Searching for Solutions

CISC481/681, Lecture #3
Ben Carterette

Example: Vacuum World

• Search for the sequence of actions that will result in all rooms being clean

Searching for Solutions

• Characterize a task or problem as a search for something
  – In the agent view, a search for a sequence of actions that will result in a predefined goal
• Many (perhaps most) computational problems can be formulated as a search

Example: Traveling Romania
Example: Eight Queens Puzzle

- Place eight queens on a chessboard such that no queen can attack any other

\[ \text{\begin{array}{cccccccc}
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
\end{array}} \]

More Examples

- Games and puzzles
  - Each move leads to a configuration of pieces
  - Search over sequences of moves to find one that wins the game or solves the puzzle
- Scheduling and planning
- Math
  - Roots of polynomials, integration and differentiation, theorem proving, ...
- Natural language processing
- Robotics
- Machine learning
  - Models, features, values of model parameters

Simple Search 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 )
        actions ← FIRST( seq )
        seq ← REST( seq )
    return actions
```

Formulating a Search Problem

- Identify four things:
  - State space (a set of possible states) and initial state
  - Actions and successor function
    - For each state, a mapping of actions to states they lead to
  - Goal test
    - Have I arrived at the state I want to be in?
  - Path cost
    - Cost of traveling from starting state to current state
    - Should be an additive function
- A solution to the problem is a sequence of actions from start state to goal state
State Space

- A **state** is an abstract representation of the most important properties of the world
  - In chess, a state might describe the locations of all the pieces on the board
  - “Abstract” because all that matters is which square each piece is in, not the absolute position
- State spaces are **discrete**: states are individually distinct and identifiable
  - The state space may be infinite, though

Successor Function

- For each state, a mapping indicating which state each possible action will lead to

Path Cost

- The **path cost** is the total cost from the start state to a given state
- A path cost function should be **additive**
  - Each action adds to the total cost
  - All actions have positive cost
- Thus we may define path cost as the sum of the costs of the actions taken
Path Cost

- The arc cost is the cost of taking a particular action
  - Chess: arc cost = 1
  - Traveling Romania: arc cost = mileage
- It is useful to assume cost is always non-zero
  - To prevent the number of actions from growing too large
  - If the arc cost of chess were 0, two players could get stuck in an infinite loop of moving their kings one space back and forth

Goal Test

- A search problem should have a well-defined goal
- That goal should be represented by at least one state
  - A state representing the solution to the problem
- The goal test function determines whether the current state is the goal state
  - Chess: goal is checkmate; goal function determines whether opponent is in check, and if so, whether opponent can move king out of check
  - Traveling Romania: goal is Bucharest; goal function checks whether we're there
- There may be many equally-viable goal states

Example: Traveling Romania

<table>
<thead>
<tr>
<th>State space</th>
<th>Initial state</th>
<th>Actions</th>
<th>Goal test</th>
<th>Path cost</th>
<th>Solution</th>
<th>Optimal solution</th>
</tr>
</thead>
</table>

Example: Vacuum World

| States: |
| Actions: |
| Path cost: |
| Goal test: |
Example: Eight Queens

- States:
- Initial state:
- Actions:
- Goal:
- Path cost:

- Choosing the state space is a search problem (think about it)

Tree Search

- Search problems comprising states, successors, and arc costs are naturally represented as a tree

Tree Nodes

- Each node encodes information about the search:
  - State, parent node, action, path cost \( g(x) \), depth

Search Strategy

- **Expanding** a node \( x \) uses \( x \)'s successor function to create a list of nodes containing states that can be reached from \( x \):
  - \( \text{expand}(\text{Fagaras}) = \{(\text{Sibiu}, \text{cost}=338), (\text{Bucharest}, \text{cost}=450)\} \)
  - Expansion nodes added to the **fringe**: the list of nodes that have not yet been expanded

- A **search strategy** describes the order in which nodes are expanded
Tree Search

```c
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)
```

Big-O Notation Review

- \( f(n) = O(g(n)) \)
- Describes rate of growth of time/space to compute \( f(n) \) as a function \( g(n) \) of \( n \)
  - \( g(n) = 1 \): constant growth; input size does not determine time/space costs
  - \( g(n) = \log n \): logarithmic growth
  - \( g(n) = n \): linear growth
  - \( g(n) = n^{k} \), \( k > 1 \): polynomial growth
  - \( g(n) = k^n \), \( k > 1 \): exponential growth

Complexity Review

- \( \text{EXP} \)
- \( \text{PSPACE} \)
- \( \text{NP} \)
- \( \text{P} \)

Measuring Search Strategy Performance

- **Completeness**: will it find a solution if one exists?
- **Optimality**: is the solution it finds optimal?
- **Time & space complexity**: how many nodes expanded and how many in memory?
  - Branching factor \( b \): maximum number of successors of any node
  - Depth \( d \): depth of least-cost solution
  - Depth \( m \): maximum depth of the search
Uninformed or “Blind” Search

Environment Properties

• Assumptions behind uninformed search:
  – Static: states can’t change
  – Discrete: states are individually distinct; there is not a smooth continuum between states
  – Observable: the world is visible from each state
  – Deterministic: state properties, arc costs, etc are fixed

Algorithms

• Breadth-first
• Uniform cost
• Depth-first
• Depth-limited
• Iterative deepening

• Their implementation depends mainly on how the fringe nodes are ordered for expansion

Breadth-First Search

• Expand nodes in the order they appear
• Treat fringe as a queue (first in, first out)

• Properties:
  – Complete: it will find a solution
  – Optimal: the solution will be optimal
  – Exponential time & space complexity: $O(b^d)$
Uniform-Cost Search

• Just like breadth-first, except order nodes on the fringe by increasing path cost
• Expand least-cost node on fringe
• Properties:
  – Complete assuming arc cost \( \geq \epsilon > 0 \)
  – Optimal
  – Time complexity: number of nodes with \( g \leq \) cost of optimal solution = \( O(b^{C*/\epsilon}) \), \( C* \) = cost of optimal
  – Space complexity: number of nodes with \( g \leq \) cost of optimal solution = \( O(b^{C*/\epsilon}) \)

Depth-First Search

• Expand the most-recently added node
• Treat fringe as a stack (first in, last out)
• Properties:
  – Not complete: may get stuck in loops; may deepen infinitely
  – Not optimal: it finds the first solution, not the best
  – Exponential time \( O(b^m) \) but linear space \( O(bm) \)

Depth-Limited Search

• Depth-first, but with a depth limit \( l \)
  – Do not continue to expand nodes at the limit
• Solves infinite-path problem
  – But still incomplete—why?

Iterative Deepening

• Use depth-limited search iteratively, increasing limit by one each time
• For each value of \( l \), every possible node at depth \( l \) is tested
  – Like breadth-first
• Properties:
  – Complete: it will find a solution
  – Optimal: if arc cost is constant
  – Complexity: exponential time \( O(b^d) \), linear space \( O(bd) \)
  – No worse than depth-first (\( d \leq m \))
Algorithm Comparison

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-Limited</th>
<th>Depth-First</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))</td>
<td>(O(b^{D+1}))</td>
<td>(O(b^{D+1}))</td>
<td>(O(b^D))</td>
<td>(O(b^D))</td>
</tr>
<tr>
<td>Space</td>
<td>(O(b^D))</td>
<td>(O(b^{D+1}))</td>
<td>(O(b^{D+1}))</td>
<td>(O(b^D))</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>

• Practical time/space implications when \(b=10\):

<table>
<thead>
<tr>
<th></th>
<th>Breadth-first</th>
<th>Iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D=2)</td>
<td>0.11s; 1Mb</td>
<td>0.01s; 20Kb</td>
</tr>
<tr>
<td>(D=4)</td>
<td>11s; 106Mb</td>
<td>1s; 40Kb</td>
</tr>
<tr>
<td>(D=8)</td>
<td>31h; 1Tb</td>
<td>2.8h; 80Kb</td>
</tr>
<tr>
<td>(D=16)</td>
<td>317,000y; 10Eb</td>
<td>32,000y; 1.6Mb</td>
</tr>
</tbody>
</table>

Bidirectional Search

• Start a search from the beginning and a search from the end
• Stop when their fringes intersect
• Requires a “predecessor” function

Avoiding Repeated States

• A finite state space can result in an infinite search
  — E.g. when states can be revisited

Graph Search

```python
def GRAPH-SEARCH(problem, fringe):
    fringe = an empty set
    fringe = INSERT(Make-Node(InitialState[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node = REMOVE-FRONT(fringe)
        fringe = INSERT-ALL(EXPAND(node, problem), fringe)
        if GOAL-TEST(Problem[State[node]]) then return SOLUTION(node)
        if State[node] is not in closed then
            add State[node] to closed
            fringe = INSERT-ALL(Expand(node, problem), fringe)
    return failure
```