Artificial Intelligence Programming
Local Search

Chris Brooks

Department of Computer Science
University of San Francisco
Local Search - When is it useful?
Hill-climbing search
Simulated Annealing
Genetic Algorithms
5-1: Local Search

- So far, the algorithms we’ve looked at store the entire path from initial state to goal state.
- This leads to memory/space issues on large problems.
- For some problems, path information is essential
  - Route finding
  - Fox and chickens
  - 8-puzzle
  - What we’re interested in is the sequence of actions to take
  - We know the goal, but not how to get to it.
For other sorts of problems, we may not care what the sequence of actions is.

- Finding the optimal solution is what’s important.
- Scheduling
- VLSI layout
- Cryptography

In these cases, we can safely discard at least some of the path information.
A search algorithm that uses only the current state (as opposed to path information) is called a *local search* algorithm.

**Advantages:**
- Constant memory requirements
- Can find solutions in incredibly large spaces.

**Disadvantages:**
- Typically not optimal - only local optima will be found.
- Can oscillate, due to lack of memory.
Local search is often useful for optimization problems

“Find parameters such that $o(x)$ is maximized/minimized”

This is a search problem, where the state space is the combination of parameters.

If there are $n$ parameters, we can imagine an $n + 1$ dimensional space, where the first $n$ dimensions are the parameters of the function, and the $n + 1$th dimension is the objective function.

We call this space a search landscape

- Optima are on hills
- Valleys are poor solutions.
- (reverse this to minimize $o(x)$)
A one-dimensional landscape:

Linear Pricing for varying C and N. w = 10, k ~ U[0.0, 0.7]
5-6: Search Landscape

A two-dimensional landscape:

(beyond 2 dimensions, they’re tough to draw)
5-7: Search landscapes

- Landscapes turn out to be a very useful metaphor for local search algorithms.
- Lets us visualize ’climbing’ up a hill.
- Gives us a way of differentiating easy problems from hard problems.
  - Easy: few peaks, smooth surfaces, no ridges/plateaus
  - Hard: many peaks, jagged or discontinuous surfaces, plateaus.
5-8: Hill-climbing search

The simplest form of local search is hill-climbing search.

Very simple: at any point, look at your “successors” (neighbors) and move in the direction of the greatest positive change.

Very similar to greedy search
- Pick the choice that myopically looks best.

Very little memory required.

Will get stuck in local optima.

Plateaus can cause the algorithm to wander aimlessly.
5-9: Local search in Calculus

6 Find roots of an equation $f(x) = 0$, $f$ differentiable.
6 Guess an $x_1$, find $f(x_1)$, $f'(x_1)$
6 Use the tangent line to $f(x_1)$ (slope = $f'(x_1)$) to pick $x_2$.
6 Repeat. $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_1)}$
6 This is a hill-climbing search.
6 Works great on smooth functions.
5-10: Improving hill-climbing

- Hill-climbing can be appealing
  - Simple to code
  - Requires little memory
  - We may not have a better approach.

- How to make it better?

- Stochastic hill-climbing - pick randomly from uphill moves
  - Weight probability by degree of slope
5-11: Improving hill-climbing

- Random-restart hill-climbing
- Run until an optimum is reached
- Randomly choose a new initial state
- Run again.
- After $n$ iterations, keep best solution.
5-12: Simulated Annealing

- Hill-climbing’s weakness is that it never moves ’downhill’
- Like greedy search, it can’t “back up”.
- Simulated annealing is an attempt to overcome this.
- “Bad” actions are occasionally chosen to move out of a local optimum.
5-13: *Simulated Annealing*

- Based on analogies to crystal formation.
- When a metal cools, lattices form as molecules fit into place.
- By reheating and recooling, a harder metal is formed
  - Small undoing leads to better solution.
  - Minimize the “energy” in the system
- Similarly, small steps away from the solution can help hill-climbing escape local optima.
5-14: Simulated Annealing

T = initial
s = initial-state
while (s != goal)
    ch = successor-fn(s)
    c = select-random-child(ch)
    if c is better than s
        s = c
    else
        s = c with probability p(T, c, s)
    update T

What is T?

What is p?
What we want to do is make “mistakes” more frequently early in the search and more rarely later in the search.

We’ll use $T$ to parameterize this.

$T$ stands for temperature.

Two questions:
- How does $T$ change over time?
- What’s the probability function wrt $T$?
The function for changing $T$ is called a cooling schedule.

The most commonly used schedules are:

- Linear: $T_{new} = T_{old} - dt$
- Proportional: $T_{new} = c \times T_{old}, c < 1$
5-17: Boltzmann distribution

- The probability of accepting a mistake is governed by a Boltzmann distribution.
- Let \( s \) be the current state, \( c \) be the child considered, and \( o \) the function to optimize.

\[
P(c) = e^{\frac{-|o(c) - o(s)|}{T}}
\]

- Consider boundary conditions:
  - \(|o(c) - o(s)| = 0\), then \( P(c) = 1 \).
  - \( T \) very high: almost all fractions near 0, so \( P(c) \) near 1.
  - \( T \) low: \( P(c) \) depends on \(|o(c) - o(s)|\), typically small.
- Boltzmann gives us a way of weighting the probability of accepting a “mistake” by its quality.
Simulated Annealing is complete and optimal as long as $T$ is lowered “slowly enough”.

Can be very effective in domains with many optima.

Simple addition to a hill-climbing algorithm.

Weakness - selecting a good cooling schedule.

No problem knowledge used in search. (weak method)
Genetic Algorithms can be thought of as a form of parallel hill-climbing search.

Basic idea:
- Select some solutions at random.
- Combine the best parts of the solutions to make new solutions.
- Repeat.

Successors are a function of two states, rather than one.
5-20: GA terminology

- Genome - a solution or state
- Trait/gene - a parameter or state variable
- Fitness - the “goodness” of a solution
- Population - a set of genomes or solutions.
pop = makeRandomPopulation
while (not done)
    foreach p in pop
        evaluate(p)
    for i = 1 to size(pop)
        select random solutions from pop
crossover solutions
mutate solutions
pop = new solutions
Assume that we can encode our problem as a bitstring.

Example: 8 queens. For each column, we use 3 bits to encode the row of the queen = 24 bits.

100 101 110 000 101 001 010 110 = 4 5 6 0 5 1 2 6

We begin by generating random bitstrings, then evaluating them according to a *fitness function* (the function to optimize).

N-queens: number of nonattacking pairs of queens (max = 28)
5-23: Generating new solutions

Our successor function will work by operating on two solutions.
This is called crossover.
Pick two solutions at random.
First method: Fitness-proportionate selection
  △ Sum all fitnesses
  △ \( P(\text{selection of } s) = \text{fitness}(s) / \text{total fitness} \)
Pick a random point on the bitstrings. (locus)
Merge the first part of \( b_1 \) with the second part of \( b_2 \) (and vice versa) to produce two new bitstrings.
Crossover Example

s1: (100 101 110) (000 101 001 010 110) = 4 5 6 0 5 1 2 6

s2: (001 000 101) (110 111 010 110 111) = 1 0 5 6 7 2 6 7

Pick locus = 9

s3 = (100 101 110) (110 111 010 110 111) 4 5 6 6 7 2 6 7

s4 = (001 000 101) (000 101 001 010 110) 1 0 5 0 5 1 2 6
6 Next, apply mutation.

6 With probability $m$ (for small $m$) randomly flip one bit in the solution.

6 After generating a new population of the same size as the old population, discard the old population and start again.
5-26: So what is going on?

6 Why would this work?

6 Crossover: recombine pieces of partially successful solutions.

6 Genes closer to each other are more likely to stay together in successive generations.
   △ This makes encoding important.

6 Mutation: inject new solutions into the population.
   △ If a trait was missing from the initial population, crossover cannot generate it unless we place the locus within a gene.

6 We’ll see a more formal proof of this (the schema theorem) on Monday.
Keep in mind that this is *not* how biological evolution works.

- Biological evolution is much more complex.
- Diploid chromosomes, phenotype/genotype, nonstationary objective functions, ...

Biology is a nice metaphor.

- GAs must stand or fail on their own merits.
There are many more GA-related issues to consider:

- Proofs of convergence
- How to encode a problem
- Using real-valued parameters
- Alternative selection criteria
- Other crossover methods.
- Learning higher-order solutions.

All this and more, on our next episode ...