Introduction to Genetic Algorithms

A Tutorial by Erik D. Goodman

Professor, Electrical and Computer Engineering

Professor, Mechanical Engineering

Co-Director, Genetic Algorithms Research and Applications Group (GARAGe)

Michigan State University

Chair, ACM SIGEVO

Thanks to:

Much of this material is based on:

- David Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, 1989 (still one of the best introductions!)
- Darrell Whitley, “Genetic Algorithm Tutorial” – on the web at www.cs.colostate.edu/~genitor/MiscPubs/tutorial.pdf
- Robert Heckendorn, 2003 GECCO Tutorial

Overview of Tutorial

- Quick intro – What IS a genetic algorithm?
  - Classical, binary chromosome
- Where used, & when better to use something else
- A little theory – why a GA works
- GA in Practice -- some modern variants

Genetic Algorithms:

- Are a method of search, often applied to optimization or learning
- Are stochastic – but are *not* random search
- Use an evolutionary analogy, “survival of fittest”
- Not *fast* in some sense; but sometimes more robust; scale relatively well, so can be useful
- Have extensions including Genetic Programming (GP) (LISP-like function trees), learning classifier systems (evolving rules), linear GP (evolving “ordinary” programs), many others

The Canonical or Classical GA

- Maintains a set or "population" of <u>strings</u> at each stage
- Each string is called a chromosome, and encodes a "candidate solution"– CLASSICALLY, encodes as a *binary string* (and now in almost any conceivable representation)

Criterion for Search

- Goodness ("fitness") or optimality of a string's solution determines its FUTURE influence on search process -- survival of the fittest
- Solutions which are good are used to generate other, similar solutions which may also be good (even better)
- The POPULATION at any time stores ALL we have learned about the solution, at any point
- Robustness (efficiency in finding good solutions in difficult searches) is key to GA success

Classical GA: The Representation

`1011101010 –` a possible 10-bit string representing a possible solution to a problem

Bit or subsets of bits might represent choice of some feature, for example. "4WD" "2-door" "4-cylinder" "closed cargo area" "blue" might be meaning of chrom above, to evaluate as the new standard vehicles for the US Post Office

Each position (or each set of positions that encodes some feature) is called a LOCUS (plural LOCI)

Each possible value at a locus is called an ALLELE

How Does a GA Operate?

- For ANY chromosome, must be able to determine a FITNESS (measure performance toward an objective)
- Objective may be maximized or minimized; usually say *fitness* is to be maximized, and if objective is to be minimized, define fitness from it as something to maximize

GA Operators: Classical Mutation

- Operates on ONE “parent” chromosome
- Produces an “offspring” with changes.
- Classically, toggles one bit in a binary representation
- So, for example: `1101000110` could mutate to: `1111000110`
- Each bit has same probability of mutating

Classical Crossover

- Operates on two parent chromosomes
- Produces one or two children or offspring
- Classical crossover occurs at 1 or 2 points:
- For example: (1-point) (2-point)

```
           1111111111 or   1111111111
      x    0000000000      0000000000
           1110000000      1110000011
and        0001111111      0001111100
```

Selection

- *Traditionally*, parents are chosen to mate with probability proportional to their fitness: *proportional selection*
- Traditionally, children replace their parents
- Many other variations now more commonly used (we’ll come back to this)
- Overall principle: survival of the fittest

Synergy – the KEY

Clearly, selection alone is no good …

Clearly, mutation alone is no good …

Clearly, crossover alone is no good …

Fortunately, using all three simultaneously is sometimes spectacular!

Contrast with Other Search Methods

- "indirect" -- setting derivatives to 0
- "direct" -- hill climber
- enumerative – search 'em all
- random – just keep trying, or can avoid resampling
- simulated annealing – single-point method, reals, changes all loci randomly by decreasing amounts, mostly keeps the better answer, …
- Tabu (another common method)

When Might a GA Be Any Good?

- Highly multimodal functions
- Discrete or discontinuous functions
- High-dimensionality functions, including many combinatorial ones
- Nonlinear dependencies on parameters (interactions among parameters) -- "epistasis" makes it hard for others
- Often used for approximating solutions to NP-complete combinatorial problems
- DON'T USE if a hill-climber, etc., will work well

The Limits to Search

- No search method is best for all problems – per the No Free Lunch Theorem
- Don't let anyone tell you a GA (or THEIR favorite method) is best for all problems!!!
- Needle-in-a-haystack is just *hard*, in practice
- Efficient search must be able to EXPLOIT correlations in the search space, or it's no better than random search or enumeration
- Must balance with EXPLORATION, so don't just find nearest local optimum

Examples of Successful Real-World GA Application

- Antenna design
- Drug design
- Chemical classification
- Electronic circuits (Koza)
- Factory floor scheduling (Volvo, Deere, others)
- Turbine engine design (GE)
- Crashworthy car design (GM/Red Cedar)
- Protein folding
- Network design
- Control systems design
- Production parameter choice
- Satellite design
- Stock/commodity analysis/trading
- VLSI partitioning/ placement/routing
- Cell phone factory tuning
- Data Mining

"Genetic Algorithm" -- Meaning?

- "classical or canonical" GA -- Holland (taught in '60's, book in '75) -- binary chromosome, population, selection, crossover (recombination), low rate of mutation
- More general GA: population, selection, (+ recombination) (+ mutation) -- may be hybridized with LOTS of other stuff

Representation Terminology

- Classically, binary string: individual or chromosome
- What's on the chromosome is GENOTYPE
- What it *means* in the problem context is the PHENOTYPE (e.g., binary sequence may map to integers or reals, or order of execution, or inputs to a simulator, etc.)
- Genotype determines phenotype, but phenotype may *look* very different

Discretization – Representation Meets Mutation!

- If problem is binary decisions, bit-flip mutation is fine
- BUT if using binary numbers to encode integers, as in [0,15] → [0000, 1111], problem with Hamming cliffs:
  - One mutation can change 6 to 7: 0110 → 0111, BUT
  - Need 4 bit-flips to change 7 to 8: 0111 → 1000
  - That's called a "Hamming cliff"
- May use Gray (or other distance-one) codes to improve properties of operators: for example: 000, 001, 011, 010, 110, 111, 101, 100

Mutation Revisited

On "parameter encoded" representations

- Binary ints
  - Gray codes and bit-flips
  - Or binary ints & 0-mean, Gaussian changes, etc.
- Real-valued domain
  - Can discretize to binary -- typically powers of 2 with lower, upper limits, linear/exp/log scaling
  - End result (classically) is a bit string
- BUT many now work with real-valued GAs, non-bit-flip (0-mean, Gaussian "noise") mutation operators

Defining Objective/Fitness Functions

- Problem-specific, of course
- Many involve using a simulator
- Don't need to know (or even HAVE) derivatives
- May be stochastic
- Need to evaluate thousands of times, so can't be TOO COSTLY
- For real-world, evaluation time is typical bottleneck

Back to the "What" Function?

- In problem-domain form -- "absolute" or "raw" fitness, or *evaluation* or *performance* or *objective* function
- *Relative fitness* (to population), may require *inverting* and/or *offsetting*, *scaling* the objective function, yielding the *fitness* function. *Fitness* should be MAXIMIZED, whereas the *objective* function might need to be MAXIMIZED OR MINIMIZED.

Selection

In a classical, "generational" GA:

- Based on fitness, choose the set of individuals (the *"intermediate"* population) that will soon:
  - survive untouched, or
  - be mutated, replaced, or
  - in pairs, be crossed over and possibly mutated, with offspring replacing parents

  One individual may appear several times in the intermediate population (or the next population)

Scaling of Relative Fitnesses

- Trouble: as evolution progresses, relative fitness differences get smaller (as chromosomes get more similar to each other – population is *converging*). Often helpful to SCALE relative fitnesses to keep about same ratio of best guy/average guy, for example.

Types of Selection

Proportional, using *relative fitness* (examples):

- **“roulette wheel” -- classical Holland -- chunk of wheel ~ *relative* fitness**
- **stochastic uniform sampling -- better sampling -- integer parts GUARANTEED; still proportional**

OR, NOT requiring *relative* fitness, nor *fitness scaling*:

- **tournament selection**
- **rank-based selection (proportional to rank or all above some threshold)**

Explaining Why a GA Works – Intro to GA Theory

- **Just touching the surface with two classical results:**
  - **Schema theorem – how search effort is allocated**
  - **Implicit parallelism – each evaluation provides information on many possible candidate solutions**

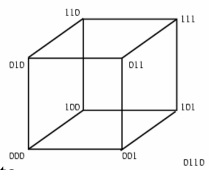
What is a GA DOING? (Schemata and Hyperstuff)

- **Schema -- adds “*”, means “don’t care”**
- **One schema, two schemata**
- **Definition: *ORDER* of schema H = o(H): # of non-*’s**
- **Def.: *Defining Length* of schema, Δ(H): distance *between* first and last non-* in a schema; for example:**

  Δ (1*01*0**) = 5 (= number of positions where 1-pt crossover can disrupt it).**

  (NOTE: diff. xover → diff. relationship to defining length)
- **Strings or chromosomes are order L schemata, where L is length of chromosome (in bits or loci). Chromosomes are *INSTANCES* (or members) of lower-order schemata**

Cube and Hypercube



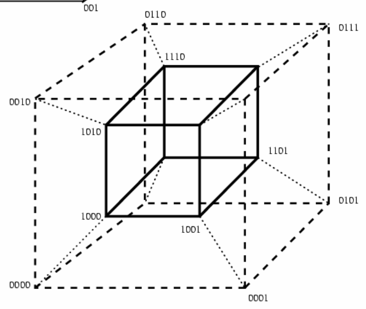
Vertices are order ? schemata

Edges are order ? schemata

Planes are order ? schemata

Cubes (a type of hyperplane) are order ? schemata

8 different order-1 schemata (cubes): 0***, 1***, *0**, *1**, **0*, **1*, ***0, ***1



Hypercubes, Hyperplanes, Etc.

- A *string* is an instance of how many schemata (a member of how many hyperplane partitions)? (not counting the "all *'s," per Holland)
- If L=3, then, for example, 111 is an instance of how many (and which) schemata: 7 schemata
- 2^3-1

GA Sampling of Hyperplanes

So, in general, string of length L is an instance of 2^L-1 schemata

But how many schemata are there in the whole search space?

(how many choices each locus?)

Since one string instances 2^L-1 schemata, how much does a population tell us about schemata of various orders?

Implicit parallelism: one string's fitness tells us something about relative fitnesses of more than one schema.

Fitness and Schema/ Hyperplane Sampling

- Look at next figure (from Whitley tutorial), for another view of hyperspaces

Fitness and Schema/ Hyperplane Sampling

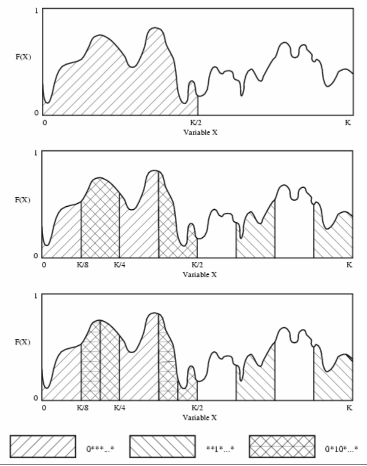
Whitley's illustration of various partitions of fitness hyperspace

Plot fitness versus *one variable* discretized as a K = 4-bit binary number: then get ➔

First graph shades 0***

Second superimposes **1*, so crosshatches are ?

Third superimposes 0*10



How Do Schemata Propagate?

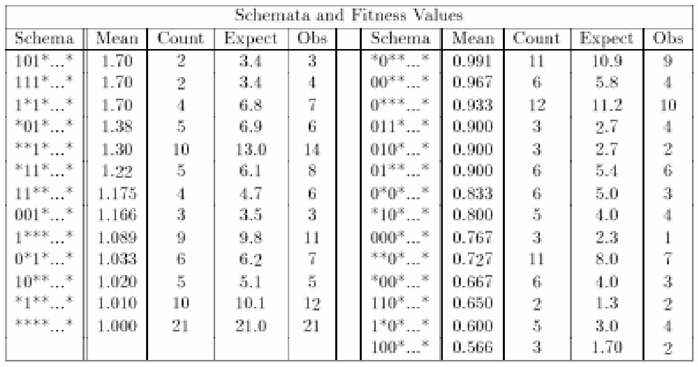
- Via *instances* -- only STRINGS appear in pop – you'll never actually see a schema
- But, in general, want schemata whose instances have higher average fitnesses (even just in the current population in which they're instanced) to get more chance to reproduce. That's how we make the fittest survive!

Proportional Selection Favors "Better" Schemata

- Select the INTERMEDIATE population, the "parents" of the next generation, via fitness-proportional selection
- Let *M(H,t)* be number of instances (samples) of schema H in population at time t. Then fitness-proportional selection yields an expectation of:

$$M(H, t + intermed) = M(H, t)\frac{f(H,t)}{\bar{f}}$$

- In an example, actual number of instances of schemata (next page) in intermediate generation tracked expected number pretty well, in spite of small pop size



Schemata and Fitness Values										
Schema	Mean	Count	Expect	Obs		Schema	Mean	Count	Expect	Obs
101*...*	1.70	2	3.4	3		*0**...*	0.991	11	10.9	9
111*...*	1.70	2	3.4	4		00**...*	0.967	6	5.8	4
1*1*...*	1.70	4	6.8	7		0***...*	0.933	12	11.2	10
01...*	1.38	5	6.9	6		011*...*	0.900	3	2.7	4
**1*...*	1.30	10	13.0	14		010*...*	0.900	3	2.7	2
11...*	1.22	5	6.1	8		01**...*	0.900	6	5.4	6
11**...*	1.175	4	4.7	6		0*0*...*	0.833	6	5.0	3
001*...*	1.166	3	3.5	3		*10*...*	0.800	5	4.0	4
1***...*	1.089	9	9.8	11		000*...*	0.767	3	2.3	1
0*1*...*	1.033	6	6.2	7		**0*...*	0.727	11	8.0	7
10**...*	1.020	5	5.1	5		*00*...*	0.667	6	4.0	3
*1**...*	1.010	10	10.1	12		110*...*	0.650	2	1.3	2
****...*	1.000	21	21.0	21		1*0*...*	0.600	5	3.0	4
						100*...*	0.566	3	1.70	2

Results of example run (Whitley) showing that observed numbers of instances of schemata track expected numbers pretty well

Now, What Does CROSSOVER Do to Schemata

- One-point Crossover Examples (blackboard) 11******** and 1********1
- Two-point Crossover Examples (blackboard) (rings)
- Closer together loci are, less likely to be disrupted by crossover. A "compact representation" tends to keep alleles together under a given form of crossover (minimizes probability of disruption).

Linkage and Defining Length

- Linkage -- "coadapted alleles" (generalization of a *compact representation* with respect to schemata)
- Example, convincing you that probability of disruption by 1-point crossover of schema H of length Δ(H) is Δ(H)/(L-1):

    1****01**1

The Fundamental Theorem of Genetic Algorithms -- "The" Schema Theorem

Holland published in ANAS in 1975, had taught it much earlier (by 1968, for example, when I started Ph.D. at UM)

It provides *lower bound* on change in sampling rate of a single schema from generation t to t+1. We'll consider it in several steps, starting from the change caused by selection alone:

$$M(H, t+intermed) = M(H,t)\frac{f(H,t)}{\bar{f}}$$

Schema Theorem Derivation (cont.)

Now we want to add effect of crossover:

A fraction p_c of pop undergoes crossover, so:

$$M(H,t+1) = (1-p_c)M(H,t)\frac{f(H,t)}{\bar{f}} + p_c[M(H,t)\frac{f(H,t)}{\bar{f}}(1-losses)+gains]$$

Conservative assumption: crossover within the defining length of H is always disruptive to H, and will ignore gains (we're after a LOWER bound -- won't be as tight, but simpler). Then:

$$M(H,t+1) \geq (1-p_c)M(H,t)\frac{f(H,t)}{\bar{f}} + p_c[M(H,t)\frac{f(H,t)}{\bar{f}}(1-disruptions)]$$

Schema Theorem Derivation (cont.)

Whitley adds a *non*-disruption case that Holland ignored:

If cross instance of H with *another*, anywhere, get no disruption. Chance of doing that, drawing second parent at random, is P(H,t) = M(H,t)/popsize: so prob. of disruption by x-over is:

$$\frac{\Delta(H)}{L-1}(1-P(H,t))$$

Then can simplify the inequality, dividing by popsize and rearranging re p_c:

$$P(H,t+1) \geq P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}(1-P(H,t))]$$

So far, we have ignored mutation and assumed second parent is chosen at random. But it's interesting, already.

Schema Theorem Derivation (cont.)

Now, we'll choose the second parent based on fitness, too:

$$P(H,t+1) \ge P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}(1-P(H,t)\frac{f(H,t)}{\bar{f}})]$$

Now, add effect of mutation. What is probability that a mutation affects schema H? (Assuming mutation always flips bit or changes allele):

Each fixed bit of schema (o(H) of them) changes with probability p_m, so they ALL stay UNCHANGED with probability:

$$(1-p_m)^{o(H)}$$

Schema Theorem Derivation (cont.)

Now we have a more comprehensive schema theorem:

$$P(H,t+1) \ge P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}(1-P(H,t)\frac{f(H,t)}{\bar{f}})](1-p_m)^{o(H)}$$

People often use Holland's earlier, simpler, but less accurate bound, first approximating the mutation loss factor as (1-*o*(H)p_m), assuming p_m<<1.

Schema Theorem Derivation (cont.)

That yields:

$$P(H,t+1) \ge P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}][1-o(H)p_m]$$

But, since p_m<<1, we can ignore small cross-product terms and get:

$$P(H,t+1) \ge P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}-o(H)p_m]$$

That is what many people recognize as the "classical" form of the schema theorem.

What does it tell us?

Using the Schema Theorem

Even a simple form helps balance initial selection pressure, crossover & mutation rates, etc.:

$$P(H,t+1) \ge P(H,t)\frac{f(H,t)}{\bar{f}}[1-p_c\frac{\Delta(H)}{L-1}-o(H)p_m]$$

Say relative fitness of H is 1.2, p_c = .5, p_m = .05 and L = 20: What happens to H, if H is long? Short? High order? Low order?

Pitfalls: slow progress, random search, premature convergence, etc.

Problem with Schema Theorem – important at beginning of search, but less useful later...

Building Block Hypothesis

Define a *Building block* as: a short, low-order, high-fitness schema

BB Hypothesis: "Short, low-order, and highly fit schemata are sampled, recombined, and resampled to form strings of potentially higher fitness… we construct better and better strings from the best partial solutions of the past samplings."

-- David Goldberg, 1989

(GA's can be good at assembling BB's, but GA's are also useful for many problems for which BB's are not available)

Using the Schema Theorem to Exploit the Building Block Hypothesis

For newly discovered *building blocks* to be nurtured (made available for combination with others), but not allowed to take over population (why?):

- Mutation rate should be:
  (but contrast with SA, ES, $(1+\lambda)$, …)
- Crossover rate should be:
- Selection should be able to:
- Population size should be (oops – what can we say about this?… so far… infinity is large…):

Traditional Ways to Do GA Search…

- Population "large"
- Mutation rate (per locus) ~ 1/L
- Crossover rate moderate (<0.3) or high (per DeJong, .7, or up to 1.0)
- Selection scaled (or rank/tournament, etc.) such that Schema Theorem allows new BB's to grow in number, but not lead to premature convergence

Schema Theorem and Representation/Crossover Types

If we use a different type of representation or different crossover operator:

- Must formulate a different schema theorem, using same ideas about disruption of some form of "schemata"

Uniform Crossover & Linkage

- 2-pt crossover is superior to 1-point
- *Uniform* crossover chooses allele for each locus at random from either parent
- Uniform crossover is thus more disruptive than 1-pt or 2-pt crossover
- BUT uniform is unbiased relative to linkage
- If all you need is small populations and a “rapid scramble” to find good solutions, uniform xover sometimes works better – but is this what you need a GA for? Hmmmm…
- Otherwise, try to lay out chromosome for good linkage, and use 2-pt crossover (or Booker’s 1987 *reduced surrogate crossover*, see Whitley tutorial)

The N^3 Argument (Implicit or Intrinsic Parallelism)

Assertion: A GA with pop size N can usefully process on the order of N^3 hyperplanes (schemata) in a generation.

(WOW! If N=100, N^3 = 1 million)

To elaborate, assume:

- Random population of size N.
- Need ϕ instances of a schema to claim we are “processing” it in a statistically significant way in one generation.

The N^3 Argument (cont.)

Example: to have 8 samples (on average) of 2nd order schemata in a pop., (there are 4 distinct (CONFLICTING) schemata in each 2-position pair – for example, *0*0**, *0*1**, *1*0**, *1*1**), we’d need 4 bit patterns x 8 instances = 32 popsize.

In general, the highest ORDER of schema, θ, that is “processed” is log (N/ϕ); in our case, log(32/8) = log(4) = 2. (log means $\log_2$)

The N^3 Argument (cont.)

Instead of general case, Fitzpatrick & Grefenstette argued:

- Assume $L \geq 64$ and $2^6 \leq N \leq 2^{20}$
- Pick ϕ=8, which implies $3 \leq \theta \leq 17$
- By inspection (plug in N’s, get θ’s, etc.), the number of schemata processed is greater than N^3. So, as long as our population size is REASONABLE (64 to a million) and L is large enough (problem hard enough), the argument holds.
- But this deals with the initial population, and it does not necessarily hold for the latter stages of evolution. Still, it may help to explain why GA’s can work so well…

Exponentially Increasing Sampling and the K-Armed Bandit Problem

Question: How much sampling should above-average schemata get?

Holland showed, subject to some conditions, using analysis of problem of allocating choices to maximize reward returned from slot machines ("K-Armed Bandit Problem") that:

- **Should allocate an exponentially increasing fraction of trials to above-average schemata**
- **The schema theorem says that, with careful choice of population size, fitness measure, crossover and mutation rates, a GA can do that:**
- **(Schema Theorem says M(H,t+1) >= k M(H,t))**

  That is, H's instances in population grow exponentially, as long as small relative to pop size and k>1 (H is a "building block").

Want More GA Theory?

Vose and Liepins ('91) produced best-known model, looking at a GA as a Markov chain – the fraction of population occupying each possible genome at time *t* is the state of the system. It's "correct", but difficult to apply for practical guidance.

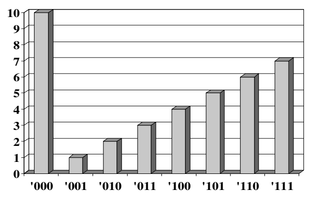
Shapiro and others have developed a model based on principles of statistical mechanics

Lots of others work on aspects of GA theory

Attend other GECCO tutorials or the FOGA Workshop for more theory!

What are Common Problems when Using GAs in Practice?

- **Hitchhiking: BB1.BB2.junk.BB3.BB4: junk adjacent to building blocks tends to get "fixed" – can be a problem**
- **Deception: a 3-bit deceptive function** →
- **Epistasis: nonlinear effects, more difficult to capture if spread out on chromosome**



In PRACTICE – GAs Do a JOB

- **DOESN'T mean necessarily finding *global optimum***
- **DOES mean *trying* to find *better* approximate answers than other methods do, within the time available!**
- **People use any "dirty tricks" that work:**
  - **Hybridize with local search operations**
  - **Use multiple populations/multiple restarts, etc.**
  - **Use problem-specific representations and operators**
- **The GOALS:**
  - **Minimize # of function evaluations needed**
  - **Balance exploration/exploitation so get best answer can during time available (AVOIDING *premature convergence*)**

Other Forms of GA

Generational vs. "Steady-State"

- "Generation gap": 1.0 means replace ALL by newly generated "children"
- at lower extreme, generate 1 (or 2) offspring per generation (called "steady-state") – no real "generations" – children ready to become parents on next operation

More Forms of GA

Replacement Policy:

1. Offspring replace parents
2. K offspring replace K worst ones
3. Offspring replace random individuals in intermediate population
4. Offspring are "crowded" in
5. "Elitism" – always keep best K

Crowding

Crowding (DeJong) helps form "niches" and reduce premature takeover by fit individuals

For each child:

- Pick K candidates for replacement, at random, from intermediate population
- Calculate pseudo-Hamming distance from child to each
- Replace individual most similar to child

Effect?

Example GA Packages – GENITOR (Whitley)

- Steady-state GA
- Two-point crossover, reduced surrogates
- Child replaces worst-fit individual
- Fitness is assigned according to rank (so no scaling is needed)
- (elitism is automatic)

Example GA Packages – CHC (Eshelman)

- Elitism -- (μ+λ) from ES: generate λ offspring from μ parents, keep best μ of the μ+λ parents and children.
- Uses incest prevention (reduction) – pick mates on basis of their Hamming dissimilarity
- HUX – form of uniform crossover, highly disruptive
- Rejuvenate with “cataclysmic mutation” when population starts converging, which is often (small populations used)
- No mutation

Hybridizing GAs – a Good Idea!

IDEA: combine a GA with local or problem-specific search algorithms

HOW: typically, for some or all individuals, start from GA solution, take one or more steps according to another algorithm, use resulting fitness as fitness of chromosome.

If also change genotype, “Lamarckian;” if don’t, “Baldwinian” (preserves schema processing)

Helpful in many constrained optimization problems to “repair” infeasible solutions to nearby feasible ones

Other Representations/Operators: Permutation/Optimal Ordering

- Chromosome has EXACTLY ONE copy of each int in [0,N-1]
- Must find optimal ordering of those ints
- 1-pt, 2-pt, uniform crossover ALL useless
- Mutations: *swap* 2 loci, *scramble* K adjacent loci, *shuffle* K arbitrary loci, etc.

Crossover Operators for Permutation Problems

What properties do we want:

- 1) Want each child to combine building blocks from both parents in a way that preserves high-order *schemata* in as meaningful a way as possible, and
- 2) Want all solutions generated to be feasible solutions.

Operators for Permutation-Based Representations, Using TSP Problem:
Example: PMX -- Partially Matched Crossover

- 2 sites picked, intervening section specifies "cities" to interchange between parents:
- A = 9 8 4 | 5 6 7 | 1 3 2 10
- B = 8 7 1 | 2 3 10 | 9 5 4 6
- A' = 9 8 4 | 2 3 10 | 1 6 5 7
- B' = 8 10 1 | 5 6 7 | 9 2 4 3
- (i.e., swap 5 with 2, 6 with 3, and 7 with 10 in both children.)
- Thus, some ordering information from each parent is preserved, and no infeasible solutions are generated
- Only one of many specialized operators developed

Other Approaches for Combinatorial Problems

Choose a less direct representation that allows using traditional operators:

- Assign an arbitrary integer to each position on chromosome
- Order phenotype by sorting the integers
- Then ordinary crossover, mutation work fine, produce legal genotypes

Parallel GAs
(Independent of Parallel Hardware)

Three primary models: coarse-grain (island), fine-grain (cellular), and micro-grain (trivial)

Trivial (not really a *parallel GA* – just a parallel *implementation* of a single-population GA): pass out individuals to separate processors for evaluation (or run lots of local tournaments, no master) – still acts like one large population

Coarse-Grain (Island) Parallel GA

N "independent" subpopulations, acting as if running in parallel (*timeshared* or *actually* on multiple processors)

Occasionally, migrants go from one to another, in pre-specified patterns

Strong capability for avoiding premature convergence while exploiting good individuals, if migration rates/patterns well chosen

Fine-Grain Parallel GAs

- Individuals distributed on cells in a tessellation, one or few per cell (often, toroidal checkerboard)
- Mating typically among near neighbors, in some defined neighborhood
- Offspring typically placed near parents
- Can help to maintain spatial "niches," thereby delaying premature convergence
- Interesting to view as a cellular automaton

Refined Island Models – Heterogeneous/ Hierarchical GAs

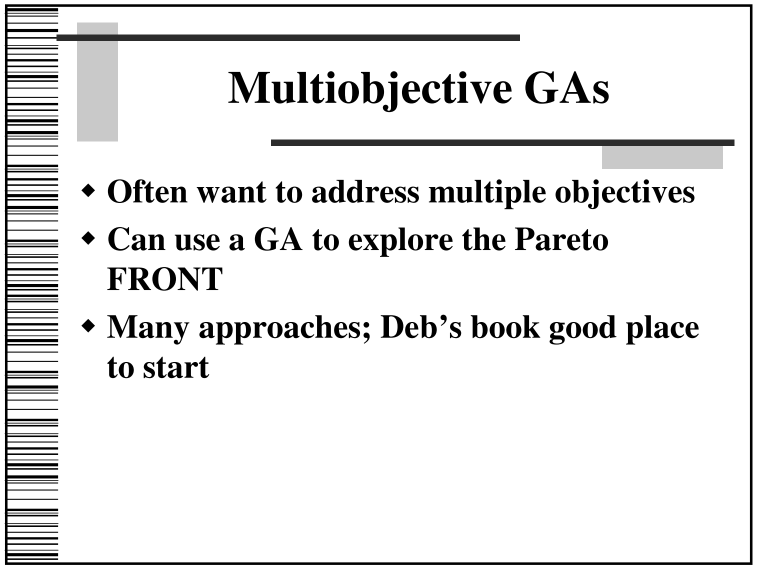
- For many problems, useful to use different representations/levels of refinement/types of models, allow them to exchange "nuggets"
- GALOPPS was first package to support this
- Injection Island architecture arose from this, now used in HEEDS, etc.
- Hierarchical Fair Competition is newest development (Jianjun Hu), breaking populations by fitness bands

Multi-Level GAs

- Island GA populations are on lower level, their parameters/operators/ neighborhoods on chromosome of a single higher-level population that controls evolution of subpopulations (for example, DAGA2, 1995)
- Excellent performance – reproducible trajectories through operator space, for example

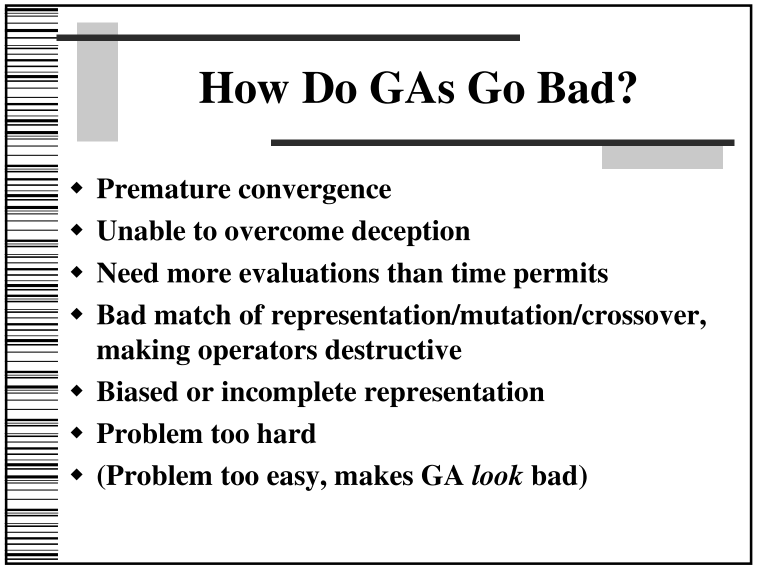
Examples of Population-to-Population Differences in a Heterogeneous GA

- Different GA parameters (pop size, crossover type/rate, mutation type/rate, etc.)
  - 2-level or without a master pop
- Examples of Representation Differences:
  - Hierarchy – one-way migration from least refined representation to most refined
  - Different models in different subpopulations
  - Different objectives/constraints in different subpops (sometimes used in Evolutionary Multiobjective Optimization ("EMOO"))



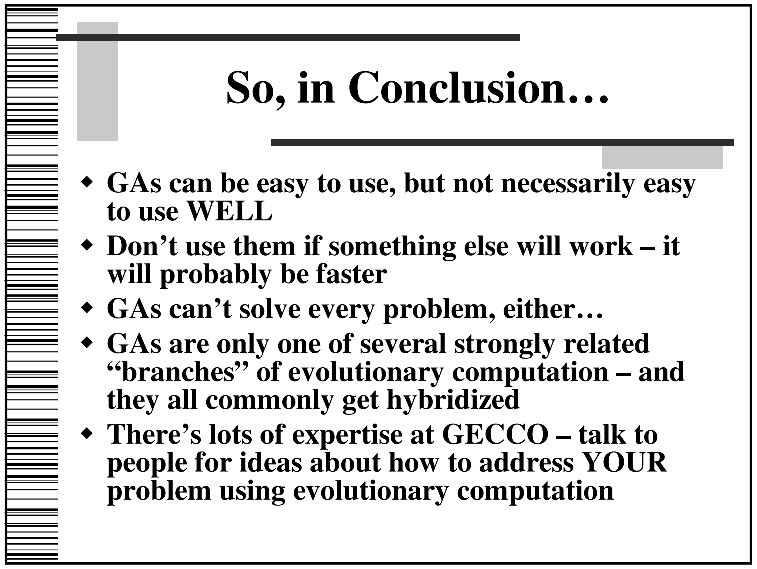
Multiobjective GAs

- Often want to address multiple objectives
- Can use a GA to explore the Pareto FRONT
- Many approaches; Deb's book good place to start



How Do GAs Go Bad?

- Premature convergence
- Unable to overcome deception
- Need more evaluations than time permits
- Bad match of representation/mutation/crossover, making operators destructive
- Biased or incomplete representation
- Problem too hard
- (Problem too easy, makes GA *look* bad)



So, in Conclusion…

- GAs can be easy to use, but not necessarily easy to use WELL
- Don't use them if something else will work – it will probably be faster
- GAs can't solve every problem, either…
- GAs are only one of several strongly related "branches" of evolutionary computation – and they all commonly get hybridized
- There's lots of expertise at GECCO – talk to people for ideas about how to address YOUR problem using evolutionary computation