CS 4700: Foundations of Artificial Intelligence

Fall 2019
Prof. Haym Hirsh

Lecture 5
September 23, 2019
Properties of A* Search

• If
  • search space is a finite graph and
  • all operator costs are positive

• Then
  • A* is guaranteed to terminate and
  • if there is a solution, A* will find a solution (not necessarily an optimal one)
Properties of A* Search

• If
  • search space is an *infinite* graph (but branching factor is finite) and
  • all operator costs are positive and are never less than some number $\varepsilon$
    (in other words, they cannot get arbitrarily close to 0)
• Then
  • if there is a solution, A* will terminate with a solution (not necessarily an
    optimal one)
    (no guarantee of termination if there is no solution)
Properties of A* Search

• If, in addition,
  • $h(s)$ is admissible
    (for all states $s$, $0 \leq h(s) \leq h^*(s)$)

• Then
  • If A* terminates with a solution it will be optimal
Properties of A* Search

• If, in addition,
  • $h(s)$ is consistent
    
    (for all states $s$, $h(s) \leq h(\text{apply}(a,s)) + \text{cost}(\text{apply}(a,s))$
  
• Then
  • $h(s)$ is admissible,
  • the first path found to any state is guaranteed to have the lowest cost
    
    (do not need to check for this in the algorithm), and
  • no other algorithm using the same $h(s)$ and the same tie-breaking rules will expand fewer nodes than A*
Properties of A* Search

• If
  • the search space is a tree,
  • there is a single goal state, and
  • for all states \( s \), \( |h^*(s) - h(s)| = O(\log(h^*(s))) \)
    (the error of \( h(s) \) is never more than a logarithmic factor of \( h^*(s) \))

• Then
  • A* runs in time polynomial in \( b \) (branching factor)
A* Variants

Weighted A*:

• If
  • $h(s)$ is admissible and
  • $A^*$ is used with $h'(s) = c \times h(s)$ where $c > 1$

• Then
  • Any goal state that $A^*$ terminates with will have cost no more than $c$ times the cost of an optimal solution
A* Variants

IDA*:  
• Use cost-bounded depth-first search with $h(\text{initial state})$ as the bound  
• Any time a successor is greater than the bound don’t expand it  
  - But store the lowest cost $C$ of any such state that you reach that exceeds the cost bound  
• If you terminate without a goal state run cost-bounded depth-first search with depth bound $C$

(= Depth-first search emulation of A* search)
A* Variants

SMA*:

• A* search with a memory bound
• If you would generate a node but don’t have space to add it to Open, remove from open the node s on Open with greatest f(s) but keep track of its parent s’ and the cost of the removed node f(s)
• If you reach a node on Open whose cost is worse than this value, you re-expand s’
Relationship to Other Methods

• A* search is a special case of
  • Branch and bound search
  • Dynamic programming
What if We’re OK with Suboptimal Solutions?
Idea 1: Beam Search

- Best-first search, but only keep the k best on Open
Beam(s,ops,open,closed,width) =
  If goal(s) Then return(s);
  Else If not(s ∈ closed)
    Then
      successors ← {}; add(s,closed);
      For each o ∈ ops that applies to s
        add apply(o,s) to successors
      open ← add successors to open;
      open ← best_f(s')(open,width)
      If not(empty(open))
        s' ← min_f(s')(open);
        open ← remove(s’,open);
        BestFS(s’,ops,open,closed,width)
    Else return(FAIL)
Initial call: Beam(initialstate,ops,{},{},width)
Beam Search

- Lose the guarantees, gain a bounded memory size, simple algorithm
Idea 2: Hill Climbing

• Loosely, beam search with width 1
Idea 2: Hill Climbing

• Loosely, beam search with width 1

• (For historical reasons seeking to maximize rather than minimize, hence the name hill *climbing*)
Idea 2: Hill Climbing

• Loosely, beam search with width 1

• (For historical reasons seeking to maximize rather than minimize, hence the name hill *climbing*)

• (Just to confuse things, it includes gradient descent, where you’re minimizing)
Idea 2: Hill Climbing

• Loosely, beam search with width 1

• (For historical reasons seeking to maximize rather than minimize, hence the name hill *climbing*)

• (Just to confuse things, it includes gradient descent, where you’re minimizing)

• (Just to confuse things even further textbook example minimizes f)
Hill Climbing Example: 8 Queens

• Initial state = random placement of 8 queens
• Operators = move a queen
• $f(s) = \# \text{ of attacked queens}$
• Want $f(s) = 0$
Hill Climbing

hillclimbing(s):
    current ← s;
    loop
        new ← lowest-valued successor of s;
        if f(new) < f(s)
            then current ← new
        else return(current)

If goal is to find a maximum valued state, switch this to largest-valued and >
Problems for Hill Climbing

• Local optima
• Plateau problem: no direction looks good (flat vs shoulder)
• Ridges: increases not aligned with axes
Hill Climbing

hillclimbing(s):
  current ← s;
  loop
    new ← lowest-valued successor of s;
    if $f(new) < f(s)$
      then current ← new
    else return(current)

If goal is to find a maximum valued state, switch this to largest-valued and $>$

This is not $\leq$
Problems for Hill Climbing

- Local optima
- Plateau problem: no direction looks good (flat vs shoulder)
- Ridges: increases not aligned with axes
Hill Climbing Variants

• Stochastic hill climbing:
  • Pick successor of a state probabilistically “proportional” to f values of successors
  • Can use a weighting scheme where early on you do this, but as you progress you become more and more likely to pick the best successor

• Slower than pure hill-climbing, but can find better solutions, such as due to ridges
Hill Climbing Variants

• Sideways moves:
  • Allow the algorithm to pick a successor with equal value if there is none with a better value
  • Do this at most some bounded number of times in a row

• Good for plateaus
Hill Climbing Variants

• First-choice hill-climbing:
  • Generate successors, stop and move ahead with the first successor that’s better than the current state
  • Good for problems where states may have a large #s of successors
Hill Climbing Variants

• Random restart:
  • If initial state is random or there are often ties that are broken randomly you can rerun hill climbing with different starting states

• Good for local optima
Hill Climbing Variants

• Combinations of the above

• Usually thought of as a tool kit and you try various options
Simulated Annealing

Stochastic Hill Climbing Search
with a small, decreasing probability
of doing a bad move
Simulated Annealing

Stochastic Hill Climbing Search
with a small, decreasing probability
of doing a bad move

Intuition: To avoid getting stuck in local optima,
let yourself wander a little, less so as time progresses

Vocabulary: the farther into the search you go the lower the “temperature”
Sample Simulated Annealing Algorithm

SA(s,ops):

\begin{align*}
\text{current} &\leftarrow s; \quad T \leftarrow \text{initial T value}; \quad [\text{For example, } T=1] \\
\text{loop} & \\
\text{op} &\leftarrow \text{random element of ops}; \\
\text{new} &\leftarrow \text{apply(op, current)}; \\
\text{delta} &\leftarrow f(\text{new}) - f(\text{current}); \\
\text{if } \text{delta} < 0 \text{ then } \text{current} &\leftarrow \text{new} \\
\text{else with probability } e^{-\frac{\text{delta}}{T}} \text{ current} &\leftarrow \text{new}; \\
\text{update } T & \quad [\text{For example, } T=\frac{1}{\text{iteration#}}] \\
\text{until } <\text{stopping criterion}> & \quad [\text{For example, some max # of iterations}]
\end{align*}
Genetic Algorithms and Evolutionary Computation

Use more than just f(s)!

• Assumes:
  • States have structure (example: 8 queen chess boards)
  • Perturbing elements of states makes sense (example: moving a queen 1 square)
  • You can mix and match pieces of states to get new ones (example: Take columns 1-3 from one state, 4-8 from a second state)
  • f(s) is called the “fitness function”

• Loosely, beam search with two operators, mutate and crossover

• Many variants, including Genetic Programming where states are programs and you’re trying to find a program that does what you want