Solving problems by searching

Chapter 3

Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Basic search algorithms
Goal-based Agents

Agents that take actions in the pursuit of a goal or goals.

What should a goal-based agent do when none of the actions it can currently perform results in a goal state?

- Choose an action that at least leads to a state that is closer to a goal than the current one is.
Goal-based Agents

Making that work can be tricky:

- What if one or more of the choices you make turn out not to lead to a goal?
- What if you’re concerned with the best way to achieve some goal?
- What if you’re under some kind of resource constraint?

Problem Solving as Search

One way to address these issues is to view goal-attainment as problem solving, and viewing that as a search through a state space.

In chess, e.g., a state is a board configuration
Problem-solving agents

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 do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH( problem)
    action ← FIRST(seq)
    seq ← REST(seq)
return action

Problem Solving

A problem is characterized as:
- An initial state
- A set of actions
- A goal test
- A cost function
Problem Solving

A problem is characterized as:
- An initial state
- A set of actions
  - successors: state \(\rightarrow\) set of states
- A goal test
  - goalp: state \(\rightarrow\) true or false
- A cost function
  - edgecost: edge between states \(\rightarrow\) cost

Example Problems

- **Toy problems (but sometimes useful)**
  - Illustrate or exercise various problem-solving methods
  - Concise, exact description
  - Can be used to compare performance
  - *Examples*: 8-puzzle, 8-queens problem, Cryptarithmetic, Vacuum world, Missionaries and cannibals, simple route finding

- **Real-world problem**
  - More difficult
  - No single, agreed-upon description
  - *Examples*: Route finding, Touring and traveling salesperson problems, VLSI layout, Robot navigation, Assembly sequencing
Toy Problems: The vacuum world

- **The vacuum world**
  - The world has only two locations
  - Each location may or may not contain dirt
  - The agent may be in one location or the other
  - 8 possible world states
  - Three possible actions: Left, Right, Suck
  - **Goal**: clean up all the dirt

---

States: one of the 8 states given earlier
Actions: move left, move right, suck
Goal test: no dirt left in any square
Path cost: each action costs one
Missionaries and cannibals

- Missionaries and cannibals
  - Three missionaries and three cannibals want to cross a river
  - There is a boat that can hold two people
  - Cross the river, but make sure that the missionaries are not outnumbered by the cannibals on either bank
- Needs a lot of abstraction
  - Crocodiles in the river, the weather and so on
  - Only the endpoints of the crossing are important
  - Only two types of people

Problem formulation

- **States**: ordered sequence of three numbers representing the number of missionaries, cannibals and boats on the bank of the river from which they started. The start state is (3, 3, 1)
- **Actions**: take two missionaries, two cannibals, or one of each across in the boat
- **Goal test**: reached state (0, 0, 0)
- **Path cost**: number of crossings
Real-world problems

- **Route finding**
  - Specified locations and transition along links between them
  - *Applications*: routing in computer networks, automated travel advisory systems, airline travel planning systems

- **Touring and traveling salesperson problems**
  - "Visit every city on the map at least once and end in Bucharest"
  - Needs information about the visited cities
  - *Goal*: Find the shortest tour that visits all cities
  - *NP-hard*, but a lot of effort has been spent on improving the capabilities of TSP algorithms
  - *Applications*: planning movements of automatic circuit board drills

What is a Solution?

- A sequence of actions that when performed will transform the initial state into a goal state (*e.g.*, the sequence of actions that gets the missionaries safely across the river)
- Or sometimes just the goal state (*e.g.*, infer molecular structure from mass spectrographic data)
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
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 $\rightarrow$ 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: The 8-puzzle

- states?
- actions?
- goal test?
- path cost?
Example: The 8-puzzle

- **states?** locations of tiles
- **actions?** move blank left, right, up, down
- **goal test?** = goal state (given)
- **path cost?** 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard]

---

Initial Assumptions

- The agent knows its current state
- Only the actions of the agent will change the world
- The effects of the agent’s actions are known and deterministic

All of these are defeasible... likely to be wrong in real settings.
Another Assumption

- Searching/problem-solving and acting are distinct activities
- First you search for a solution (in your head) then you execute it

Tree search algorithms

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```plaintext
function Tree-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of 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 the resulting nodes to the search tree
```
Tree search example

Tree search example
Tree search example

Implementation: general tree search

```plaintext
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 SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
    end loop

function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each action 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
    end loop
    return successors
```

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 **state**, **parent node**, **action**, **path cost** $g(x)$, **depth**

![State Diagram]

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

Search strategies

- A search 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
  - **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 $\infty$)
Uninformed search strategies

- **Uninformed** search 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

Breadth-first search

- Expand shallowest unexpanded node
- **Implementation:**
  - *fringe* is a FIFO queue, i.e., new successors go at end

![Tree diagram](image_of_tree_diagram)
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - fringe is a FIFO queue, i.e., new successors go at end

![Breadth-first search diagram]

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - fringe is a FIFO queue, i.e., new successors go at end

![Breadth-first search diagram]
Breadth-first search

- Expand shallowest unexpanded node

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

Properties of breadth-first search

- **Complete?** Yes (if \( b \) is finite)
- **Time?** \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \)
- **Space?** \( O(b^{d+1}) \) (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step)

- **Space** is the bigger problem (more than time)
Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - fringe = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost \( \geq \varepsilon \)
- Time? # of nodes with \( g \leq \) cost of optimal solution, \( O(b^{\text{ceiling}(C*/\varepsilon)}) \) where \( C^* \) is the cost of the optimal solution
- Space? # of nodes with \( g \leq \) cost of optimal solution, \( O(b^{\text{ceiling}(C*/\varepsilon)}) \)
- Optimal? Yes – nodes expanded in increasing order of \( g(n) \)

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front

\[ \text{Diagram of a tree with nodes labeled A, B, C, D, E, F, G, H, I, J, K, L, M, N, O.} \]
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
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:**
  - \( \text{fringe} = \) LIFO queue, i.e., put successors at front

---

Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
  - \( \text{fringe} = \) 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

Properties of depth-first search

- **Complete?** No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → 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-first search with depth limit \( l \),
i.e., nodes at depth \( l \) have no successors

Recursive implementation:

function Depth-Limited-Search( problem, limit) returns soln/fail/cutoff

Recursion-DFS(Make-Node(Initial-State(problem)), problem, limit)

function Recursive-DFS(node, problem, limit) returns soln/fail/cutoff
cutoff-occurred ← false
if Goal-Test(problem)(State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result ← Recursive-DFS(successor, problem, limit)
if result = cutoff then cutoff-occurred ← true
else if result ≠ failure then return result
if cutoff-occurred then return cutoff else return failure

Iterative deepening search

function Iterative-Deepening-Search( problem) returns a solution, or failure
inputs: problem, a problem
for depth ← 0 to ∞ do
result ← Depth-Limited-Search(problem, depth)
if result ≠ cutoff then return result
Iterative deepening search $l = 0$

Iterative deepening search $l = 1$
Iterative deepening search $l = 2$

Iterative deepening search $l = 3$
Iterative deepening search

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
  
  $$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  
  $$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead = $(123,456 - 111,111)/111,111 = 11\%$

Properties of iterative deepening search

- **Complete?** Yes
- **Time?** $(d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- **Space?** $O(bd)$
- **Optimal?** Yes, if step cost = 1
Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^{*}/c})$</td>
<td>$O(b^{m})$</td>
<td>$O(b^{l})$</td>
<td>$O(b^{d})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^{*}/c})$</td>
<td>$O(b^{m})$</td>
<td>$O(b^{l})$</td>
<td>$O(b^{d})$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms