**Reminder**

- Only a very small fraction of AI is about making computers play games intelligently
- Recall: computer vision, natural language, robotics, machine learning, computational biology, etc.
- That being said: games tend to provide relatively simple example settings which are great to illustrate concepts and learn about algorithms which underlie many areas of AI
Reflex Agent

- 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
- Act on how the world IS
- Can a reflex agent be rational?

A reflex agent for pacman

While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food

4 actions: move North, East, South or West
A reflex agent for pacman (2)

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food

A reflex agent for pacman (3)

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food
  - But, if other options are available, exclude the direction we just came from
A reflex agent for pacman (4)

- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

A reflex agent for pacman (5)

- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from
Reflex Agent

- 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
- Act on how the world IS
- Can a reflex agent be rational?

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 Problems

- A search problem consists of:
  - A state space
  - A successor function
  - 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:**
  - 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), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false
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)

State Space Sizes?

- Search Problem: Eat all of the food
- Pacman positions: $10 \times 12 = 120$
- Food count: 30
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

Another 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

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?

```python
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
```

Example: Tree Search
State Graphs vs. Search Trees

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

We construct both on demand – and we construct as little as possible.

Review: Depth First (Tree) Search

Strategy: expand deepest node first

Implementation: Fringe is a LIFO stack
Review: Breadth First (Tree) Search

Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue

Search Tiers

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 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?

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^s)</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+1})</td>
</tr>
</tbody>
</table>

- When is BFS optimal?

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 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>N*</td>
<td>O(b^{s+1})</td>
<td>O(b^{s+1})</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>O(b^{s+1})</td>
<td>O(bs)</td>
</tr>
</tbody>
</table>

Costs on Actions

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 (Tree) Search

Expand cheapest node first:
Fringe is a priority queue

Priority Queue Refresher

- 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><code>pq.push(key, value)</code></td>
<td>inserts <em>(key, value)</em> 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
### Uniform Cost (Tree) Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time (in nodes)</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>N</td>
<td>O(b^{s+1})</td>
<td>O(b^{s+1})</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>O(b^{C*/e})</td>
<td>O(b^{C*/e})</td>
</tr>
</tbody>
</table>

* UCS can fail if actions can get arbitrarily cheap

### 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 Search Example

Search Heuristics

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

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)

Greedy

Uniform Cost
Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Best-first orders by goal proximity, or *forward cost* $h(n)$

$A^*$ Search orders by the sum: $f(n) = g(n) + h(n)$

Example: Teg Grenager

When should $A^*$ terminate?

- Should we stop when we enqueue a goal?

- No: only stop when we dequeue a goal
Is A* Optimal?

- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Admissible Heuristics

- A heuristic $h$ is **admissible** (optimistic) if:
  \[ h(n) \leq h^*(n) \]
  where $h^*(n)$ is the true cost to a nearest goal

- Examples:

- Coming up with admissible heuristics is most of what’s involved in using A* in practice.
Optimality of A*: Blocking

Proof:
- What could go wrong?
- We’d have to have to pop a suboptimal goal G off the fringe before G*
- This can’t happen:
  - Imagine a suboptimal goal G is on the queue
  - Some node n which is a subpath of G* must also be on the fringe (why?)
  - n will be popped before G

\[ f(n) = g(n) + h(n) \]
\[ g(n) + h(n) \leq g(G^*) \]
\[ g(G^*) < g(G) \]
\[ g(G) = f(G) \]
\[ f(n) < f(G) \]

Properties of A*

Uniform-Cost  
\[ f(n) = g(n) + h(n) \]
\[ g(n) + h(n) \leq g(G^*) \]
\[ g(G^*) < g(G) \]
\[ g(G) = f(G) \]
\[ f(n) < f(G) \]

A*  
\[ f(n) = g(n) + h(n) \]
\[ g(n) + h(n) \leq g(G^*) \]
\[ g(G^*) < g(G) \]
\[ g(G) = f(G) \]
\[ f(n) < f(G) \]
UCS vs A* Contours

- Uniform-cost expanded in all directions

- A* expands mainly toward the goal, but does hedge its bets to ensure optimality

Example: Explored States with A*

Heuristic: manhattan distance ignoring walls
<table>
<thead>
<tr>
<th>Comparison</th>
<th><img src="image1.png" alt="Image" /></th>
<th><img src="image2.png" alt="Image" /></th>
<th><img src="image3.png" alt="Image" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform Cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A star</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics.

- Often, admissible heuristics are solutions to *relaxed problems*, with new actions ("some cheating") available.

- Inadmissible heuristics are often useful too (why?)
Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?

---

8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- \( h(\text{start}) = 8 \)
- This is a \text{relaxed-problem} heuristic

<table>
<thead>
<tr>
<th>Average nodes expanded when optimal path has length...</th>
<th>UCS</th>
<th>TILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 steps</td>
<td>112</td>
<td>13</td>
</tr>
<tr>
<td>8 steps</td>
<td>6,300</td>
<td>39</td>
</tr>
<tr>
<td>12 steps</td>
<td>(3.6 \times 10^6)</td>
<td>227</td>
</tr>
</tbody>
</table>
8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total Manhattan distance
- Why admissible?

\[ h(\text{start}) = 3 + 1 + 2 + \ldots = 18 \]

<table>
<thead>
<tr>
<th>TILES</th>
<th>13</th>
<th>39</th>
<th>227</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANHATTAN</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>

8 Puzzle III

- How about using the actual cost as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What’s wrong with it?

- With A*: a trade-off between quality of estimate and work per node!
Trivial Heuristics, Dominance

- Dominance: $h_a \geq h_c$ if
  \[ \forall n : h_a(n) \geq h_c(n) \]

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
    \[ h(n) = \max(h_a(n), h_b(n)) \]
  - Trivial heuristics
    - Bottom of lattice is the zero heuristic (what does this give us?)
    - Top of lattice is the exact heuristic

Other A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work. Why?

Graph Search

- In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + list of expanded states (closed list)
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state is new

- Python trick: store the closed list as a set, not a list

- Can graph search wreck completeness? Why/why not?

- How about optimality?

---

Graph Search

- Very simple fix: never expand a state twice

```
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 — INSERT-ALL(Expand(node, problem), fringe)
  end
```

- Can this wreck completeness? Optimality?
Optimality of A* Graph Search

Proof:
- New possible problem: nodes on path to $G^*$ that would have been in queue aren’t, because some worse $n'$ for the same state as some $n$ was dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor which was on the queue when $n'$ was expanded
- Assume $f(p) < f(n)$
- $f(n) < f(n')$ because $n'$ is suboptimal
- $p$ would have been expanded before $n'$
- So $n$ would have been expanded before $n'$, too
- Contradiction!

Consistency

- Wait, how do we know parents have better f-values than their successors?
- Couldn’t we pop some node $n$, and find its child $n'$ to have lower f value?
- YES:

  - What can we require to prevent these inversions?
  - Consistency: $c(n, a, n') \geq h(n) - h(n')$
  - Real cost must always exceed reduction in heuristic
A* Graph Search Gone Wrong

State space graph

Search tree

C is already in the closed-list, hence not placed in the priority queue

Consistency

The story on Consistency:
• Definition:
  \( \text{cost(A to C)} + h(C) \geq h(A) \)
• Consequence in search tree:
  Two nodes along a path: \( N_A, N_C \)
  \( g(N_C) = g(N_A) + \text{cost(A to C)} \)
  \( g(N_C) + h(C) \geq g(N_A) + h(A) \)
• The f value along a path never decreases
• Non-decreasing f means you’re optimal to every state (not just goals)
Optimality Summary

- **Tree search:**
  - A* optimal if heuristic is admissible (and non-negative)
  - Uniform Cost Search is a special case (h = 0)

- **Graph search:**
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)

- **Consistency implies admissibility**
  - Challenge: Try to prove this.
  - Hint: try to prove the equivalent statement *not admissible implies not consistent*

- In general, natural admissible heuristics tend to be consistent

- Remember, costs are always positive in search!

Summary: A*

- A* uses both backward costs and (estimates of) forward costs

- A* is optimal with admissible heuristics

- Heuristic design is key: often use relaxed problems
A* Memory Issues → IDA*

## IDA* (Iterative Deepening A*)

1. set $f_{\text{max}} = 1$ (or some other small value)
2. Execute DFS that does not expand states with $f > f_{\text{max}}$
3. If DFS returns a path to the goal, return it
4. Otherwise $f_{\text{max}} = f_{\text{max}} + 1$ (or larger increment) and go to step 2

- Complete and optimal
- Memory: $O(bs)$, where $b$ – max. branching factor, $s$ – search depth of optimal path
- Complexity: $O(kb^s)$, where $k$ is the number of times DFS is called

Recap Search I

- Agents that plan ahead → formalization: Search
- Search problem:
  - States (configurations of the world)
  - Successor function: a function from states to lists of (state, action, cost) triples; drawn as a graph
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)
- Search Algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
Recap Search II

- Tree Search vs. Graph Search
- Priority queue to store fringe: different priority functions → different search method
  - Uninformed Search Methods
    - Depth-First Search
    - Breadth-First Search
    - Uniform-Cost Search
  - Heuristic Search Methods
    - Greedy Search
    - A* Search → heuristic design!
      - Admissibility: \( h(n) \leq \text{cost of cheapest path to a goal state} \). Ensures when goal node is expanded, no other partial plans on fringe could be extended into a cheaper path to a goal state.
      - Consistency: \( c(n \rightarrow n') \geq h(n) - h(n') \). Ensures when any node \( n \) is expanded during graph search the partial plan that ended in \( n \) is the cheapest way to reach \( n \).
- Time and space complexity, completeness, optimality
- Iterative Deepening: enables to retain optimality with little computational overhead and better space complexity