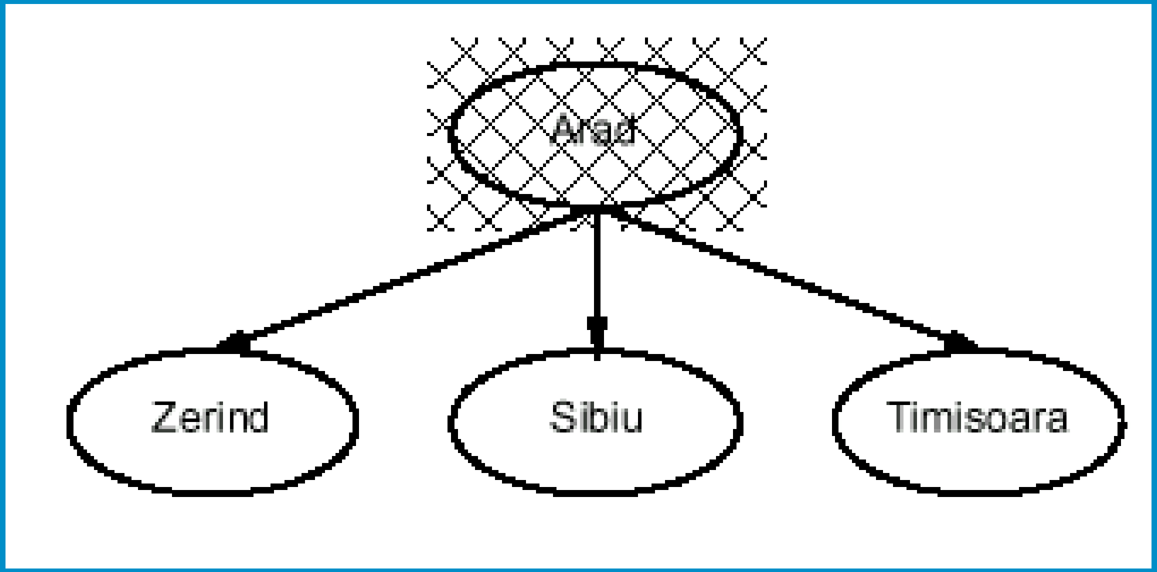
Basic idea:

- offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

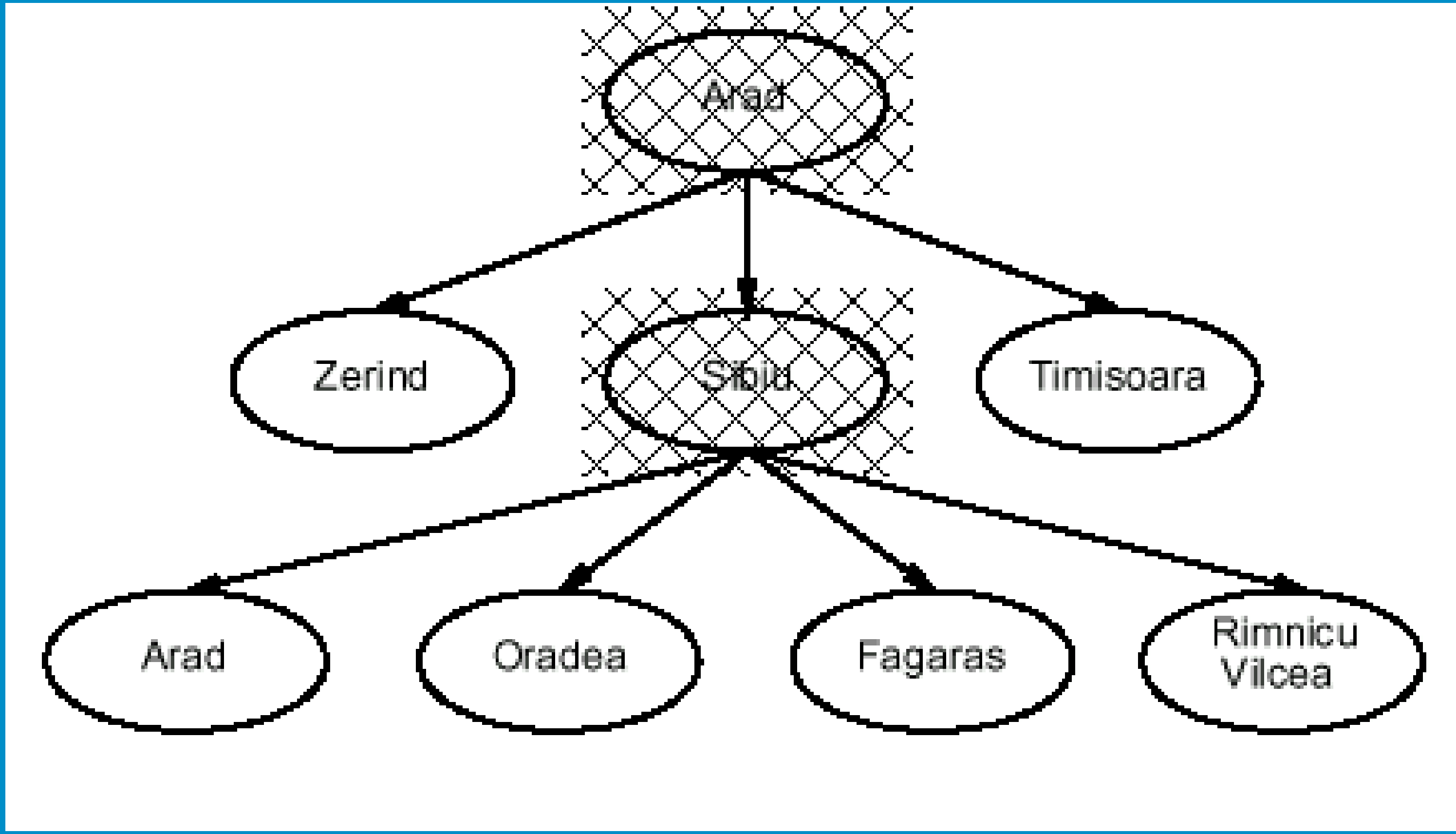
Function General-Search(*problem*, *strategy*) returns a *solution*, or failure
    initialize the search tree using the initial state problem
    loop do
        if there are no candidates for expansion **then return** failure
        choose a leaf node for expansion according to strategy
        if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

WE ARE HUMBER

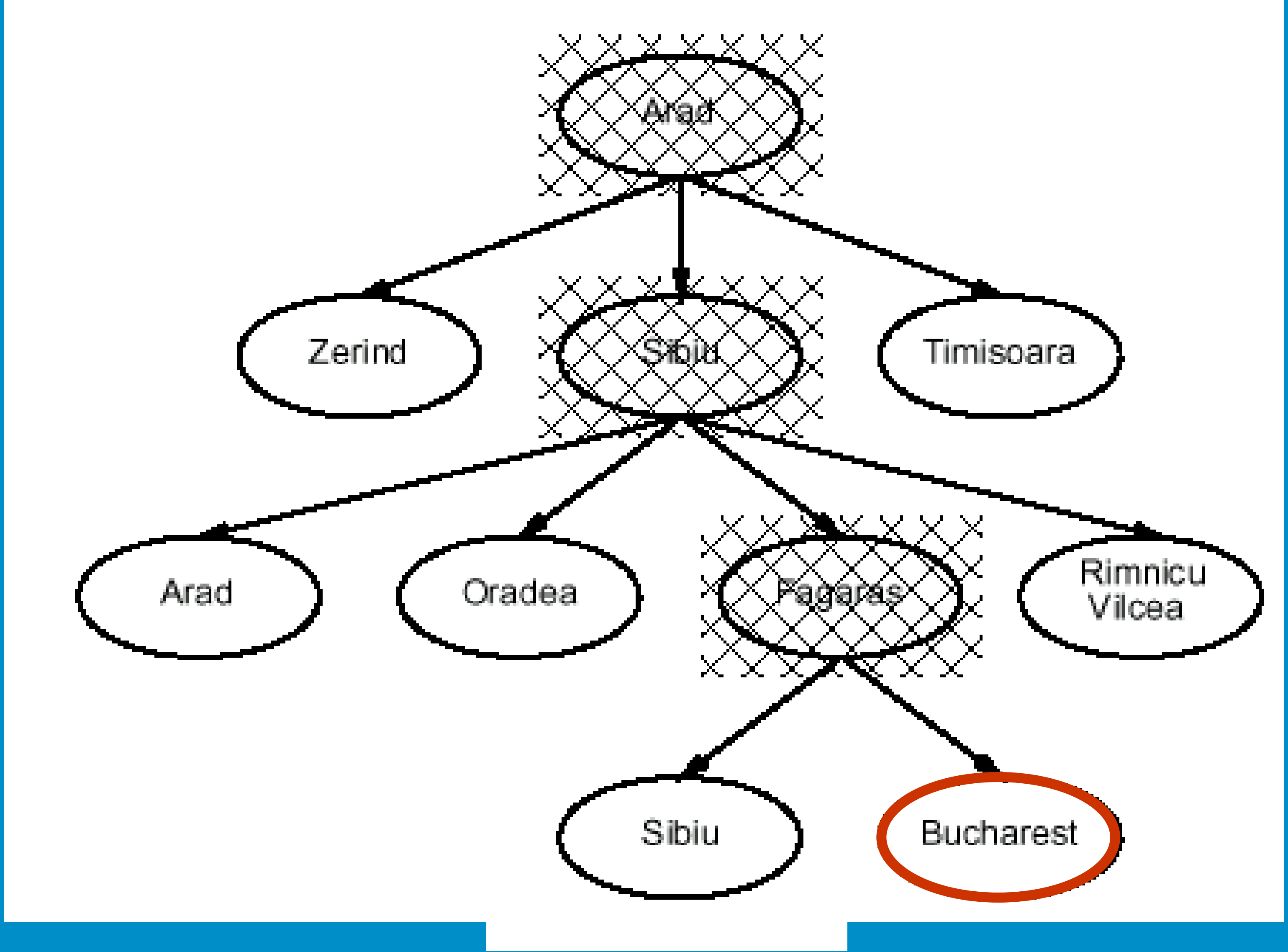
General search example



General search example



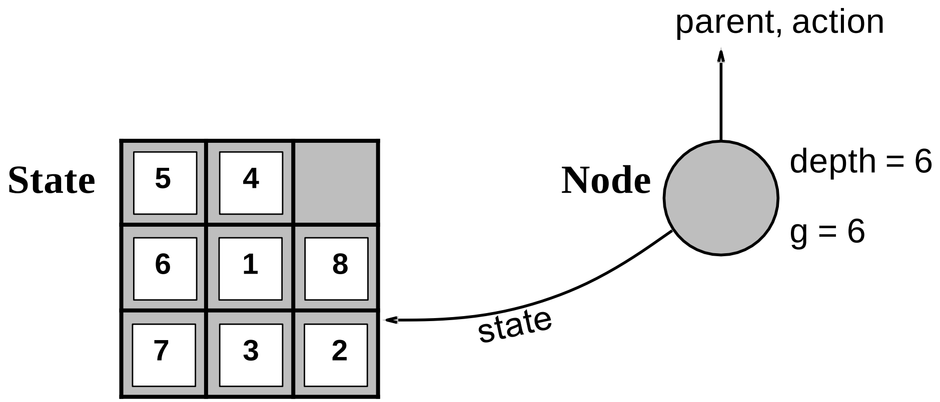
General search example



WE ARE HUMBER

Implementation: states vs. nodes

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

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 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
```


Pearson

WE ARE HUMBER

Search strategies

- A 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?
- Time and space complexity are measured in terms of
  - b—maximum branching factor of the search tree (maximum number of successors of any node)
  - d—depth of the least-cost solution
  - m—maximum depth of the state space (may be ∞)

Pearson

WE ARE HUMBER

Uninformed search strategies

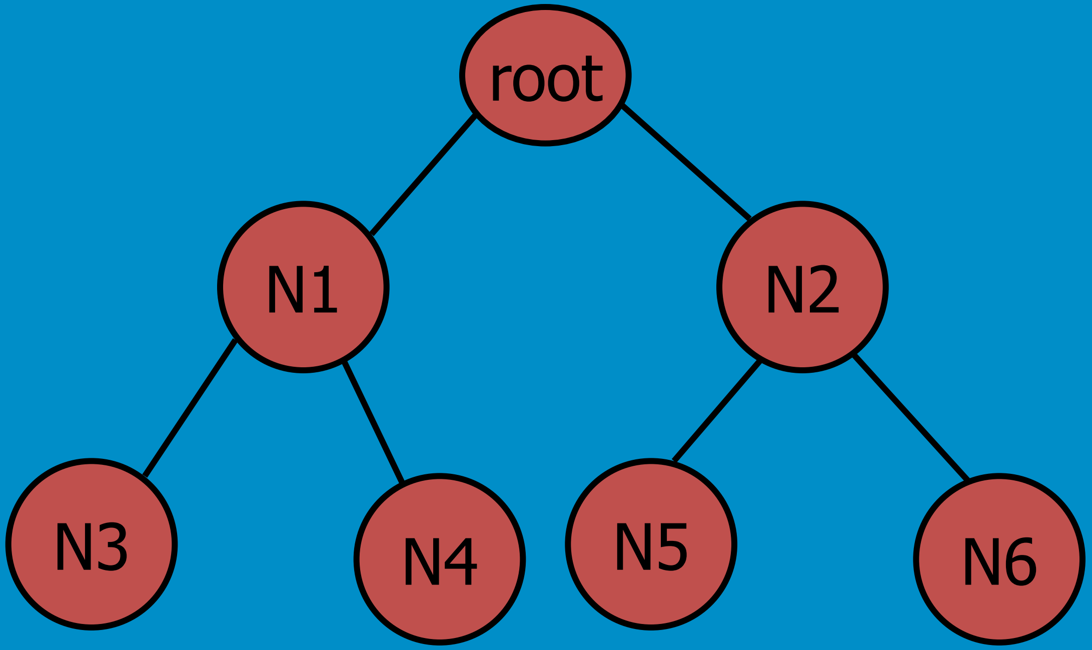
- Uninformed strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Binary Tree Example

Depth = 0

Depth = 1

Depth = 2

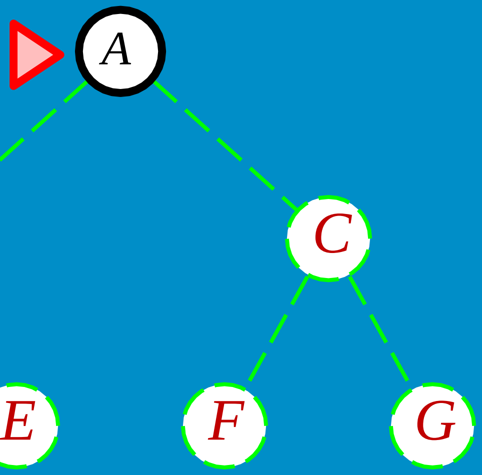


Number of nodes: $n = 2^{\text{max depth}}$
Number of levels (max depth) = log(n) (could be n)

Breadth-first search

Expand shallowest unexpanded node

Implementation:
fringe is a FIFO queue, i.e., new successors go at end

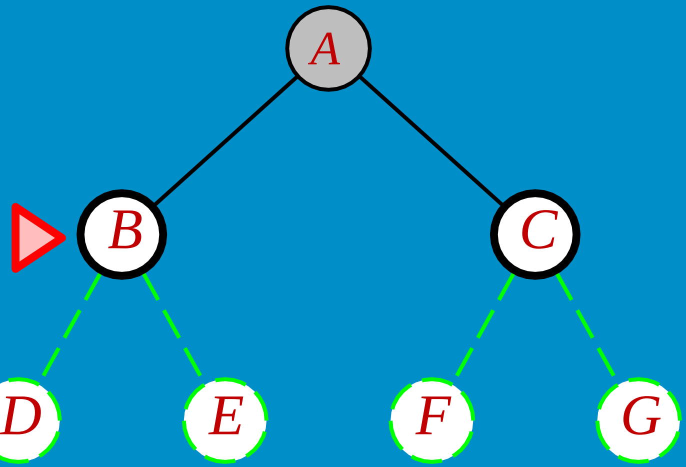


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

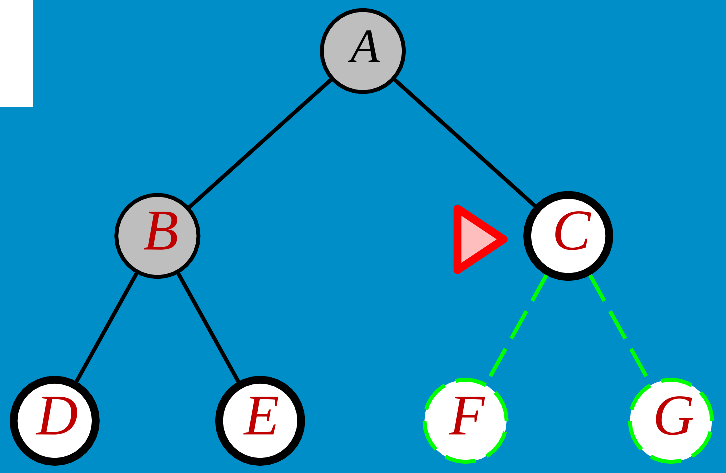


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



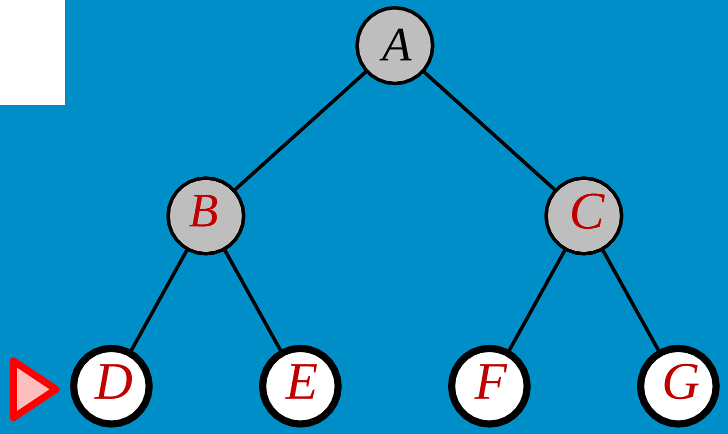
WE ARE HUMBER

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



A B C D E F G

Properties of breadth-first search

- Complete?? Yes (if b is finite)
- Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d
- Space?? $O(b^{d+1})$ (keeps every node in memory)
- Optimal?? Yes (if cost = 1 per step); not optimal in general

- Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation**: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

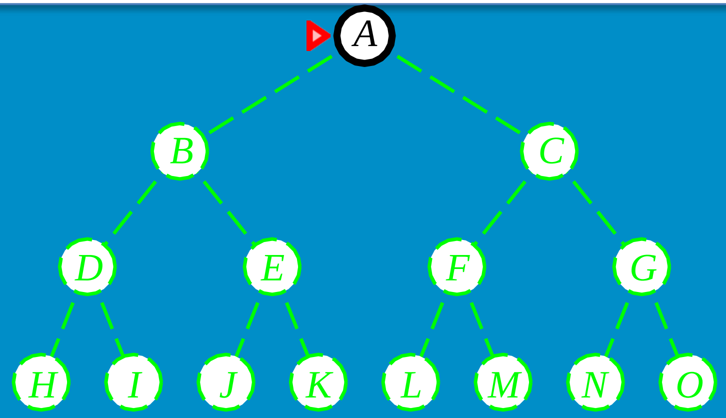
Pearson

WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

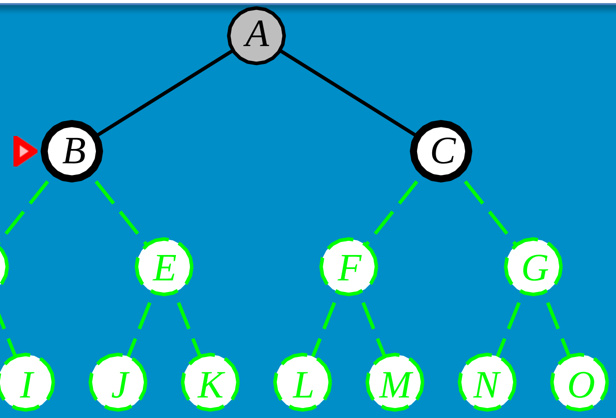


WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

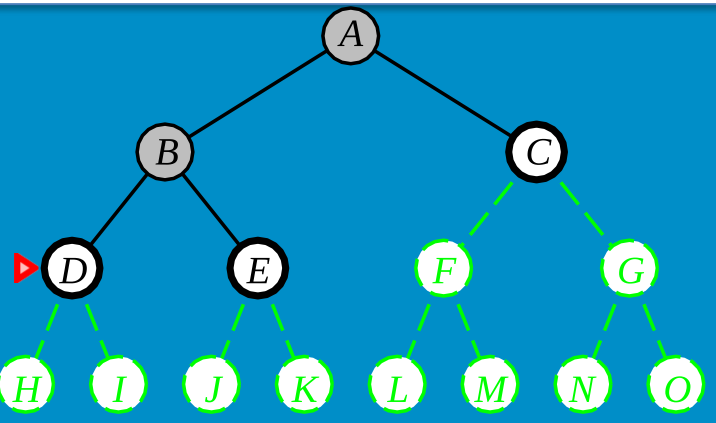


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Pearson

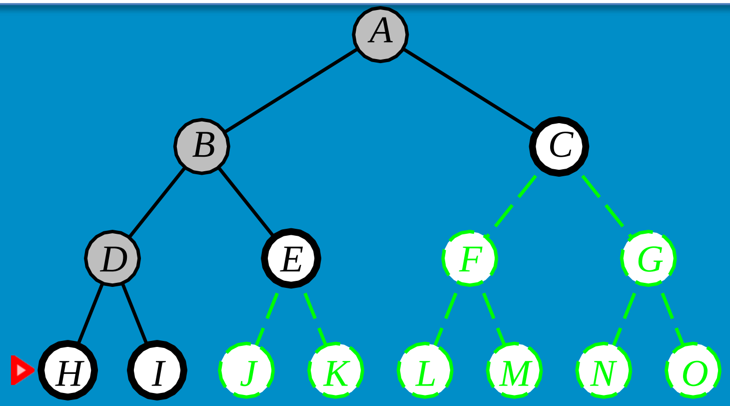
WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

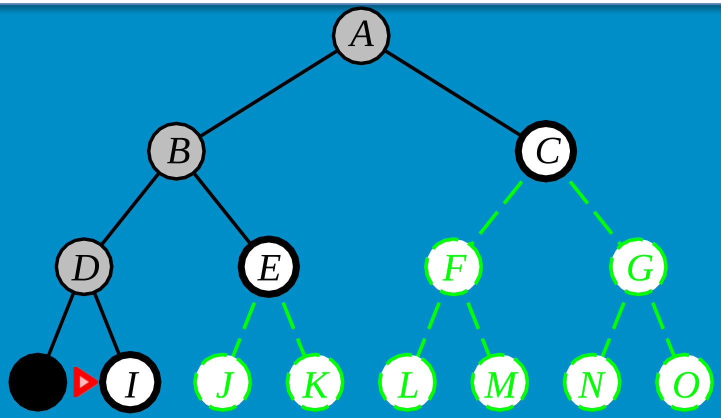


Depth-first search

Expand deepest unexpanded node

Implementation:

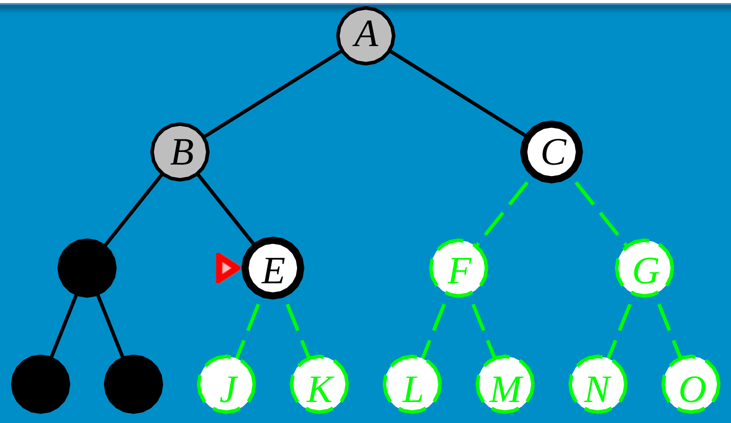
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front



Pearson

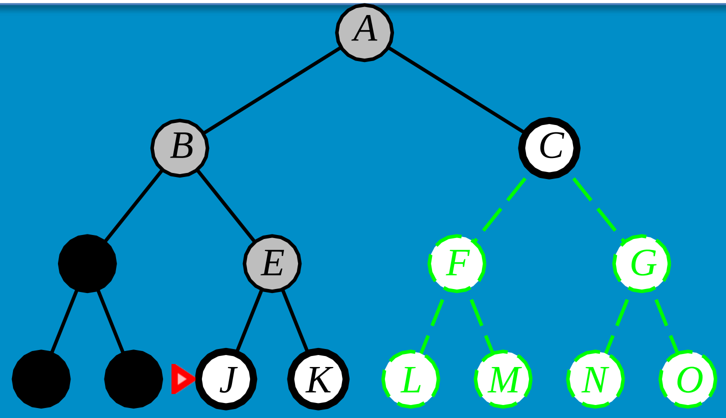
WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

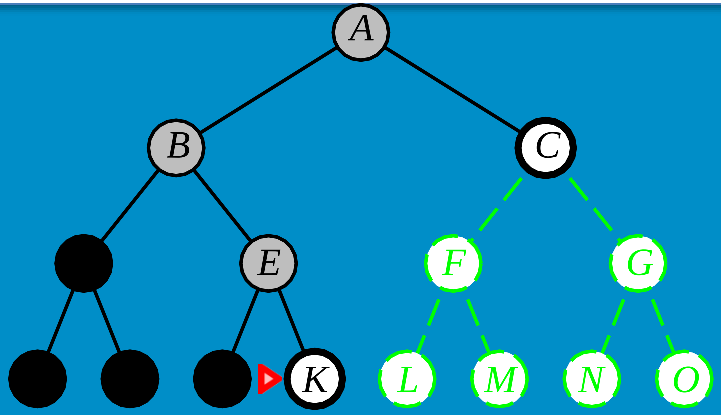


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



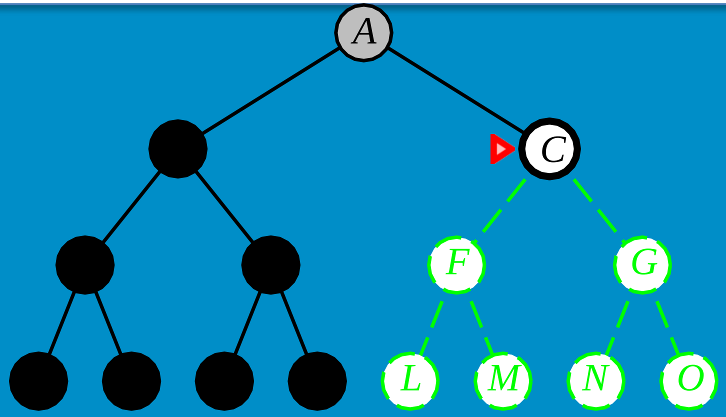
WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

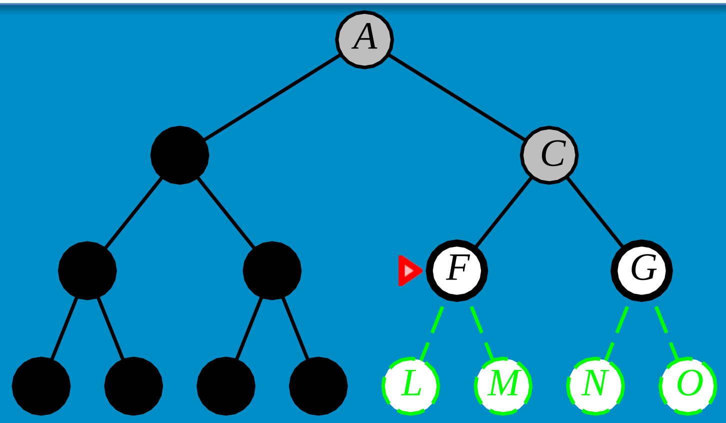


WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

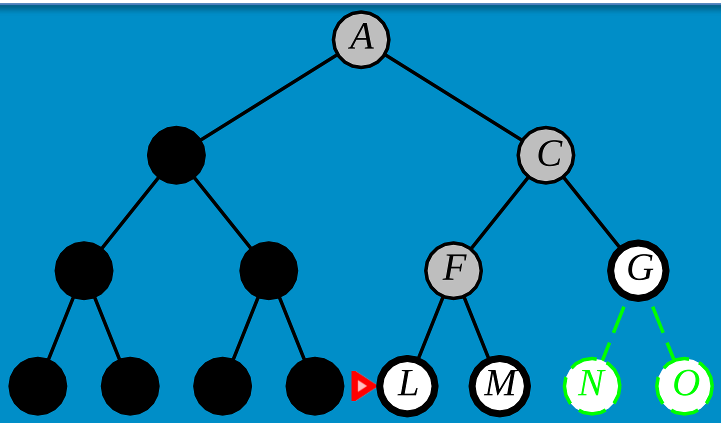


WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

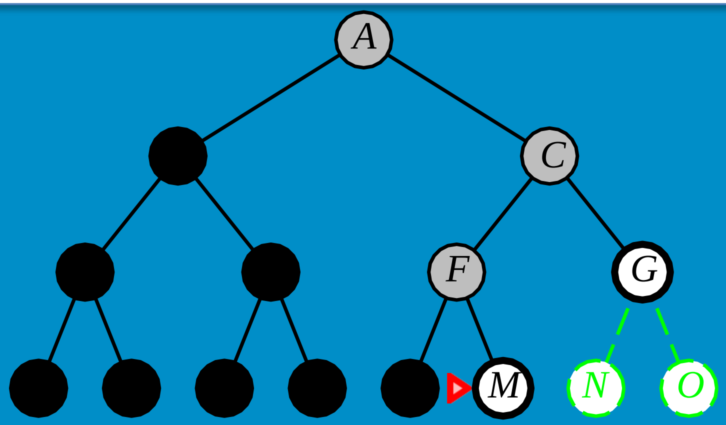


WE ARE HUMBER

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front



A B D H I J K C F L M G N O

WE ARE HUMBER

Depth-first search, example 2

- **Preorder** DFS variation(Root -> Left -> Right)
- **Order** of nodes visited:

A -> B -> D -> G -> H -> E -> I -> J -> C -> F -> K

Depth-first search, example 2

- **Inorder** DFS variation(Left -> Root -> Right)
- **Order** of nodes visited:

G -> D -> H -> B -> I -> E -> J -> A -> C -> K -> F

WE ARE HUMBER

Depth-first search, example 2

- **Postorder** DFS variation(Left -> Right -> Root)
- **Order** of nodes visited:

G -> H -> D -> I -> J -> E -> B -> K -> F -> C -> A

Properties of depth-first search

- Complete?? No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path
  - $\Rightarrow$ complete in finite spaces
- Time?? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space?? $O(bm)$, i.e., linear space!
- Optimal?? No

Summary of Uninformed Search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Pearson

WE ARE HUMBER

Informed (Heuristic) Search Strategies

- Uses problem-specific knowledge beyond the definition of the problem itself
- Can find solutions more efficiently than can an uninformed strategy

Best-first search

- Idea: use an evaluation function (heuristic) for each node
  - estimate of "desirability"

- ⇒ Expand most desirable unexpanded node

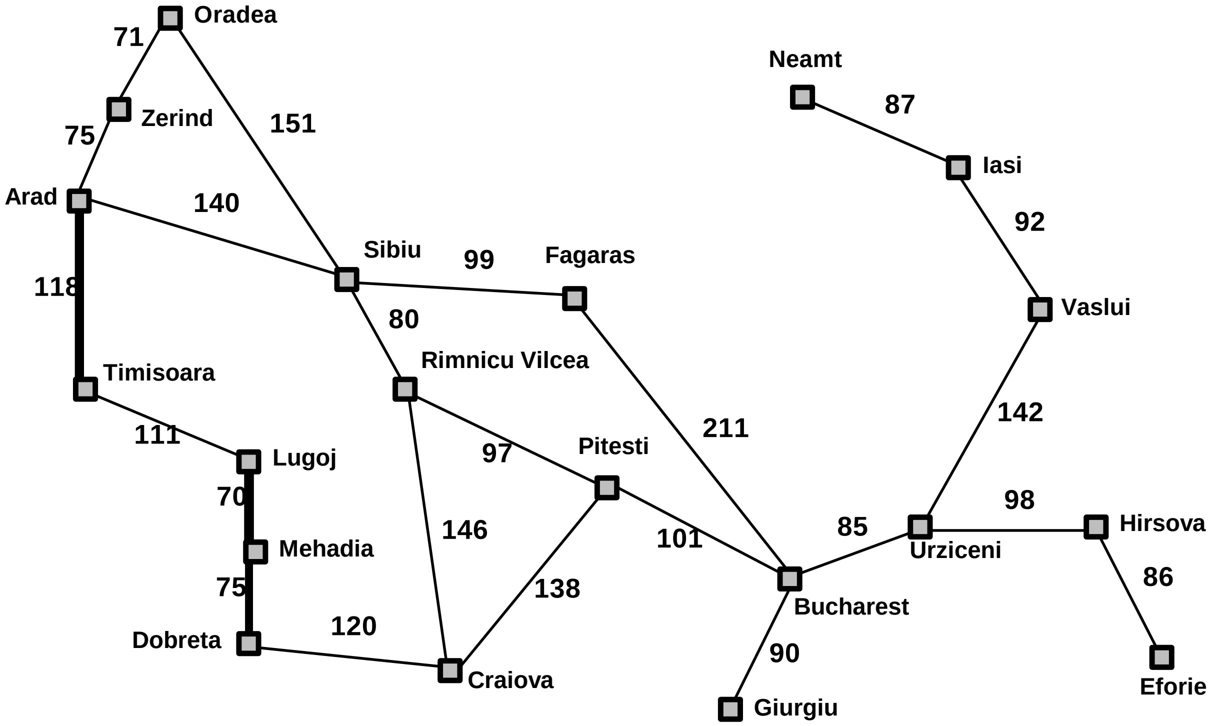
Implementation:

- *fringe* is a queue sorted in decreasing order of desirability

- Special cases:
  - greedy search
  - A* search

WE ARE HUMBER

Romania with step costs in km



Straight−line distance
to Bucharest

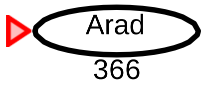
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

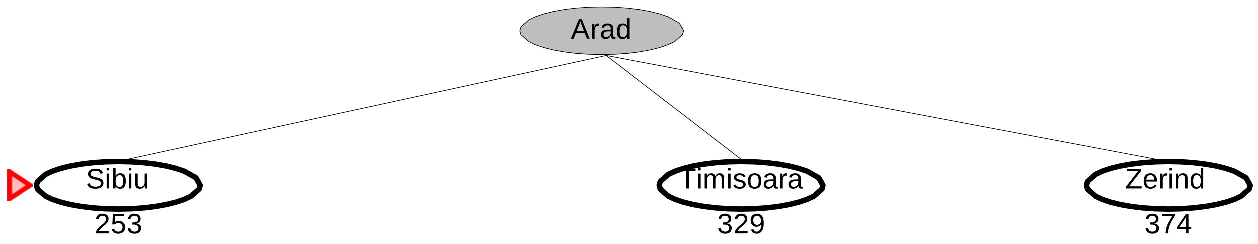
- Evaluation function $h(n)$ (heuristic)
  - $=$ estimate of cost from n to the closest goal
- E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest
- Greedy search expands the node that $\mathrm{appears}$ to be closest to goal

WE ARE HUMBER

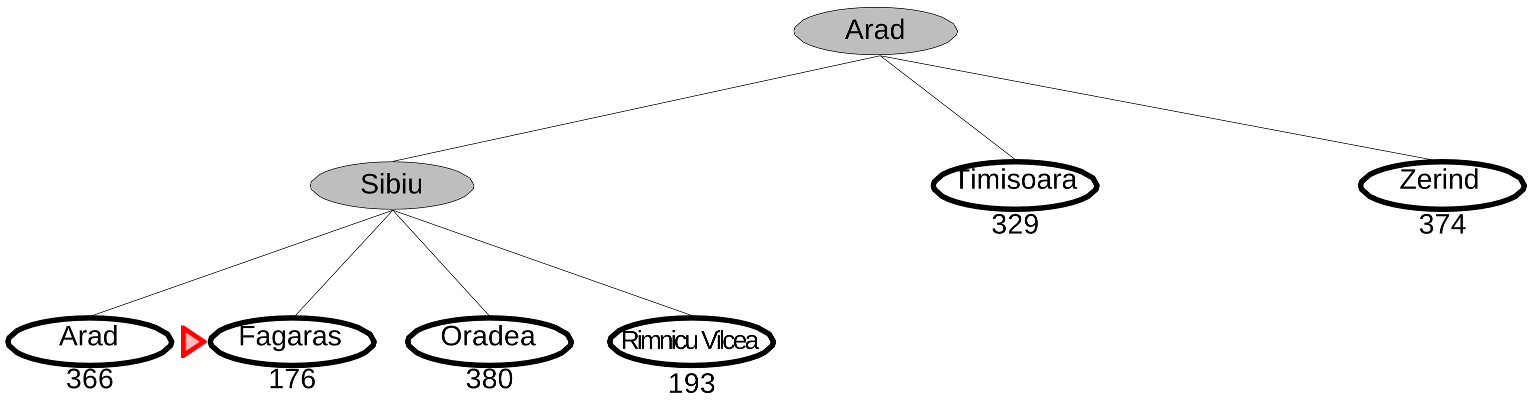
Greedy search example



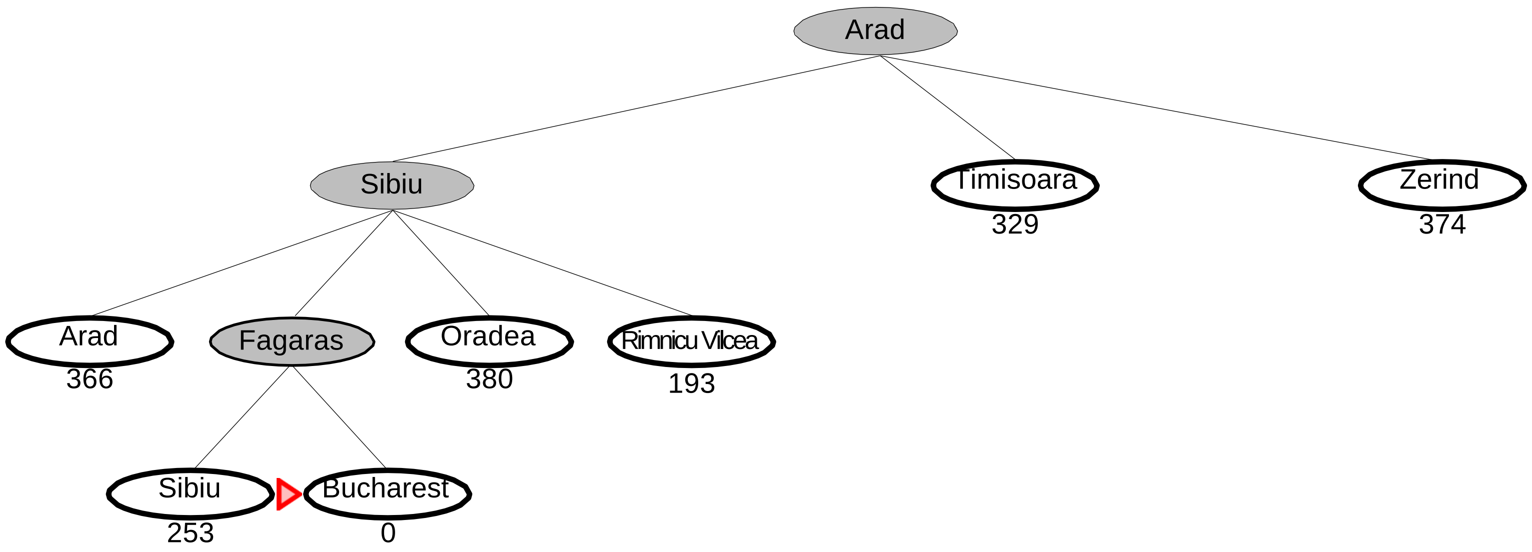
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

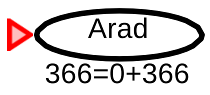
- Complete?? No–can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
  - Complete in finite space with repeated-state checking
- Time?? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?? $O(b^m)$—keeps all nodes in memory
- Optimal?? No

WE ARE HUMBER

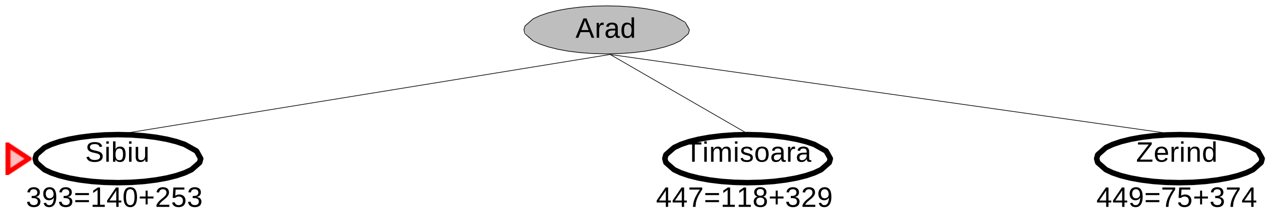
A* Search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost to goal from n
  - $f(n)$ = estimated total cost of path through n to goal
- A$*$ search uses an admissible heuristic
  - i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n. (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)
- E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance
- Theorem: A$*$ search is optimal

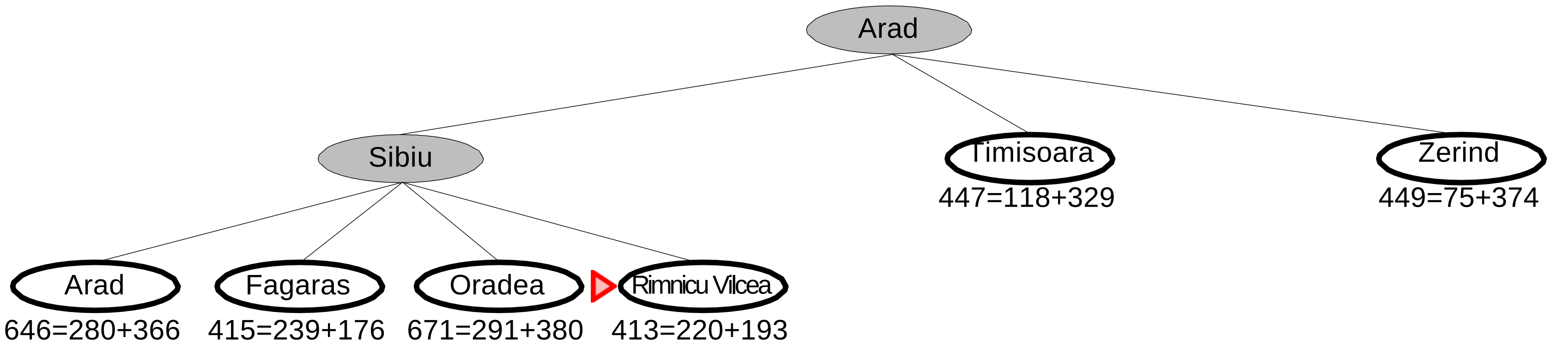
A* Search Example



A* Search Example

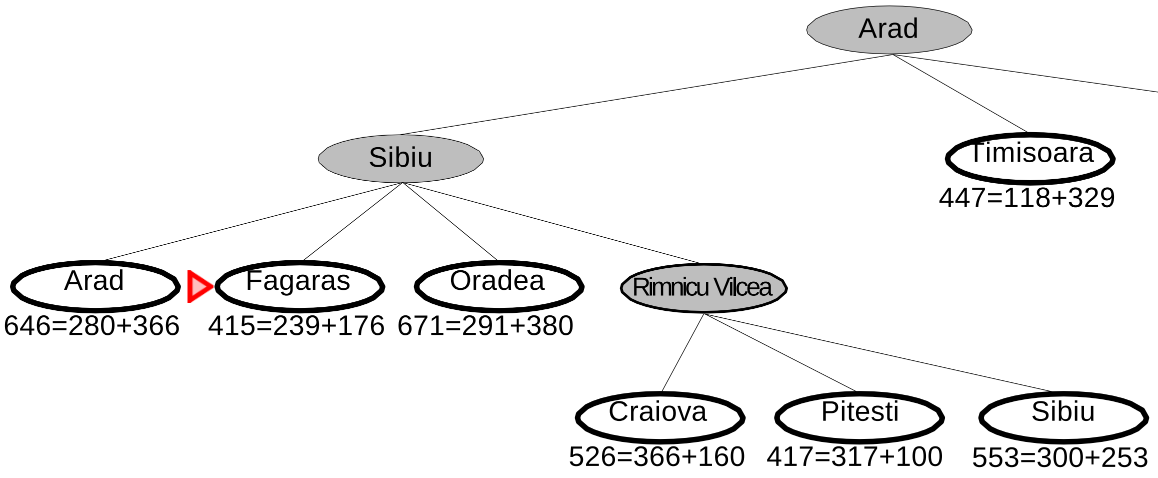


A* Search Example



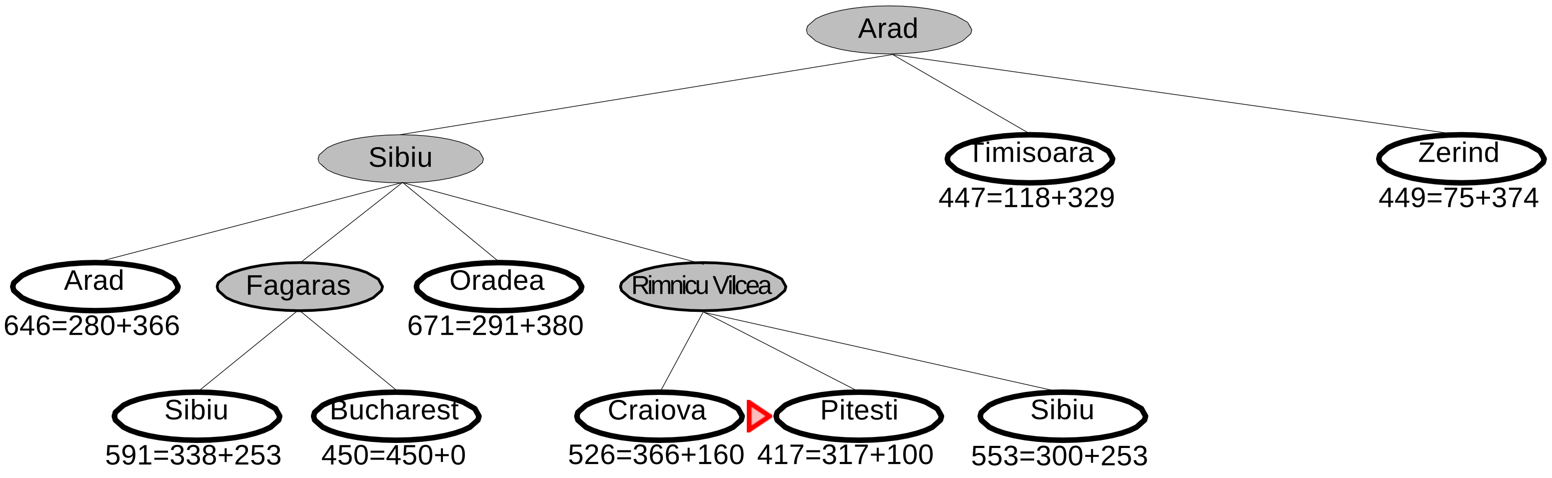
WE ARE HUMBER

A* Search Example



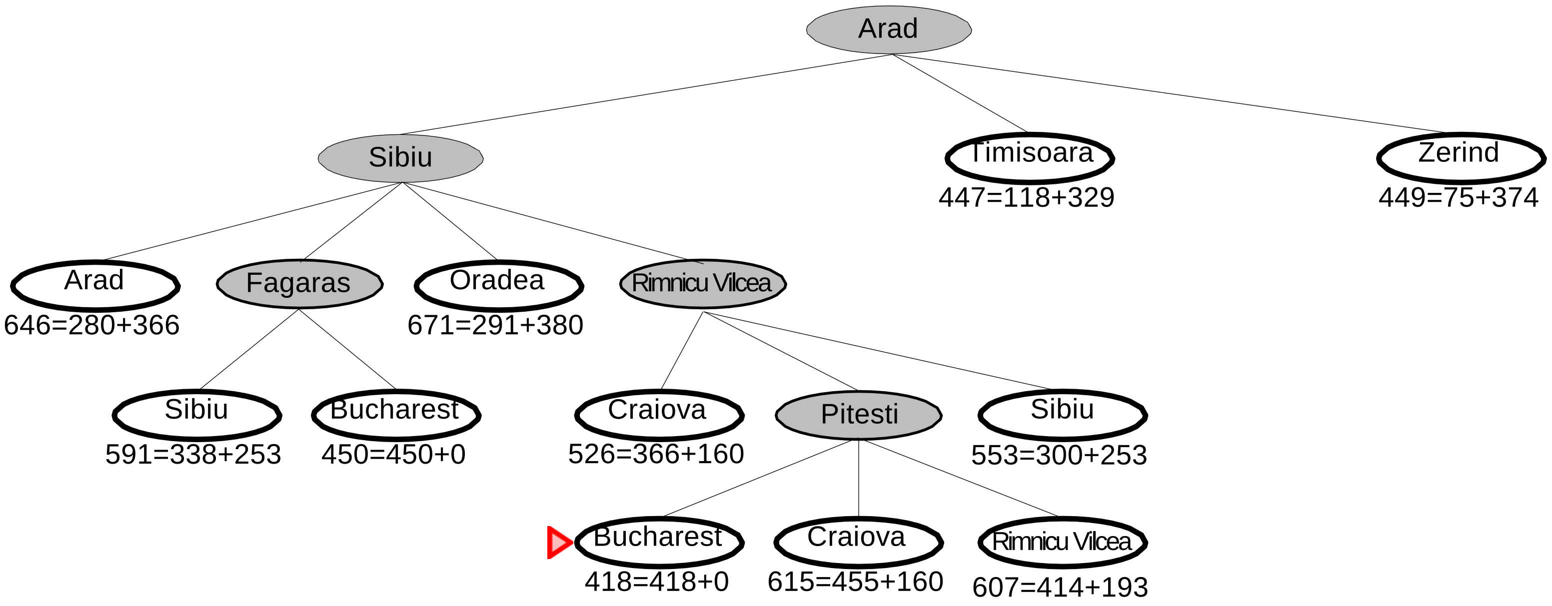
WE ARE HUMBER

A* Search Example



WE ARE HUMBER

A* Search Example



Properties of A*

- Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time?? Exponential in [relative error in $h \times$ length of soln.]
- Space?? Keeps all nodes in memory
- Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Summary

- A problem consists of five parts: the initial state, a set of actions, a transition model describing the results of those actions, a set of goal states, and an action cost function.
- Uninformed search methods have access only to the problem definition. Algorithms build a search tree in an attempt to find a solution.
- Informed search methods have access to a heuristic function $h(n)$ that estimates the cost of a solution from n.