Outline

- Agents that Plan Ahead
- Search Problems
  - Uninformed Search Methods (part review for some)
    - Depth-First Search
    - Breadth-First Search
    - Uniform-Cost Search
  - Heuristic Search Methods (new for all)
    - Best First / Greedy Search
Review: Agents

An agent:
- Perceives and acts
- Selects actions that maximize its utility function
- Has a goal

Environment:
- Input and output to the agent

Search -- the environment is:
fully observable, single agent, deterministic, static, discrete
Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - Do not consider the future consequences of their actions
  - Act on how the world IS

- Can a reflex agent achieve goals?
Goal Based Agents

- **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
Search thru a
Problem Space / State Space

• **Input:**
  - Set of states
  - Successor Function [and costs - default to 1.0]
  - Start state
  - Goal state [test]

• **Output:**
  - Path: start ⟷ a state satisfying goal test
  - [May require shortest path]
  - [Sometimes just need state passing test]
Example: Simplified Pac-Man

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

- **Output:**
  - N, 1.0
  - E, 1.0
Ex: Route Planning: Romania → Bucharest

- **Input:**
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)

- **Output:**
Example: N Queens

- **Input:**
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)

- **Output**
Input:
- Set of states
- Operators [and costs]
- Start state
- Goal state (test)

Output:

\[
\begin{align*}
\partial_r^2 u &= - \left[ E' - \frac{l(l+1)}{r^2} - r^2 \right] u(r) \\
E^{-2s} \left( \partial_s^3 - \partial_s^1 \right) u(s) &= - \left[ E' - l(l+1)E^{-2s} - e^{2s} \right] u(s) \\
e^{-2s} \left[ e^{s/2} \left( e^{-s/2} u(s) \right)'''' - \frac{1}{4} u \right] &= - \left[ E' - l(l+1)e^{-2s} - e^{2s} \right] u(s) \\
e^{-2s} \left[ e^{s/2} \left( e^{-s/2} u(s) \right)''' \right] &= - \left[ E' - \left( l + \frac{1}{2} \right)^2 e^{-2s} - e^{2s} \right] u(s) \\
v'' &= e^{2s} \left[ E' - \left( l + \frac{1}{2} \right)^2 e^{-2s} - e^{2s} \right] v
\end{align*}
\]
What is in State Space?

- **A world state** includes every detail of the environment.

- **A search state** includes only details needed for planning.

**Problem: Pathing**
- States: \{x,y\} locations
- Actions: NSEW moves
- Successor: update location
- Goal: is \((x,y)\) End?

**Problem: Eat-all-dots**
- States: \{(x,y), dot booleans\}
- Actions: NSEW moves
- Successor: update location and dot boolean
- Goal: dots all false?
State Space Sizes?

- World states:
- Pacman positions: $10 \times 12 = 120$
- Pacman facing: up, down, left, right
- Food Count: 30
- Ghost positions: 12
State Space Sizes?

- How many?
- World State:
  \[120 \times (2^{30}) \times (12^2) \times 4\]
- States for Pathing:
  120
- States for eat-all-dots:
  \[120 \times (2^{30})\]
Problem: eat all dots while keeping the ghosts perma-scared
What does the state space have to specify?
State Space Graphs

- State space graph:
  - Each node is a state
  - The successor function is represented by arcs
  - Edges may be labeled with costs
- We can rarely build this graph in memory (so we don’t)
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
Example: Tree Search

State Graph:

What is the search tree?

Ridiculously tiny search graph for a tiny search problem
State 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 problem graph.
States vs. Nodes

- 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

Search Nodes

- Depth 5
- Depth 6
Quiz: State 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!
Building Search Trees

- **Search:**
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - 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
```

Detailed pseudocode is in the book!
Search Methods

- Uninformed Search Methods (part review for some)
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search

- Heuristic Search Methods (new for all)
  - Best First / Greedy Search
Review: Depth First Search

Strategy: expand deepest node first

Implementation:
Fringe is a LIFO queue (a stack)
Review: Depth First Search

Expansion ordering:
(d,b,a,c,a,e,h,p,q,q,r,f,c,a,G)
Review: Breadth First Search

Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue
Review: Breadth First Search

Expansion order:

(S, d, e, p, b, c, e, h, r, q, a, a, h, r, p, q, f, p, q, f, q, c, G)
Search Algorithm Properties

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

Variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of states in the problem</td>
</tr>
<tr>
<td>b</td>
<td>The maximum branching factor B (the maximum number of successors for a state)</td>
</tr>
<tr>
<td>C*</td>
<td>Cost of least cost solution</td>
</tr>
<tr>
<td>d</td>
<td>Depth of the shallowest solution</td>
</tr>
<tr>
<td>m</td>
<td>Max depth of the search tree</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)

* Or graph search – next lecture.
### 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 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>

- BFS is **optimal** when searching for the shortest path in an unweighted graph.
- DFS with Path Checking is **complete** and **optimal** for unweighted graphs, but has higher time complexity than BFS.

**Diagram:**

- The diagram illustrates a tree structure with \(b\) nodes at each tier, leading to a total of \(b^d\) nodes at depth \(d\).
- The number of nodes grows exponentially with depth.

**Example:**

- With \(b = 2\) and \(d = 3\), we have:
  - 1 node
  - 2 nodes
  - 4 nodes
  - 8 nodes
  - 16 nodes

- \(b^m\) nodes at depth \(m\).
Comparisons

- When will BFS outperform DFS?
- When will DFS outperform BFS?
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>
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
Best-First Search

- Generalization of breadth-first search
- *Priority* queue of nodes to be explored
- Cost function $f(n)$ applied to each node

Add initial state to priority queue
While queue not empty
  Node = head(queue)
  If goal?(node) then return node
  Add children of node to queue
A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pq.push(key, value)</code></td>
<td>inserts (key, value) into the queue.</td>
</tr>
<tr>
<td><code>pq.pop()</code></td>
<td>returns the key with the lowest value, and removes it from the queue.</td>
</tr>
</tbody>
</table>

- You can decrease a key’s priority by pushing it again.
- Unlike a regular queue, insertions aren’t constant time, usually $O(\log n)$.
- We’ll need priority queues for cost-sensitive search methods.
Expand cheapest node first:
Fringe is a priority queue
Uniform Cost Search

Expansion order:
(S, p, d, b, e, a, r, f, e, G)
# 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*/\varepsilon})</td>
<td>O(b^{C*/\varepsilon})</td>
</tr>
</tbody>
</table>

\[ C^{*/\varepsilon} \text{ tiers} \]
Uniform Cost Issues

- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one
Search Heuristics

- Any estimate of how close a state is to a goal
- Designed for a particular search problem

- Examples: Manhattan distance, Euclidean distance
Heuristics
Best First / Greedy Search

Best first with $f(n) = \text{heuristic estimate of distance to goal}$
Best First / Greedy Search

- Expand the node that seems closest…

- What can go wrong?
Best First / Greedy Search

- A common case:
  - Best-first takes you straight to the (wrong) goal

- Worst-case: like a badly-guided DFS in the worst case
  - Can explore everything
  - Can get stuck in loops if no cycle checking

- Like DFS in completeness (finite states w/ cycle checking)