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)
Search Gone Wrong?
Example: Pancake Problem


text content
Example: Pancake Problem

State space graph with costs as weights
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

Action: flip top two
Cost: 2

Path to reach goal:
Flip four, flip three
Total cost: 7
Uniform Cost Search

- **Strategy**: expand lowest path cost

- **The good**: UCS is complete and optimal!

- **The bad**:
  - Explores options in every “direction”
  - No information about goal location
Example: Heuristic Function

Heuristic: the largest pancake that is still out of place

$h(x)$
Example: Heuristic Function

$h(x)$
Best First (Greedy)

- **Strategy:** expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state

- **A common case:**
  - Best-first takes you straight to the (wrong) goal

- **Worst-case:** like a badly-guided DFS
Combining UCS and Greedy

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

- **A* Search** orders by the sum: $f(n) = g(n) + h(n)$
When should A* terminate?

- Should we stop when we enqueue a goal?
  - No: only stop when we dequeue a goal

![Diagram showing the A* search process with nodes S, A, B, and G, and their respective costs and heuristics.]

- No: only stop when we dequeue a goal
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

Notation:

- $g(n) = \text{cost to node } n$
- $h(n) = \text{estimated cost from } n \text{ to the nearest goal (heuristic)}$
- $f(n) = g(n) + h(n) = \text{estimated total cost via } n$
- $G^* = \text{a lowest cost goal node}$
- $G = \text{another goal node}$
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*
UCS vs A* Contours

- Uniform-cost expanded in all directions

- A* expands mainly toward the goal, but does hedge its bets to ensure optimality
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*, where new actions are 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>Average nodes expanded when optimal path has length…</th>
<th>…4 steps</th>
<th>…8 steps</th>
<th>…12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>112</td>
<td>6,300</td>
<td>3.6 x 10^6</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
</tbody>
</table>
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?

\[ h(\text{start}) = 3 + 1 + 2 + \ldots = 18 \]
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 + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state is new
  - If not new, skip it

- Important: store the closed set 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 ← INSERTALL(EXPAND(node, problem), fringe)
    end
  end
```

- Can this wreck completeness? Optimality?
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)
A (1+4) B (1+1)
C (2+1) C (3+1)
G (5+0) G (6+0)
Consistency of Heuristics

- Stronger than admissibility

- Definition:
  \[
  \text{cost}(A \text{ to } C) + h(C) \geq h(A)
  \]
  \[
  \text{cost}(A \text{ to } C) \geq h(A) - h(C)
  \]
  Real cost $\geq$ cost implied by heuristic

- Consequences:
  - The f value along a path never decreases
  - A* graph search is optimal
Optimality of A* Graph Search

Proof:

- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because $n'$ is suboptimal
- $p$ would have been expanded before $n'$
- Contradiction!
Optimality

- **Tree search:**
  - A* is 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, most natural admissible heuristics tend to be consistent, especially if from relaxed problems
Summary: A*

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems
Mazeworld Demos
Graph Search, Reconsidered

- 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?
Optimality of A* Graph Search

- Consider what A* does:
  - Expands nodes in increasing total f value (f-contours)
  - Proof idea: optimal goals have lower f value, so get expanded first

We're making a stronger assumption than in the last proof… What?
Optimality of A* Graph Search

- Consider what A* does:
  - Expands nodes in increasing total f value (f-contours)
    Reminder: \( f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic} \)
  - Proof idea: the optimal goal(s) have the lowest f value, so it must get expanded first

There’s a problem with this argument. What are we assuming is true?
Course Scheduling

- From the university’s perspective:
  - Set of courses \( \{c_1, c_2, \ldots, c_n\} \)
  - Set of room / times \( \{r_1, r_2, \ldots, r_n\} \)
  - Each pairing \((c_k, r_m)\) has a cost \(w_{km}\)
  - What’s the best assignment of courses to rooms?

- States: list of pairings
- Actions: add a legal pairing
- Costs: cost of the new pairing

- Admissible heuristics?
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:

![Diagram]

- What can we require to prevent these inversions?
- Consistency:
  - Real cost must always exceed reduction in heuristic
  - Like admissibility, but better!