Introduction to Genetic Algorithms

Peter G. Anderson

Dealing with Hard Problems

Some Problems We Just Don't Know How to Solve!
. . . but we do know how to critique a “solution.”

Dealing with Hard Problems

Some Problems We Just Don't Know How to Solve!
. . . but we do know how to critique a "solution."
Coloring graphs is hard, but counting colors and violations is easy (a violation is two adjacent vertices with the same color).

Dealing with Hard Problems

Some Problems We Just Don't Know How to Solve!
. . . but we do know how to critique a "solution."
Coloring graphs is hard, but counting colors and violations is easy (a violation is two adjacent vertices with the same color).
Finding the shortest salesman's path is hard, but measuring a path is easy.

Dealing with Hard Problems

Some Problems We Just Don't Know How to Solve!
. . . but we do know how to critique a "solution."
Coloring graphs is hard, but counting colors and violations is easy (a violation is two adjacent vertices with the same color).
Finding the shortest salesman's path is hard, but measuring a path is easy.
Scheduling examinations or assigning teachers to classes is hard, but counting the conflicts (ideally there are none) is easy.

Dealing with Hard Problems

Some Problems We Just Don't Know How to Solve!

. . . but we do know how to critique a "solution."

Coloring graphs is hard, but counting colors and violations is easy (a violation is two adjacent vertices with the same color).

Finding the shortest salesman's path is hard, but measuring a path is easy.

Scheduling examinations or assigning teachers to classes is hard, but counting the conflicts (ideally there are none) is easy.

Computer programs are hard to write, but counting bugs is easy.

GAs Emulate Selective Breeding

Designing tender chickens is hard; taste-testing them is easy.
Designing thick-skinned tomatoes is hard; dropping is easy.
So, the breeders iterate:

GAs Emulate Selective Breeding

Designing tender chickens is hard; taste-testing them is easy.
Designing thick-skinned tomatoes is hard; dropping is easy.
So, the breeders iterate:

- **Selection:** Cull their population of the inferior members.

GAs Emulate Selective Breeding

Designing tender chickens is hard; taste-testing them is easy.
Designing thick-skinned tomatoes is hard; dropping is easy.
So, the breeders iterate:

- **Selection:** Cull their population of the inferior members.
- **Crossover:** Let the better members breed.

GAs Emulate Selective Breeding

Designing tender chickens is hard; taste-testing them is easy.
Designing thick-skinned tomatoes is hard; dropping is easy.
So, the breeders iterate:

- **Selection:** Cull their population of the inferior members.
- **Crossover:** Let the better members breed.
- **Mutation:** X-ray them.

Applications I Have Known

Choosing among 1,500 features for OCR. (My first GA!)
Scheduling the Chili, NY, annual soccer invitational.
Scheduling my wife's golf league.
Designing LED lenses.
Programming a synchronizing cellular automaton.
Designing halftone screens for laser printers.
N Queens, Coloring Graphs, Routing Salesmen, etc., etc.

Subsetting 1,500 OCR Features

The polynet OCR engine trains and executes rapidly.

Performance was competitive.

We wanted to embed it in hardware, but it used 1,500 features.

We could deal with 300 features.

So, we bred high-performance feature subsets.

Soccer Scheduling

Bill Gustafson's MS Project, May, 1998

The Chili Soccer Association hosts an annual soccer tournament.

131 teams, 209 games, 14 fields, 17 game times.

a long weekend for a group of schedulers,

. . . . . and then some teams back out. . .

Soccer Scheduling Hard Constraints

A field can have one game at a time.
A team can only play one game at a time.
Teams must play on appropriate size fields.
Late games must be played on lighted fields.
A team must rest one game period (two is better) between games.
Teams can only play when they can be there (some can't come Friday)

Soccer Scheduling Soft Constraints

A team's games should be distributed evenly over the playing days.
Teams should play in at most two playing areas.
Each team should play at least once in the main playing area.
Teams should play in areas where they have a preference.
Games should finish as early as possible on Sunday.
Etc...

GAs Use Selective Breeding to . . .

discover a really good bit string

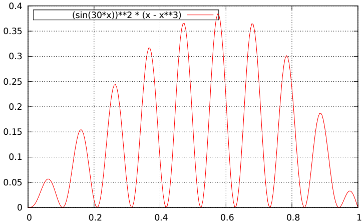
CS Dogma: Bit Strings Represent Everything

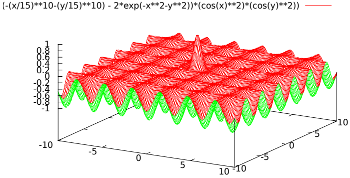
$$B = \{b_1, b_2, \ldots, b_n\}$$

1. A subset of an *n*-set (where the 1's are)
2. A number x in $[0,1)$: $x = \sum_1^n b_k 2^{-k}$
3. A pair (x, y) in $[0,1)^2$: $x = \sum_1^{n/2} b_k 2^{-k} \quad y = \sum_{n/2+1}^{n} b_k 2^{-k}$

Examples

1. Set searching
   1. Search for the biggest subset possible (maximize 1's count)
   2. Knapsack, bin packing
   3. Maximum independent set, map coloring
2. Maximize $f(x)$
3. Maximize $f(x, y)$





How to Search for Good Bit Stings

1. Enumerate all possibilities (but 2^n gets big)
2. Random search – “explore”
3. Hill climb – “exploit”
4. Genetic algorithm
5. Simulated annealing
6. Firefly algorithm

The Parts of a GA

1. A population of genotypes – e.g., bit strings.

The Parts of a GA

1. A population of genotypes – e.g., bit strings.
2. Fitness of phenotypes – the selection mechanism.

The Parts of a GA

1. A population of genotypes – e.g., bit strings.
2. Fitness of phenotypes – the selection mechanism.
3. Crossover – sexual reproduction of relatively superior individuals.

The Parts of a GA

1. A population of genotypes – e.g., bit strings.
2. Fitness of phenotypes – the selection mechanism.
3. Crossover – sexual reproduction of relatively superior individuals.
4. Mutation.

The Parts of a GA

1. A population of genotypes – e.g., bit strings.
2. Fitness of phenotypes – the selection mechanism.
3. Crossover – sexual reproduction of relatively superior individuals.
4. Mutation.
5. Elimination of low fit individuals – more selection.

My Algorithm

See Octave code.

Parameters: Exploration vs. Exploitation

1. Population size

Parameters: Exploration vs. Exploitation

1. Population size
2. Mutation rate

Parameters: Exploration vs. Exploitation

1. Population size
2. Mutation rate
3. Tournament size

Crossover Variations

1. One point: interchange random-size prefix.

Crossover Variations

1. One point: interchange random-size prefix.
2. Two point: interchange random-positioned substring.

Crossover Variations

1. One point: interchange random-size prefix.
2. Two point: interchange random-positioned substring.
3. Uniform (in my code):
   $\text{child}_{1,i}$ = either $\text{parent}_{1,i}$ or $\text{parent}_{2,i}$
   $\text{child}_{2,i}$ = complementary

Crossover Variations

1. One point: interchange random-size prefix.
2. Two point: interchange random-positioned substring.
3. Uniform (in my code):
   $\text{child}_{1,i} = \text{either parent}_{1,i} \text{ or parent}_{2,i}$
   $\text{child}_{2,i} = \text{complementary}$
   Uniform crossover is twice as fast for “baby problem” – why?

Crossover Variations

1. One point: interchange random-size prefix.
2. Two point: interchange random-positioned substring.
3. Uniform (in my code):
   $\text{child}_{1,i}$ = either $\text{parent}_{1,i}$ or $\text{parent}_{2,i}$
   $\text{child}_{2,i}$ = complementary
   Uniform crossover is twice as fast for "baby problem" – why?
4. Bizarre: pick and sort three individuals, $X > Y > Z$
   if $X_i == Y_i$, then $\text{child}_i := X_i$
   else $\text{child}_i := \textit{not}\ Z_i$

How to Compare Algorithms

1. Count number of fitness evaluations,
   the lion’s share of the computation cost.
   (Profile!)

How to Compare Algorithms

1. Count number of fitness evaluations,
   the lion's share of the computation cost.
   (Profile!)
2. Test a large number of times.
   Report the *median*.

Variations on an Algorithm

I choose simplicity. (Does it matter?)

1. Use generations of populations.
   Initial population$_0$ is random.
   Population$_{t+1}$ is mutated children from Population$_t$.

Variations on an Algorithm

I choose simplicity. (Does it matter?)

1. Use generations of populations.
   Initial population$_0$ is random.
   Population$_{t+1}$ is mutated children from Population$_t$.
2. A student suggested:
   Iteratively create a mutated child from *any two* individuals and replace the current worst.

Variations on an Algorithm

I choose simplicity. (Does it matter?)

1. Use generations of populations.
   Initial population$_0$ is random.
   Population$_{t+1}$ is mutated children from Population$_t$.
2. A student suggested:
   Iteratively create a mutated child from *any two* individuals and replace the current worst.
3. Parallelize: use islands of populations.
   Occasionally allow immigration.

Variations on an Algorithm

I choose simplicity. (Does it matter?)

1. Use generations of populations.
   Initial population$_0$ is random.
   Population$_{t+1}$ is mutated children from Population$_t$.
2. A student suggested:
   Iteratively create a mutated child from *any two* individuals and replace the current worst.
3. Parallelize: use islands of populations.
   Occasionally allow immigration.
4. Iteratively: Remove worst half of population.
   Randomly line up the survivors.
   For every adjacent pair, create and mutate a child.

Upcoming

- Programming cellular automata
- Permutation-based GA