Beyond Classical Search

Professor Marie Roch
Chapter 4, Russell & Norvig

Local search

• Single state node
  • paths not usually retained
  • typically move only to neighbors of state

• The good
  • Low memory usage
  • Appropriate for large (possibly infinite) state spaces

• The bad
  • Lose advantages from search-tree retention (e.g. backtracking)
Optimization problems

• Find best state – find extrema of objective function $f(state)$

Optimization problems

• Optimal solutions (global extrema) can be problematic

  • Complete search $\rightarrow$ local extrema
    *easy to get stuck*

  • Optimal search $\rightarrow$ global extrema
Hill-climbing (aka greedy) search

def hillclimb(state):
    done = False
    while not done:
        next = successors of state
        find s in next such that maximizes f(s) – f(state)
        if f(s) – f(state) > 0 then state = s
        else done = True
    return state

Troublesome for hill climbing...

- Local extrema – trapped!
- Ridges – no real way out
- Plateaus – what should we do for sideways moves? Continue?
Hill-climbing variants

• Stochastic – Assign probabilities related to steepness of choice and pick randomly (slow convergence).
• First-choice – Generate successors randomly, pick the first one that’s better than current state.
• Random-restart – Pick a new initial state if we don’t find what we are looking for.

Speed at which search converges to a “good” state?

Simulated annealing

• Annealing
  • Process to harden metals
  • Subject to high heat
    • metals enter high energy state
    • slowly cool
    • allows molecules to realign, reducing stress
• Simulated annealing
  • Simulate temperature
  • Volatility of action choices is related to temperature
    • high temperature – more likely to pick “risky” decisions
    • low temperature – more likely to pick “good” decisions
Simulated annealing

- Simulated annealing
  - “temperature” starts hot and cools (function of time)
  - A successor state is chosen at random
    - improvement + or degradation - of state fitness
      \[ \Delta E = \text{fitness(child)} - \text{fitness(current)} \]
    - If \( \Delta E > 0 \)
      then update state
    otherwise
      update based on odds of picking a bad node
      \[ 1 + e^{\Delta E/\text{Temp}} \]

Beam search

- Differs in treatment of successors from standard search
  - Only keep the k most successful children
  - May add stochastic component to increase diversity of population
- Frequently used to explore multiple hypotheses while keeping frontier set small
- Example: Speech recognition systems often use this
Genetic algorithms

• Search-state nodes are measured by a fitness function
• Successors
  • Generated from random pair in frontier (called population)
    • new state from crossover (mixture of parent states)
    • new state may be further mutated
  • Only fittest nodes are retained (beam search)

Genetic algorithms

• States need to be represented in a way that parameters can be mixed
• Example
  • 8 queens with all queens placed
  • state – row # of queen (1,6,2,5,7,4,8,3) or 16257483
  • fitness function:
    # non-attacking pairs
Genetic algorithm example

- How are random pairs selected?
  Assigned probabilities
  \[ P(\text{node}) = \frac{\text{fitness(node)}}{\sum_{i \in \text{population}} \text{fitness}(i)} \]

- Population of four nodes
  \[ \text{fitness}(24748552) = 24 \rightarrow 31\% \quad (24/(24+23+20+11)) \]
  \[ \text{fitness}(32752411) = 23 \rightarrow 29\% \]
  \[ \text{fitness}(24415124) = 20 \rightarrow 26\% \]
  \[ \text{fitness}(32543213) = 11 \rightarrow 14\% \]

Mutation changes a random position
Local search in continuous spaces

Place three airports to minimize distance to nearest city

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

$$C_i = \{ \text{cities closest to airport } i \}$$
Local search in continuous spaces

Possible approaches
- Discretize the search space
  - increment state by \( \pm \varepsilon \)
  - with 6 variables, 12 possible successors (if constrained to one direction)
  - what size \( \varepsilon \)?

- Compute the gradient
  - Gives us the direction of steepest ascent.
  \[
  \nabla f = \left( \frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta y_1}, \frac{\delta f}{\delta x_2}, \frac{\delta f}{\delta y_2}, \frac{\delta f}{\delta x_3}, \frac{\delta f}{\delta y_3} \right)
  \]

Local search in continuous space

Gradient approaches
- If gradient exists in closed form, may be able to solve for maximum: \( \nabla f = 0 \)

- Many objective functions cannot be solved in closed form, e.g.
  \[
  f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{j \in C_i} (x_i - x_j)^2 + (y_i - y_j)^2
  \]
  has discontinuities as cities change \( C_i \) membership.
Local search in continuous space

• Local gradient might be possible

\[ \nabla f(x_1, y_1, x_2, y_2, x_3, y_3) = \left( 2 \sum_{c \in C_1} (x_1 - x_c), 2 \sum_{c \in C_1} (y_1 - y_c) \right) \]

• If objective function not differentiable
  evaluate \( f \) in the neighborhood and compute \textit{empirical gradient}.

• Update requires step size \( \alpha \)

\[ \text{state} \leftarrow \text{state} + \alpha \nabla f(\text{state}) \]

Local search in continuous space

• Choice of \( \alpha \)
  • too small... learning slow
  • too large... might overshoot extrema or gradient change

• Line search
  • double \( \alpha \) repeatedly until objective function \( f \) starts to decrease
  • choose new direction
Newton-Raphson method

• Method for finding roots \( f(x) = 0 \)
• Find root \( x \):
  • start with a “good” estimate \( x_0 \)
  • improve it iteratively

• Suppose we pick \( x_0 = a \) and actual root is \( r; f(r) = 0 \)
• Let \( a + h = r \)

So, we have
\[
f(r) = 0, x_0 = a \quad \text{and let } r = a + h
\]
\[
f(r) = f(a + h)
\]

• Consider the line tangent to \( f(a) \)
given by \( \nabla f(a) \).
• It intercepts the x axis at \( b \)
Newton-Raphson method

Tangent line through $(b,0)$ and $(a,f(a))$: $y = (x-a)f'(a) + f(a)$

Let's find $b$'s value by setting $y=0$

$0 = (x-a)f'(a) + f(a) \implies x = a - \frac{f(a)}{f'(a)}$

Newton-Raphson method

• Linear approximation $x_{i+1} = a - \frac{f(a)}{f'(a)}$ provides a new approximation of the root.

• Iterate until convergence

• Very good with good starting points, not so good with bad ones...
Newton-Raphson and local search

- We want to find states where gradient of optimization function is zero: \( \nabla f(x) = 0 \)

- Newton-Raphson lets us find this, but we use the derivative of the gradient, or second derivative

Newton-Raphson method

- In airport optimization, we computed \( \frac{\partial f}{\partial x_i} \) and \( \frac{\partial f}{\partial y_j} \)

- As we find the roots of the derivative, we need to find \( \frac{\partial^2 f}{\partial x_i \partial x_j} \) and \( \frac{\partial^2 f}{\partial y_i \partial y_j} \) and \( \frac{\partial^2 f}{\partial x_i \partial y_j} \)

\[
\frac{\partial^2 f}{\partial x_i \partial y_j} \sum_{i=1}^{2} \sum_{c \in C_i} (x_i - x_c)^2 + (y_j - y_c)^2
\]

\[
= \frac{\partial f}{\partial y_j} \left( \sum_{c \in C_i} (x_i - x_c) \right) \bar{x}_i
\]

\[
= 0
\]

\[
= \frac{\partial f}{\partial x_i} \left( \sum_{c \in C_i} (x_i - x_c) \right) \bar{y}_j
\]

\[
= 2
\]
Newton-Raphson method

• Derivatives can be arranged in Hessian matrix

\[ H_f(x) = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}
\end{bmatrix} \]

In this case diagonals are 2, off diagonals are 0

Newton-Raphson method

Update function becomes

\[ x_{i+1} = x - H_f^{-1}(x_i) \nabla f(x_i) \]

where \( H_f^{-1}(x_i) \) is the inverse of the Hessian matrix \( H_f(x_i) \)

We will not cover constrained optimization
which lets us add conditions that must hold, e.g.:

\( (x_i, y_i) \) cannot be on a mountain
\( (x_i, y_i) \) cannot be in a lake
Actions and contingency plans

• Deterministic
  • Percepts only needed for initial state
  • We know the results of every action

• Non-deterministic
  • No longer sure what the next state is

• Partially observable
  • Might not be certain of initial state

Non-deterministic/partially observable environments require *contingency plans* (aka strategies)

Contingency plans

• We redefine the result of an action such that it returns multiple possible states.

Example for a partially observable environment

```
result(state(xy(32,45), ok), deltaxy_m(0,3)) ←
{ state(xy(32,48), falling), state(xy(32,48), ok) }
```

See erratic vacuum world section
4.3.1 for a more developed example
And-or search trees

allow representation of multiple outcomes

Agent makes choice at or node
And node represents possible outcomes of that choice.

Solution: subtree with
• Goal at every leaf
• 1+ action for each or node
• includes all outcome branches from each and node

And-or search

• To simplify, assume a single start state
• Expand the node and take actions
  • or nodes – represent deterministic choices
  • and nodes – environment decides outcome of an action
    (nondeterministic as far as agent is concerned)
• With or nodes, we continue searching for a solution.
• With and nodes, there needs to be a solution along every node of the and.
And-or search

function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
   OR-SEARCH(problem. INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
   if problem.GOAL-TEST(state) then return the empty plan
   if state is on path then return failure
   for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
   return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
   return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_n-1 then plan_{n-1} else plan_n]