CSE 473: Introduction to Artificial Intelligence
Fall 2019
Hanna Hajishirzi
Problem Spaces and Search

slides adapted from
Dan Klein, Pieter Abbeel ai.berkeley.edu
And Dan Weld, Luke Zettelmoyer
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
Agents that Plan
Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world’s current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS

- Can a reflex agent be rational?
Video of Demo Reflex Optimal
Video of Demo Reflex Odd
Planning Agents

- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE

- Optimal vs. complete planning

- Planning vs. replanning
Video of Demo Replanning
Video of Demo Mastermind
Search Problems
Search Problems

- A search problem consists of:
  - A state space
  - A successor function (with actions, costs)
  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state
Search: it is not just for agents

- Route Planning
- Hardware verification
- Planning optimal repair sequences

- Search: Modeling the world
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?**
What’s in a State Space?

The **world state** includes every last detail of the environment

![Diagram of Pac-Man level]

A **search state** keeps only the details needed for planning (abstraction)

- **Problem: Pathing**
  - States: \((x,y)\) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is \((x,y)\)=END

- **Problem: Eat-All-Dots**
  - States: \(\{(x,y), \text{dot booleans}\}\)
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

The world state includes every last detail of the environment
### Input:
- **Set of states**
- **Operations**
- **Start state**
- **Goal state (test)**

### Output:

This lecture is about search algorithms.
State Space Sizes?

- **World state:**
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- **How many**
  - **World states?**
    - $120 \times (2^{30}) \times (12^2) \times 4$
  - **States for pathing?**
    - 120
  - **States for eat-all-dots?**
    - $120 \times (2^{30})$
State Representation

- Real-world applications:
  - Requires approximations and heuristics
  - Need to design state representation so that search is feasible
    - Only focus on important aspects of the state
    - E.g., Use features to represent world states
Problem: eat all dots while keeping the ghosts perma-scared

What does the state space have to specify?

- (agent position, dot booleans, power pellet booleans, remaining scared time)
State Space Graphs and Search Trees
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

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

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

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

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea
Search Trees

- A search tree:
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

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

We construct both on demand—and we construct as little as possible.

Each NODE in the search tree is an entire PATH in the state space graph.
State Space Graphs vs. Search Trees

- Nodes in state space graphs are problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors

- Nodes in search trees are plans
  - Represent a plan (sequence of actions) which results in the node’s state
  - Have a problem state and one parent, a path length, a depth & a cost
  - The same problem state may be achieved by multiple search tree nodes

**Problem States**

- Depth 5
- Depth 6

**Search Nodes**

- Parent
- Action
- Node
- Depth 5
- Depth 6
State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?
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
Searching with a Search Tree

- **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

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

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?
Example: Tree Search
Example: Tree Search

SDERFG

s
s \rightarrow d
s \rightarrow e
s \rightarrow p
s \rightarrow d \rightarrow b
s \rightarrow d \rightarrow c
s \rightarrow d \rightarrow e
s \rightarrow d \rightarrow e \rightarrow h
s \rightarrow d \rightarrow e \rightarrow r
s \rightarrow d \rightarrow e \rightarrow r \rightarrow f
s \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow c
s \rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow G
Search Algorithms

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

- Heuristic Search Methods
  - Best First / Greedy Search
  - A*
Depth-First Search
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 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 \( O(b^m) \)

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

- **Is it optimal?**
  - No, it finds the “leftmost” solution, regardless of depth or cost
**DFS**

<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>No</td>
<td>No</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

Infinite paths make DFS incomplete…
- How can we fix this?
  - Check new nodes against path from S

Infinite search spaces still a problem
- If the left subtree has unbounded depth
DFS

Algorithm | Complete | Optimal | Time | Space
--- | --- | --- | --- | ---
DFS w/ Path Checking | Y if finite | N | $O(b^m)$ | $O(bm)$
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 $s$
  - Search takes time $O(b^s)$

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

- **Is it complete?**
  - $s$ must be finite if a solution exists, so yes!

- **Is it optimal?**
  - Only if costs are all 1 (more on costs later)
### 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>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</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>

Diagram:
- **d tiers**
  - 1 node
  - $b$ nodes
  - $b^2$ nodes
  - $b^d$ nodes
  - $b^m$ nodes
Quiz: DFS vs BFS
DFS vs BFS

- When will BFS outperform DFS?

- When will DFS outperform BFS?
Video of Demo Maze Water DFS/BFS (part 1)
Video of Demo Maze Water DFS/BFS (part 2)
Iterative Deepening

- Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution...
  - Run a DFS with depth limit 2. If no solution...
  - Run a DFS with depth limit 3. ..... 

- Isn’t that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!
BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.
Uniform Cost Search
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! (Proof next lecture via A*)
Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location

- We’ll fix that soon!
Video of Demo Empty UCS
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 1)
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 2)
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 3)
Example: Pancake Problem

- Action: Flip over top $n$ pancakes
- Cost: Number of pancakes
Example: Pancake Problem

BOUND S FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation \( \sigma \) of the integers from 1 to \( n \), let \( f(\sigma) \) be the smallest number of prefix reversals that will transform \( \sigma \) to the identity permutation, and let \( f(n) \) be the largest such \( f(\sigma) \) for all \( \sigma \) in (the symmetric group) \( S_n \). We show that \( f(n) \leq (5n + 5)/3 \), and that \( f(n) \geq 17n/16 \) for \( n \) a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function \( g(n) \) is shown to obey \( 3n/2 - 1 \leq g(n) \leq 2n + 3 \).
Pancake Problem

- State graph with costs as weights
Uniform Cost Search

Action: flip top two
Cost: 2

Action: flip all four
Cost: 4

Path to reach goal:
Flip four, flip three
Total cost: 7
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.
Search and Models

- Search operates over models of the world
  - The agent doesn’t actually try all the plans out in the real world!
  - Planning is all “in simulation”
- Your search is only as good as your models…
Search Gone Wrong?
To Do:

- Try python practice (PS0)
  - Won’t be graded
- Check out PS1 in the webpage
  - Start ASAP
  - Submission: Canvas
- Website:
  - Do readings for search algorithms
  - Try this search visualization tool
    - http://qiao.github.io/PathFinding.js/visual/