Search Gone Wrong?
Announcements

- **Assignments:**
  - Project 0 (Python tutorial): due Friday 1/28 at 4:59pm
  - Project 1 (Search): due Friday 2/4 at 4:59pm
    - Watch for office hour specifics --- GSI project Czar!
  - Still looking for project partners? --- Come to front after lecture.
  - Try pair programming, not divide-and-conquer
  - Account forms available up front during break and after lecture

- **Lecture Videos:** will be linked from lecture schedule

- **Sections start tomorrow**
  - Have fun solving exercises! Solutions will be posted online on Friday after last section.
  - After 2 weeks of section we will evaluate potential overcrowdedness issues and find a solution

---

Today

- Time and space complexity of DFS and BFS
- Iterative deepening --- “best of both worlds”
- Uniform cost search
- Greedy search
- A* search
  - Heuristic design
  - Admissibility, Consistency
- Tree search → Graph search
Recap: 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)

---

General Tree Search

```plaintext
def 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?
Example Search Tree

- **Search:**
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

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>
DFS

- 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</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>
</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</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td></td>
<td>w/ Path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Checking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>$O(b^{m+1})$</td>
<td>$O(b^{m+1})$</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>$O(b^{m+1})$</td>
<td>$O(bs)$</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.

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 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>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
# Uniform Cost 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>$O(b^m)$</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>$O(b^{C*/\varepsilon})$</td>
<td>$O(b^{C*/\varepsilon})$</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
Heuristics

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* \( h(n) \)

\[ f(n) = g(n) + h(n) \]

Example: Teg Grenager
When should A* terminate?

- Should we stop when we enqueue a goal?

  - No: only stop when we dequeue a goal

- When should A* terminate?

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 \( G \)

\[
\begin{align*}
f(n) &= g(n) + h(n) \\
g(n) + h(n) &\leq g(G^*) \\
g(G^*) &< g(G) \\
g(G) &= f(G) \\
f(n) &< f(G)
\end{align*}
\]
Properties of A*

Uniform-Cost

\[ b \]

A*

\[ b \]

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

Comparison

Greedy

Uniform Cost

A star
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 relaxed-problem heuristic

<table>
<thead>
<tr>
<th></th>
<th>Average nodes expanded when optimal path has length...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...4 steps</td>
</tr>
<tr>
<td>UCS</td>
<td>112</td>
</tr>
<tr>
<td>TILES</td>
<td>13</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?
- \( h(\text{start}) = 3 + 1 + 2 + ... \)
  \[ = 18 \]

<table>
<thead>
<tr>
<th></th>
<th>Average nodes expanded when optimal path has length...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...4 steps</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
</tr>
<tr>
<td>MANHATTAN</td>
<td>12</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 Diagram]

- 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?
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
Optimality

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

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

- Consistency implies admissibility

- In general, natural admissible heuristics tend to be consistent

---

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