Announcements

- **Project 0: Python Tutorial**
  - Due Jan 16th before midnight

- **Homework 1**
  - Due Jan 18th before midnight
  - Covers today’s lecture.
  - You can start today!
  - Look at the practice problems first!
CS 6300: Search

Instructor: Daniel Brown

University of Utah

[Based on slides created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley http://ai.berkeley.edu.]
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
- Informed (heuristic) Search
Agents that Plan
Planning Agents

- **Planning agents:**
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world **WOULD BE**

- **Optimal Planning**
  - Returns a least cost solution.

- **Complete Planning**
  - If there exists a solution it will find it.

- **Planning vs. replanning**
Video of Demo Mastermind
Video of Demo Replanning
Search Problems
Search Problems

- A search problem consists of:
  - A state space
  - A successor function (with actions, costs)
  - 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

"N", 1.0

"E", 1.0
Search Problems Are Models
Example: Traveling in Romania

- **State space:**
  - Cities

- **Successor function:**
  - Roads: Go to adjacent city with cost = distance

- **Start state:**
  - Arad

- **Goal test:**
  - Is state == Bucharest?

- **Solution?**
What’s in a State Space?

- **Problem: Pathing (go from location A to B)**
  - States: \((x,y)\) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is \((x,y) = \text{END}\)

- **Problem: Eat-All-Dots**
  - States: \(\{(x,y), \text{dot booleans}\}\)
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

The **world state** includes every last detail of the environment.

A **search state** keeps only the details needed for planning (abstraction).
State Space Sizes?

- **World state:**
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- **How many**
  - World states?
    \[120 \times (2^{30}) \times (12^2) \times 4 \approx 74 \text{ trillion}\]
  - States for pathing?
    120
  - States for eat-all-dots?
    \[120 \times (2^{30})\]
Quiz: Safe Passage

- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
  - (agent position, dot booleans, power pellet booleans, remaining scared time)
State Space Graphs and Search Trees
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a search graph, each state occurs only once!

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea

Tiny state space graph for a tiny search problem
A search tree:
- A “what if” tree of plans and their outcomes
- The start state is the root node
- Children correspond to successors
- Nodes show states, but correspond to PLANS that achieve those states
- For most problems, we can never actually build the whole tree

This is now / start
Possible futures

"E", 1.0
"N", 1.0
We construct both on demand – and we construct as little as possible.

Each NODE in the search tree is an entire PATH in the state space graph.
Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

```
S ——— a ——— G
     |   |   |
     v   v   v
  a ——— G ——— b
```

What does the search tree look like?

How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!
Search Example: Romania
Searching with a Search Tree

- **Search:**
  - Expand out potential plans (tree nodes)
  - Maintain a **fringe** of partial plans under consideration
  - Try to expand as few tree nodes as possible
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

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?
Example: Tree Search
Depth-First Search
**Depth-First Search**

*Strategy:* expand a deepest node first

*Implementation:* Fringe is a LIFO stack
Search Algorithm Properties
Search Algorithm Properties

- **Complete:** Guaranteed to find a solution if one exists?
- **Optimal:** Guaranteed to find the least cost path?
- **Time complexity?**
- **Space complexity?**

**Cartoon of search tree:**
- $b$ is the branching factor
- $m$ is the maximum depth
- solutions at various depths

**Number of nodes in entire tree?**
- $1 + b + b^2 + \ldots + b^m = O(b^m)$
**Depth-First Search (DFS) Properties**

- **What nodes DFS expand?**
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If \( m \) is finite, takes time \( O(b^m) \)

- **How much space does the fringe take?**
  - Only has siblings on path to root, so \( O(bm) \)

- **Is it complete?**
  - \( m \) could be infinite, so only if we prevent cycles (more later)

- **Is it optimal?**
  - No, it finds the “leftmost” solution, regardless of depth or cost
Breadth-First Search
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be $s$
  - Search takes time $O(b^s)$

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$

- Is it complete?
  - $s$ must be finite if a solution exists, so yes!

- Is it optimal?
  - Only if costs are all 1 (more on costs later)
BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.
Uniform Cost Search
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs $C^*$ and arcs cost at least $\varepsilon$, then the “effective depth” is roughly $C^*/\varepsilon$
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- Is it optimal?
  - Yes! (Proof via A*)
Uniform Cost Issues

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

\[ \text{[c \leq 1, c \leq 2, c \leq 3]} \]

[Demo: empty grid UCS (L2D5)]
[Demo: maze with deep/shallow water DFS/BFS/UCS (L2D7)]
Video of Demo Empty UCS
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 1)
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 2)
Video of Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 3)
Graph Search
Tree Search: Extra Work!

- 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?
Tree 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
Graph Search Pseudo-Code

```plaintext
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
                if STATE[child-node] is not in closed then fringe ← INSERT(child-node, fringe)
            end
        end
    end
```

Use this version for the homeworks and exams!
Some Hints for P1

- Implement your closed list (explored set) as a set!
- Nodes are conceptually paths, but better to represent with a state, cost, last action, and reference to the parent node.
- Pseudo code from Russell and Norvig book. Good example of how a child node is created from a parent node.

```plaintext
function CHILD-NODE(problem, parent, action) returns a node
return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```
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
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

$h(x)$
Greedy Search
Example: Heuristic Function

$h(x)$
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)]
Video of Demo Contours Greedy (Pacman Small Maze)
A* Search
A* Search
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
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 $n$, *that is* along the optimal path to $A$, is on the fringe, too (maybe $A$!)
- Claim: $n$ will be expanded before $B$
  1. $f(n)$ is less or equal to $f(A)$

\[
f(n) = g(n) + h(n) \quad \text{Definition of f-cost}
\]
\[
f(n) \leq g(n) + h^*(n)
\]
\[
= g(A) \\
= f(A) \quad \text{Admissibility of h}
\]

$h = 0$ at a goal
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
- Some ancestor \( n \), *that is* along the optimal path to 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 \( f(B) \)

\[ g(A) < g(B) \quad \text{B is suboptimal} \]
\[ f(A) < f(B) \quad h = 0 \text{ at a goal} \]
Proof:

- Imagine B is on the fringe
- Some ancestor \( n \), *that is* along the optimal path to 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 along optimal path to A expand before B
- A expands before B
- A* search is optimal

\[
f(n) \leq f(A) < f(B)
\]
Properties of 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)]
Video of Demo Contours (Empty) -- UCS
Video of Demo Contours (Empty) -- Greedy
Video of Demo Contours (Empty) – A*
Video of Demo Contours (Pacman Small Maze) – A*
Comparison

Greedy

Uniform Cost

A*
A* Applications
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 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.
Heuristics

- 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
A* Graph Search Gone Wrong?

State space graph

Search tree

C is already in closed set so not expanded again
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
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

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

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal
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