CS 343H: Artificial Intelligence

Lecture 3: Uninformed Search
Tues 1/21/14

Slides courtesy of Dan Klein at UC-Berkeley
Unless otherwise noted
Announcements

- PS0 due this Thurs 1/23 by 11:59 pm
- All remaining reading response deadlines are firm
- Printing slides before lecture
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
Recall: Rational Agents

- 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.
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?
Planning 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
- Consider how the world WOULD BE
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
Example: Romania

- **State space:**
  - Cities

- **Successor function:**
  - Roads: Go to adj city with cost = dist

- **Start state:**
  - Arad

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

- **Solution?**
What’s in a State Space?

The world state specifies every last detail of the environment.

A search state keeps only the details needed (abstraction).

- **Problem 1: Pathing**
  - States: \((x,y)\) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is \((x,y) = \text{END}\)

- **Problem 2: Eat-All-Dots**
  - States: \{\((x,y)\), dot booleans\}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false
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 Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes: abstracted world configurations
  - Arcs: successors (action results)
  - Goal test is set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this graph in memory (so we don’t)
A search tree:
- This is a “what if” tree of plans and outcomes
- Start state at the root node
- Children correspond to successors
- Nodes contain states, correspond to PLANS to those states
- For most problems, we can never actually build the whole tree
Recall: Romania example
Searching with a search tree

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

Detailed pseudocode is in the book!
Example: Tree Search

Fringe (potential plans)  Tree
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.
Depth First Search

Strategy: expand deepest node first

State graph

Search tree
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>$n$</th>
<th>Number of states in the problem (huge)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>The average branching factor $B$ (the average number of successors)</td>
</tr>
<tr>
<td>$C^*$</td>
<td>Cost of least cost solution</td>
</tr>
<tr>
<td>$s$</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?

<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>Depth First Search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph with START, a, b, and GOAL nodes connected with arrows]
DFS

- With cycle checking, DFS is complete.*

![Diagram showing DFS](image)

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- When is DFS optimal?

* Or graph search – next lecture.
Breadth First Search

Strategy: expand shallowest node first
When is BFS optimal?
### BFS complexity: concretely

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 ms</td>
<td>107 KB</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 ms</td>
<td>10.6 MB</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 s</td>
<td>1 GB</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 h</td>
<td>10 TB</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 d</td>
<td>1 PB</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 y</td>
<td>99 PB</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 y</td>
<td>10 EB</td>
</tr>
</tbody>
</table>

**Figure 3.13** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- Russell & Norvig
Comparisons

- When will BFS outperform DFS?

- When will DFS outperform BFS?
Iterative Deepening

Iterative deepening: BFS using 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 w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></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.
Uniform Cost Search

Expand cheapest node first:
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>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq.push(key, value)</td>
<td>inserts <em>(key, value)</em> into the queue.</td>
</tr>
<tr>
<td>pq.pop()</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
Uniform Cost Search

- Remember: explores increasing cost contours
# 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>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCS</td>
<td></td>
<td></td>
<td></td>
<td></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
Search Gone Wrong?
Summary

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Next time: informed search, A*