



Artificial Intelligence Programming

Genetic Algorithms and Local Search

Chris Brooks

Department of Computer Science

University of San Francisco



Overview

- Local Search - When is it useful?
- Hill-climbing search
- Simulated Annealing
- Genetic Algorithms

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
 - Water Jugs
 - 8-puzzle
 - The solution *is* the sequence of actions to take.
 - We know what the goal state is, but not how to reach it.

Local Search

- For other sorts of problems, we may not care what the sequence of actions is.
- CSPs fit this description.
 - Finding the optimal (or satisfactory) solution is what's important.
 - Scheduling
 - VLSI layout
 - Cryptography
 - Function optimization
 - Protein folding, gene sequencing
- The solution is an assignment of values to variables that maximizes some objective function.
- In these cases, we can safely discard at least some of the path information.

Local Search

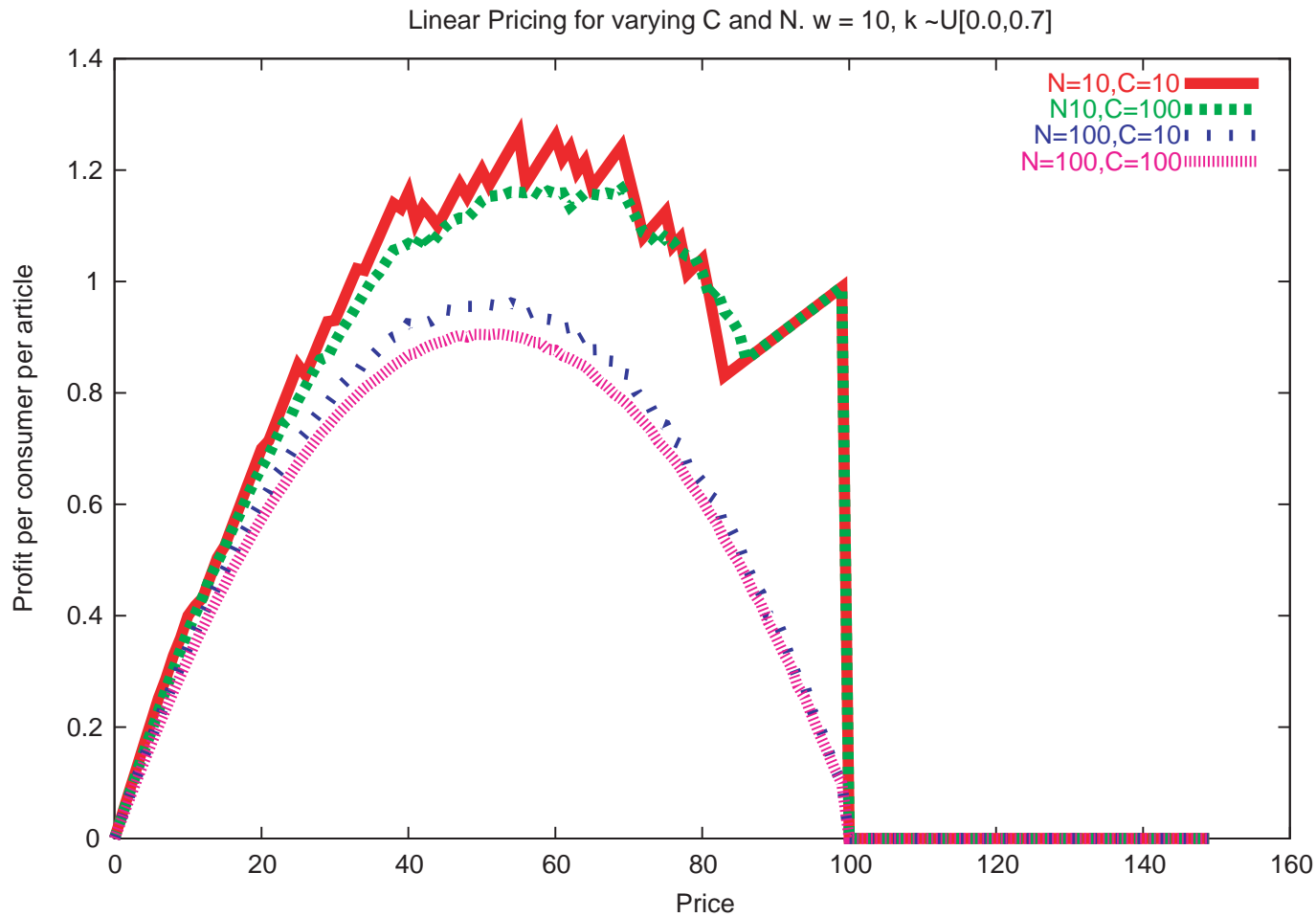
- A search algorithm that uses only the current state (as opposed to path information) is called a *local search* algorithm.
- Advantages:
 - Constant memory requirements
 - Can find solutions in incredibly large spaces.
- Disadvantages:
 - Hard to guarantee optimality; we might only find a local optimum
 - Lack of memory may cause us to revisit states or oscillate.

Search Landscape

- 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 value assignments to 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 $o(x)$)

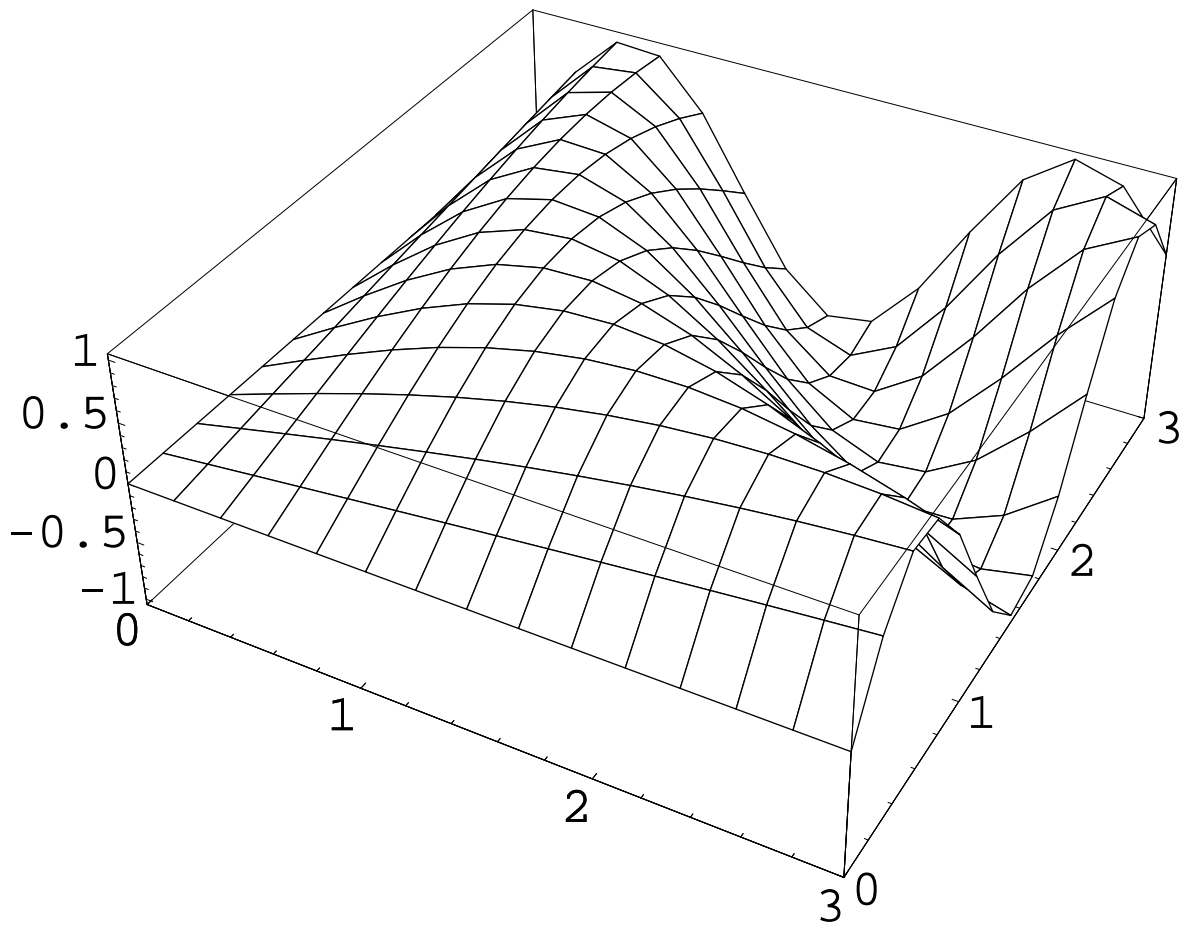
Search Landscape

• A one-dimensional landscape:



Search Landscape

- A two-dimensional landscape:



(beyond 2 dimensions, they're tough to draw)

Search landscapes

- Landscapes turn out to be a very useful metaphor for local search algorithms.
- Lets us visualize ‘climbing’ up a hill (or descending a valley).
- 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.

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.

Local search in Calculus

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

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

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.
 - If we have a guess as to the number of optima, we can choose an n .

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.

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.

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?

Simulated Annealing

- 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 with respect to T ?

Cooling schedule

- The function for changing T is called a cooling schedule.
- The most commonly used schedules are:
 - Linear: $T_{new} = T_{old} - dt$
 - Proportional: $T_{new} = c * T_{old}, c < 1$

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.
- $P(c) = \exp\left(\frac{-|o(c) - o(s)|}{T}\right)$
- 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

- 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

- 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.

GA applications

- Function optimization
- Job Shop Scheduling
- Factory scheduling/layout
- Circuit Layout
- Molecular structure
- etc

GA terminology

- Chromosome - a solution or state
- Trait/gene - a parameter or state variable
- Fitness - the “goodness” of a solution
- Population - a set of chromosomes or solutions.

A Basic GA

```
pop = makeRandomPopulation
while (not done)
  foreach p in pop
    p.fitness = evaluate(p)
  for i = 1 to size(pop) by 2:
    parent1, parent2 = select random solutions from pop
    child1, child2 = crossover (parent1, parent2)
    mutate child1, child2
  replace old population with new population
```

Analogies to Biology

- 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.

Encoding a Problem

- Encoding a problem for use by a GA can be quite challenging.
- Traditionally, GA problems are encoded as bitstrings
- 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).
 - 8-queens: number of nonattacking pairs of queens (max = 28)

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 b1 with the second part of b2 (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

Mutation

- Next, apply mutation.
- With probability m (for small m) randomly flip one bit in the solution.
- After generating a new population of the same size as the old population, discard the old population and start again.

So what is going on?

- Why would this work?
- Crossover: recombine pieces of partially successful solutions.
- Genes closer to each other are more likely to stay together in successive generations.
 - This makes encoding important.
- 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.
- The *schema theorem* provides a formal proof of why GAs work.

Selection

- How should we select parents for reproduction?
- Use the best n percent?
 - Want to avoid premature convergence
 - No genetic variation
 - Also, sometimes poor solutions have promising subparts.
- Purely random?
 - No selection pressure

Roulette Selection

- *Roulette Selection* weights the probability of a chromosome being selected by its relative fitness.
- $$P(c) = \frac{fitness(c)}{\sum_{chr \in pop} fitness(chr)}$$
- This normalizes fitnesses; total relative fitnesses will sum to 1.
- Can directly use these as probabilities.

Example

- Suppose we want to maximize $f(x) = x^2$ on $[0, 31]$
 - Let's assume integer values of x for the moment.
- Five bits used to encode solution.
- Generate random initial population

String	Fitness	Relative Fitness
01101	169	0.144
11000	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

Example

- Select parents with roulette selection.
- Choose a random *locus*, and crossover the two strings

String	Fitness	Relative Fitness
0110 1	169	0.144
1100 0	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

- Children: 01100, 11001

Example

- Select parents with roulette selection.
- Choose a random *locus*, and crossover the two strings

String	Fitness	Relative Fitness
01101	169	0.144
11 000	576	0.492
01000	64	0.055
10 011	361	0.309
Total	1170	1.0

- Children: 01100, 11001
- Children: 10000, 11011

Example

- Replace old population with new population.
- Apply mutation to new population.
 - With a small population and low mutation rate, mutations are unlikely.
- New generation:
 - 01100, 11001, 11011, 10000
- Average fitness has increased (293 to 439)
- Maximum fitness has increased (576 to 729)

What's really going on here?

- The subsolutions 11^{***} and $***11$ are recombined to produce a better solution.
- There's a correlation between strings and fitness.
 - Having a 1 in the first position is correlated with fitness.
 - This shouldn't be shocking, considering how we encoded the input.
- We'll call a 1 in the first position a building block.
- GAs work by recombining smaller building blocks into larger building blocks.

Schemata

- We need a way to talk about strings that are similar to each other.
- Add '*' (don't care) symbol to $\{0,1\}$.
- A *schema* is a template that describes a set of strings using $\{0,1,*\}$
 - 111^{**} matches 11100, 11101, 11110, 11111
 - $0^{*}11^{*}$ matches 00110, 00111, 01110, 01111
 - $0^{***}1$ matches 00001, 00011, 00101, 00111, 01001, 01011, 01101, 01111
- Premise: Schemata are correlated with fitness.
- In many encodings, only some bits matter for a solution. Schemata give us a way of describing all the important information in a string.

Schemata

- GAs actually process schemata, rather than strings.
- Crossover may or may not damage a schema
 - $**11*$ vs $0***1$
- Short, highly fit low-order schemata are more likely to survive.
 - Order: the number of fixed bits in a schema
 - $1****$ - order 1
 - $0*1*1$ - order 3
- Building Block Hypothesis: GAs work by combining low-order schemata into higher-order schemata to produce progressively more fit solutions.

Propagation of schemata

- Suppose that there are m examples of a schema H in a population of size n at time t .: $m(H, t)$

- Strings are selected according to relative fitness.

$$p = \frac{f}{\sum f}$$

- At time $t + 1$, we will have

$m(H, t + 1) = M(H, t) * n * \frac{f(H)}{\sum f}$, where $f(H)$ is the average fitness of strings represented by this schema.

- In other words, schema grow or decay according to their fitness relative to the rest of the population.

- Above average schemata receive more samples

- Below average schemata receive fewer samples.

- But how many more or less?

Propagation of schemata

- Assume that schema H remains above average by an amount c .
- We can rewrite the schema difference equation as:
 - $m(H, t + 1) = (1 + c)m(H, t)$
 - Starting at $t = 0$, we get:
 - $m(H, t) = m(H, 0)(1 + c)^t$
 - This is a geometric progression. (Also the compound interest formula)
- Reproduction selects exponentially more (fewer) above (below) average schemata.

Schemata and crossover

- Selection is only half the story
- Schemata with longer defining length are more likely to be damaged by crossover.
- $P(\textit{survival}) \geq 1 - \frac{\textit{defininglength}}{\textit{Strlen}-1}$
- We can combine this with the previous equation to produce:
 - $m(H, t + 1) \geq m(H, t) \frac{f(H)}{\sum f} \left(1 - \frac{\textit{defininglength}}{\textit{Strlen}-1}\right)$
- In words, short, above-average schemata are sampled at exponentially increasing rates.
- This is known as the *schema theorem*.

Schema in our example

- Consider $H - 1 = 1****$, $H_2 = *10**$, $H_3 = 1***0$ in our

previous example.

String	Fitness	Relative Fitness
01101	169	0.144
11000	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

- Parents: 2 (twice), 3, 4
 - 2 and 4 are both instances of H_1
- Children: 01100, 11001, 11011, 10000
 - 3 instances of H_1
 - Schema theorem predicts $m \frac{f(H)}{\sum f} = \frac{468.5}{293} = 3.2$ copies

Theory vs. Implementation

- Schema Theorem shows us *why* GAs work.
- In practice, implementation details can make a big difference in the effectiveness of a GA.
- This includes algorithmic improvements and encoding choices.

Tournament Selection

- Roulette selection is nice, but can be computationally expensive.
 - Every individual must be evaluated.
 - Two iterations through entire population.
- *Tournament selection* is a much less expensive selection mechanism.
- For each parent, choose two individuals at random.
- Higher fitness gets to reproduce.

Elitism

- In practice, discarding all solutions from a previous generation can slow down a GA.
 - Bad draw on RNG can destroy progress
 - You may need monotonic improvement.
- *Elitism* is the practice of keeping a fraction of the population from the previous generation.
- use Roulette selection to choose a fraction of the population to carry over without crossover.
- Varying the fraction retained lets you trade current performance for learning rate.

Knowing when to stop

- In some cases, you can stop whenever your GA finds an acceptably good solution.
- In other cases, it's less clear
 - How do we know we've found the best solution to TSP?
- Stop when population has 'converged'
 - Without mutation, eventually one solution will dominate the population
- After 'enough' iterations without improvement

Encoding

- The most difficult part of working with GAs is determining how to encode problem instances.
 - Schema theorem tells us that short encodings are good
 - Parameters that are interrelated should be located near each other.
- N queens: Assume that each queen will go in one column.
- Problem: find the right row for each queen
- N rows requires $\log_2 N$ bits
- Entire length of string: $N * \log_2 N$

Encoding Real-valued numbers

- What if we want to optimize a real-valued function?
- $f(x) = x^2, x \in \text{Reals}[0, 31]$
- Decide how to discretize input space; break into m “chunks”
- Each chunk coded with a binary number.
- This is called *discretization*

Permutation operators

- Some problems don't have a natural bitstring representation.
- e.g. Traveling Salesman
 - Encoding this as a bitstring will cause problems
 - Crossover will produce lots of invalid solutions
- Encode this as a list of cities: [3,1,2,4,5]
- Fitness: MAXTOUR - tour length (to turn minimization into maximization.)

Partially Matched Crossover

- How to do crossover in this case?
- Exchange *positions* rather than substrings.
- Example:
 - t1: 3 5 4 6 1 2 8 7
 - t2: 1 3 6 5 8 7 2 4
- First, pick two loci at random.

Partially Matched Crossover

- t1: 3 5 | 4 6 1 2 | 8 7
- t2: 1 3 | 6 5 8 7 | 2 4
- Use pairwise matching to exchange corresponding cities on each tour:
 - In each string, 4 and 6 trade places, as do 6 and 5, 1 and 8, and 2 and 7.
 - New children:
 - c1: 2 6 5 4 8 7 1 2
 - c2: 8 3 4 6 1 2 7 5
- Intuition: Building blocks that are sections of a tour should tend to remain together.

Permutation strategies

- PMX is just one of many approaches to using GAs to solve permutation problems.
 - Scheduling, TSP, route finding, etc
- Can also encode the *position* of each city.
- Can try to replace subtours.
- Fertile research field, with many practical applications.

Summary

- GAs use bitstrings to perform local search through a space of possible schema.
- Quite a few parameters to play with in practice.
- Representation is the hardest part of the problem.
- Very effective at searching vast spaces.