COSC343: Artificial Intelligence

Lecture 17: Problem solving and search

Alistair Knott

Dept. of Computer Science, University of Otago

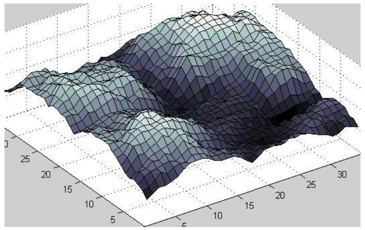
In today's lecture

- The concept of state-space search
- Basic search algorithms

The concept of state-space search

Recap: in an optimisation task,

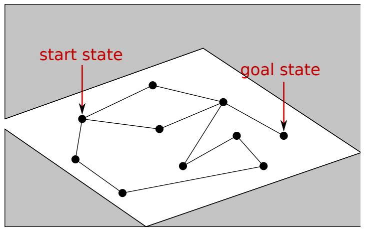
- There's a set of possible states;
- You can establish any state at any time;
- You don't know which state is best.



The concept of state-space search

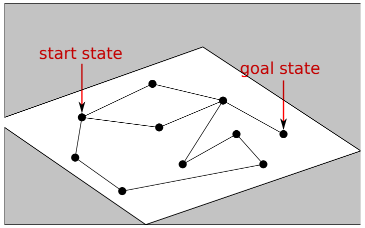
In a state-space search task,

- There's a set of possible states;
- To get into a given state, you may have to go through *other* states.
- You *know* which state is best: but you don't know how to get there.



Actions and state transitions

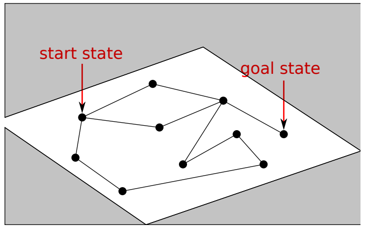
Lots of problems can be formalised as state-space search tasks.



- The agent is in a given start state.
- The agent wants to reach some goal state.
- In each state, there is a set of actions the agent can perform.
  - In different states, there are different possible actions.
  - Each action gets you into a *new* state.

Look-ahead search

If agents can *represent* their world, and *reason* about it a little, they can solve state-space search tasks 'in their heads', without acting at all.

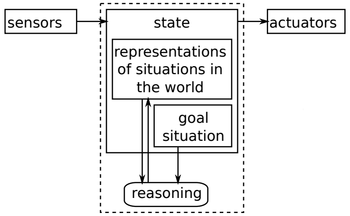


This is a distinctively human skill. It requires the ability

- to represent states of the world *other than the actual state*;
- to represent the results of actions *before they're executed*.

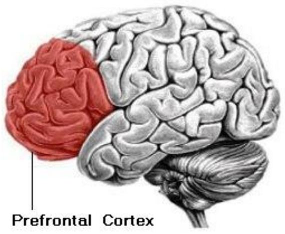
Recap from Lecture 1: types of agent

Goal-based agents can *search* for a way of achieving a desired situation.



Representation and reasoning

Our ability to reason about the consequences of our actions, in service of our goals, depends in large part on the prefrontal cortex.



Not fully developed in humans until age 21!

Formalising look-ahead search

Formally, a look-ahead search problem has three components:

- The agent's INITIAL STATE.
- A function SUCCESSOR-FN(s) that takes a state s and returns the set of actions that are possible in s, along with the *new* state resulting from each action. (A set of $\langle action, result \rangle$ pairs)
- A GOAL TEST. (Either a set of states, or a Boolean fn on states.)

The first two of these define the state space of the problem.

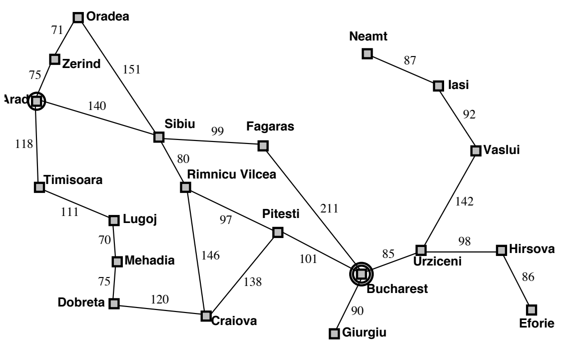
Example: Romania

Let's say the agent is on holiday in Romania, is currently in Arad, and needs to get to Bucharest.

- Let's model Romania as having 20 towns, linked by various roads.
- We can represent the ACTIONS and RESULT functions as a state space graph, in which the nodes are states (towns) and the arcs are actions (travelling down roads which connect towns together).
- The initial state is Arad; the goal state is Bucharest.

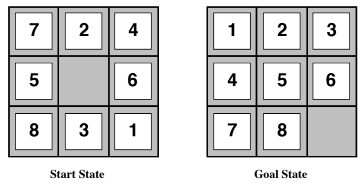
We can now define a formal search problem: is there a sequence of actions that takes the agent from Arad to Bucharest?

A state space graph for Romania



(Note: action costs are shown as labels on the arcs of the graph.)

Another example: The 8-puzzle



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

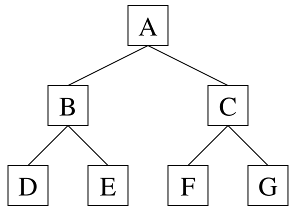
Tree search algorithms

To solve a look-ahead search problem, we build a search tree which systematically explores the state space graph.

- We begin by creating a root node at the start state.
- Then we expand this node, by finding all the states we can get to from this state, and adding these nodes as children of the root node.
- This creates a fringe of new nodes. We now expand all of these new nodes in turn.
- Each time we expand a node, we add the new nodes to the fringe.
- Before we expand a node, we check to see if it's the goal state.

A search tree

Here's a simple search tree, whose root node is A.



Nodes and states

Don't confuse *nodes* in the search tree with *states* in the state-space graph!

- A state is a (representation of a) possible state of the world.
- A node is the computer's record of a state as it is encountered during a systematic search of the state space.

A node is a data structure that has several attributes.

- It has a parent node and a set of successor nodes.
- Since each node represents a state, one of its attributes is a state.
- We also store the action which got us *to* this state in the search.
- We can also store the depth of the node in the search tree.
- And we can add a path cost, storing the combined step costs of all the actions taken from the start state.

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(STATE[node], action,
result)
       DEPTH[s] ← DEPTH[node] + 1
       add s to successors
   return successors
```

Search strategies

There are different ways of building the search tree—i.e. different search strategies.

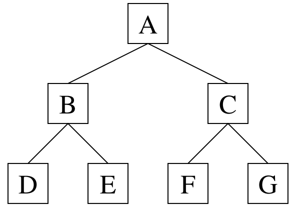
- The strategy is determined by the order in which nodes in the fringe are expanded.
- If the search algorithm always picks the first node in the fringe to expand, then the strategy is determined by how new nodes are *added* to the fringe.

Breadth-first search

In breadth-first search, we expand nodes 'row-by-row' in the search tree.

- Algorithm: put new successor nodes at the *end* of the fringe.

Q: What order would nodes be expanded in the following search tree?



Evaluating a search strategy

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
- d—depth of the least-cost solution
- m—maximum depth of the state space (may be ∞)

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., exponential in d
- Space?? $O(b^{d+1})$ (keeps every node in the lowest ply of the tree in memory)
- Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem.

- Breadth-first search can easily generate nodes at 100MB/sec
- so 24hrs = 8640GB!

Uniform-cost search

Uniform-cost search is a variant of breadth-first search.

- We associate each node with a path cost g.
- When choosing a node to expand, always pick the node with lowest g.

Implementation: Keep the fringe ordered by path cost of nodes (with the lowest path cost first).

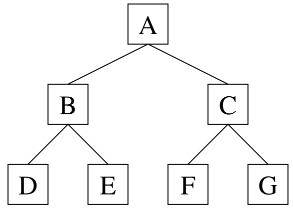
- Complete?? Yes, if step cost $\geq \epsilon$
- Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution
- Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
- Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

In depth-first search, we expand the most recent successor node we generated.

- Algorithm: put new successor nodes at the *front* of the fringe.

Q: What order would nodes be expanded in the following search tree?



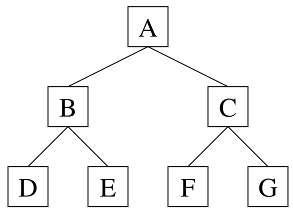
Properties of depth-first search

- Complete?? No: fails in infinite-depth spaces, spaces with loops.
  - You can modify the algorithm to avoid repeated states.
  - Then it's 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

Depth-limited search

Depth-limited search is a depth-first search with a depth limit of l. (I.e. nodes at depth l have no successors.)

If $l = 0$, we just search the start node.



Iterative deepening search

In iterative deepening search, we iteratively perform a depth-limited search in the tree, setting l first to 0, then to level 1, and so on.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```


Properties of iterative deepening search

- Complete?? Yes
- Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?? $O(bd)$
- Optimal?? Yes, if step cost = 1.
  (The algorithm can be modified to explore uniform-cost tree.)

Q: How many nodes must be stored to search a tree with $b = 10$ and $d = 5$, if the solution is at the far right leaf?

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded.

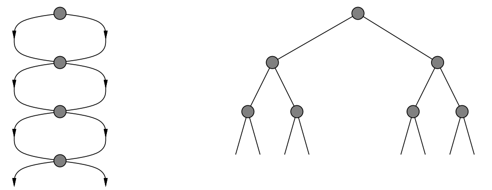
Summary of search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!

For instance: the search space on the left is linear, but if we don't ignore repeated states, we produce a binary branching search tree.



Graph search

In graph search:

- We keep a list called *closed*, holding all the states we have encountered so far.
- We only add a node to the fringe if it's not in *closed*.

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
```


Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- There are several different uninformed search strategies.
- Iterative deepening is pretty much the best of these.
- Graph search can be exponentially more efficient than tree search.

Reading

The reading for today's lecture is AIMA Sections 3.1–3.4.

For next lecture: AIMA Section 3.5–3.6.