Artificial Intelligence
Strategy Making as Searching

Lecture 2
Week 2
Strategy Making

- **Objective**: Achieving the *goal*
- **Environment**: Single Agent
  - Simple world (static, known)
    - Adversarial world (tic-tac-toe, chess)
    - Non-adversarial (shortest path to a point in a known city)
  - Complex world (dynamic, unknown)
    - Adversarial world (Road Rash)
    - Non-adversarial world (Angry Birds)
- **Environment**: Multiple Agents
  - Simple world
    - Adversarial world (Robot Soccer)
    - Non-adversarial world (Cargo Transportation robots)
  - Complex world
    - Adversarial world (Contra)
    - Non-adversarial world (Robot Rescue)
Strategy Making as Search

- Goal defined as *state* or *action*
  - *State goal*
    - How can the agent *reach* the world state?
    - How can the agent *create* the world state?
  - *Action goal*
    - How can the agent *reach* a particular “decided” action?
    - How can the agent *create* a particular “decided” action?
  - *Should the agent attach a value to the path to the goal?*
Search Problem

- The world:

- The State Space:

- Successor Function:

- Start State & Goal State check condition

- Solution: Sequence of actions (plan) from start to goal
Example: Romania

- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with cost = dist
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?
Search Problem: Graphical Representation

- 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)
The Pacman Problem

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: \((x, y) = \text{END}\)

- **Problem: 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 Size

- **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?
    \[120 \times (2^{30})\]
Search Tree (What if Tree)

- Nodes – States
  - Also, *plan till now*
- Root node – Start State
- Successor function produces children
- Impossible to generate complete tree in most cases!
Search Strategy: Outline

- **Expand** out possible *plans*
- Maintain a *fringe* of unexpanded plans (what the agent has not yet explored)
  - *Which fringe node to explore?*
- Try to get to the goal without too much expansion

```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
```
The Romania Problem

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

- Search is completely *uninformed*
- Strategy: *Expand* deepest node first
- Philosophy: Dig deep to get to the goal ("Dilli dur hai!")
- Implementation (Data Structure): Keep the *fringe* on a stack
Strategy B: Breadth First Search

- Search is completely \textit{uninformed}
- Strategy: Expand shallowest node first
- Philosophy: Dig low to get to the goal ("Dilli nasdeek hai!")
- Implementation (Data Structure): Keep the \textit{fringe} on a \textit{queue}
### Search Algorithm Analysis

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Number of states in the problem (huge)</td>
</tr>
<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>
DFS - Analysis

<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</td>
<td>N</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

- Depth First Search

- Infinite paths make DFS incomplete…
- How can we fix this?
DFS: Breaking the cycle

- Cycle checking

<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>$O(b^{n+1})$</td>
<td>$O(b^m)$</td>
</tr>
</tbody>
</table>

- Why not optimal?

Max stack size at any point
When is BFS optimal?

Max queue size at any point
Which strategy is better
DFS or BFS?
Strategy C: Iterative Deepening (Dig down but slow!)

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</td>
<td>Y</td>
<td>N</td>
<td>$O(b^{m+1})$</td>
<td>$O(b^m)$</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>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>
What if the search comes with a cost?
The Romania Problem with cost
Strategy D: Uniform Cost Search
(Jump Start from what looks good)

Expand cheapest node first:
Fringe is a priority queue
(priority: cumulative cost)

Why e before G?
What data structure should we use?
Priority 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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq.push(key, value)</td>
<td>inserts (key, value) 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.
## Strategy UCS Analysis

<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>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^{C^*/\varepsilon})$</td>
<td>$O(b^{C^*/\varepsilon})$</td>
</tr>
</tbody>
</table>

* UCS can fail if actions can get arbitrarily cheap

* Epsilon is minimum action cost

Why?

$C^*$ | Cost of least cost solution
What’s bad about UCS?

- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location
How is UCS different than Dijkstra’s?
How does they relate to Angry Birds?
How does they relate to Robot Soccer
- Attack?
How does they relate to Robot Soccer - Defense?