AI Adjacent Fields

- Philosophy:
  - Logic, methods of reasoning
  - Mind as physical system
  - Foundations of learning, language, rationality

- Mathematics
  - Formal representation and proof
  - Algorithms, computation, (un)decidability, (in)tractability
  - Probability and statistics

- Psychology
  - Adaptation
  - Phenomena of perception and motor control
  - Experimental techniques (psychophysics, etc.)

- Economics: formal theory of rational decisions
- Linguistics: knowledge representation, grammar
- Neuroscience: physical substrate for mental activity
- Control theory:
  - Homeostatic systems, stability
  - Simple optimal agent designs

This slide deck courtesy of Dan Klein at UC Berkeley
How Much of AI is Math?

- A lot, but not right away
- Understanding probabilities will help you a great deal
- In later weeks, there will be many more equations
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?
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
  - 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: 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
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
  - For every search problem, there’s a corresponding state space graph
  - The successor function is represented by arcs

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

- 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
- **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
State Graphs vs. Search Trees

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

We construct both on demand – and we construct as little as possible.
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
Review: Depth First Search

Strategy: expand deepest node first

Implementation: Fringe is a LIFO stack
Review: Breadth First Search

**Strategy:** expand shallowest node first

**Implementation:** Fringe is a FIFO queue
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?
With cycle checking, DFS is complete.*

When is DFS optimal?

* 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>$O(b^{m+1})$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>$O(b^{s+1})$</td>
<td>$O(b^s)$</td>
</tr>
</tbody>
</table>

- **Complete**: Indicates whether the algorithm can complete the task.
- **Optimal**: Indicates whether the algorithm is optimal.
- **Time**: Big O notation for time complexity.
- **Space**: Big O notation for space complexity.

#### Diagram
- **s tiers**: Number of tiers in the structure.
- **1 node**: Top node.
- **b nodes**: Nodes in the first tier.
- **$b^2$ nodes**: Nodes in the second tier.
- **$b^s$ nodes**: Nodes in the s-th tier.
- **$b^m$ nodes**: Total nodes in the entire structure.

---

- **When is BFS optimal?**
Comparisons

- When will BFS outperform DFS?

- When will DFS outperform BFS?
Iterative Deepening

Iterative deepening: BFS using DFS as a subroutine:

3. Do a DFS which only searches for paths of length 1 or less.
4. If “1” failed, do a DFS which only searches paths of length 2 or less.
5. 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>Y</td>
<td>N</td>
<td>O(b^{m+1})</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>O(b^{s+1})</td>
<td>O(b^s)</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>O(b^{s+1})</td>
<td>O(b^s)</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. We will quickly cover an algorithm which does find the least-cost path.
Uniform Cost Search

Expand cheapest node first:

Fringe is a priority queue (priority: cumulative cost)
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>pq.push(key, value)</th>
<th>inserts (key, value) into the queue.</th>
</tr>
</thead>
<tbody>
<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

<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>$O(b^{m+1})$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^{s+1})$</td>
<td>$O(b^s)$</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>

$C*/\varepsilon$ tiers

* UCS can fail if actions can get arbitrarily cheap
Uniform Cost Search

- What will UCS do for this graph?

- What does this mean for completeness?
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 Heuristics

- Any *estimate* of how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance

![Diagram of a maze with estimated distances marked: 5, 10, and 11.2]
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)
Search Gone Wrong?
Extra Work?

- Failure to detect repeated states can cause exponentially more work (why?)
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
end
```

- Can this wreck completeness? Why or why not?
- How about optimality? Why or why not?
Some Hints

- Graph search is almost always better than tree search (when not?)

- Implement your closed list as a dict or set!

- Nodes are conceptually paths, but better to represent with a state, cost, last action, and reference to the parent node
# Best First Greedy Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy Best-First Search</td>
<td>$\exists^*$</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
</tbody>
</table>

- **What do we need to do to make it complete?**
- **Can we make it optimal?** Next class!
Best First / Greedy Search

- Strategy: expand the closest node to the goal

[Diagram showing a graph with nodes labeled with distances and heuristics, and arrows indicating paths with corresponding costs.]
Example: Tree Search