Announcements

- Project 1 is due **Tuesday, September 6, 11:59 PM PT**
  - Start soon if you haven’t already! It’s one of the longer projects.
- HW1 is due **Friday, September 9, 11:59 PM PT**
Today

- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search
- Graph Search
Recap: Search
Recap: Search

- **Search problem:**
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - 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)
  - Optimal: finds least-cost plans
Example: Pancake Problem

Cost: Number of pancakes flipped
Example: Pancake Problem

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation $\sigma$ of the integers from 1 to $n$, let $f(\sigma)$ be the smallest number of prefix reversals that will transform $\sigma$ to the identity permutation, and let $j(n)$ be the largest such $f(\sigma)$ for all $\sigma$ in (the symmetric group) $S_n$. We show that $f(n) \leq (5n + 5)/3$, and that $f(n) \geq 17n/16$ for $n$ a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n + 3$. 

Example: Pancake Problem

State space graph with costs as weights
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

Action: flip top two
Cost: 2

Path to reach goal:
Flip four, flip three
Total cost: 7
All these search algorithms are the same except for fringe strategies

- Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
- Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
- Can even code one implementation that takes a variable queuing object
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

[Demo: contours UCS empty (L3D1)]
[Demo: contours UCS pacman small maze (L3D3)]
Informed Search
Search Heuristics

- A heuristic is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing
Example: Heuristic Function

$h(x)$
Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place
Greedy Search
Greedy Search

- Expand the node that seems closest...

- What can go wrong?
Greedy Search

- **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

[Demo: contours greedy empty (L3D1)]
[Demo: contours greedy pacman small maze (L3D4)]
A* Search
A* Search

UCS

Greedy

A*
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: $f(n) = g(n) + h(n)$

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
Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs
Admissible Heuristics

- A heuristic $h$ is **admissible** (optimistic) if:

  $$0 \leq h(n) \leq h^*(n)$$

  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* Tree Search
Optimality of A* Tree Search

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

- A will exit the fringe before B
Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B

1. $f(n)$ is less or equal to $f(A)$

$$f(n) = g(n) + h(n)$$  \hspace{1cm} \text{Definition of f-cost}

$$f(n) \leq g(A)$$  \hspace{1cm} \text{Admissibility of h}

$$g(A) = f(A)$$  \hspace{1cm} h = 0 at a goal
Optimality of A* Tree Search: Blocking

1. \( f(n) \) is less than or equal to \( f(A) \)
   - Definition of \( f \)-cost says:
     \[
     f(n) = g(n) + h(n) = (\text{path cost to } n) + (\text{est. cost of } n \text{ to } A)
     \]
     \[
     f(A) = g(A) + h(A) = (\text{path cost to } A) + (\text{est. cost of } A \text{ to } A)
     \]
   - The admissible heuristic must underestimate the true cost
     \[
     h(A) = (\text{est. cost of } A \text{ to } A) = 0
     \]
   - So now, we have to compare:
     \[
     f(n) = g(n) + h(n) = (\text{path cost to } n) + (\text{est. cost of } n \text{ to } A)
     \]
     \[
     f(A) = g(A) = (\text{path cost to } A)
     \]
   - \( h(n) \) must be an underestimate of the true cost from \( n \) to \( A \)
     \[
     (\text{path cost to } n) + (\text{est. cost of } n \text{ to } A) \leq (\text{path cost to } A)
     \]
     \[
     g(n) + h(n) \leq g(A)
     \]
     \[
     f(n) \leq f(A)
     \]
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B
  1. $f(n)$ is less or equal to $f(A)$
  2. $f(A)$ is less than $f(B)$

$g(A) < g(B)$  B is suboptimal
$f(A) < f(B)$  $h = 0$ at a goal
Optimality of A* Tree Search: Blocking

2. \( f(A) \) is less than \( f(B) \)
   - We know that:
     \[
     f(A) = g(A) + h(A) = \text{(path cost to A)} + \text{(est. cost of A to A)}
     \]
     \[
     f(B) = g(B) + h(B) = \text{(path cost to B)} + \text{(est. cost of B to B)}
     \]
   - The heuristic must underestimate the true cost:
     \[
     h(A) = h(B) = 0
     \]
   - So now, we have to compare:
     \[
     f(A) = g(A) = \text{(path cost to A)}
     \]
     \[
     f(B) = g(B) = \text{(path cost to B)}
     \]
   - We assumed that B is suboptimal! So
     \[
     \text{(path cost to A)} < \text{(path cost to B)}
     \]
     \[
     g(A) < g(B)
     \]
     \[
     f(A) < f(B)
     \]
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
- Some ancestor \( n \) of A is on the fringe, too (maybe A!)
- Claim: \( n \) will be expanded before B
  1. \( f(n) \) is less or equal to \( f(A) \)
  2. \( f(A) \) is less than \( f(B) \)
  3. \( n \) expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal

\[ f(n) \leq f(A) < f(B) \]
Properties of A*
Properties of A*

Uniform-Cost

A*
UCS vs A* Contours

- Uniform-cost expands equally in all “directions”

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

[Demo: contours UCS / greedy / A* empty (L3D1)]
[Demo: contours A* pacman small maze (L3D5)]
Comparison

Greedy  Uniform Cost  A*
A* Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- ...

[Demo: UCS / A* pacman tiny maze (L3D6,L3D7)]
[Demo: guess algorithm Empty Shallow/Deep (L3D8)]
Creating Heuristics

YOU GOT

HEURISTIC UPGRADE!
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 relax problems, where new actions are available.

- Inadmissible heuristics are often useful too.
Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- How many successors 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

Start State

Goal State

### Average nodes expanded when the optimal path has...

<table>
<thead>
<tr>
<th>Steps</th>
<th>UCS</th>
<th>TILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>6,300</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td>$3.6 \times 10^6$</td>
<td>227</td>
</tr>
</tbody>
</table>

Statistics from Andrew Moore
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 is it admissible?

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

<table>
<thead>
<tr>
<th>Average nodes expanded when the optimal path has...</th>
<th>...4 steps</th>
<th>...8 steps</th>
<th>...12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
<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
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself
Semi-Lattice of Heuristics
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
Graph Search
- Failure to detect repeated states can cause exponentially more work.
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 has never been expanded before
  - If not new, skip it, if new add to closed set

- Important: store the closed set as a set, not a list

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

- How about optimality?
A* Graph Search Gone Wrong?

State space graph

Search tree

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

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]

- Consequences of consistency:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
Optimality of A* Graph Search
Optimality

- **Tree search:**
  - $A^*$ is optimal if heuristic is admissible
  - 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
A*: Summary
A*: Summary

- 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
Search and Models

- Search operates over models of the world
  - The agent doesn’t actually try all the plans out in the real world!
  - Planning is all “in simulation”
  - Your search is only as good as your models...
Search Gone Wrong?
Search Gone Wrong?
Appendix: Search Pseudo-Code
function Tree-Search(problem, fringe) return a solution, or failure
    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
        for child-node in Expand(state[node], problem) do
            fringe ← Insert(child-node, fringe)
        end
    end
function GRAPH-SEARCH(problem, fringe) return 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
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
        end
    end