What is Search?

- Search is a basic problem-solving method
- We start in an initial state
- We examine states that are (usually) connected by a sequence of actions to the initial state
- Note: Search is (usually) a thought experiment (separate topic: Real Time Search)
- We aim to find a solution, which is a sequence of actions that brings us from the initial state to the goal state, minimizing cost

Search vs. Web Search

- When we issue a search query using Google, does Google really go poking around the web for us?
- Not in real time!
- Google spiders the web continually, caches results
- Uses page rank algorithm to find the most “popular” web pages that are consistent with your query

Overview

- Problem Formulation
- Uninformed Search
  - DFS, BFS, IDDFS, etc.
- Informed Search
  - Greedy, A*
- Properties of Heuristics
Problem Formulation

- Four components of a search problem
  - Initial State
  - Actions
  - Goal Test
  - Edge costs (uniform, or varying per edge?)
- Optimal solution = lowest path cost to goal

Example: Path Planning

Find shortest route from one city to another using highways.

Example 8(15)-puzzle

Possible Start State

<table>
<thead>
<tr>
<th>8</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Solution

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Goal State

Actions: UP, DOWN, RIGHT, LEFT

“Real” Problems

- Robot motion planning
- Drug design
- Logistics
  - Route planning
  - Tour Planning
- Assembly sequencing
- Internet routing
Why Use Search?

• Other algorithms exist for these problems:
  – Dijkstra’s Algorithm
  – Dynamic programming
  – All-pairs shortest path

• Use search when it is too expensive to enumerate all states
• 8-puzzle has 362,800 states
• 15-puzzle has 1.3 trillion states
• 24-puzzle has $10^{25}$ states

Basic Search Concepts

• Assume a tree-structured space (for now)
• Nodes: Places in search tree
  (states exist in the problem space)
• Search tree: portion of state space visited so far
• Actions: Connect states to next states
• Expansion: Generation of next states for a state
• Frontier: Set of states visited, but not expanded
• Branching factor: Max no. of successors = $b$
• Goal depth: Depth of shallowest goal = $d$

Example Search Tree

8-puzzle
Generic Search Algorithm

Function Graph-Search(problem)
  Initialize frontier using initial state of problem
  Initialize explored set to empty
  loop do
    if empty(frontier) then return failure
    remove leaf node from the frontier
    if goal(leaf) then return (and possibly path to leaf)
    else if leaf is not in explored set
      add leaf to explored set
      add children of leaf to frontier (“expand leaf”)
    end
  end

Most important detail = implementation of frontier

Generic Search Implementation

• Methods needed
  – Frontier management
  – Goal test
  – Node expansion
  – Explored set management

• Details
  – Pointers from children to parents to reconstruct paths
  – A way of uniquely identifying states

Tree Search Algorithm

Function Tree-Search(problem)
  Initialize frontier using initial state of problem
  Initialize explored set to empty
  loop do
    if empty(frontier) then return failure
    remove leaf node from the frontier
    if goal(leaf) then return (and possibly path to leaf)
    else if leaf is not in explored set
      add leaf to explored set
      add children of leaf to frontier “expand leaf”
    end

Most important detail = implementation of frontier

Evaluating Search Algorithms

• Completeness:
  – Is the algorithm guaranteed to find a solution when there is one?

• Optimality:
  – Does the algorithm find the optimal solution?

• Time complexity
• Space complexity
Uninformed Search: BFS

Frontier is a FIFO

Uninformed Search: DFS

Frontier is a LIFO

BFS Properties

- Completeness: \( \gamma \)
- Optimality: \((Y \text{ for constant cost, } N \text{ for arbitrary cost})\)
- Time complexity: \(O(b^{d+1})\)
- Space complexity: \(O(b^{d+1})\)

DFS Properties

- Completeness: \((Y \text{ for finite trees, } N \text{ for infinite trees})\)
- Optimality: \(N\)
- Time complexity: \(O(b^{m+2})\) (\(m = \text{depth we hit, } m>d?)\)
- Space complexity: \(O(bm)\) (for trees)
Iterative Deepening

- Want:
  - DFS memory requirements
  - BFS optimality, completeness
- Idea:
  - Do a depth-limited DFS for depth m
  - Iterate over m

IDDFS Properties

- Completeness: \( \top \)
- Optimality: (whenever BFS is optimal)
- Time complexity: \( O(b^{d+2}) \)
- Space complexity: \( O(bd) \)

IDDFS vs. BFS

Theorem: IDDFS visits no more than twice as many nodes for a binary tree as BFS.

Proof: Assume the tree bottoms out at depth \( d \), BFS visits:

\[
2^{d+1} - 1
\]

In the worst case, IDDFS does no more than:

\[
\sum_{i=0}^{d} (2^{i+1} - 1) = \sum_{i=0}^{d} 2^{i+1} - \sum_{i=1}^{d} 1 = 2(2^{d+1} - 1) - d < 2(2^{d+1} - 1) = 2 \times BFS(d)
\]

What about b-ary trees? IDDFS relative cost is lower!
Bi-directional Search

$b^d/2 + b^d/2 << b^d$

Issues with Bi-directional Search

- Uniqueness of goal
  - Suppose goal is parking your car
  - Huge no. of possible goal states (configurations of other vehicles)
- Invertability of actions

Informed Search

- Idea: Give the search algorithm hints
- Heuristic function: $h(x)$
- $h(x)$ = estimate of cost to goal from $x$
- If $h(x)$ is 100% accurate, then we can find the goal in $O(bd)$ time

Greedy Search

- Expand node with lowest $h(x)$
- Optimal if $h(x)$ is 100% correct
- How can we get into trouble with this?
What Price Greed?

A*

- Path cost so far: \( g(x) \)
- Total cost estimate: \( f(x) = g(x) + h(x) \)
- Maintain frontier as a priority queue (on \( f \))
- \( O(bd) \) time if \( h \) is 100% accurate
- We want \( h \) to be an admissible heuristic
- Admissible: never overestimates cost

Some A* Properties

- Admissibility implies \( h(x)=0 \) if \( x \) is a goal state
- Above implies \( f(x) \)=cost to goal if \( x \) is a goal state and \( x \) is popped off the queue

- What if \( h(x)=0 \) for all \( x \)?
  - Is this admissible?
  - What does the algorithm do?

Optimality of A*

- If \( h \) is admissible, A* is optimal
- Proof (by contradiction):
  - Suppose a suboptimal solution node \( n \) with solution value \( f(n) > C^* \) is about to be expanded (where \( C^* \) is optimal)
  - Let \( n^* \) be a goal state found on optimal path
  - There must be some node \( n' \) that is currently in the fringe and on the path to \( n^* \)
  - We have \( f(n) > C^* \), and \( f(n') = g(n') + h(n') \leq C^* \)
  - But then, \( n' \) should be expanded first (contradiction b/c we are using a priority queue prioritized on \( f \))
Does A* fix the greedy problem?

![Diagram of a graph with initial state and goal nodes, showing the cost and heuristic values for each step.]

Between Greedy and A*

- Q: Is there a way to get some of the benefit of A* without requiring a heuristic?
- A: Uniform cost search
  - Like A*:
    - Maintains a priority queue
    - Expands node with lowest cost in queue
  - Unlike A*:
    - Ignores h(x), only uses actual costs incurred so far
- Same as assuming h(x)=c everywhere

A* is optimally efficient

- A* is optimally efficient: Any other optimal algorithm must expand at least the nodes A* expands (assuming both use the same, admissible h)
- Proof:
  - Besides solution, A* expands the nodes with g(n)+h(n) < A*
  - Any other optimal algorithm must expand at least these nodes (since there may be a better solution there)

Properties of Heuristics

- h2 dominates h1 if h2(x) >= h1(x) for all x
- (strict dominance if h2(x) > h1(x))
- Does this mean that h2 is better?
- Suppose you have multiple admissible heuristics. How do you combine them?
Designing heuristics

- One strategy for designing heuristics: relax the problem
- “Number of misplaced tiles” heuristic corresponds to relaxed problem where tiles can jump to any location, even if something else is already there
- “Sum of Manhattan distances” corresponds to relaxed problem where multiple tiles can occupy the same spot
- The ideal relaxed problem is
  - easy to solve computationally,
  - close in cost to the real problem
- Some programs can successfully automatically create heuristics