CSE 473: Artificial Intelligence
Spring 2018

Problem Spaces & Search

Steve Tanimoto

With slides from:
Dieter Fox, Dan Weld, Dan Klein, Stuart Russell, Andrew Moore, Luke Zettlemoyer
Outline

- Search Problems

- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search

- Heuristic Search Methods
  - Best-First, Greedy Search
  - A*
Agent vs. Environment

- An **agent** is an entity that perceives and acts.
- A **rational agent** selects actions that maximize its utility function.
- Characteristics of the **percepts, environment, and action space** dictate techniques for selecting rational actions.
Types of Agents

- Reflex
- Goal oriented
- Utility-based
Goal Based Agents

- Plan ahead
- Ask “what if”
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Act on how the world WOULD BE
Types of Environments

- Fully observable vs. partially observable
- Single agent vs. multiagent
- Deterministic vs. stochastic
- Episodic vs. sequential
- Discrete vs. continuous
Search thru a
Problem Space (aka State Space)

• Input:
  ▪ Set of states
  ▪ Operators [and costs]
  ▪ Start state
  ▪ Goal state [test]

• Output:
  • Path: start → a state satisfying goal test
    [May require shortest path]
    [Sometimes just need a state that passes test]
Example: Traveling in Romania

- **State space:**
  - Cities

- **Successor function:**
  - Roads: Go to adjacent city with cost = distance

- **Start state:**
  - Arad

- **Goal test:**
  - Is state == Bucharest?

- **Solution?**
Example: Simplified Pac-Man

- **Input:**
  - A state space
  - A successor function
  - A start state
  - A goal test

- **Output:**
  - “N”, 1.0
  - “E”, 1.0
State Space Sizes?

- **Search Problem:**
  Eat all of the food

- **Pacman positions:**
  \[10 \times 12 = 120\]

- **Pacman facing:**
  up, down, left, right

- **Food configurations:** \(2^{30}\)

- **Ghost1 positions:** 12

- **Ghost 2 positions:** 11

\[120 \times 4 \times 2^{30} \times 12 \times 11 = 6.8 \times 10^{13}\]
State Space Graphs

- **State space graph:**
  - Each node is a state
  - The successor function is represented by arcs
  - Edges may be labeled with costs

- In a search graph, each state occurs only once!

- We can rarely build this graph in memory (so we don’t)

*Ridiculously tiny search graph for a tiny search problem*
Search Trees

- A search tree:
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, correspond to PLANS to those states
  - Edges are labeled with actions and costs
  - For most problems, we can never actually build the whole tree

This is now / start
Possible futures
State Space Graphs vs. Search Trees

Each NODE in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.
State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!
Tree Search
Search Example: Romania
- **Search:**
  - Expand out potential plans (tree nodes)
  - Maintain a *fringe* of partial plans under consideration
  - Try to expand as few tree nodes as possible
General Tree Search

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?

```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
    end
```
Tree Search Example
Depth-First Search
Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack
Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack
Search Algorithm Properties
Search Algorithm Properties

- **Complete**: Guaranteed to find a solution if one exists?
- **Optimal**: Guaranteed to find the least cost path?
- **Time complexity**?
- **Space complexity**?

**Cartoon of search tree:**
- $b$ is the branching factor
- $m$ is the maximum depth
- Solutions at various depths

**Number of nodes in entire tree?**
- $1 + b + b^2 + \ldots + b^m = O(b^m)$
Depth-First Search (DFS) Properties

- What nodes does DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If \( m \) is finite, takes time \( O(b^m) \)

- How much space does the fringe take?
  - Only has siblings on path to root, so \( O(bm) \)

- Is it complete?
  - \( m \) could be infinite, so only if we prevent cycles

- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost
Breadth-First Search
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be \( d \)
  - Search takes time \( O(b^d) \)

- How much space does the fringe take?
  - Has roughly the last tier, so \( O(b^d) \)

- Is it complete?
  - \( d \) must be finite if a solution exists, so yes!

- Is it optimal?
  - Only if costs are all 1 (more on costs later)
DFS vs BFS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>w/ Path Checking</td>
<td>N unless finite</td>
<td>N</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
</tbody>
</table>
Memory a Limitation?

Suppose:

- 4 GHz CPU
- 32 GB main memory
- 100 instructions / expansion
- 5 bytes / node

- 40 M expansions / sec
  - Memory filled in 160 sec ... 3 min
Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>O(b^d)</td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>O(b^d)</td>
</tr>
</tbody>
</table>
BFS vs. Iterative Deepening

- For b = 10, d = 5:
  - BFS = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - IDS = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
  - Overhead = (123,456 - 111,111) / 111,111 = 11%
  - Memory BFS: 100,000; IDS: 50
Costs on Actions

Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
Expand cheapest node first:
Fringe is a priority queue
Uniform Cost Search

Strategy: expand a cheapest node first:
Fringe is a priority queue (priority: cumulative cost)
Uniform Cost Search (UCS) Properties

- **What nodes does UCS expand?**
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs $C^*$ and arcs cost at least $\varepsilon$, then the “effective depth” is roughly $C^*/\varepsilon$
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)

- **How much space does the fringe take?**
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

- **Is it complete?**
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- **Is it optimal?**
  - Yes!
Uniform Cost Search

- **Strategy:** expand lowest path cost

- **The good:** UCS is complete and optimal!

- **The bad:**
  - Explores options in every “direction”
  - No information about goal location
## Uniform Cost Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^{C*/\epsilon})$</td>
<td>$O(b^{C*/\epsilon})$</td>
</tr>
</tbody>
</table>

$C^{*}/\epsilon$ tiers
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one
The One Queue

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object
To Do:

- Look at the course website:
  - http://courses.cs.washington.edu/courses/cse473/18sp/
- Do the readings (Ch 3)
- Do Project 0 if new to Python
- Start Project 1.