Reminder: Technology Policy

No technology except for first four rows of left and right sides
Jupyter Notebooks

“Primer”: Friday 5pm in Gates G01
“Informed” Search

To formulate a problem:

• States: \( S \)
• Operators: \( \text{Ops} \)
• Initial state
• Goal condition: \( \text{goal(s)} \)
• Heuristic Evaluation Function \( f(s): S \rightarrow \mathbb{R} \) (usually \( \geq 0 \))
  • \( f \) is an estimate of the merit of \( s \)
  • Typically \( f(s_1) < f(s_2) \) means \( s_1 \) is “better” than \( s_2 \)
A* Search

Best-first search using

\[ f(s) = g(s) + h(s) \]

- \( g(s) \) = sum of costs from initial state to \( s \)
- \( h(s) \) = estimate of cost from \( s \) to nearest goal
A* Search

Initial call: \( A^*(\text{initialstate}, \text{ops}, \{\}, \{\}) \)

\[
A^*(s, \text{ops}, \text{open}, \text{closed}) = \\
\text{If goal}(s) \text{ Then return}(s)\; \\
\text{Else If not}(s \in \text{closed}) \\
\quad \text{Then successors } \leftarrow \{\} \text{; add}(s, \text{closed}); \\
\quad \text{For each } o \in \text{ops} \text{ that applies to } s \text{ add } \text{apply}(o, s) \text{ to successors} \\
\quad \text{For each } s \in \text{successors} \\
\qquad \text{If } s \in \text{open} \text{ and } g(s) \text{ of current path is less, update } g(s) \text{ in open} \\
\qquad \text{Else If } s \in \text{closed} \text{ and } g(s) \text{ of current path is less, add } s \text{ to open w/ new } g(s) \\
\qquad \text{Else If } s \notin \text{open} \text{ and } s \notin \text{closed}, \text{ add } s \text{ to open} \\
\text{If not}(\text{empty}(\text{open})) \\
\quad s' \leftarrow \text{argmin}_{s \in \text{open}} (f(s)); \\
\quad \text{open } \leftarrow \text{remove}(s', \text{open}); \\
\quad A^*(s', \text{ops}, \text{open}, \text{closed}) \\
\quad \text{Else return}(\text{FAIL})
\]
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) ≤ h(apply(a,s)) + cost(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
  • A* is “optimal” - 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)
Properties of A* Search

And many others

(Extremely widely used, so well-understood)
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 \( C \)

\( (= \text{Depth-first search emulation of A* search}) \)
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’
A* Variants

And many others

(Extremely widely used, so explored)
Search Methods Thus Far

- DFS
- BFS
- IDS
- Best-First/A*
Search Methods Thus Far

DFS
BFS
IDS
Best-First/A*

Focus is on optimality
What if We’re OK with Suboptimal Solutions?
What if We’re OK with Suboptimal Solutions?

(A* variants, but what else?)
Idea 1: Beam Search

Best-first search, but only keep the k best on Open

k is called the “beam width”
Search Algorithm Template

Initial call: Search(initialstate,ops,{},{})

Search(s,ops,open,closed) =
    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;
            If not(empty(open))
                s’ ← select(open);
                open ← remove(s’,open);
                search(s’,ops,open,closed)
            Else return(FAIL)
Initial call: Search(initialstate,ops,{},{})

Search(s,ops,open,closed) =
  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;
  If not(empty(open))
    s′ ← select(open);
    open ← remove(s′,open);
    search(s′,ops,open,closed)
  Else return(FAIL)
Beam Search

Initial call: `BeamSearch(initialstate,ops,{},{},width)`

`BeamSearch(s,ops,open,closed) =`
  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 ← top-k_f(open,width)
      If not(empty(open))
        s’ ← best_f(open);
        open ← remove(s’,open);
        `BeamSearch(s’,ops,open,closed)`
      Else return(FAIL)
Beam Search

Initial call: \texttt{BeamSearch(initialstate,ops,{},{}),width)}

\texttt{BeamSearch(s,ops,open,closed)} =
\begin{enumerate}
  \item If goal(s) Then return(s);
  \item Else If not(s \in closed)
  \begin{enumerate}
    \item successors \leftarrow \{\}; add(s,closed);
    \item For each o \in ops that applies to s
      \begin{enumerate}
        \item add apply(o,s) to successors
      \end{enumerate}
    \item open \leftarrow add successors to open;
  \end{enumerate}
  \end{enumerate}
\end{enumerate}

open \leftarrow \texttt{best-k_f(open,width)} \quad \text{\texttt{best-k_f(x,k)}} = \text{the k best items on x according to f}

If not(empty(open))
\begin{enumerate}
  \item s' \leftarrow \texttt{best_f(open)};
  \item open \leftarrow remove(s',open);
  \item \texttt{BeamSearch(s',ops,open,closed)}
\end{enumerate}

Else return(FAIL)
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 \textit{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, 1 per column
• Operators = pick a column and move its 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 ≤
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 with high branching factor
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):

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

Totally different approach to search inspired by molecular genetics
A form of beam search
Genetic Algorithms and Evolutionary Computation

Totally different approach to search inspired by molecular genetics
A form of beam search

• States: Assume have a structured representation
  • Example: N Queens – (position queen 1, ..., position queen N) [n-tuples]
Genetic Algorithms and Evolutionary Computation

Totally different approach to search inspired by molecular genetics
A form of beam search

• States: Assume have a structured representation
  • Example: N Queens – (position queen 1, ..., position queen N) [n-tuples]
• Initial state: a set (“population”) of random states
  • Example: a bunch of boards with N random queens
Genetic Algorithms and Evolutionary Computation

Totally different approach to search inspired by molecular genetics
A form of beam search

- States: Assume have a structured representation
  - Example: N Queens – (position queen 1, ..., position queen N) [n-tuples]
- Initial state: a set (“population”) of random states
  - Example: a bunch of boards with N random queens
- Operators: Apply generically, not (in its purist form) domain specific
  - Mutate: Perturb a state
    - Example: Change a queen position by 1
  - Crossover: Take two states and combine elements of both
    - Example: Take two N Queens boards, take k from one and N-k from the other
Genetic Algorithms and Evolutionary Computation

Totally different approach to search inspired by molecular genetics
A form of beam search

- States: Assume have a structured representation
  - Example: N Queens – (position queen 1, ..., position queen N) [n-tuples]
- Initial state: a set (“population”) of random states
  - Example: a bunch of boards with N random queens
- Operators: Apply generically, not (in its purist form) domain specific
  - Mutate: Perturb a state
    - Example: Change a queen position by 1
  - Crossover: Take two states and combine elements of both
    - Example: Take two N Queens boards, take k from one and N-k from the other
- \( f(s) \): “fitness function”
Genetic Algorithms and Evolutionary Computation

Algorithm sketch:

• Create an initial population of individuals (states) [population size]
• On each generation (iteration) create a new population by a combination of
  • Crossover:
    • Take two elements of the population biased by fitness function
    • Create a new individual (state) by taking pieces of each
  • Mutation:
    • Take an element of the population biased by fitness function
    • Create a new individual (state) by perturbing it
Genetic Algorithms and Evolutionary Computation

There are MANY variants
Genetic Algorithms and Evolutionary Computation

There are MANY variants

One that’s distinctive enough to get its own mention:

Genetic programming:
• States are programs in a structured language
• Crossover and mutation create new programs from old ones