CSE 573: Artificial Intelligence

Problem Spaces & Search

Dan Weld

With slides from
Dan Klein, Stuart Russell, Andrew Moore, Luke Zettlemoyer, Dana Nau…
Logistics

- Read Ch 3

- Form 2-person teams.
  - Post on forum if you want a partner

- Start PS1
Outline

- Search Problems

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

- Heuristic Search Methods
- Heuristic Generation
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.
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: It’s not just for Agents

**Hardware verification**

**Planning optimal repair sequences**

![Hardware diagram](image1)

![Space Shuttle](image2)
Search thru a Problem Space (aka State Space)

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

- **Output:**
  - Path: start $\Rightarrow$ a state satisfying goal test
    - [May require shortest path]
    - [Sometimes just need a state that passes test]
Example: Simplified Pac-Man

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

- **Output:**
  - “N”, 1.0
  - “E”, 1.0
Ex: Route Planning: Arad \(\rightarrow\) Bucharest

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

**Output:**

Different operators may be applicable in different states
Ex: Dock Worker Robots

- A harbor with several locations
  - e.g., docks, docked ships, storage areas, parking areas
- Containers
  - going to/from ships
- Robot vehicles
  - can move containers
- Cranes
  - can load and unload containers
- Multiple robots can operate at the same time
- Move, load & other actions have different durations

Slide adapted from Dana Nau “Automated Planning”, licensed under creative commons NC share alike
Dock Worker 2

Input:

- Set of states
  
  Partially specified plans

- Operators [and costs]
  
  Plan modification operators

- Start state
  
  The null plan (no actions)

- Goal test
  
  A plan which provably achieves the desired world configuration

Slide adapted from Dana Nau “Automated Planning”, licensed under creative commons NC share alike
Plan Space

Blue boxes are plans = states in search space
Operators modify plans
Successors(p) = all possible ways of modifying p
Multiple Problem Spaces

Real World

States of the world (e.g. loading dock configurations)
Actions (take one world-state to another)

Robot’s Head

• **Problem Space 1**
  - PS states =
    - models of world states
  - Operators =
    - models of actions

• **Problem Space 2**
  - PS states =
    - partially spec. plan
  - Operators =
    - plan modificat’n ops
Algebraic Simplification

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

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

- **Output:**
State Space Graphs

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

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

\[120 \times 4 \times 2^{30} \times 12 \times 11 = 6.8 \times 10^{13}\]
Search Methods

- **Blind Search**
  - Depth first search
  - Breadth first search
  - Iterative deepening search
  - Uniform cost search

- **Local Search**

- **Informed Search**

- **Constraint Satisfaction**

- **Adversary Search**
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?
We construct both on demand – and we construct as little as possible.

Each NODE in the search tree denotes an entire PATH in the problem graph.

State Graphs vs. Search Trees
States vs. Nodes

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

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

- **Search:**
  - Expand out possible nodes (plans) in the tree
  - Maintain a *fringe* of unexpanded nodes
  - Try to expand as few nodes as possible
General Tree Search

Important ideas:
- Fringe (leaves of tree)
- Expansion (adding successors of a leaf)
- Exploration strategy

which fringe node to expand next?

Detailed pseudocode is in the book!

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
Review: Depth First Search

**Strategy:** expand **deepest** node first

**Implementation:**
Fringe is a stack - LIFO
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 queue - FIFO
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>
**Depth-First Search**

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

<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>No</td>
<td>No</td>
<td>O(b^m)</td>
</tr>
</tbody>
</table>

- START
- GOAL
- d depth of solution
- m max depth of tree
DFS Search (w/ cycle checking)

- Complete: Y if finite
- Optimal: N
- Time: $O(b^m)$
- Space: $O(b \cdot m)$

Only if finite tree
## BFS Tree 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>$N$ unless finite</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>

* Assuming finite branching factor

![Diagram](image_url)

- $d$ tiers
- 1 node
- $b$ nodes
- $b^2$ nodes
- $b^d$ nodes
- $b^m$ nodes
Memory a Limitation?

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

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

- DFS with limit; incrementally grow limit
- Evaluation
Iterative Deepening Search

- DFS Tree Search with limit; incrementally grow limit
- Evaluation
Iterative Deepening Search

- DFS Tree Search with limit; incrementally grow limit
- Evaluation
  - Complete?
- Time Complexity?
- Space Complexity?
Iterative Deepening Search

- DFS with limit; incrementally grow limit
- Evaluation
  - Complete?
    - Yes *
  - Time Complexity?
    - $O(b^d)$
  - Space Complexity?
    - $O(bd)$

* Assuming branching factor is finite

Important Note: no cycle checking necessary!
## Cost of Iterative Deepening

<table>
<thead>
<tr>
<th>b</th>
<th>ratio ID to DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>1.2</td>
</tr>
<tr>
<td>25</td>
<td>1.08</td>
</tr>
<tr>
<td>100</td>
<td>1.02</td>
</tr>
</tbody>
</table>
### Speed

Assuming 10M nodes/sec & sufficient memory

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Puzzle</td>
<td>$10^5$</td>
<td>.01 sec</td>
<td>$10^5$</td>
<td>.01 sec</td>
</tr>
<tr>
<td>2x2x2 Rubik’s</td>
<td>$10^6$</td>
<td>.2 sec</td>
<td>$10^6$</td>
<td>.2 sec</td>
</tr>
<tr>
<td>15 Puzzle</td>
<td>$10^{13}$</td>
<td>6 days</td>
<td>$10^{17}$</td>
<td>20k yrs</td>
</tr>
<tr>
<td>3x3x3 Rubik’s</td>
<td>$10^{19}$</td>
<td>68k yrs</td>
<td>$10^{20}$</td>
<td>574k yrs</td>
</tr>
<tr>
<td>24 Puzzle</td>
<td>$10^{25}$</td>
<td>12B yrs</td>
<td>$10^{37}$</td>
<td>$10^{23}$ yrs</td>
</tr>
</tbody>
</table>

Why the difference?

- Rubik has higher branch factor
- 15 puzzle has greater depth

# of duplicates

Slide adapted from Richard Korf presentation
Search Methods

- Depth first search (DFS)
- Breadth first search (BFS)
- Iterative deepening depth-first search (IDS)
Search Methods

- Depth first search (DFS)
- Breadth first search (BFS)
- Iterative deepening depth-first search (IDS)
- Best first search
- Uniform cost search (UCS)
- Greedy search
- A*
- Iterative Deepening A* (IDA*)
- Beam search
- Hill climbing
Blind vs Heuristic Search

- Costs on Actions
- Heuristic Guidance

Separable Issues, but usually linked.
Costs on Actions

Objective: Path with smallest overall cost
What will BFS return?

... 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
- Fringe = *Priority* queue of nodes to be explored
- Cost function $f(n)$ applied to each node
In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- Very simple fix: never expand a state type twice

```plaintext
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```
Some Hints

- **On small problems**
  - Graph search almost always better than tree search
  - Implement your closed list as a dict or set!

- **On many real problems**
  - Storage space is a huge concern
  - Graph search impractical
Best-First Search

- Generalization of breadth-first search
- Fringe = *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 *new* children of node to queue

“expanding the node”
Old Friends

- **Breadth First** =
  - Best First
  - with $f(n) = \text{depth}(n)$

- **Dijkstra’s Algorithm (Uniform cost)** =
  - Best First
  - with $f(n) = \text{the sum of edge costs from start to n}$
Uniform Cost Search

Best first, where

\[ f(n) = \text{“cost from start to } n\text{”} \]

aka “Dijkstra’s Algorithm”
Uniform Cost Search

Expansion order:
S, p, d, b, e, a, r, f, e, G

Cost contours (not all shown)
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 if finite</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>
<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>

$C^*$ = Optimal cost

$\varepsilon$ = Minimum cost of an action
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
What is a Heuristic?

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

Examples: Manhattan distance: $10 + 5 = 15$
Euclidean distance: 11.2
What is a **Heuristic**?

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

Actual distance to goal: $2 + 4 + 2 + 1 + 8 = 15$
Greedy Search

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

Expand the node that seems closest...

What can go wrong?
Greedy Search

- Common case:
  - Best-first takes you straight to a (suboptimal) goal

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

- Like DFS in completeness
  - Complete w/ cycle checking
  - If finite # states
A* Search

Hart, Nilsson & Rafael 1968

Best first search with $f(n) = g(n) + h(n)$

- $g(n) = \text{sum of costs from start to } n$
- $h(n) = \text{estimate of lowest cost path } n \rightarrow \text{goal}$
  \[ h(\text{goal}) = 0 \]

Can view as cross-breed:

- $g(n) \sim \text{uniform cost search}$
- $h(n) \sim \text{greedy search}$

Best of both worlds…