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
- 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

Overview

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

Problem Formulation

- Four components of a search problem
  - Initial State
  - Actions
  - Goal Test
  - Path Cost
- Optimal solution = lowest path cost to goal

Example: Path Planning

Find shortest route from one city to another using highways.

Example 8(15)-puzzle

Solution

Possible Start State

Goal State

Actions: UP, DOWN, RIGHT, LEFT
“Real” Problems

- Robot motion planning
- Drug design
- Logistics
  - Route planning
  - Tour Planning
- Assembly sequencing
- Internet searching

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 (distinct from states, which exist in the problem’s state space)
- Search tree: portion of state space visited so far
- Expansion: Generation of successors 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

`b=2`

Interesting details are in the implementation of Add-To-Queue

Generic Search Algorithm

```plaintext
Function Tree-Search(problem, Queuing-Fn)
    fringe = Make-Queue(Make-Node(Initial-State[problem]))
    loop do
        if empty(fringe) then failure
        node = pop(fringe)
        if Goal-Test[problem](state) then return node
        fringe = Add-To-Queue(fringe, expand(node, problem))
    end

Interesting details are in the implementation of Add-To-Queue
```

Evaluating Search Algorithms

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution?
- Time complexity
- Space complexity
**Uninformed Search: BFS**

Frontier is a FIFO

![BFS Diagram](image)

**BFS Properties**

- Completeness: \( \checkmark \)
- Optimality: \( \checkmark \)
- Time complexity: \( O(b^{d+1}) \)
- Space complexity: \( O(b^{d+1}) \)

**Uninformed Search: DFS**

Frontier is a LIFO

![DFS Diagram](image)

**DFS Properties**

- Completeness: \( \times \)
- Optimality: \( \times \)
- Time complexity: \( O(b^m) \) where \( m \) is search depth
- Space complexity: \( O(bm+1) \) where \( m \) is search depth

**Iterative Deepening**

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

**IDDFS**

![IDDFS Diagram](image)
IDDFS Properties

- Completeness: Y
- Optimality: Y
- Time complexity: \(O(b^d)\)
- Space complexity: \(O(bd+1)\) where \(m\) is search depth

IDDFS vs. BFS

Theorem: IDDFS makes no more than twice as many node expansions for a binary tree as BFS.

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

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

In the worst case, IDDFS does the following:

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

Bi-directional Search

Initial State

Goal

\[b^{d/2} + b^{d/2} << b^d\]

Issues with Bi-directional Search

- Uniqueness of goal
  - Suppose goal is on(block1,table)
  - Huge portion of state space may be considered a goal state (need to rework state space)
- 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?

What’s broken with greedy search?

A*

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

A* Properties

Theorem: A* is optimal if \( h(x) \) is admissable.

Proof: Suppose a suboptimal goal node \( g_2 \) appears on the fringe. If \( C^* \) is the optimal cost, \( f(g_2) > C^* \). Since \( h \) never overestimates the cost, there must exist some unexpanded node along the optimal path that has not yet been expanded. Thus, as long as we have not yet found the optimal path, we will continue to expand nodes.

Does A* fix the greedy problem?

Properties of Heuristics

- \( H_2 \) dominates \( h_1 \) if \( h_2(x) > h_1(x) \) for all \( x \)
- Does this mean that \( h_2 \) is better?
- Suppose you have multiple admissible heuristics. How do you combine them?

Developing Heuristics

- Is it hard to develop admissible heuristics?
- What are some heuristics for the 8 puzzle?
- What is a general strategy for developing admissible heuristics?
Other Issues

- Graphs
- Non-uniform costs
- Accuracy of heuristic
- A* is optimally efficient