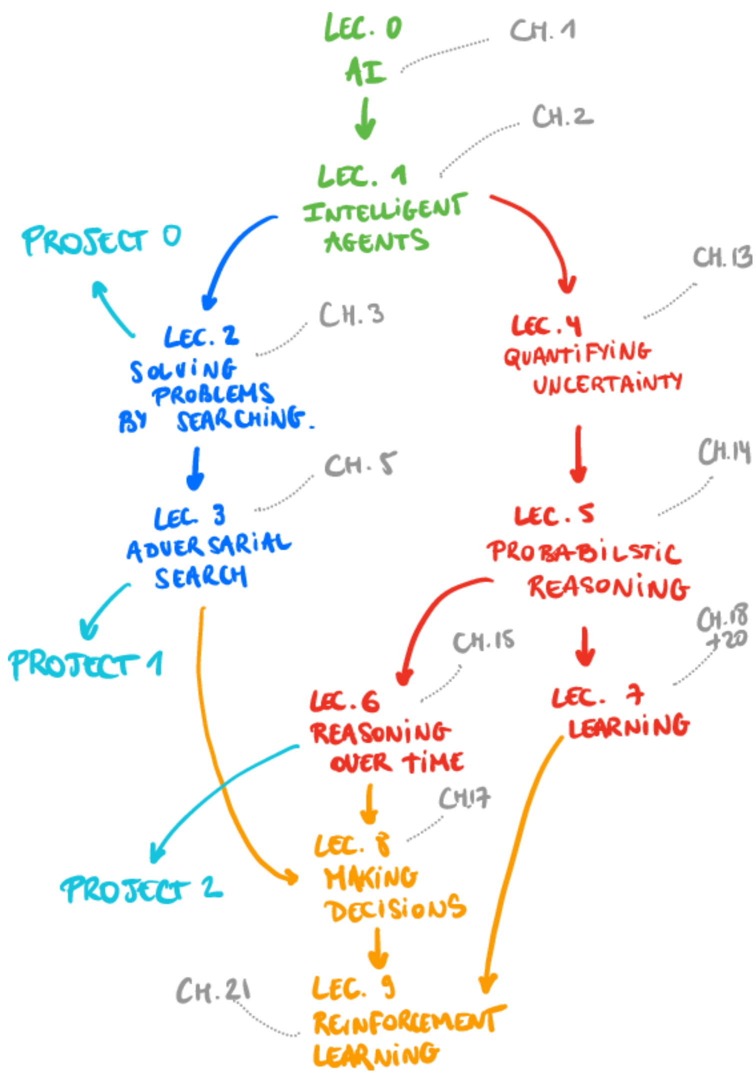
Introduction to Artificial Intelligence

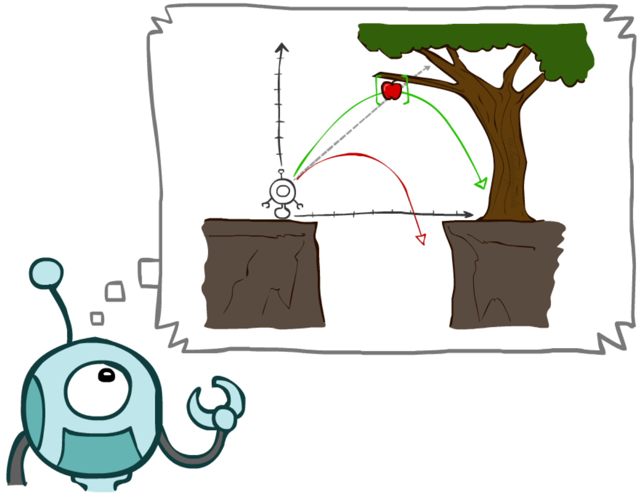
Lecture 2: Solving problems by searching

Prof. Gilles Louppe
g.louppe@uliege.be


LEC. 0
AI
CH. 1
CH. 2
LEC. 1
INTELLIGENT AGENTS
PROJECT 0
LEC. 2
SOLVING PROBLEMS BY SEARCHING.
CH. 3
LEC. 3
ADVERSARIAL SEARCH
CH. 5
PROJECT 1
CH. 13
LEC. 4
QUANTIFYING UNCERTAINTY
CH. 14
LEC. 5
PROBABILSTIC REASONING
CH. 15
CH. 18 +20
LEC. 6
REASONING OVER TIME
LEC. 7
LEARNING
CH. 17
LEC. 8
MAKING DECISIONS
PROJECT 2
CH. 21
LEC. 9
REINFORCEMENT LEARNING

Today

- Planning agents
- Search problems
- Uninformed search methods
  - Depth-first search
  - Breadth-first search
  - Uniform-cost search
- Informed search methods
  - A*
  - Heuristics



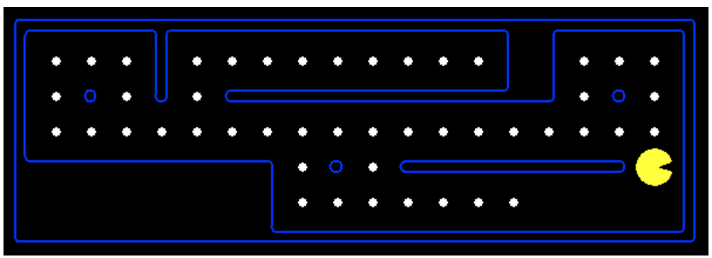
—
Image credits: CS188, UC Berkeley.

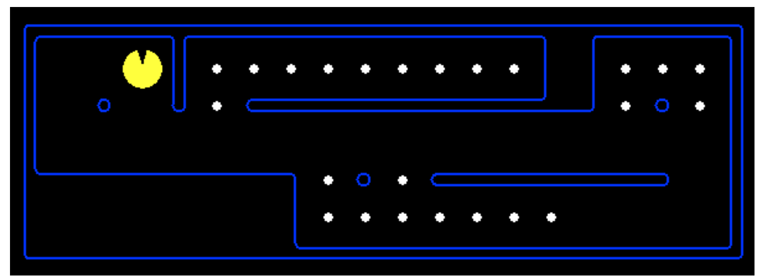
Planning agents

Reflex agents

Reflex agents

- select actions on the basis of the current percept;
- may have a model of the world current state;
- do not consider the future consequences of their actions;
- consider only how the world is now.





For example, a simple reflex agent based on condition-action rules could move to a dot if there is one in its neighborhood. No planning is involved to take this decision.

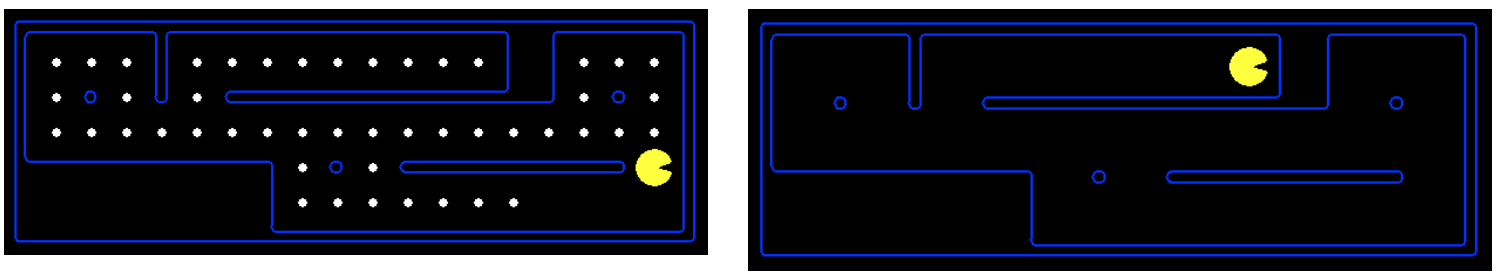
Problem-solving agents

Assumptions:

- Single-agent, observable, deterministic and known environment.

Problem-solving agents

- take decisions based on (hypothesized) consequences of actions, by considering how the world could be;
- must have a model of how the world evolves in response to actions;
- formulate a goal, explicitly.



A planning agent looks for sequences of actions to eat all the dots.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Offline vs. Online solving

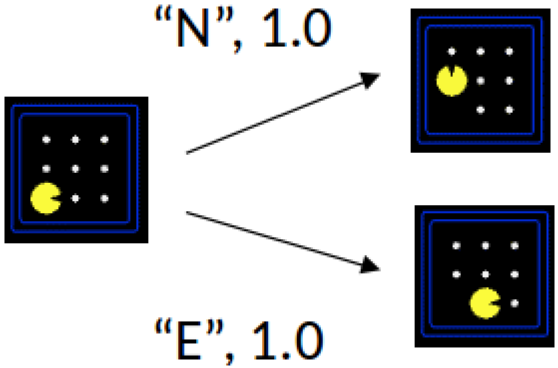
- Problem-solving agents are offline. The solution is executed "eyes closed", ignoring the percepts.
- Online problem solving involves acting without complete knowledge. In this case, the sequence of actions might be recomputed at each step.

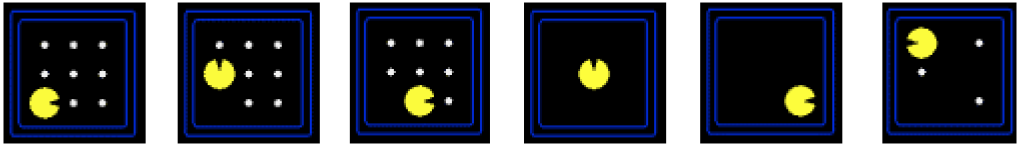
Search problems

Search problems

A search problem consists of the following components:

- A representation of the states of the agent and its environment.
- The initial state of the agent.
- A description of the actions available to the agent given a state s, denoted $\text{actions}(s)$.
- A transition model that returns the state $s' = \text{result}(s, a)$ that results from doing action a in state s.
    - We say that s' is a successor of s if there is an acceptable action from s to s'.





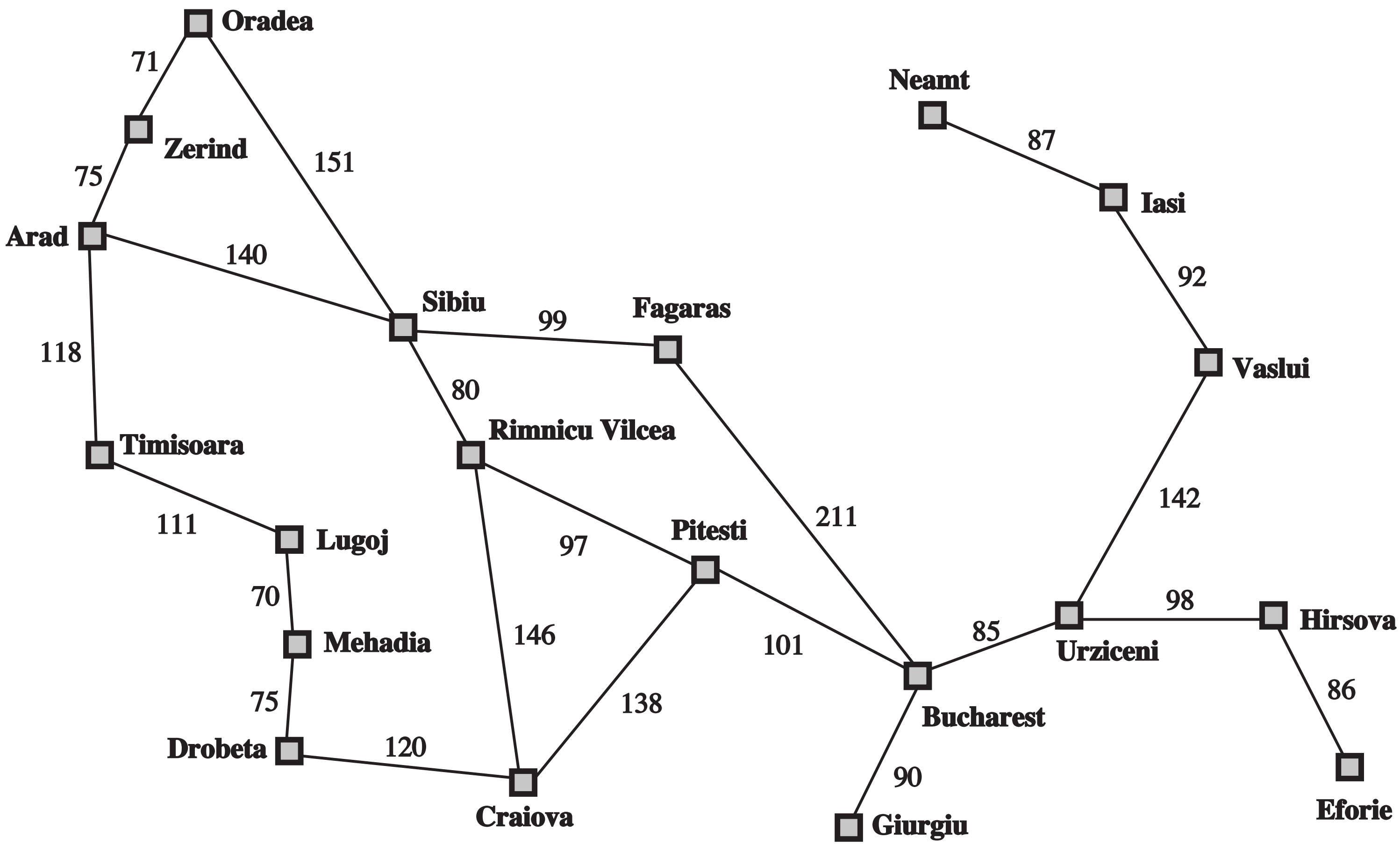
- Together, the initial state, the actions and the transition model define the state space of the problem, i.e. the set of all states reachable from the initial state by any sequence of action.
  - The state space forms a directed graph:
    - nodes = states
    - links = actions
  - A path is a sequence of states connected by actions.
- A goal test which determines whether the solution of the problem is achieved in state s.
- A path cost that assigns a numeric value to each path.
  - In this course, we will also assume that the path cost corresponds to a sum of positive step costs $c(s,a,s')$ associated to the action a in s leading to s'.

A solution to a problem is an action sequence that leads from the initial state to a goal state.

- A solution quality is measured by the path cost function.
- An optimal solution has the lowest path cost among all solutions.

Exercise
What if the environment is partially observable? non-deterministic?

Example: Traveling in Romania



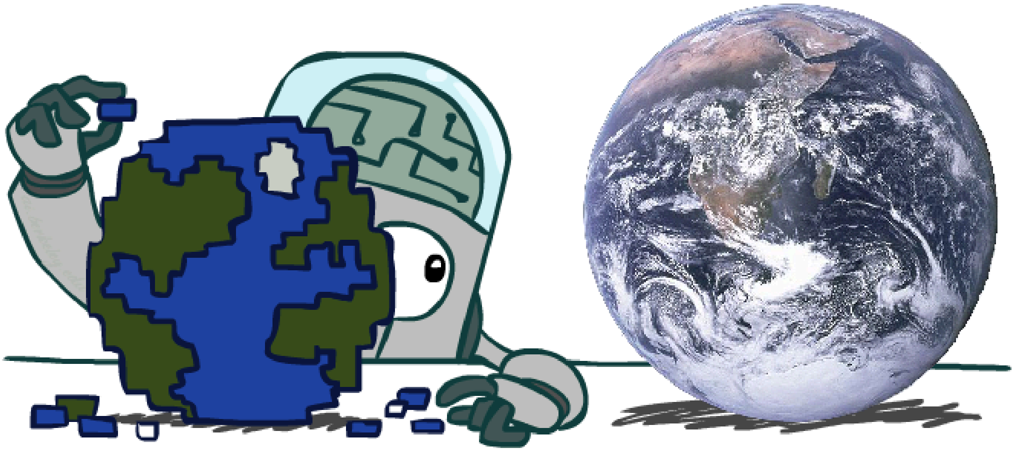
How to go from Arad to Bucharest?

- Initial state = the city we start in.
  - $s_0 = \mathrm{in}(\mathrm{Arad})$
- Actions = Going from the current city to the cities that are directly connected to it.
  - $\mathrm{actions}(s_0) = \{\mathrm{go}(\mathrm{Sibiu}), \mathrm{go}(\mathrm{Timisoara}), \mathrm{go}(\mathrm{Zerind})\}$
- Transition model = The city we arrive in after driving to it.
  - $\mathrm{result}(\mathrm{in}(Arad), \mathrm{go}(Zerind)) = \mathrm{in}(Zerind)$
- Goal test: whether we are in Bucharest.
  - $s \in \{\mathrm{in}(\mathrm{Bucharest})\}$
- Step cost: distances between cities.

Selecting a state space

The real world is absurdly complex.

- The world state includes every last detail of the environment.
- A search state keeps only the details needed for planning.

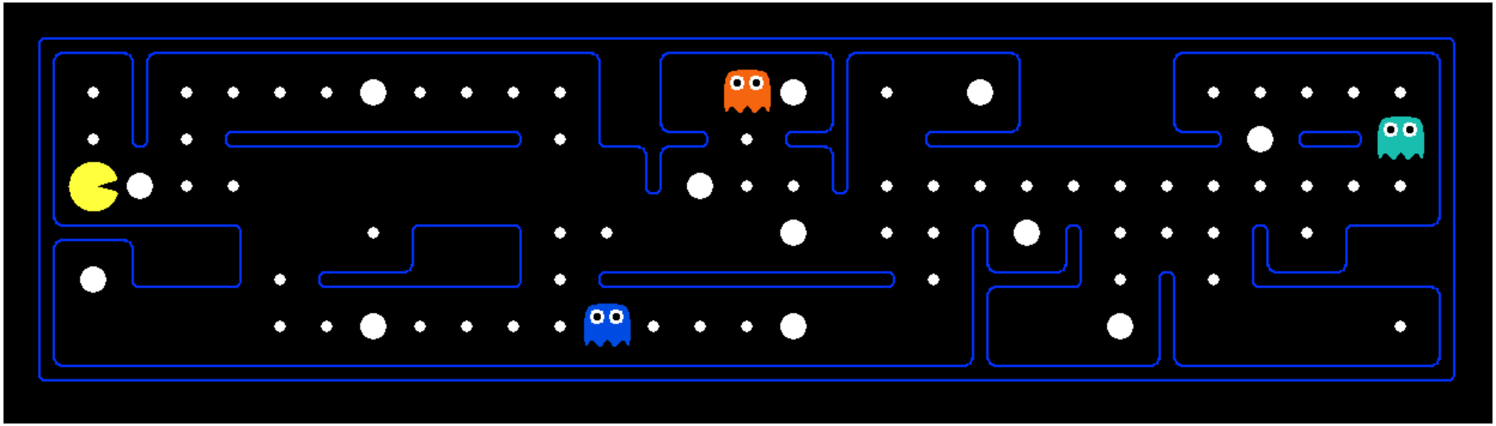


Search problems are models.

Image credits: CS188, UC Berkeley.

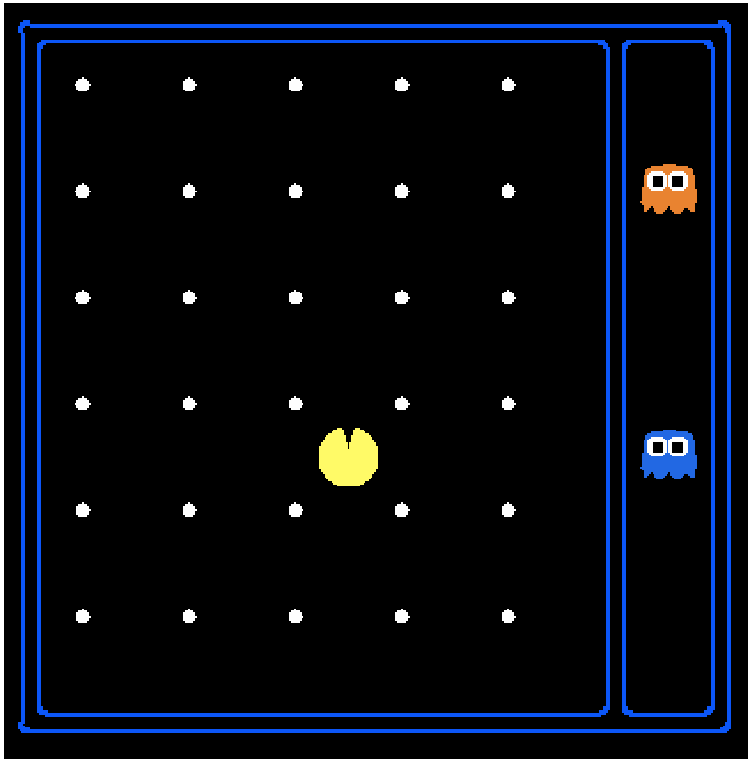
Example: eat-all-dots

- States: $\{(x, y), \text{dot booleans}\}$
- Actions: NSEW
- Transition: update location and possibly a dot boolean
- Goal test: dots all false



State space size

- World state:
    - Agent positions: 120
    - Found count: 30
    - Ghost positions: 12
    - Agent facing: NSEW
- How many?
    - World states?
        - $120 \times 2^{30} \times 12^2 \times 4$
    - States for eat-all-dots?
        - 120×2^{30}

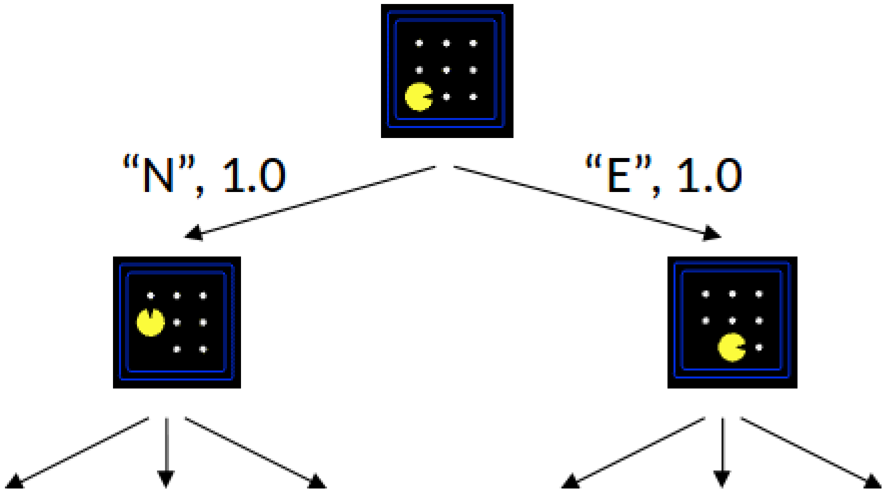


Search trees

The set of acceptable sequences starting at the initial state form a search tree.

- Nodes correspond to states in the state space, where the initial state is the root node.
- Branches correspond to applicable actions, with child nodes corresponding to successors.

For most problems, we can never actually build the whole tree. Yet we want to find some optimal branch!



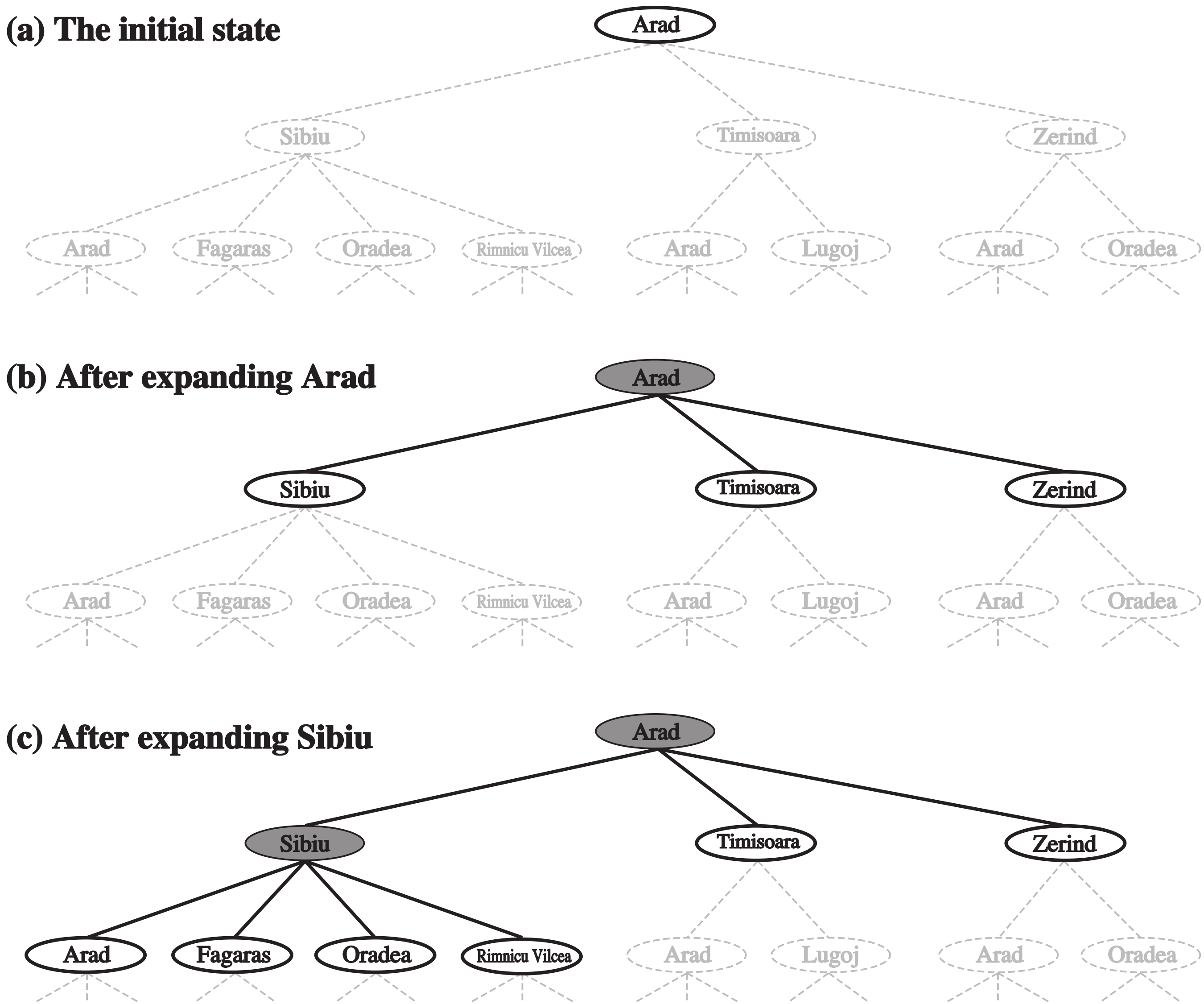
Tree search algorithms

```
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)
```

Important ideas

- Fringe (or frontier) of partial plans under consideration
- Expansion
- Exploration

Exercise
Which fringe nodes to explore? How to expand as few nodes as possible, while achieving the goal?


(a) The initial state
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Uninformed search strategies

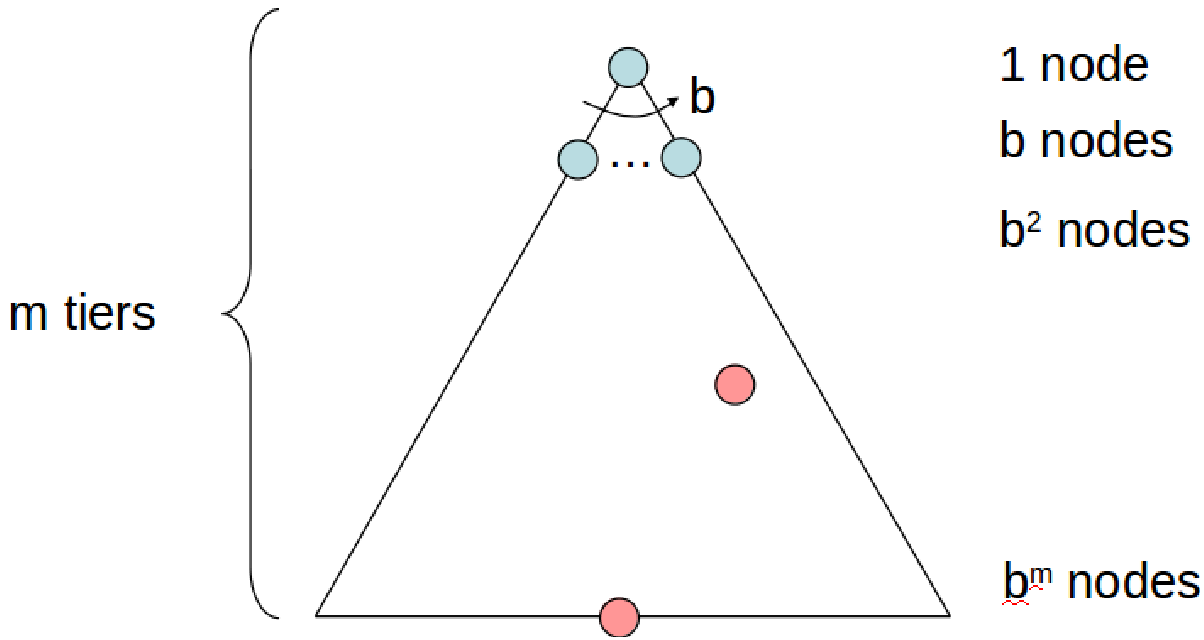
Uninformed search strategies use only the information available in the problem definition. They do not know whether a state looks more promising than some other.

Strategies

- Depth-first search
- Breadth-first search
- Uniform-cost search
- Iterative deepening

Properties of search strategies

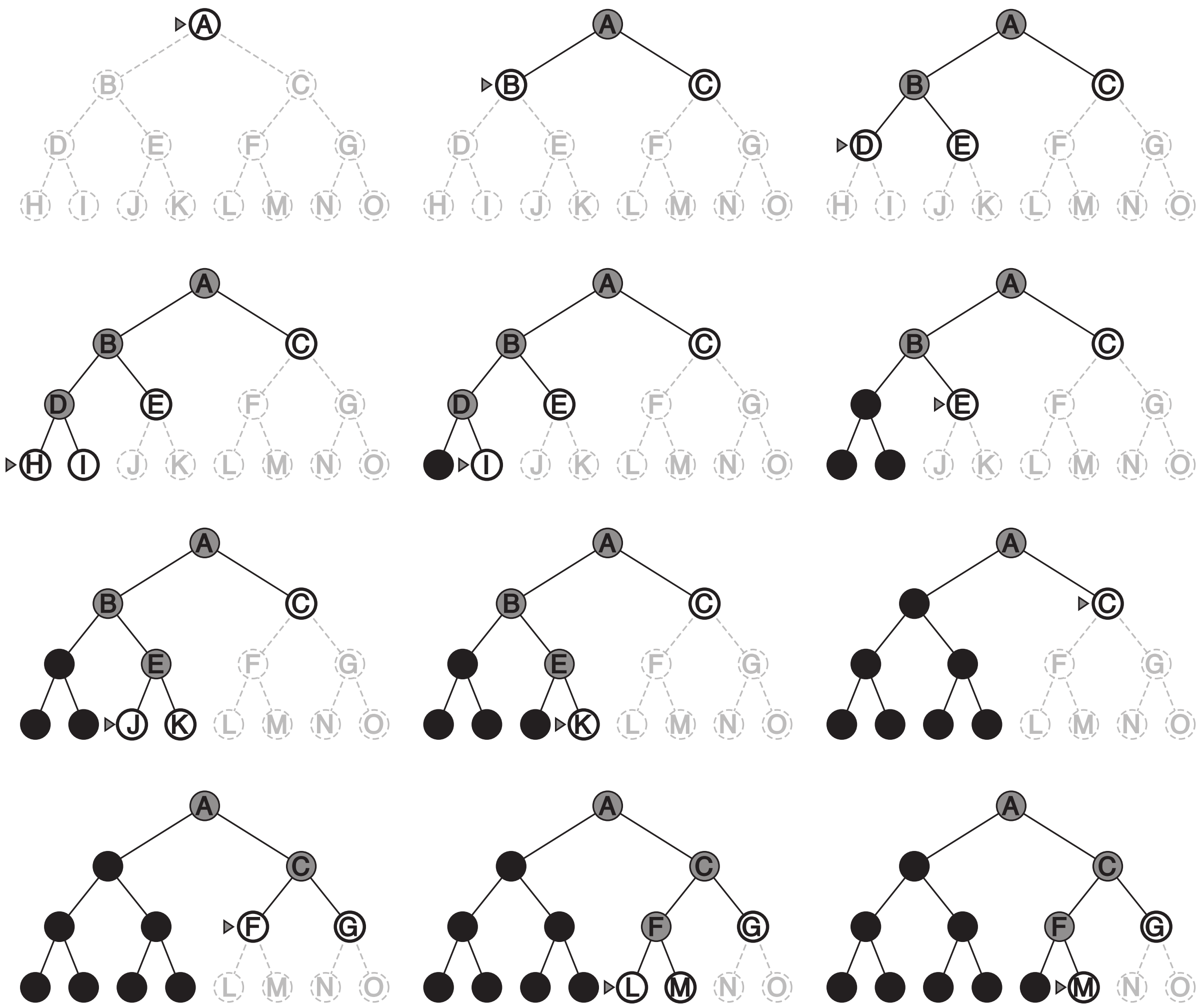
- A strategy is defined by picking the order of expansion.
- Strategies are evaluated along the following dimensions:
  - Completeness: does it always find a solution if one exists?
  - Optimality: does it always find the least-cost solution?
  - Time complexity: how long does it take to find a solution?
  - Space complexity: how much memory is needed to perform the search?
- Time and complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
    - the depth of s is defined as the number of actions from the initial state to s.
  - m: maximum length of any path in the state space (may be ∞)

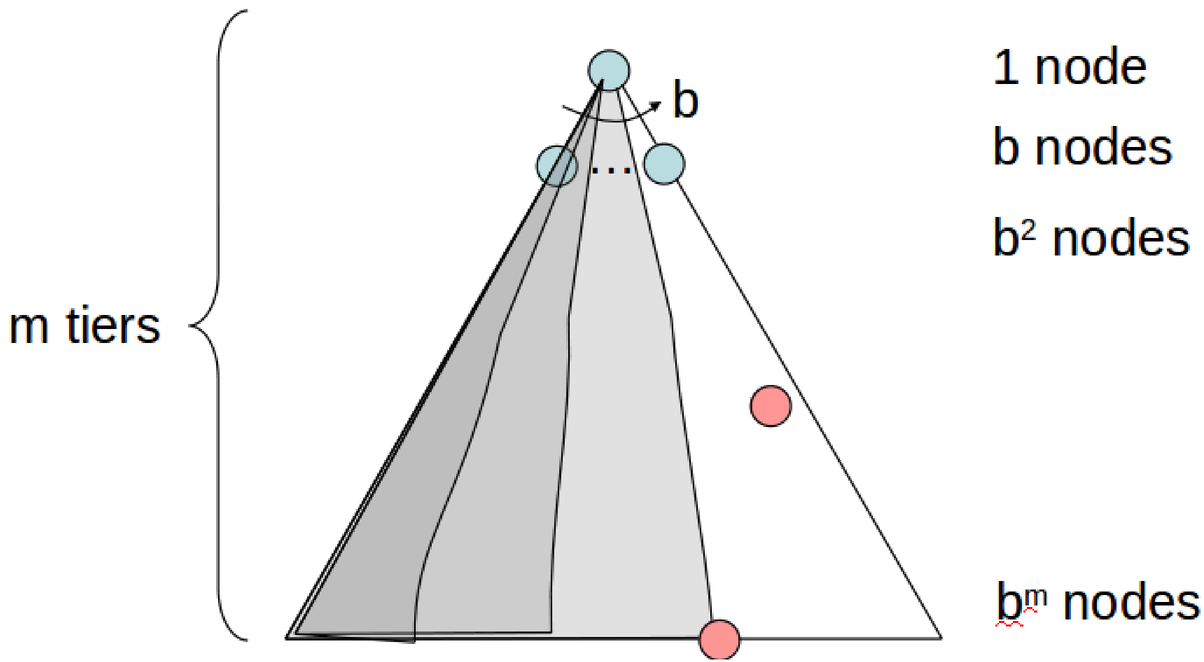

m tiers
b
…
1 node
b nodes
b2 nodes
bm nodes

Depth-first search

—
Image credits: CS188, UC Berkeley.

- Strategy: expand the deepest node in the fringe.
- Implementation: fringe is a LIFO stack.



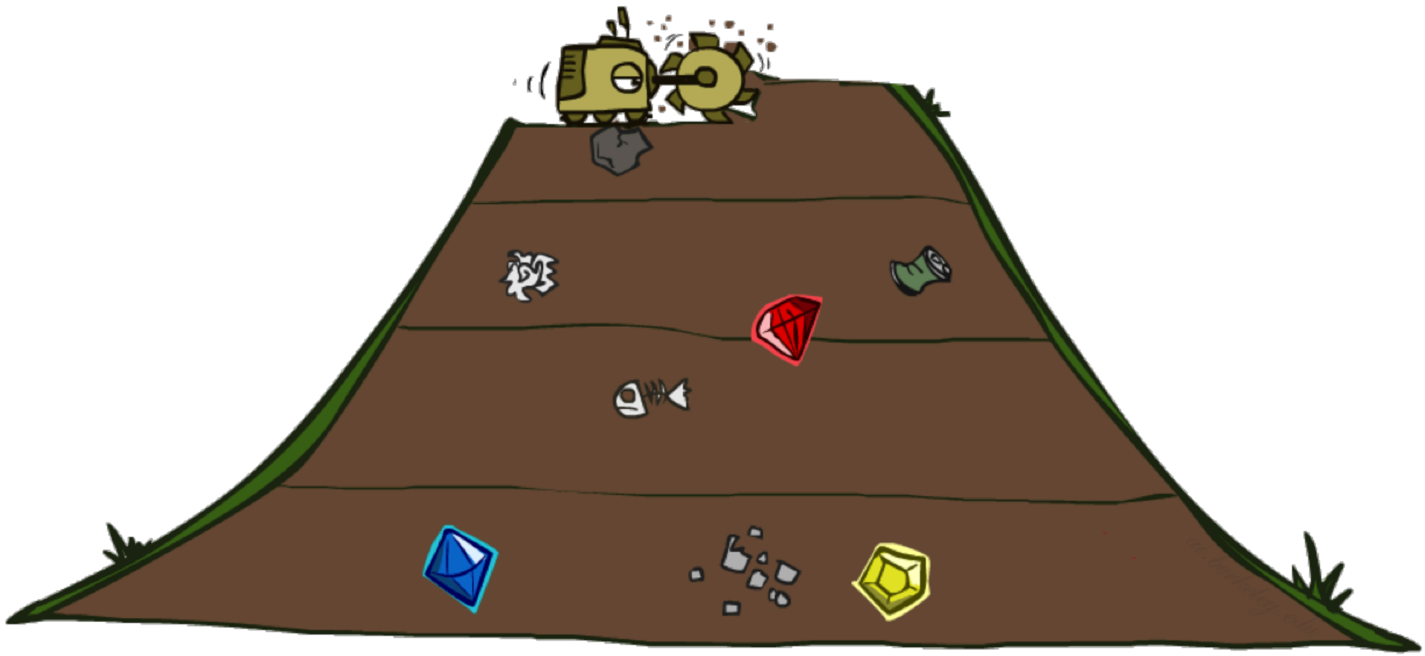


m tiers
b
1 node
b nodes
b² nodes
bᵐ nodes

Properties of DFS

- Completeness:
    - m could be infinite, so only if we prevent cycles (more on this later).
- Optimality:
    - No, DFS finds the leftmost solution, regardless of depth or cost.
- Time complexity:
    - May generate the whole tree (or a good part of it, regardless of d). Therefore $O(b^m)$, which might be much greater than the size of the state space!
- Space complexity:
    - Only store siblings on path to root, therefore $O(bm)$.
    - When all the descendants of a node have been visited, the node can be removed from memory.

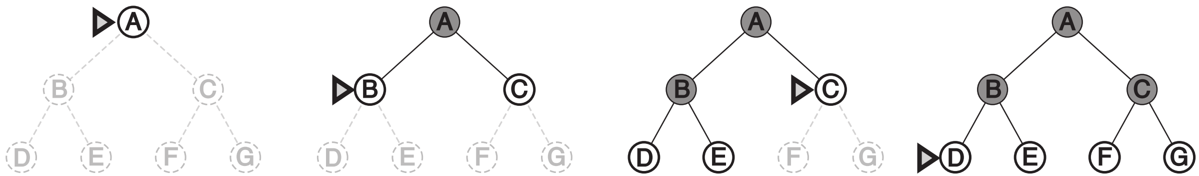
Breadth-first search

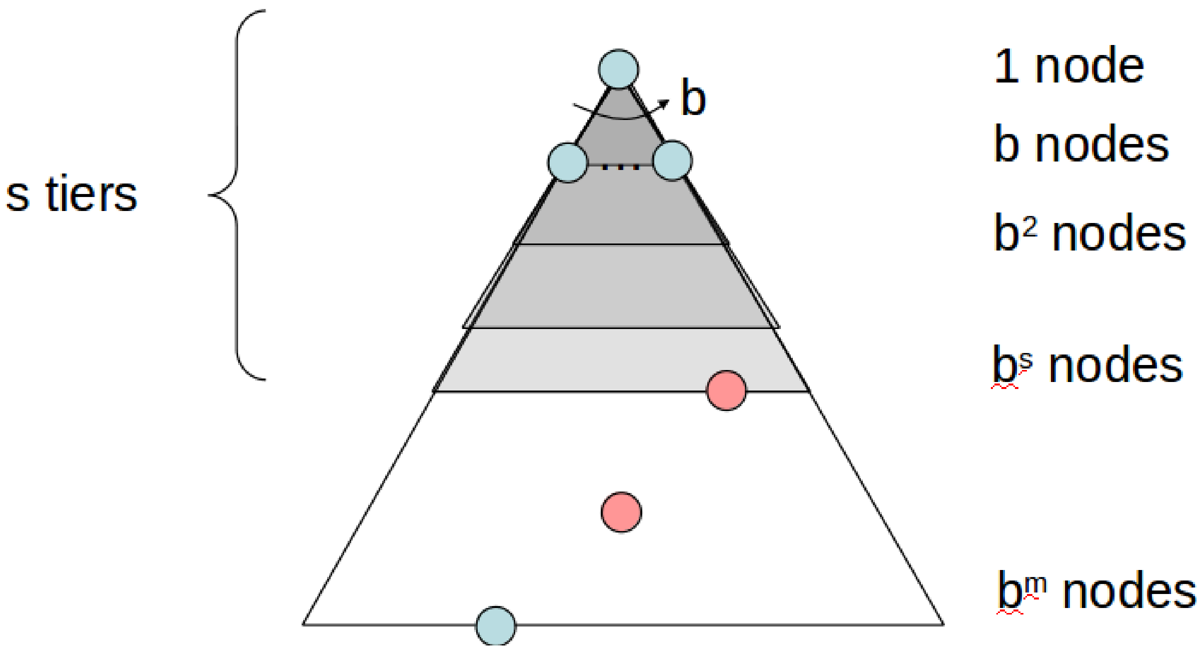


—

Image credits: CS188, UC Berkeley.

- Strategy: expand the shallowest node in the fringe.
- Implementation: fringe is a FIFO queue.




s tiers
b
...
1 node
b nodes
b^2 nodes
b^s nodes
b^m nodes

Properties of BFS

- Completeness:
  - If the shallowest goal node is at some finite depth d, BFS will eventually find it after generating all shallower nodes (provided b is finite).
- Optimality:
  - The shallowest goal is not necessarily the optimal one.
  - BFS is optimal only if the path cost is a non-decreasing function of the depth of the node.
- Time complexity:
  - If the solution is at depth d, then the total number of nodes generated before finding this node is $b + b^2 + b^3 + ... + b^d = O(b^d)$
- Space complexity:
  - The number of nodes to maintain in memory is the size of the fringe, which will be the largest at the last tier. That is $O(b^d)$

(demo)

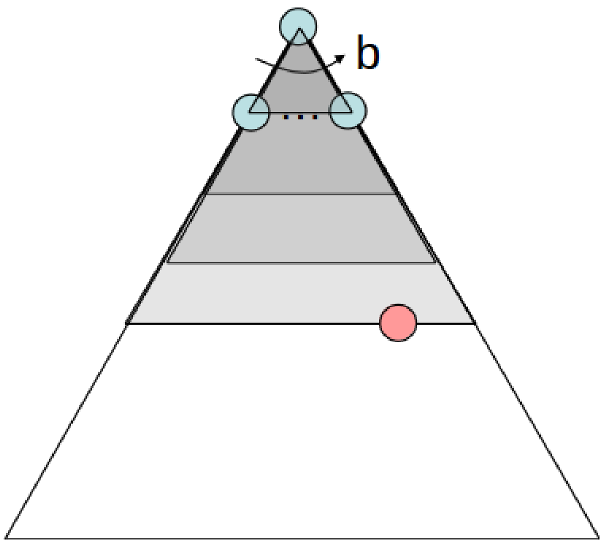
Iterative deepening

Idea: get DFS's space advantages with BFS's time/shallow solution advantages.

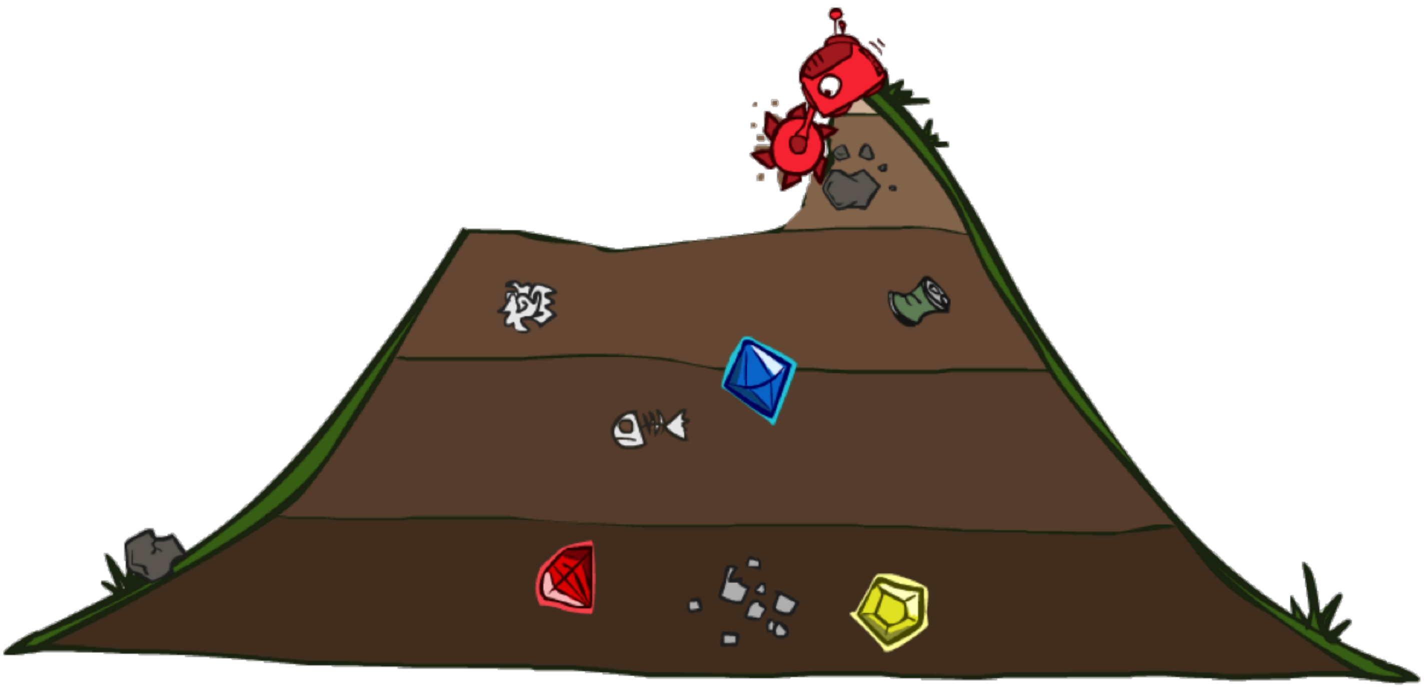
- Run DFS with depth limit 1.
- If no solution, run DFS with depth limit 2.
- If no solution, run DFS with depth limit 3.
    - ...

Exercise

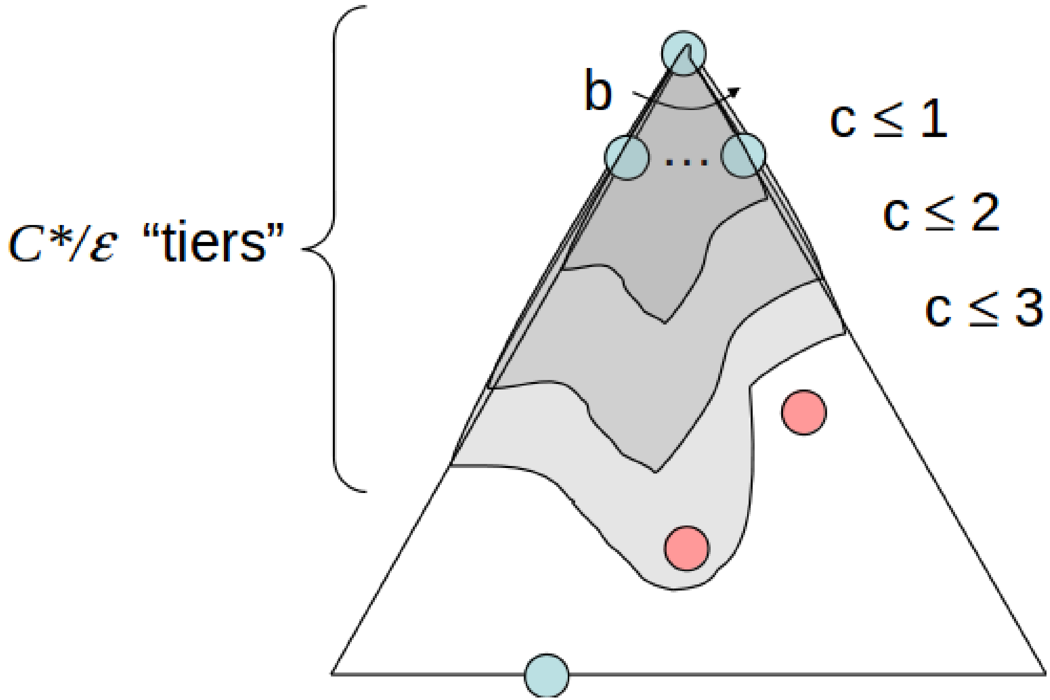
- What are the properties of iterative deepening?
- Isn't this process wastefully redundant?



Uniform-cost search



- Strategy: expand the cheapest node in the fringe.
- Implementation: fringe is a priority queue, using the cumulative cost $g(n)$ from the initial state to node n as priority.

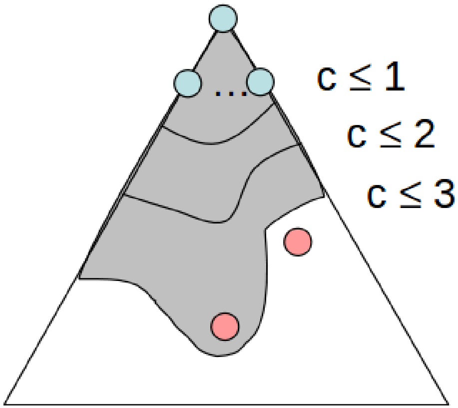

b
...
c ≤ 1
c ≤ 2
c ≤ 3
C*/ε "tiers"

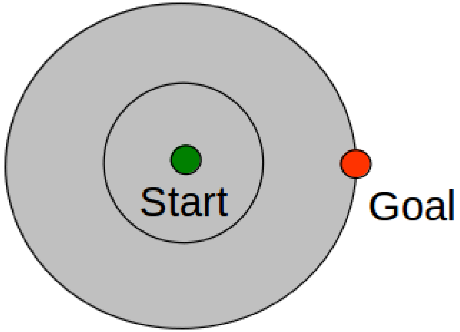
Properties of UCS

- Completeness:
  - Yes, if step cost are all such that $c(s,a,s') \geq \epsilon > 0$. (Why?)
- Optimality:
  - Yes, sinces UCS expands nodes in order of their optimal path cost.
- Time complexity:
  - Assume C^* is the cost of the optimal solution and that step costs are all $\geq \epsilon$.
  - The "effective depth" is then roughly C^*/ϵ.
  - The worst-case time complexity is $O(b^{C^*/\epsilon})$.
- Space complexity:
  - The number of nodes to maintain is the size of the fringe, so as many as in the last tier $O(b^{C^*/\epsilon})$.

(demo)

Informed search strategies





One of the issues of UCS is that it explores the state space in every direction, without exploiting information about the (plausible) location of the goal node.

Informed search strategies aim to solve this problem by expanding nodes in the fringe in decreasing order of desirability.

- Greedy search
- A*

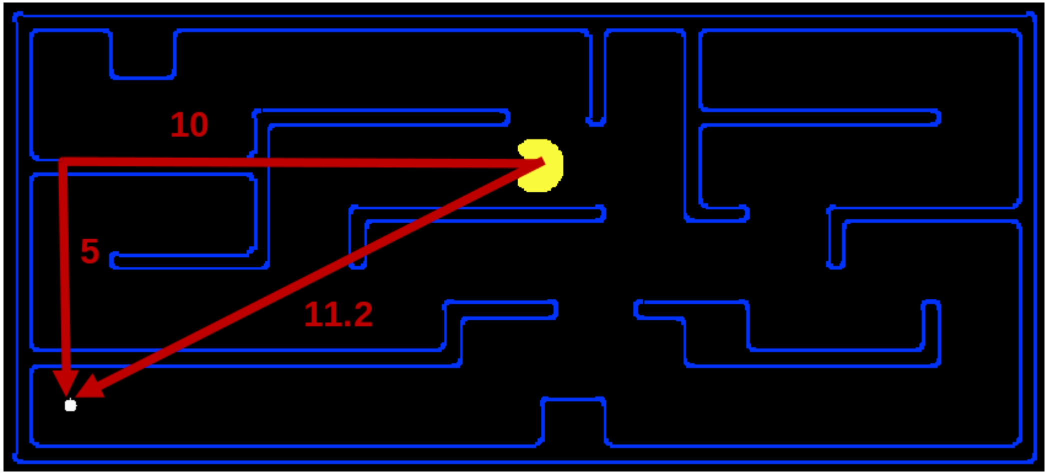
Greedy search

—
Image credits: CS188, UC Berkeley.

Heuristics

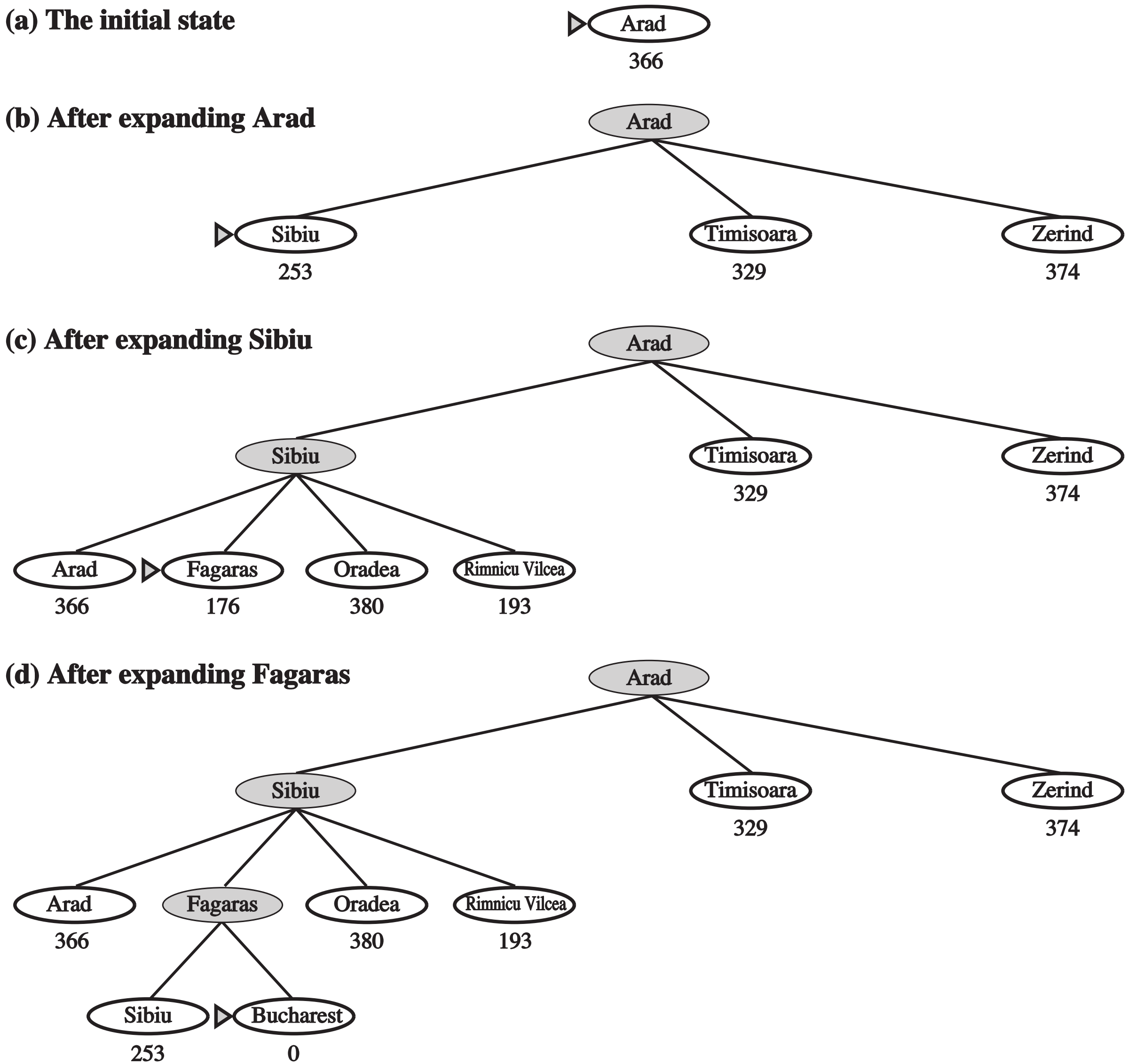
A heuristic (or evaluation) function $h(n)$ is:

- a function that estimates the cost of the cheapest path from node n to a goal state;
  - $h(n) \geq 0$ for all nodes n
  - $h(n) = 0$ for a goal state.
- is designed for a particular search problem.

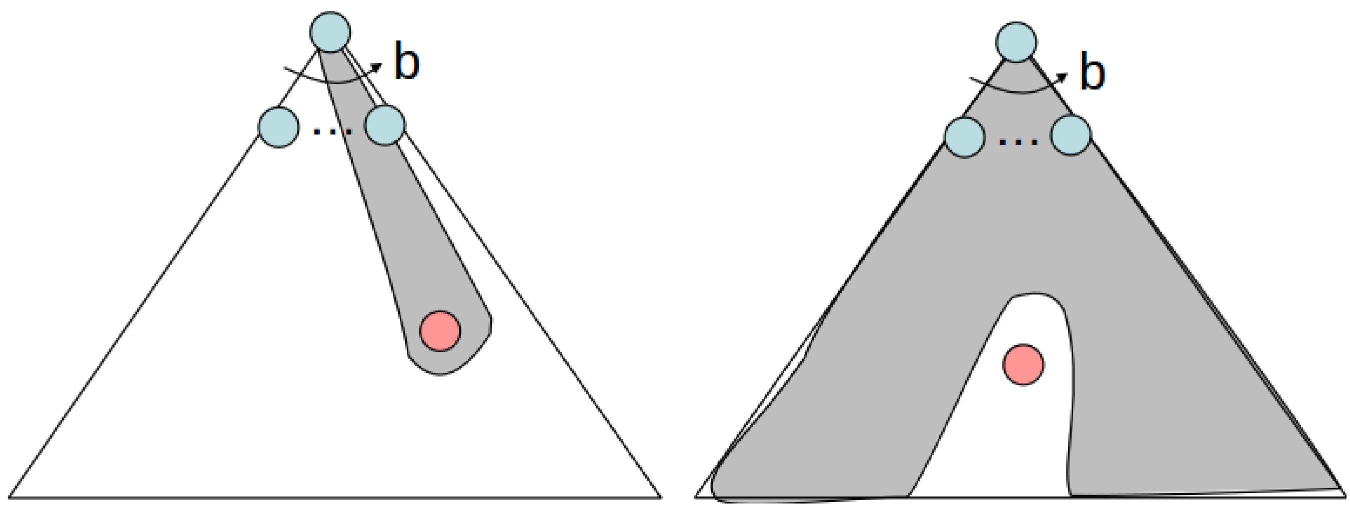


Greedy search

- Strategy: expand the node n in the fringe for which $h(n)$ is the lowest.
- Implementation: fringe is a priority queue, using $h(n)$ as priority.



$h(n)$ = straight line distance to Bucharest.



At best, greedy search takes you straight to the goal.
At worst, it is like a badly-guided BFS.

Properties of greedy search

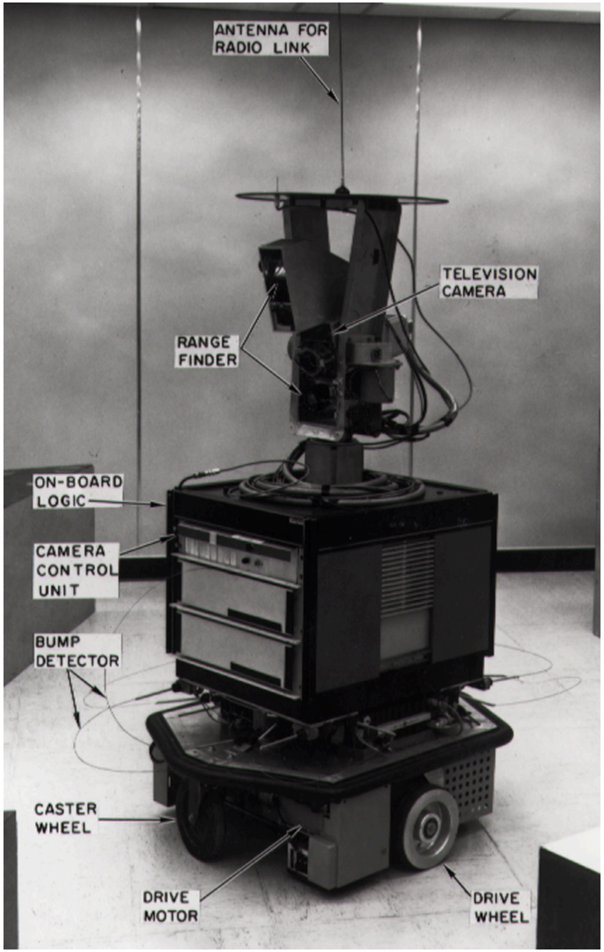
- Completeness:
    - No, unless we prevent cycles (more on this later).
- Optimality:
    - No, e.g. the path via Sibiu and Fagaras is 32km longer than the path through Rimnicu Vilcea and Pitesti.
- Time complexity:
    - $O(b^m)$, unless we have a good heuristic function.
- Space complexity:
    - $O(b^m)$, unless we have a good heuristic function.

A*

—
Image credits: CS188, UC Berkeley.

Shakey the Robot

- A* was first proposed in 1968 to improve robot planning.
- Goal was to navigate through a room with obstacles.

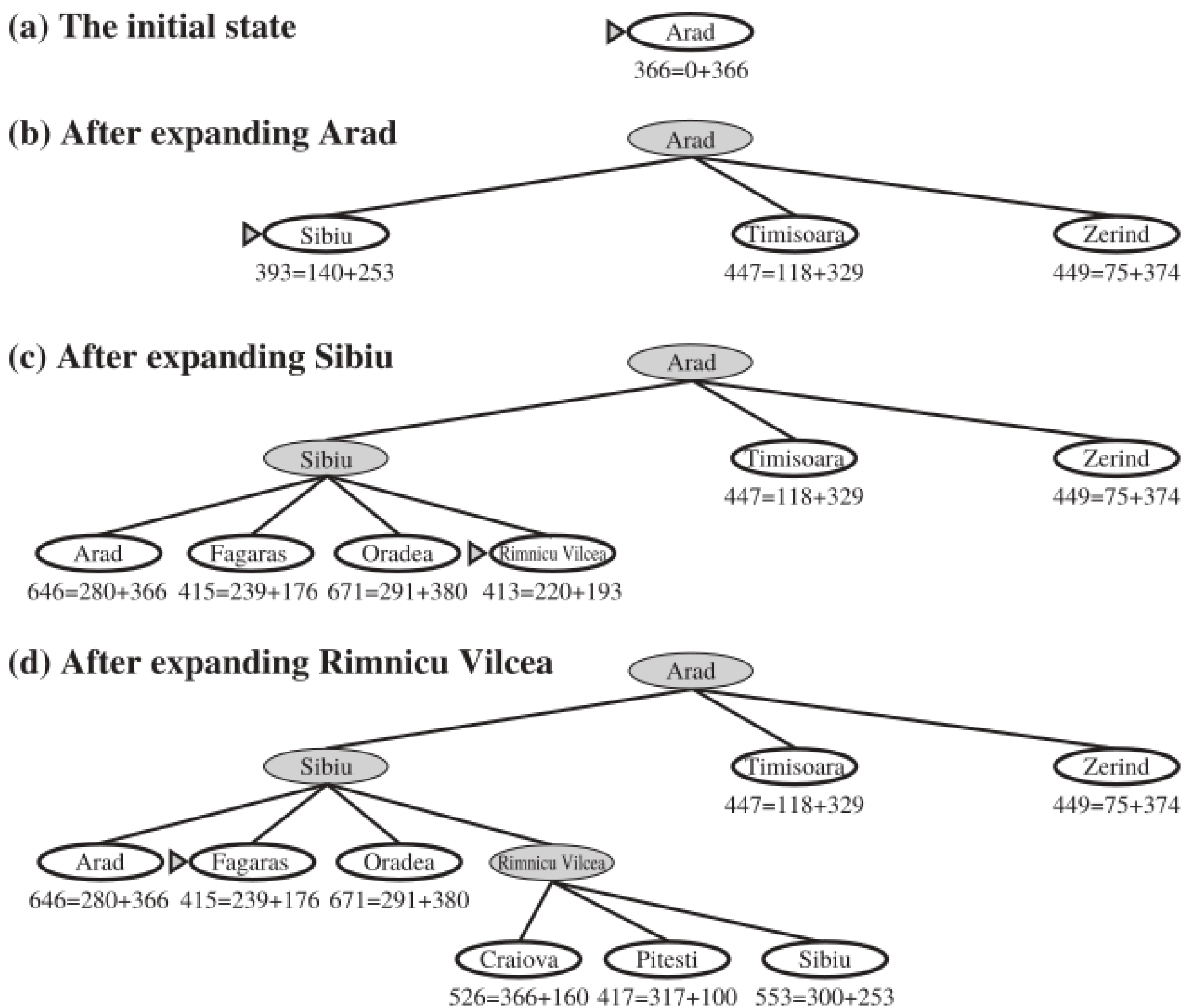


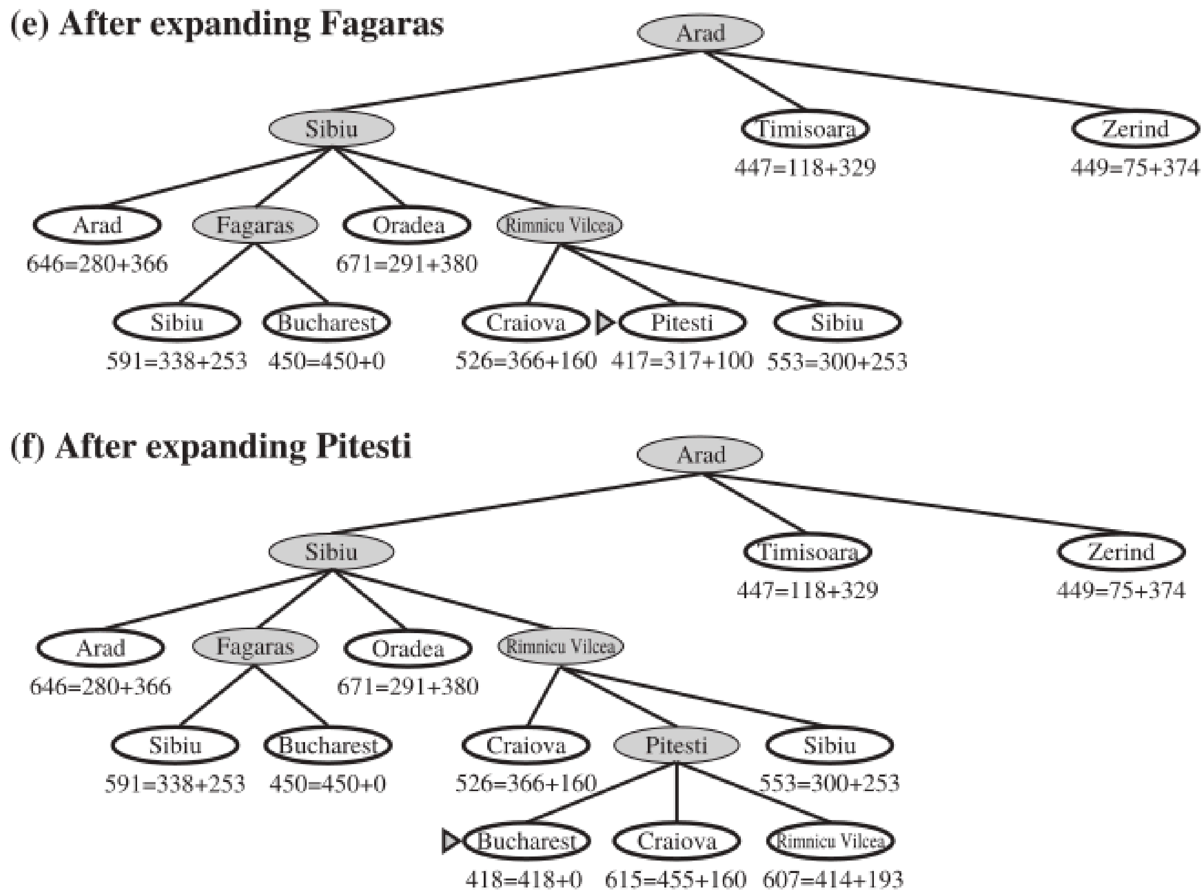
A*

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Greedy orders by goal proximity, or forward cost $h(n)$
- A* combines the two algorithms and orders by the sum

$$f(n) = g(n) + h(n)$$

- $f(n)$ is the estimated cost of cheapest solution through n.


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253



Exercise

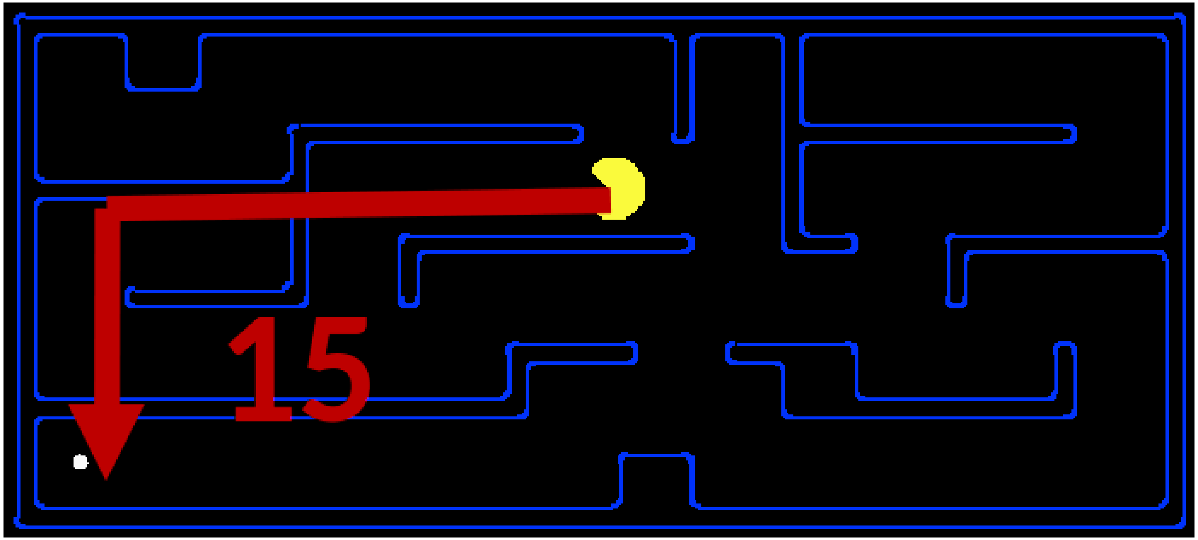
Why doesn't A* stop at step (e), since Bucharest is in the fringe?

Admissible heuristics

A heuristic h is admissible if

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal.



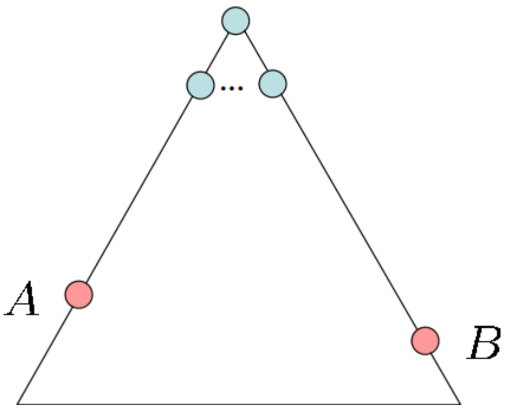
The Manhattan distance is admissible

Optimality of A*

Assumptions:

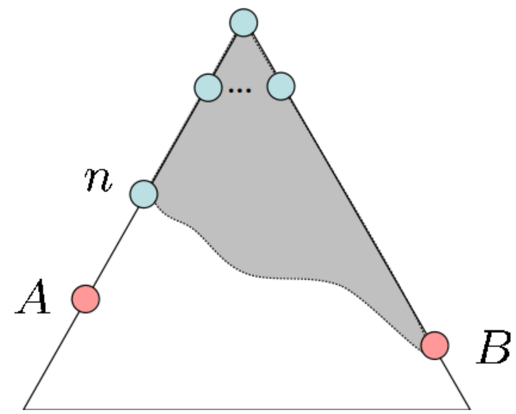
- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim: A will exit the fringe before B.



Proof

Assume B is on the fringe. Some ancestor n of A is on the fringe too.

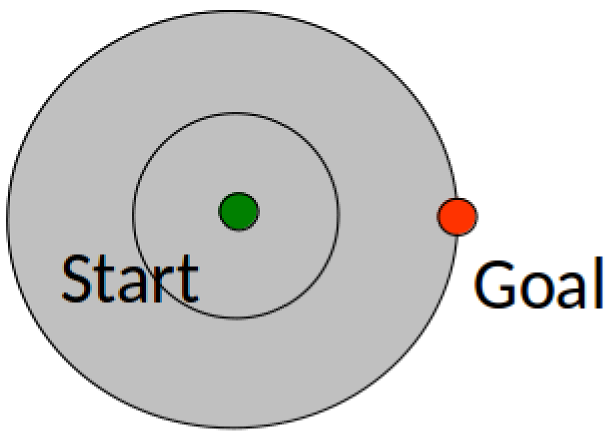


- $f(n) \leq f(A)$
  - $f(n) = g(n) + h(n)$ (by definition)
  - $f(n) \leq g(A)$ (admissibility of h)
  - $f(A) = g(A) + h(A) = g(A)$ ($h = 0$ at a goal)
- $f(A) < f(B)$
  - $g(A) < g(B)$ (B is suboptimal)
  - $f(A) < f(B)$ ($h = 0$ at a goal)
- Therefore, n expands before B.
  - since $f(n) \leq f(A) < f(B)$

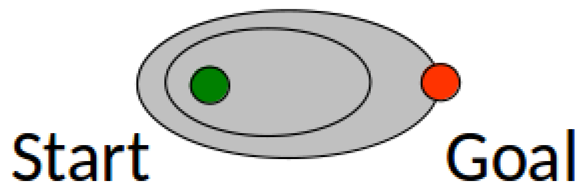
Similarly, all ancestors of A expand before B, including A. Therefore A* is optimal.

A* contours

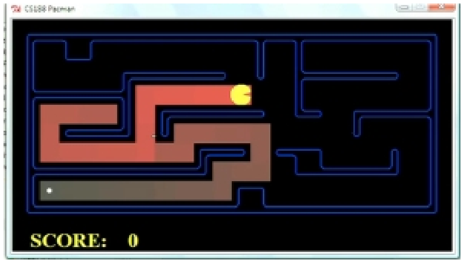
- Assume f-costs are non-decreasing along any path.
- We can define contour levels t in the state space, that include all nodes n for which $f(n) \leq t$.



For UCS ($h(n) = 0$ for all n), bands are circular around the start.

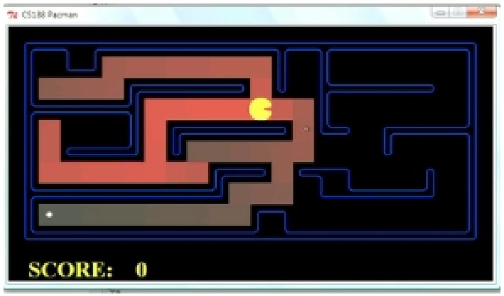


For A* with accurate heuristics, bands stretch towards the goal.



Greedy search

UCS



A*

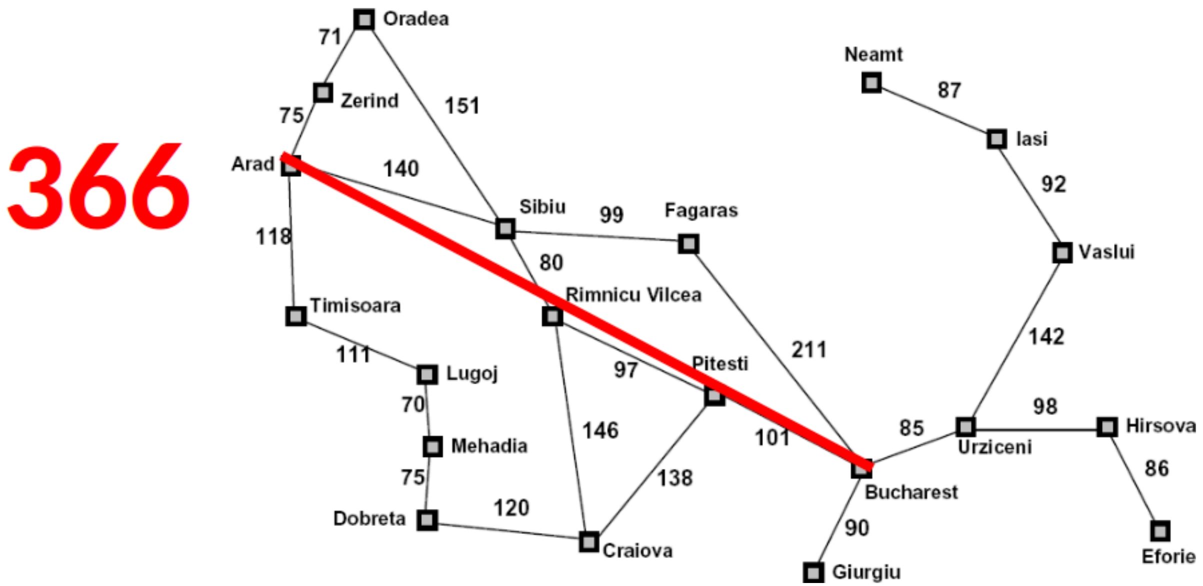
Image credits: CS188, UC Berkeley.

(demo)

Creating admissible heuristics

Most of the work in solving hard search problems optimally is in finding admissible heuristics.

Admissible heuristics can be derived from the exact solutions to relaxed problems, where new actions are available.



Dominance

- If h_1 and h_2 are both admissible and if $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1 and is better for search.
- Given any admissible heuristics h_a and h_b,

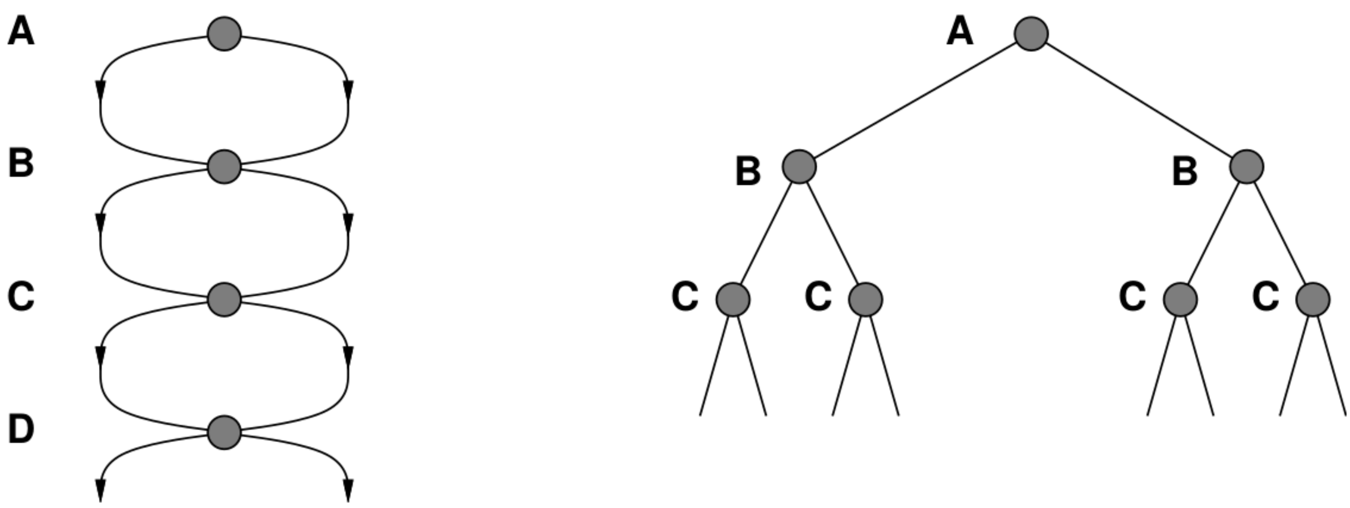
$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a and h_b.

Learning heuristics from experience

- Assuming an episodic environment, an agent can learn good heuristics by playing the game many times.
- Each optimal solution s^* provides training examples from which $h(n)$ can be learned.
- Each example consists of a state n from the solution path and the actual cost $g(s^*)$ of the solution from that point.
- The mapping $n \rightarrow g(s^*)$ can be learned with supervised learning algorithms.
    - Linear models, Neural networks, etc.

Graph search



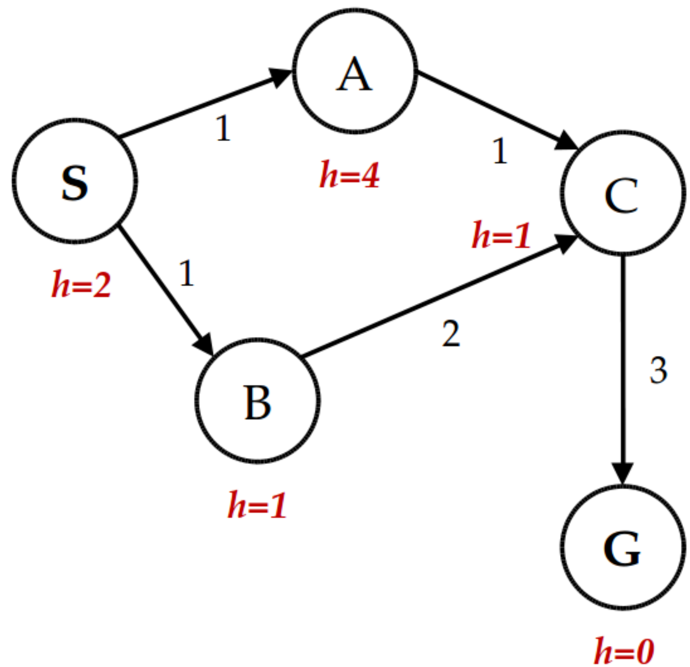
The failure to detect repeated states can turn a linear problem into an exponential one. It can also lead to non-terminating searches.

Redundant paths and cycles can be avoided by keeping track of the states that have been explored. This amounts to grow a tree directly on the state-space graph.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    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
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

A* graph-search gone wrong?

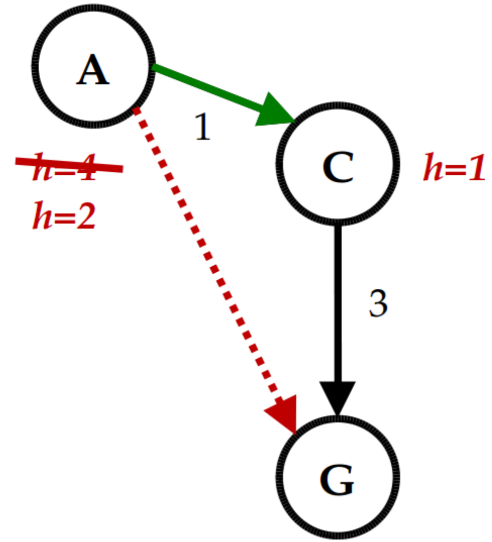
- We start at S and G is a goal state.
- Which path does graph search find?



Consistent heuristics

A heuristic h is consistent if for every n and every successor n' generated by any action a,

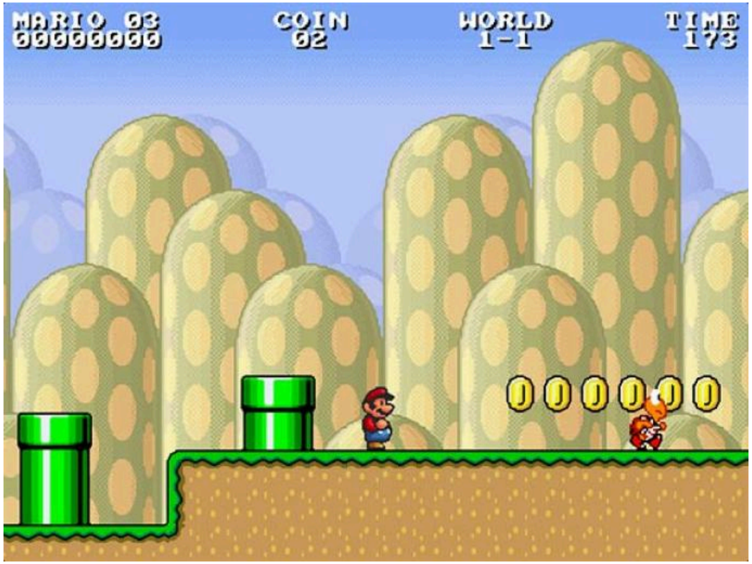
$$h(n) \leq c(n, a, n') + h(n').$$



Consequences of consistent heuristics:

- $f(n)$ is non-decreasing along any path.
- $h(n)$ is admissible.
- With a consistent heuristic, graph-search A* is optimal.

Recap example: Super Mario



- Task environment?
  - performance measure, environment, actuators, sensors?
- Type of environment?
- Search problem?
  - initial state, actions, transition model, goal test, path cost?
- Good heuristic?

A* in action

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- Variety of uninformed search strategies (DFS, BFS, UCS, Iterative deepening).
- Heuristic functions estimate costs of shortest paths. Good heuristic can dramatically reduce search cost.
- Greedy best-first search expands lowest h, which shows to be incomplete and not always optimal.
- A* search expands lowest $f = g + h$. This strategy is complete and optimal.
- Graph search can be exponentially more efficient than tree search.

The end.