CS 5522: Artificial Intelligence II

Search Algorithms

Instructor: Wei Xu
Ohio State University

[These slides were adapted from CS188 Intro to AI at UC Berkeley.]
Agents that Plan Ahead

Search Problems

Uninformed Search Methods
- Depth-First Search
- Breadth-First Search
- Uniform-Cost Search

Informed Search Methods
- Heuristics
- Greedy Search
- A* Search
Agents that Plan
Reflex Agents

- Reflex agents:
  - 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
  - Consider how the world IS

- Can a reflex agent be rational?
Video of Demo Reflex Optimal
Video of Demo Reflex Odd
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 vs. complete planning

- Planning vs. replanning

[Demo: replanning (L2D3)]
[Demo: mastermind (L2D4)]
Video of Demo Replanning
Video of Demo Mastermind
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
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?

The world state includes every last detail of the environment.

A search state keeps only the details needed for planning (abstraction).

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

- **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 Sizes?

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

- **How many**
  - World states?
    \[120 \times (2^{30}) \times (12^2) \times 4\]
  - 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
Search Trees

- 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
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.
Tree Search
Search Example: Romania
Searching with a Search Tree

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?
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 $O(b^m)$

- **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)
Video of Demo Maze Water DFS/BFS (part 1)
Video of Demo Maze Water DFS/BFS (part 1)
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 next lecture via A*)
Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

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

- We’ll fix that soon!
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)
Uninformed Search
Video of Demo Contours UCS Empty
Video of Demo Contours UCS Pacman Small Maze

SCORE: 0
Informed Search
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...
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 (Empty)
Video of Demo Contours Greedy (Pacman Small Maze)
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
What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!
Admissible Heuristics
Idea: Admissibility

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.
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)]
Video of Demo Contours (Empty) -- UCS
Video of Demo Contours (Empty) -- Greedy
Video of Demo Contours (Empty) - A*
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?**
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
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
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

- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility:** heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - **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 of A* Graph Search

- **Sketch:** consider what A* does with a consistent heuristic:
  - **Fact 1:** In tree search, 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* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems