Chapter 3: Evolutionary Computation Concepts and Paradigms

Chapter 3: Outline

- Review of the EC field
- Provide info to use EC tools to solve practical problems
  - Terminology and key concepts
    - Paradigms
    - Review history
- Discuss five main areas of EC

Evolutionary Computation History

- We will examine five areas of evolutionary computation:
  - Genetic algorithms*
  - Evolutionary programming
  - Evolution strategies
  - Genetic programming
  - Particle swarm optimization*

- Most work has been done in genetic algorithms (GAs), so this history (arbitrarily) concentrates there.

- Focus is on people:
  - * A broad sample, not exhaustive
  - * In chronological order for each field

Genetic Algorithms
A. S. Fraser

- Worked in 1950s in Australia
- Biologist using computers to simulate natural genetic systems
- Worked on epistasis (suppression of the effect of a gene)
- Used a 15-bit string as representation; 5 bits per parameter
- Selected "parents" according to "fitness function" values [-1,1]
- Didn't consider applications to artificial systems

John Holland

- Has had more influence on GA field than any other person
- Has been at the Univ. of Michigan since the early 1960s
- Received first Ph.D. in Computer Science in the U. S. (under Arthur Burks)
- Holland "… created the genetic algorithm field …"
- Published and taught in the field of ***adaptive systems*** in 1960s
- Was a pioneer in the use of a *population* of individuals to do search
- Derived *schema theorem* which shows that more fit schema are more likely to reproduce
- Used reproduction, crossover, and mutation as early as 1960s
- GA was originally called a "genetic plan"

J. D. Bagley

- Student of John Holland
- First used term "genetic algorithm" in his 1967 dissertation
- Used GAs in game playing (Hexapawn)

1975

- Holland published ***Adaptation in Natural and Artificial Systems*** (2**nd** Edition in 1992)
- DeJong's dissertation
  - Five test functions for GAs
  - Metrics for convergence and ongoing performance
  - Measured effects of varying population size, crossover probability, mutation rate, and generation gap.

(DeJong went to U. Pittsburgh, then to Naval Research Lab.)

John Grefenstette

- Student of DeJong at Pittsburgh
- Went to Vanderbilt where one of his students was Dave Schaffer, who developed the first multiobjective GA
- Developed GENESIS, the "back-propagation" of GAs
- Instrumental in founding (and editing proceedings of) ICGA
- Now at NRL

Dave Goldberg

- Student of John Holland
- Dissertation on gas pipeline project
- Published landmark book, "Genetic Algorithms in Search, Optimization, and Machine Learning" (1989)
- Now at Univ. of Illinois Urbana-Champaign

Lawrence (Dave) Davis

- Self-taught in GAs
- Used GAs for 2-D bin packing in chip layout at TI
- Edited "Handbook of Genetic Algorithms"
  - Tutorial
  - Case studies
  - Software diskette: OOGA (Lisp) and Genesis ©

Evolutionary Programming
Larry J. Fogel

- Developed evolutionary programming (EP) in 1960s
- EP utilizes survival of fittest (or "survival of more skillful"), but only operation on structures is mutation
- Interests in finite state machines and machine intelligence
- EP "… abstracts evolution as a top-down process of adaptive behavior, rather than a bottom-up process of adaptive genetics …"
- 1960s book was controversial
- EP suffered from lack of funding due to numerics versus symbolics controversy
- David Fogel and others now carrying on the work

Evolution Strategies
I. Rechenberg

- Developed *Evolutionstrategie* (evolution strategies) during 1960s in Germany
- ES uses *recombination* rather than crossover, and Gaussian-based mutation is done
- Worked on engineering optimization of airfoil designs
- Published book in 1973 that is foundation of ES.
- Student was H.-P. Schwefel, who developed computer simulations
- Student of Schwefel is Thomas Baeck, now a leader in the field

Genetic Programming
R. M. Friedberg

- Early (1950s) work related to genetic programming
- Used one program to write and optimize another fixed-length program
- Each program was 64 instructions, each instruction 14 bits
- Results were disappointing:
  - Program was judged to have failed if not successfully terminated in 64 instructions
  - Population of only one
  - Inappropriate fitness functions

John Koza

- Student of John Holland
- Developed genetic programming (GP) in late 1980s
- GP evolves computer programs
  - Population of programs
  - Tree structures
  - Uses LISP symbolic expressions
- Published books and video tapes

Particle Swarm Optimization
A Social Psychology Paradigm Tour

- Latané's dynamic social impact theory
- Axelrod's culture model
- Kennedy's adaptive culture model

Latané's Dynamic Social Impact Theory

- Behaviors of individuals can be explained in terms of the self-organizing properties of their social systems
- Clusters of individuals develop similar beliefs
- Subpopulations diverge from one another (polarization)

Dynamic Social Impact Theory Characteristics

- Consolidation: Opinion diversity is reduced as individuals are exposed to majority arguments
- Clustering: Individuals become more like their neighbors in social space
- Correlation: Attitudes that were originally independent tend to become associated
- Continuing diversity: Clustering prevents minority views form complete consolidation

Dynamic Social Impact Theory: Summary

- Individuals influence one another, and in doing so become more similar
- Patterns of belief held by individuals tend to correlate within regions of a population
- This model is consistent with findings in the fields of social psychology, sociology, economics, and anthropology

Axelrod's Culture Model

- Populations of individuals are pictured as strings of symbols, or "features"
- Probability of interaction between two individuals is a function of their similarity
- Individuals become more similar as a result of interactions
- The observed dynamic is *polarization*, homogeneous subpopulations that differ from one another

Kennedy's Adaptive Culture Model

- No effect of similarity on probability of interaction
- The effect of similarity is negative in that it is *dis*similarity that creates boundaries between cultural regions
- Interaction occurs if *fitnesses* are different

Culture and Cognition Summary

- Individuals searching for solutions learn from the experiences of others (individuals learn from their neighbors)

- An observer of the population perceives phenomena of which the individuals are the parts (individuals that interact frequently become similar)

- Culture affects the performance of individuals that comprise it (individuals gain benefit by imitating their neighbors)

So, what about intelligence?

- Social behavior increases the ability of an individual to adapt
- There is a relationship between adaptability and intelligence
- Intelligence arises from interactions among individuals

Toward unification

- 1994 IEEE World Congress on Computational Intelligence (WCCI)
  First ICEC chaired by Zbigniew Michalewicz

- 1998, 2002 and 2006 IEEE WCCIs
  Workers in different groups now working together

Evolutionary Computation

- Cover evolutionary computation theory and paradigms
- Emphasize use of EC to solve practical problems
- Compare with other techniques - see how EC fits in with other approaches

Definition: Evolutionary Computation

Evolutionary computation consists of machine learning optimization and classification paradigms that are roughly based on evolution mechanisms such as biological genetics and natural selection. The EC field comprises four main areas: genetic algorithms, evolutionary programming, evolution strategies and genetic programming.

EC Paradigms are Unique

EC paradigms differ from traditional search and optimization paradigms in that EC paradigms:

1) Use a population of points in their search,

2) Use direct "fitness" information, instead of function derivatives or other related knowledge, and ,

3) Use probabilistic rather than deterministic transition rules.

Population

- Each member is a point in the hyperspace problem domain, and thus is a potential solution
- A new population is generated each epoch (generation)
- Population typically remains the same size
- Operators such as crossover and mutation significantly enhance parallel search capabilities

Fitness Information

- Auxiliary information such as derivatives used to minimize sum-squared error in neural nets is not used
- The fitness value optimized is directly proportional to the function value being optimized
- If fitness is proportional to profit, for example, then the fitness rises as the profit rises
- Provides environmental influence on individuals

Probabilistic Transition Rules

- Search is *not* random
- Search is directed toward regions that are *likely* to have higher fitness values
- Different EC paradigms make different uses of stochasticity

Encoding of Parameters

- Here Parameters means components of the solution, for example each disease is a bit; we prefer to call user defined inputs to the GA the Parameters
- Often encoded as binary strings
- Any finite alphabet can be used
- Typically, population member string is of fixed length

Generic EC Procedure

1. Initialize the population
2. Calculate the fitness for each individual
3. Perform evolutionary operations (e.g., mate selection, crossover, mutation)
4. Go to step 2 until some condition is met

Generic EC Procedure, Cont'd.

- Initialization often done by randomizing initial population (can use promising values sometimes; be careful here)

- Fitness value is proportional to value of function being optimized (often scaled 0–1)

- Selection for reproduction based on fitness values (some paradigms such as PSO retain all population members)

Applying EC Tools

- Optimization and classification
- Mostly optimization
  - non-differentiable
  - many local optima
  - may not know optimum
  - system may be dynamic, changing with time, or even chaotic
- Optimization versus meliorization (perhaps try other approaches first)

Note: EC is usually the second-best way to solve a problem!

EC Tools are Robust

- Can be used to solve many problems, and many kinds of problems, with minimal adjustments
- Are fast and easy to implement

Genetic Algorithms: Outline

- Overview
- Terminology
- GA example problem
- Review of GA operations
- Schemata and the schema theorem

Genetic Algorithms

- Reflect in a *primitive* way some of the natural processes of evolution
- GAs perform highly efficient search of the problem hyperspace
- GAs work with a *population* of *individuals* (chromosomes)
- Number of elements in each individual equals the number of parameters in the optimization problem
- If *w* is the number of parameters, and we have *b* bits per parameter, then the search space is 2^{wb}

The variables being optimized comprise the *phenotype* space.
The binary strings upon which the operators work comprise the *genotype* space.

General Procedure for Implementing a GA

1. Initialize the population (usually randomized binary strings)
2. Calculate the fitness for each individual in the population
3. Reproduce selected individuals to form new population
4. Perform crossover and mutation (or other operations)
5. Loop to step 2 until some condition is met

Implementing a Simple GA Involves:

- Fitness evaluation
- Copying strings
- Exchanging portions of strings
- Flipping bits in strings

GA Example Problem

Optimize value of $f(x) = \sin\left(\frac{\pi x}{256}\right)$ over $0 \le x \le 255$

where x is restricted to integers.

The maximum value of 1 occurs at x = 128.

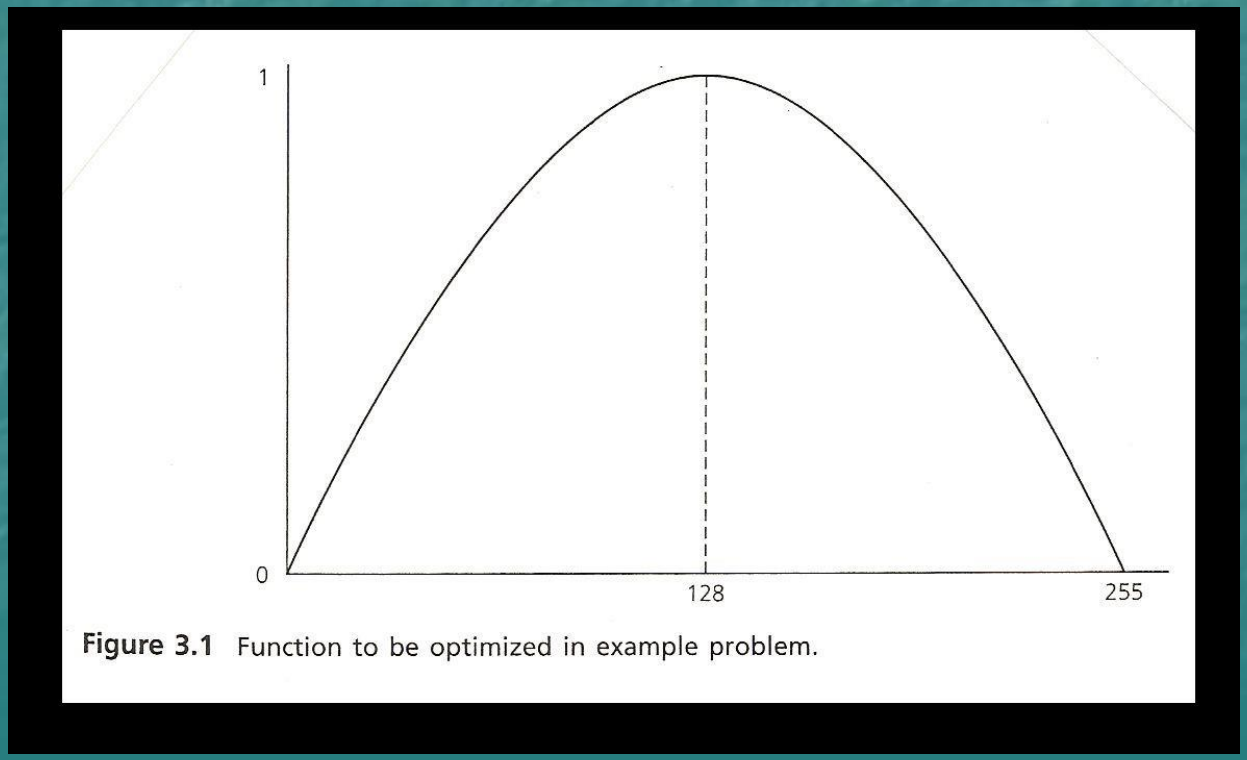


Figure 3.1 Function to be optimized in example problem.

Note: Function space is identical to fitness space.

GA Example Problem

- One variable
- Use binary alphabet
- Therefore, represent each individual as 8-bit binary string
- Set number of population members to (artificially low) 8
- Randomize population
- Calculate fitness for each member of (initialized) population

GA Example Problem

Individuals	x	$f(x)$	f_{norm}	cumulative f_{norm}
1 0 1 1 1 1 0 1	189	0.733	0.144	0.144
1 1 0 1 1 0 0 0	216	0.471	0.093	0.237
0 1 1 0 0 0 1 1	99	0.937	0.184	0.421
1 1 1 0 1 1 0 0	236	0.243	0.048	0.469
1 0 1 0 1 1 1 0	174	0.845	0.166	0.635
0 1 0 0 1 0 1 0	74	0.788	0.155	0.790
0 0 1 0 0 0 1 1	35	0.416	0.082	0.872
0 0 1 1 0 1 0 1	53	0.650	0.128	1.000
		$\Sigma f(x) = 5.083$		

Figure 3.2 Initial population and $f(x)$ values for GA example.

GA Example Problem - Reproduction

Reproduction is used to form new population of *n* individuals
- Select members of current population
- Use stochastic process based on fitnesses

First, compute normalized fitness value for each individual by dividing individual fitness by sum of all fitnesses
- (f_i / 5.083 in the example case)

Generate a random number between 0 and 1 *n* times to form new population (sort of like spinning roulette wheel)

Roulette Wheel Selection

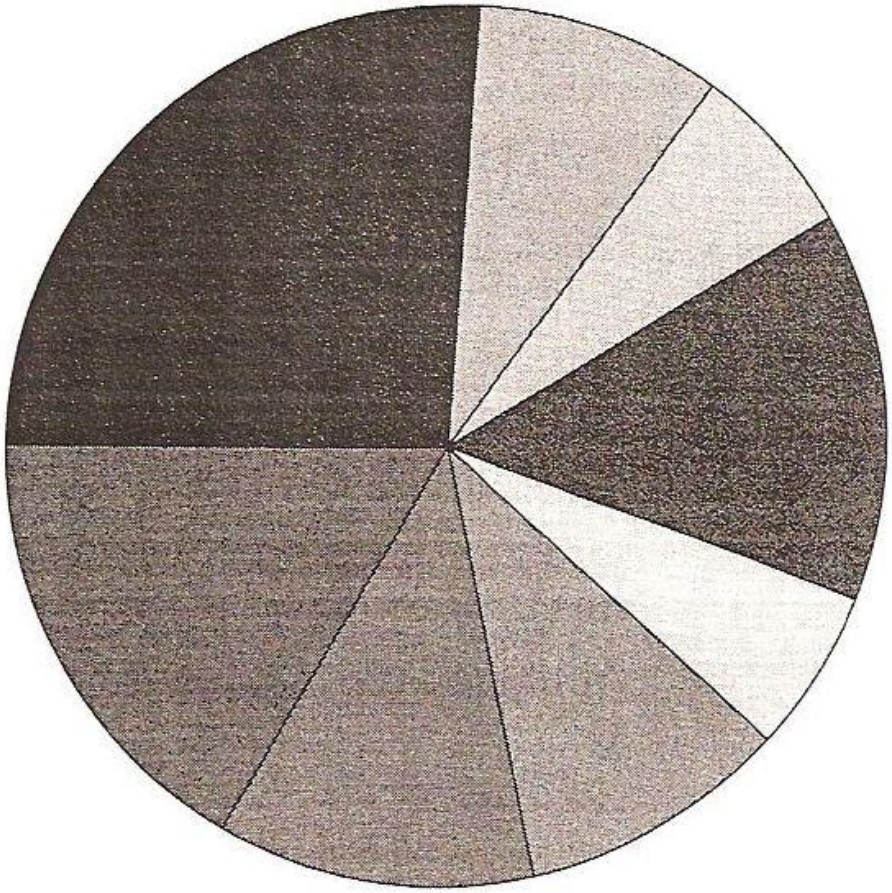
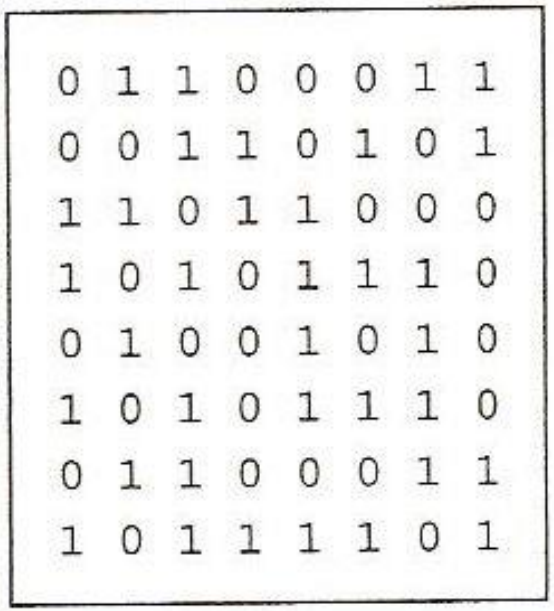


Figure 3.5 Roulette wheel selection, in which the probability of an individual being selected is proportional to its fitness.

Population after Reproduction

The eight random numbers generated, presented in random order, are .293, .971, .160, .469, .664, .568, .371, and .109. This results in initial population member numbers 3, 8, 2, 5, 6, 5, 3, and 1 being chosen to make up the population after reproduction, as shown below.



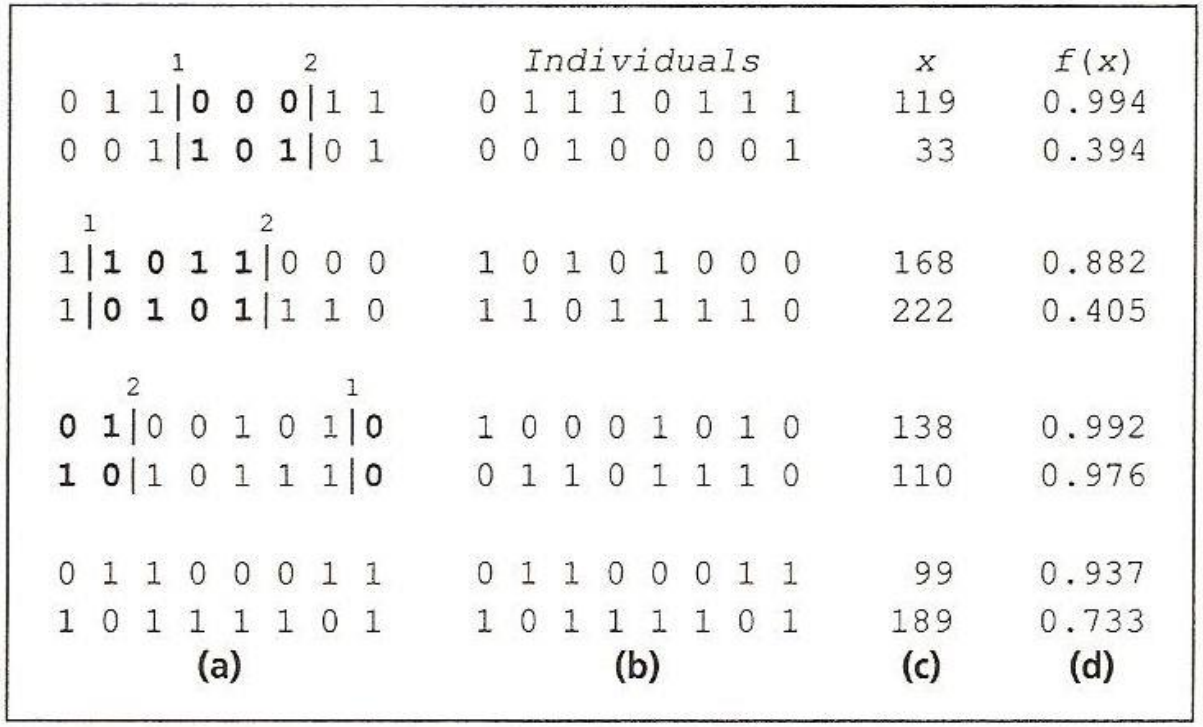
0	1	1	0	0	0	1	1
0	0	1	1	0	1	0	1
1	1	0	1	1	0	0	0
1	0	1	0	1	1	1	0
0	1	0	0	1	0	1	0
1	0	1	0	1	1	1	0
0	1	1	0	0	0	1	1
1	0	1	1	1	1	0	1

Figure 3.3 Population after reproduction.

GA Example Problem: Crossover

- Crossover: the exchange of portions of two individuals (the "parents")
- Probability of crossover usually ~ .65 - .80, .75 used here
- Randomly pair off population, determine whether or not crossover will occur (in example, 3 of 4 crossover)
- Randomly select two crossover points for pairs undergoing crossover
- Exchange portions of strings between 1st and 2nd crossover points, treating individuals as "ring" structures

Crossover: Before and After



(a)	(b)	(c)	(d)
0 1 1 | 0 0 0 | 1 1	0 1 1 1 0 1 1 1	119	0.994
0 0 1 | 1 0 1 | 0 1	0 0 1 0 0 0 0 1	33	0.394
1 | 1 0 1 1 | 0 0 0	1 0 1 0 1 0 0 0	168	0.882
1 | 0 1 0 1 | 1 1 0	1 1 0 1 1 1 1 0	222	0.405
0 1 | 0 0 1 0 1 | 0	1 0 0 0 1 0 1 0	138	0.992
1 0 | 1 0 1 1 1 | 0	0 1 1 0 1 1 1 0	110	0.976
0 1 1 0 0 0 1 1	0 1 1 0 0 0 1 1	99	0.937
1 0 1 1 1 1 0 1	1 0 1 1 1 1 0 1	189	0.733

Figure 3.4 Population before crossover showing crossover points (a); after crossover (b); and values of x (c) and $f(x)$ (d) after crossover.

GA Example Problem: Mutation

- Mutation is final operation in Gas
- Consists of flipping bits with probability of flip ~ .001 - .01
- Example problem, assume no bit flips (64 bits total in population)
- Now have *first generation*, with total fitness of 6.313, and two individuals each with fitness > .99
- Now ready for more generations until stopping condition met:
  - * Number of generations
  - * Population member(s) with > specified fitness
  - * No change in max fitness in *n* generations

Summary of GA Process

1. Select the initial population (usually randomly).
2. Select percent probability of crossover (often .6-.8) and of mutation (often about .001).
3. Calculate the fitness value for each population member.
4. Normalize fitness values and use to determine probabilities for reproduction.
5. Reproduce new generation with the same number of members, using probabilities from 3.
6. Pair off strings to cross over randomly.
7. Select crossing sites (often 2) randomly for each pair.
8. Mutate on a bit-by-bit basis.
9. If more generations, go to step 2.
10. If completed, stop and output results.

Review of GA Operations

- Representation of variables
- Population size
- Population initialization
- Fitness calculation
- Reproduction
- Crossover
- Inversion
- Mutation
- Selecting number of generations

Representation of Variables

Consider example problem,
where 127 is 01111111 and 128 is 10000000

- The smallest fitness change requires change in every bit
- Gray coding has representation such that adjacent values vary by a single bit
- Some software converts dynamic range and resolution into appropriate bit strings
- Different alphabets possible

Gray Coding

Table 3.1 Gray Codes and Binary Codes for Integers 0–15

Integer	*Binary code*	*Gray code*
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

To get Gray code from binary code, the leftmost bit is the same, then

G_i = XOR(B_i, B_{i-1}) for i>=2, where G_i is the i^{th} Gray code bit, Bi is i^{th} binary bit.

Dynamic Range and Resolution

Some software allows you to set the dynamic range and resolution for each variable.

Example: Dynamic range is 2.5 to 4.5, and you want a resolution of 3 decimal places (1 in 1000), you need 11 bits.

Problems can occur: If we have a dynamic range of 5 and a resolution of 3 decimal places, then we need 13 bits, the same as for a dynamic range of 8. We can now get numbers from crossover and mutation out of range, and repairs or penalties must be implemented.

Population Size

- Start with moderate sized population
- 50-500 is often a good starting place for a GA
- Population size tend to increase *linearly* with individual string length (not exponentially)

Population Initialization

- Usually selected stochastically
- Sometimes seeded with a *few* promising individuals
- Do not skew population significantly

Fitness Calculation

- Most GAs scale fitness values
- Example problem illustrates two common problems:
    1. "Bunching" of fitness values near top of scale, thereby lowering fitness differentials
    2. Fitness values often are nearly constant

Fitness Calculation, Cont'd.

- Solve bunching problem by:
  - *Equally spacing fitness values (don't use 0, and keep track of f(x) for evaluation purposes)
  - *Use "scaling" window over last *w* (~5-10) generations that keeps track of minimum fitness value

- Solve "flat" fitness problem (in problems like example problem) by defining a new f(x): $f(x)_{new} = f(x)^{n}_{old}$

- Accomplish minimization with a GA by setting fitness equal to $f_{max} - f(x)$, where f_{max} is set by scaling

Reproduction

- Reproduction often done by normalizing fitnesses and generating *n* random numbers between 0 and 1

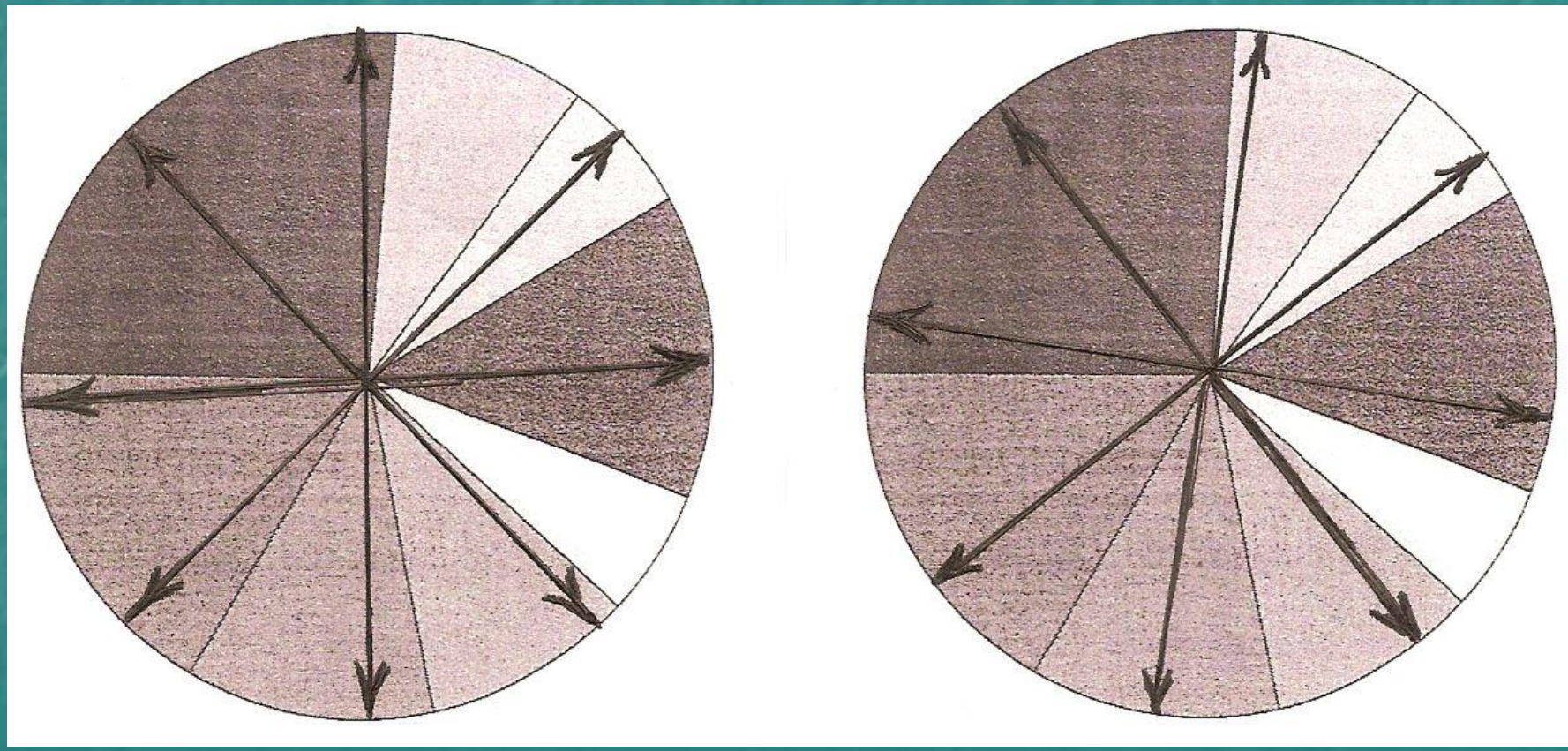
- Baker's method: Use roulette wheel with *n* pointers spaced 1/*n* apart; use normalized fitness; spin wheel *once*.

- "Tournament selection" - Select two individuals at random; the individual with the higher fitness is selected for the next population

- Generation gap approach: Replace *x* percent that have worst fitness values (*x* is defined as the generation gap)

- "Elitist strategy" ensures that individual with highest fitness is copied into next generation (most GAs use this)

Baker's Method: Two Examples



Crossover

- Two-point crossover, with p(c) of 60-80% is common

- Often start with relatively high crossover rate, and reduce it during the run

- The most basic crossover is one-point:

  10110|**010** > 10110100

  01001|**100** > 01001010

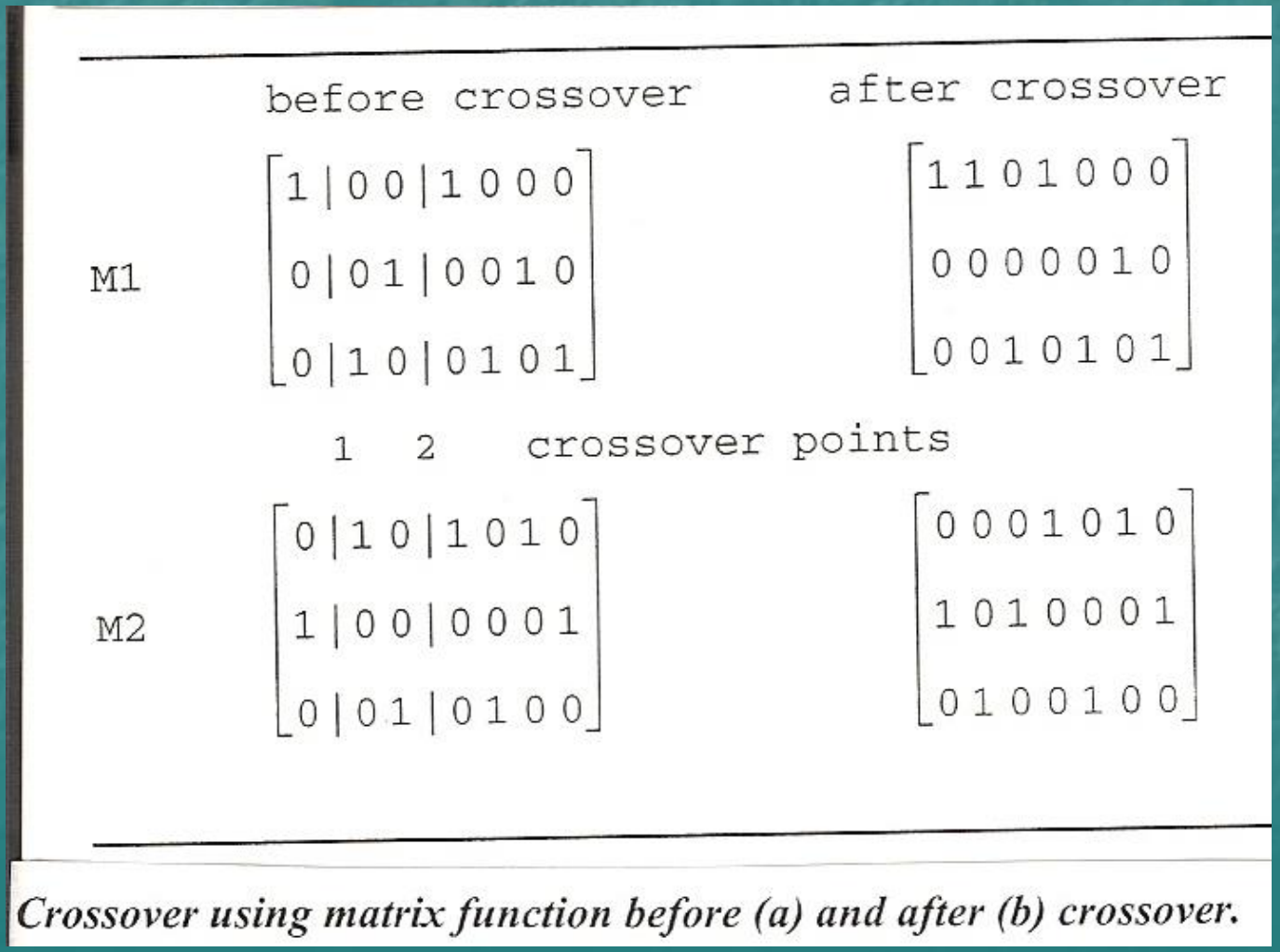
- Uniform crossover (Syswerda)

  * Randomly choose 2 parents & stochastically decide whether to do crossover
  * At each bit position, exchange bits between the 2 strings with p(c) <= 50% [constant for run]

Matrix Crossover

- Example of matrix crossover after Bezdek (1994)
- Working with a population of *c* by *n* matrices, where *c* = no. of classes and *n* is number of patterns
- Each matrix represents classes assigned to all patterns (each column represents a pattern; only one 1 is allowed in each column)
- This case is analogous to working with strings that have an alphabet of *c* characters

Example of Crossover Using Matrix Function



Crossover using matrix function before (a) and after (b) crossover.

c=3

n=7

Inversion

- Single parent produces single child
- Not used much today...it destroys “schemata” (defined later)
- Example: 10|**01101**|0 -> 10101100

Mutation

- Stochastically flipping bits often with p(*m*) ~ .001
- If real-valued parameters used, mutation can assign any value in parameters allowed range
- Probability of mutation usually held constant or increased during run
- Can increase mutation rate when fitness variablility drops below some threshold

Selecting the Number of Generations

- Often a trial and error process
- Optimum value (if known) a function of the problem
- Multiple runs often done; overall best solution selected

Schemata and the Schema Theorem

Schemata are similarity templates for strings (features of strings)

Schema: a subset of strings with identical values at specified string locations

A "don't care" or "wildcard" symbol such as "*" or "#" is used in schema locations where the value doesn't matter

The schema **1**0** has 4 matching strings:
1000, 1010, 1100, 1110

Schemata, cont'd.

For a string of length *l* and alphabet of *a* characters, the *total possible* number of schemata is $(a+1)^l$

For a population of *n* individuals, there are between 2^l and $n2^l$ *unique* schemata

Schemata in highly fit individuals are more likely to be reproduced in the new population

If 2^l then all populatoin members are identical
If $n2^l$ thenno two schemata are the same

Population diversity is proportional to the number of schemata

Schemata, cont'd.

Crossover and mutation are *needed* to guide search into new regions

Effect of crossover dependent on "defining length" of schemata

In case of **1******0 and **10**, the first is more likely to be** disrupted by crossover

Mutation is as likely to disrupt one as the other

Short, highly fit, schemata will appear in increasing numbers in successive generations

The Schema Theorem

The Schema Theorem predicts the number of times a specific schema will appear in the next generation of a GA, given the average fitness of individuals containing the schema, the average fitness of the population, and other parameters.

Fundamental Theorem of Genetic Algorithms: Holland's Schema Theorem

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\overline{f}}\left[1-p_c\frac{\delta(H)}{l-1}-o(H)p_m\right]$$

m is the number of examples of a particular schema H
$o(H)$ is the "order" of the schema (number of fixed positions)
$f(H)$ is the average fitness of strings representing schema H
$\overline{f}$ is the average fitness of the entire population
$\delta(H)$ is the "defining length," the distance between the first and last specific string position: *11*01* has $\delta(H) = 4$ → Number of potential cut points.
l is the total length of the string (population member)
p_c is the probability of crossover
p_m is the probability of mutation

Schema Theorem Consequences

- A genetic algorithm works with all schemata *simultaneously*, an attribute named *intrinsic parallelism* by Holland.
- The most rapidly increasing representation in any population will be of highly fit, short schemata, which will experience growth approaching exponential.
- The Schema Theorem does ***not*** indicate how well a GA will solve a particular problem.

Final Comment on GAs

"…[T]here is something profoundly moving about linking a genetic algorithm to a difficult problem and returning later to find that the algorithm has evolved a solution that is better than the one a human found. With genetic algorithms we are not optimizing; we are creating conditions in which optimization occurs, as it may have occurred in the natural world. One feels a kind of resonance at such times that is uncommon and profound."

Dave Davis (1991)

(This could apply to other EC paradigms as well.)

Evolutionary Programming

Similar to genetic algorithms (GAs)
- Uses population of solutions
- Evolves a solution using survival of the fittest
- Uses mutation (slightly differently than GAs)

Different from GAs
- Concentrates on "top-down" processes of adaptive behavior
- Simulates evolution (phenotypic behavior) rather than genetics (genotypic behavior)
- Does not use crossover

Works in phenotype space of observable behaviors

Generates the same number of "children" as "parents"

Evolutionary Programming Procedure

1. Initialize the population
2. Expose the population to the environment
3. Calculate the fitness for each member
4. Randomly mutate each "parent" population member
5. Evaluate parents and children
6. Select members of new population
7. Go to step 2 until some condition is met

Population Initialization

Component variables are usually real-valued

Dynamic range constraints usually exist and are observed

Random initial values within dynamic ranges are used

Population is often a few dozen to a few hundred

Example Application – Finite State Machine Evolution

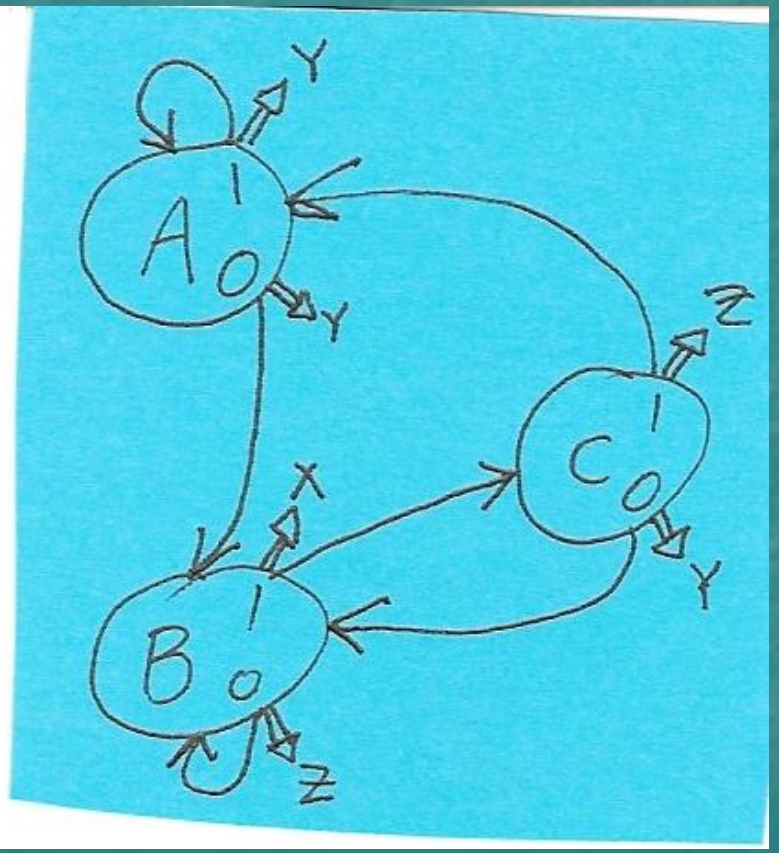
- Illustrates use of EP for prediction
- Analyze a symbol sequence and predict next symbol - or otherwise optimize a fitness function
- Definition of Finite State Machine (Fogel): A transducer that can be stimulated by a finite alphabet of input symbols, can respond in a finite alphabet of output symbols, and possesses some finite number of internal states
- Can have different input and output symbol alphabets
- Initial state must be specified; and specification table must be provided

Three-state Finite State Machine

Table 3.2 Specification Table for a Three-State Finite State Machine

Existing state	A	A	B	B	C	C
Input symbol	1	0	1	0	1	0
Output symbol	Y	Y	X	Z	Z	Y
Next state	A	B	C	B	A	B

Source: Fogel (1991).



Finite State Machine Evolution

Finite state machines (FSMs) are similar to Turing machines, which can solve all mathematical problems of a defined general class

EP uses only mutation; for FSMs, 5 main types of mutation exist:

* Change initial state
* Delete a state
* Add a state
* Change a state transition
* Change an output signal

Note: first two mutations require that >1 state exist

Finite State Machine Evolution

Each parent produces one child; population temporarily doubles

Typically keep best 1/2 of doubled population for new population

Problems can exist with respect to state transition tables:

* States can be added that are not utilized
* Deleted states may make state transitions impossible (can be fatal error)

Finite State Machine Evolution

Example of fixed length structure with maximum of 4 states, 2 input symbols, and 3 output symbols:

Can use 9 bits per state; 36 bits total per population member

bit 1: activity of state (enabled or not)
bits 2 and 3: output symbol for input of 0
bits 4 & 5: next state for input of 0
bits 6 & 7: output symbol for input of 1
bits 8 & 9: next state for input of 1

(Note: must deal with non-existent output symbols and non-existent states.)

Finite State Machine Evolution

- Lengths of individuals can either be variable (Fogel) or fixed

- One possible mutation method (given 5 possible kinds of mutation):
  1. Generate random number between 0 and 1
  2. If it is between 0 and .2, do first kind of mutation, if between .8 and 1.0, do fifth, etc.
  3. Mutation is done across possible values with flat probability
  4. Reassign infeasible state transitions with flat probabilities
  5. Evaluate fitnesses

- Keep best 50% of population

- Number of mutations per individual, and probability of each of the five types of mutation can be varied...even evolved!

Axelrod's Iterated Prisoners' Dilemma

Payoff Function

		Player 1 C	Player 1 D
Player 2	C	3|3	0|5
	D	5|0	1|1

(Format is Player2|Player1)

A 7-State Finite State Machine

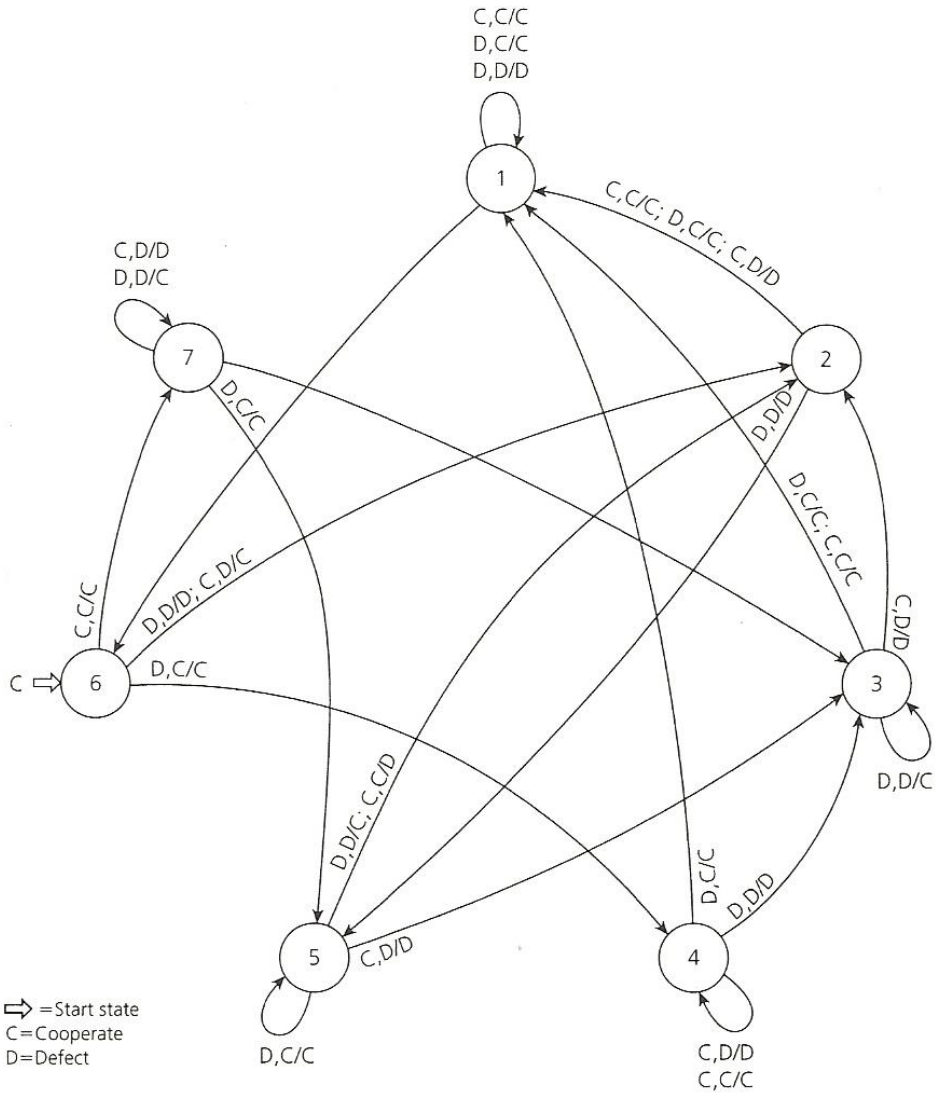


Figure 3.8 A seven-state finite state machine to play prisoner's dilemma. *Source*: Fogel 1995; © IEEE. Used with permission.

Function Optimization

- Mutation is done with Gaussian random function
- Example: optimize $F(x,y) = x^2 + y^2$, i.e., find the minimum at (0,0)
- Procedure:
  - * Set population size (=50 for this problem)
  - * Initialize values of variables over [-5,5]
  - * Calculate fitness values (1/Euclidean_dist_fm_origin)
  - * Mutate each parent to produce one child

EP Mutation Process

The process of mutation is illustrated in equation 3.2:

$$p_{i+k,j} = p_i + N(0, \beta_j \phi_{p_i} + z_j), \forall_j = 1..., n, \tag{3.2}$$

where

$p_{i,j}$ is the j^{th} element of the i^{th} organism

$N(\mu, \sigma^2)$ is a Gaussian random variable with mean μ and variance σ^2

ϕ_{p_i} is the fitness score for p_i

β_j is a constant of proportionality to scale ϕ_{p_i}

z_j represents an offset

For the function used in the example, it has been shown that the optimum rate of convergence is represented by $\sigma = \frac{1.224\sqrt{f(x)}}{n}$, where n is the number of dimensions (Bäck and Schwefel 1993).

Function Optimization - Mutation

* Mutate by adding a Gaussian random variable with zero mean and variance equal to parent's error to each parent vector component

* Next generation can be best 1/2, or can be selected probabilistically in tournament competitions with other individuals

Tournament Selection for EP

Example: Perform 10 "competitions" for each individual using error scores:

1. Randomly select opponent
2. Probability of scoring a point = error of opponent divided by sum of individual and opponent errors
3. Repeat 10 times

Each individual gets a total score; highest 50 percent survive to next generation

Selection is thus a probabilistic function of fitness

Evolution Strategies

- Evolution strategies field is based on the evolution of evolution (evolution optimizing itself)
- Evolution strategies utilize mutation and recombination (similar to crossover)
- Phenotypic behavior stressed, similar to EP
- Recombination uses real values for variables, rather than the binary coding of GAs

Strategy Parameters

- Individuals include *strategy parameters* that are variances and covariances that are evolved
- Each parameter can have a variance associated with it
- With *n* variables, there can be up to *n* variances
- Usually, only one is implemented
- Usually use standard deviation to set the mutation step size
- There are up to *n(n-1)/2* covariances for each individual; covariances are usually not implemented

*Note: Both strategy parameters and individual variables are evolved

Evolution Window

- *Evolution window concept*: Mutation improves fitness only if it occurs within a defined "window"

- If evolution operators stay within the window (std. dev. is optimal), probability of successful mutation is approximately 0.2

- Evolving the window itself can result in "meta-evolution"

ES Mutation

- Variables: Gaussian noise functions with zero mean determine mutation magnitudes for variables.

- Strategic parameters (variances and covariances): use log-normal distributions for standard deviations of strategic parameters

- Mutation rate is inversely proportional to number of variables in individual

- Mutation rate sometimes made proportional to error (distance from optimum; if opt. not known, even limited knowledge helps guide search)

ES Recombination (Crossover)

* Recombination (crossover) manipulates entire variable values using one of two methods:
  * Local method - Form 1 new individual using two randomly-selected parents (usual method used)
  * Global method - Uses entire population as potential sources for each new individual
* Global and local methods can each be implemented in one of two ways (often use local method):
  * Discrete recombination - Select value from one parent or the other (usually used for parameter values)
  * Intermediate recombination - Each parameter value for child is a point between parents' values (used for strategy parameters)

$$x_i^{new} = x_{A,i}^{old} + C\left(x_{B,i}^{old} - x_{A,i}^{old}\right)$$ (Often, C=0.5)

Selection Methods

- Ranking – Only the best are chosen
- Roulette Wheel – Probability of selection is proportional to fitness
- Tournament Selection – Local competitions determine survivors
  - Best of random pair
  - Best of random subgroup
  - Winner of defined number of competitions

(Note: Having *n*-1 tournaments is the same as ranking.)

ES Selection

* ES usually operates with a surplus of descendents
* Ratio of number of children to number of parents is often about 7
* Two main types of selection:

(μ, λ): μ individuals with highest fitness out of λ children selected (parents not eligible)

$(\mu + \lambda)$: μ most fit out of μ *and* λ individuals selected

(μ, λ) version generally reported to give better performance (it is *not* elitist!)

ES Procedure Summary

1. Initialize population
2. Perform recombination using the μ parents to form λ children
3. Mutate all children
4. Evaluate λ or $\mu + \lambda$ population members
5. Select μ fittest for new population
6. If termination criteria *not* met, go to step 2

Genetic Programming

- Genetic programming (GP) evolves hierarchical computer programs
- Programs are represented as tree structure populations (most work originally done in Lisp)
- Structures may vary in size and complexity
- The goal is to obtain the desired output for a given set of inputs, within the search domain

Differences between GPs and Generic GAs

- Population members of GPs are executable structures (generally, computer programs) rather than strings of bits and/or variables
- The fitness of an individual population member in a GP is measured by executing it. (Generic GAs measure of fitness depends on the problem being solved.)
- Hierarchical structure of GPs

Representation and Preparation

- Each program is represented as a parse tree
  - Functions are at internal tree points
  - Variables and constants are at external (leaf) points

- Preliminary steps are to specify:
  - Terminal set (variables and constants)
  - Function set
  - Fitness measure
  - System control parameters
  - Termination conditions

Terminal Set & Function Set

- Consists of system state variables and constants relevant to problem

- Example: cart centering problem uses position *x*, velocity *v*, and a constant such as 1

- Functions selected are limited only by programming language implementation
  - Math functions (sin, exp, ...)
  - Arithmetic functions
  - Conditional operators (if...then, etc.)
  - Boolean operators (AND, NOT, ...)

- Select a minimal yet sufficient set

- Arity of each must be honored

Closure and Sufficiency

* Closure - Each function must successfully operate on all functions and terminals (any value of any data type)

* Closure requirement gives rise to “protected functions”: functions redefined so as to return acceptable values (ex: divide by 0)

Closure and Sufficiency

Sufficiency - Set of functions and set of terminals must be sufficient to allow a solution to be evolved

* Some knowledge of problem required

* Some problems are easier than others
  - Boolean easy (AND, OR, NOT)
  - Control system equations may be difficult, since having more than minimally sufficient function set usually degrades performance significantly (but sometimes improves it!)

* Koza generally limits the number of terminals

Fitness Measures and Control Parameters

- Variety of possible fitness measures
  - * Inversely proportional to error
  - * Game score

- Control parameters
  - * Population size
  - * Max. no. of generations
  - * Reproduction probability
  - * Crossover probability
  - * Max. depth allowed (initial and final)

Termination of GP

- Usually set by maximum number of generations specified
- Best program created thus far usually selected as winner

GP Process

1. Initialize population of computer programs
2. Determine fitness of each program
3. Reproduce according to fitness values and reproduction probability
4. Perform crossover of subexpressions
5. Go to step 2 unless termination conditions are met

Initialization

- Initialize with randomly-generated programs using defined functions and terminals (build rooted tree)
- No restrictions apply except maximum depth (which is significant!)
- Population size varies, but 500 is common with Koza *et al*.

Randomly Generated Programs

Start with "root" function with number of branches = arity

Examples of generating population: 2 approaches:

Grow method - select randomly from functions *and* terminals, except at maximum depth, where terminal must be selected (Ratio of #functns:#terms is important)

Full method - Each limb extends for full allowed depth. Only functions are selected until max depth is reached, then only terminals are selected.

Ramped Half-and-Half Approach

* Evenly distribute depth parameter from 2 to max. depth. So if max. depth is 11, 10 percent of population will have each depth (2, 3, ..., 11)

* Within each depth, 1/2 of the programs are built using *grow* approach, 1/2 using *full* approach

GP Fitness

- Fitness must be calculated over a number of cases for each program. Average value often taken to be fitness. (Use perhaps 100 cases).

- Same cases usually used throughout the run.

- Four ways to calculate fitness defined by Koza:
  - Raw fitness
  - Standardized fitness
  - Adjusted fitness
  - Normalized fitness

GP Raw Fitness

- Can be calculated various ways
- Direct - score, profit, miles traveled (max. or min.)
- A frequently-used method is to sum over all cases of the absolute value of the error (sum-squared, Hamming, etc.)

GP Standardized Fitness

- Lower values desirable (min. usually set = 0)
- Sometimes same as "raw" fitness (error minimized)
- If more is better, subtract raw fitness from max. fitness

Adjusted Fitness

- Adjusted fitness Fa = 1/(1-Fs), where Fs is standardized fitness
- Values range from 0 to 1, where 1 is optimum
- Koza uses adjusted fitness for many problems
- Small changes in standardized fitnesses near optimum have greater impact than those far away
- Provides a sort of scaling feature

GP Normalized Fitness

- Calculated using adjusted fitnesses
- Calculated in similar manner to GA normalized fitnesses

GP Reproduction and Crossover

- Often carried out in "parallel"
- Sum of probabilities for reproduction and crossover = 100
  - 10 % for reproduction typical
  - 90 % for crossover typical
- Both done after fitnesses are calculated
- If reproduction selected, roulette wheel selection used
- Over-selection of highly-fit individuals in large (<1000) populations is sometimes used

Crossover

- If crossover selected, pick two parents using roulette wheel selection
- Pick one point randomly in each parent for crossover - point can be *anywhere* in program
- Exchange crossover point roots and everything below them
- Entire programs (root on down) can be exchanged
- When resulting program after crossover would result exceed maximum depth, copy unaltered program into new population

GP Final Thoughts

* Preprocessing as used in other EC tools not as important in GP
But

* Selection of functions and terminals is similar in concept to preprocessing

* For assistance with formulating problems, see Chapter 26 in Koza (1992)

Particle Swarm Optimization

- Introduction and basics
- Comparison of GAs and PSO
- Advanced PSO concepts

Particle Swarm Optimization Overview

- Developed with Dr. Jim Kennedy, Bureau of Labor Statistics, Washington, DC
- A concept for optimizing nonlinear functions using particle swarm methodology
- Has roots in artificial life and evolutionary computation
- Simple in concept
- Easy to implement
- Computationally efficient
- Effective on a wide variety of problems

Evolution of concept and paradigms

- Discovered through simplified social model simulation
- Related to bird flocking, fish schooling and swarming theory
- Related to evolutionary computation: genetic algorithms and evolution strategies
- Kennedy developed the "cornfield vector" for birds seeking food
- Bird flock became a swarm
- Expanded to multidimensional search
- Incorporated acceleration by distance
- Paradigm simplified

Particle Swarm Optimization Process

1. Initialize population in hyperspace
2. Evaluate fitness of individual particles
3. Modify velocities based on previous best and global (or neighborhood) best
4. Terminate on some condition
5. Go to step 2

PSO Velocity Update Equations

Original global version:

$$v_{id} = v_{id} + c_1 rand()(p_{id} - x_{id}) + c_2 Rand()(p_{gd} - x_{id})$$

$$x_{id} = x_{id} + v_{id}$$

Where ***d*** **is the dimension,** c_1 **and** c_2 **are positive constants,** ***rand*** **and** ***Rand*** **are random functions, and** ***w*** **is the inertia weight.**

For the neighborhood version, change p_{gd} **to** p_{ld} **.**

Basic Principles of Swarm Intelligence

- Proximity principle: the population should be able to carry out simple space and time computations

- Quality principle: the population should be able to respond to quality factors in the environment

- Diverse response principle: the populations should not commit its activities along excessively narrow channels

- Stability principle: the population should not change its mode of behavior every time the environment changes

- Adaptability principle: the population must be able to change behavior mode when it's worth the computational price

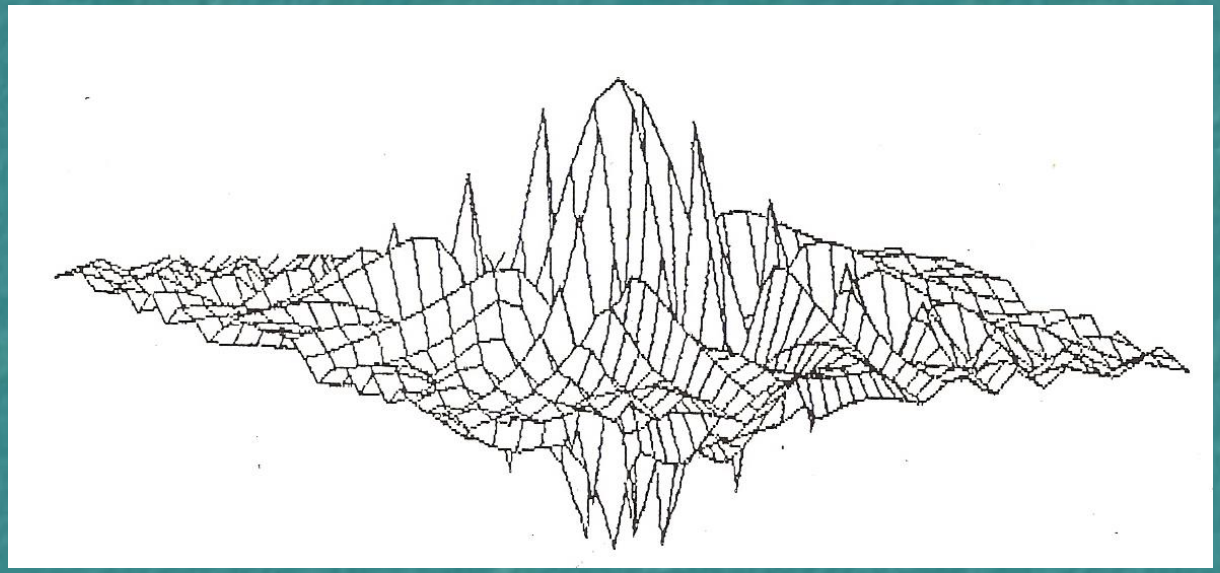
Adherence to Swarm Intelligence Principles

- Proximity: *n*-dimensional space calculations carried out over series of time steps
- Quality: population responds to quality factors *pbest* and *gbest* (or *lbest*)
- Diverse response: responses allocated between *pbest* and *gbest* (or *lbest*)
- Stability: population changes state only when *gbest* (or *lbest*) changes
- Adaptability: population *does* change state when *gbest* (or *lbest*) changes

Applications and Benchmark Tests

- Function optimization
  - De Jong's test set
  - Schaffer's *f6* function
- Neural network training
  - XOR
  - Fisher's iris data
  - EEG data
  - 2500-pattern SOC test set
- Benchmark tests
  - Compare *gbest* and *lbest*
  - Vary neighborhood in *lbest*

Schaffer's F6 Function



VMAX

- An important parameter in PSO; sometimes the only one adjusted
- Clamps particles' velocities on each dimension
- Determines "fineness" with which regions are searched
  - If too high, can fly past optimal solutions
  - If too low, can get stuck in local minima

PSO Initial Version

- Fundamental assumptions seem to be upheld: Social sharing of information among conspeciates provides and evolutionary advantage.

- PSO has a memory: it is thus related to the "elitist" version of GAs

Roles in the PSO Equation and the Inertia Weight

- Without changes, particles fly at constant speed to boundary
- Without ***v*** term, the best particle would have 0 velocity, and other particles would statistically contract to optimum
- Originally, we set **c1** and **c2** to 2.0, then tried values between 1 and 2.
- A parameter that balances global and local search was introduced: an inertia weight ***w***.

PSO Equations with Inertia Weight

$$v_{id} = wv_{id} + c_1 rand()(p_{id} - x_{id}) + c_2 Rand()(p_{gd} - x_{id})$$

$$x_{id} = x_{id} + v_{id}$$

where ***w*** is the inertia weight.

Inertia Weights and Constriction Factors in Particle Swarm Optimization: Introduction

- We compare the performance of particle swarm optimization using an inertia weight versus using a constriction factor
- Several benchmark functions are used

PSO Update Equations Using Constriction Factor Method

$$v_{id} = K * [v_{id} + c_1 * rand( ) * (p_{id} - x_{id}) + c_2 * Rand( ) * (p_{gd} - x_{id})]$$

$$K = \frac{2}{\left| 2 - \phi - \sqrt{\phi^2 - 4\phi} \right|}$$

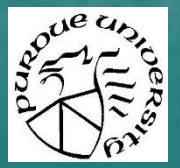
where $\phi = c_1 + c_2, \ \phi > 4$

Phi was set to 4.1, so that K = 0.729; multiplier is thus 1.49445.

Benchmark Functions

- Parabolic function
  - 30 dimensions, *Xmax* = 100, error < 0.01
- Rosenbrock function
  - 30 dimensions, *Xmax* = 30, error < 100
- Rastrigrin function
  - 30 dimensions, *Xmax* = 5.12, error < 100
- Griewank function
  - 30 dimensions, *Xmax* = 600, error < 0.05
- Schaffer's f6 function
  - 2 dimensions, *Xmax* = 100, error < 0.00001

Each version of each function was run 20 times for each benchmark function.



IUPUI

Parabolic Function

	Average No. of Iterations	Range (No. of Iter.)
Inertia Weight	1538	130
Constriction F. Vmax=100K	552	96
Constriction F. Vmax=Xmax	530	78

Rosenbrock Function

	Average No. of Iterations	Range (No. of Iter.)
Inertia Weight	3517	1640
Constriction F. Vmax=100K	1424	4318
Constriction F. Vmax=Xmax	669	992

Rastrigrin Function

	Average No. of Iterations	Range (No. of Iter.)
Inertia Weight	1321	961
Constriction F. Vmax=100K	943*	6823
Constriction F. Vmax=Xmax	213	175

* Note: Target error not achieved for one run

Griewank Function

	Average No. of Iterations	Range (No. of Iter.)
Inertia Weight	2901	1335
Constriction F. Vmax=100K	437*	279
Constriction F. Vmax=Xmax	313	84

* Note: Target error not achieved for three runs; error relaxed to 0.1

Schaffer f6 Function

	Average No. of Iterations	Range (No. of Iter.)
Inertia Weight	512	409
Constriction F. Vmax=100K	431	794
Constriction F. Vmax=Xmax	532*	1952

* Note: Avg. = 453, range=803 with one outlier removed

Conclusions

- Best approach is to use constriction factor, limiting the maximum velocity ***Vmax*** to the dynamic range ***Xmax***
- Performance on benchmark functions is superior to any other results known to the authors
- Method has been incorporated into several applications

IUPUI

Fuzzy Adaptive Inertia Weight: A Preview

- Uses a set of fuzzy rules to adapt an inertia weight
- First application to Rosenbrock function with asymmetric initialization
  - Nine rules
  - Two inputs: global best fitness and current inertia weight
  - One output: change in inertia weight
- Can significantly improve system performance
- We'll look at fuzzy systems in Chapter 7

Current Method

- Best approach is to use constriction factor, limiting the maximum velocity ***Vmax*** to the dynamic range ***Xmax***

- Performance on benchmark functions is superior to any other results known to the authors

- Method has been incorporated into several applications

Tracking and Optimizing Dynamic Systems with Particle Swarms

IUPUI

Outline

- Types of dynamic systems
- Practical application requirements
- Previous work
- Experimental design
- Results
- Conclusions and future effort

IUPUI

Constriction Factor Version

$$v_{id} = K * [v_{id} + c_1 * rand( ) * (p_{id} - x_{id}) + c_2 * Rand( ) * (p_{gd} - x_{id})]$$

$$K = \frac{2}{\left| 2 - \phi - \sqrt{\phi^2 - 4\phi} \right|}$$

where $\phi = c_1 + c_2, \ \phi > 4$

(ϕ was set to 4.1, so K = .729)

IUPUI

Dynamic System Types

- Location of optimum value can change
- Optimum value can vary
- Number of optima can change
- Combinations of the above can occur

In this work, we varied the location of the optimum.

IUPUI

Practical Application Requirements

- Few practical problems are static; most are dynamic
- Most time is spent re-optimizing (re-scheduling, etc.)
- Many systems involve machines and people
  - These systems have inertia
  - 10-100 seconds often available for re-optimization
- Eberhart's Law of Sufficiency applies: If the solution is good enough, fast enough, and cheap enough, then it is sufficient

IUPUI

Previous Work

- Testing Parabolic Function

$$error = \sum_{i=1}^{N} (x_i - offset)^2$$

  - Offset = offset + severity
  - Severity 0.01, .1, .5
  - 2000 evaluations per change
  - 3 dimensions, dynamic range –50 to +50

IUPUI

Previous Work: References

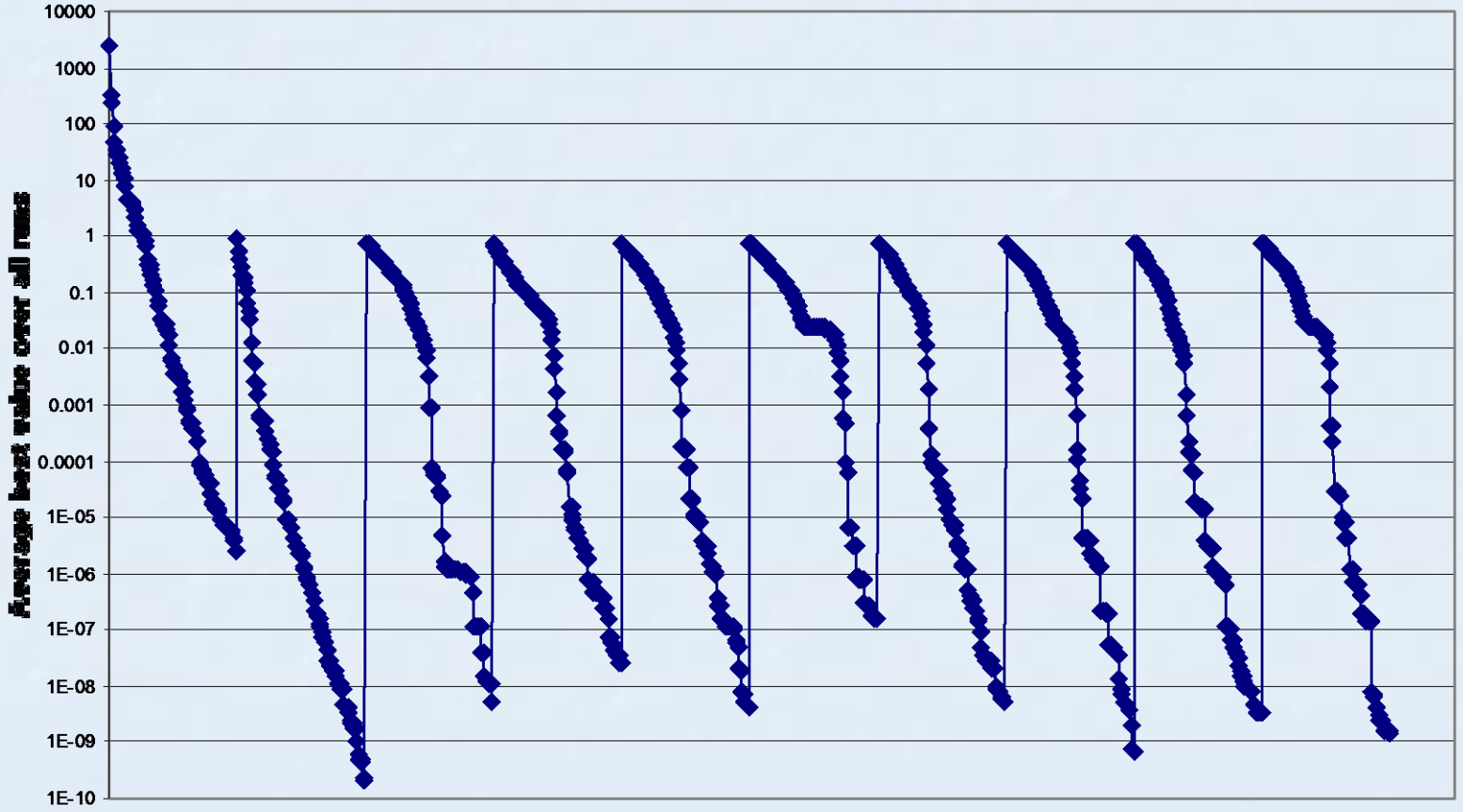
- Angeline, P.J. (1997) Tracking extrema in dynamic environments. *Proc. Evol. Programming VI*, Indianapolis, IN, Berlin: Springer-Verlag, pp. 335-345
- Bäck, T. (1998). On the behavior of evolutionary algorithms in dynamic environments. *Proc. Int. Conf. on Evol. Computation*, Anchorage, AK. Piscataway, NJ: IEEE Press, pp. 446-451

IUPUI

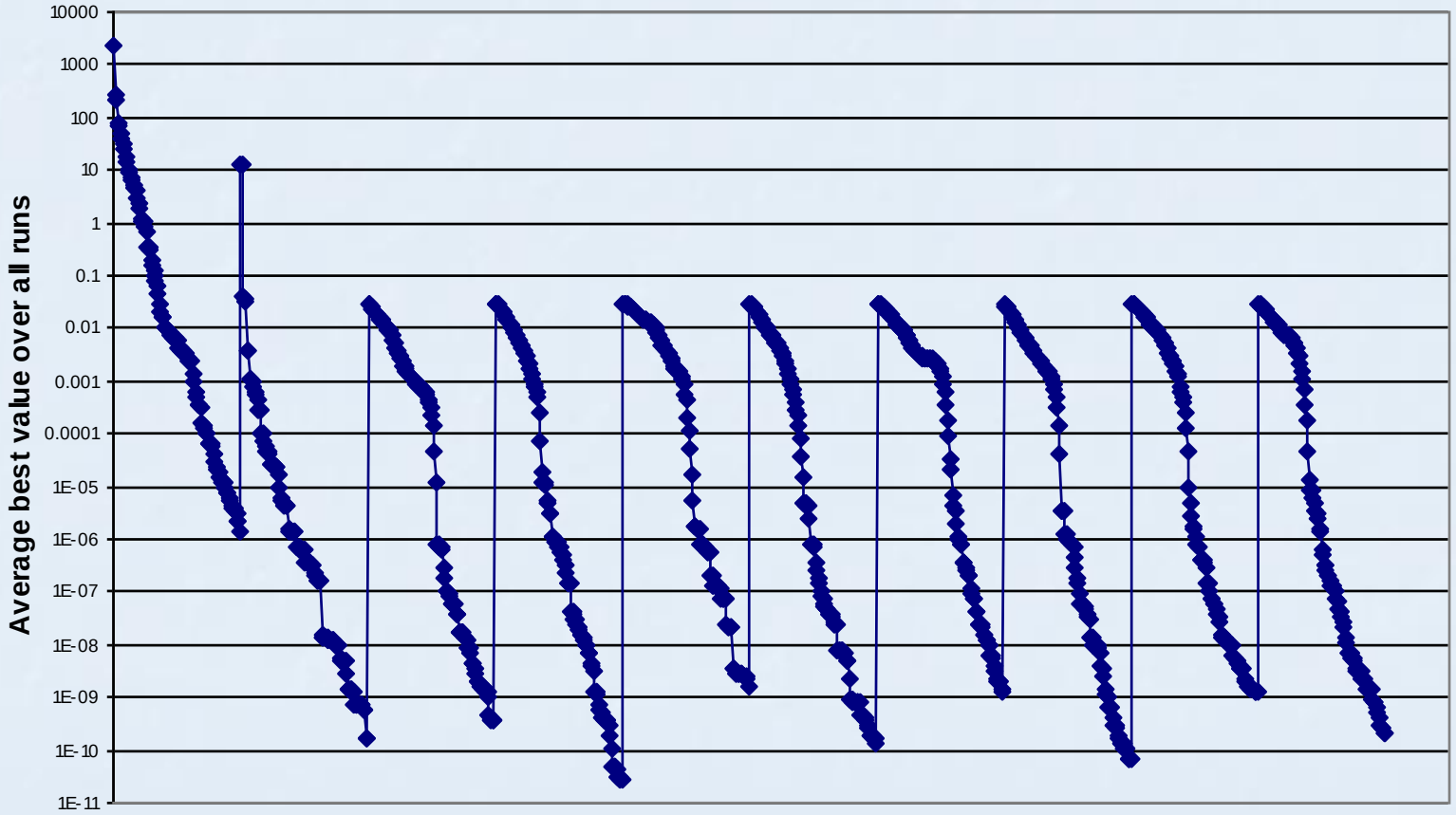
Experimental Design

- Two possibilities with swarm
  - Continue on from where we were
  - Re-initialize the swarm
- Inertia weight of [0.5+(Rnd/2.0)] used
- 20 particles; update interval of 100 generations
- When change occurred:
  - Retained the position of each particle
  - Reset values of pbest (also of gbest)

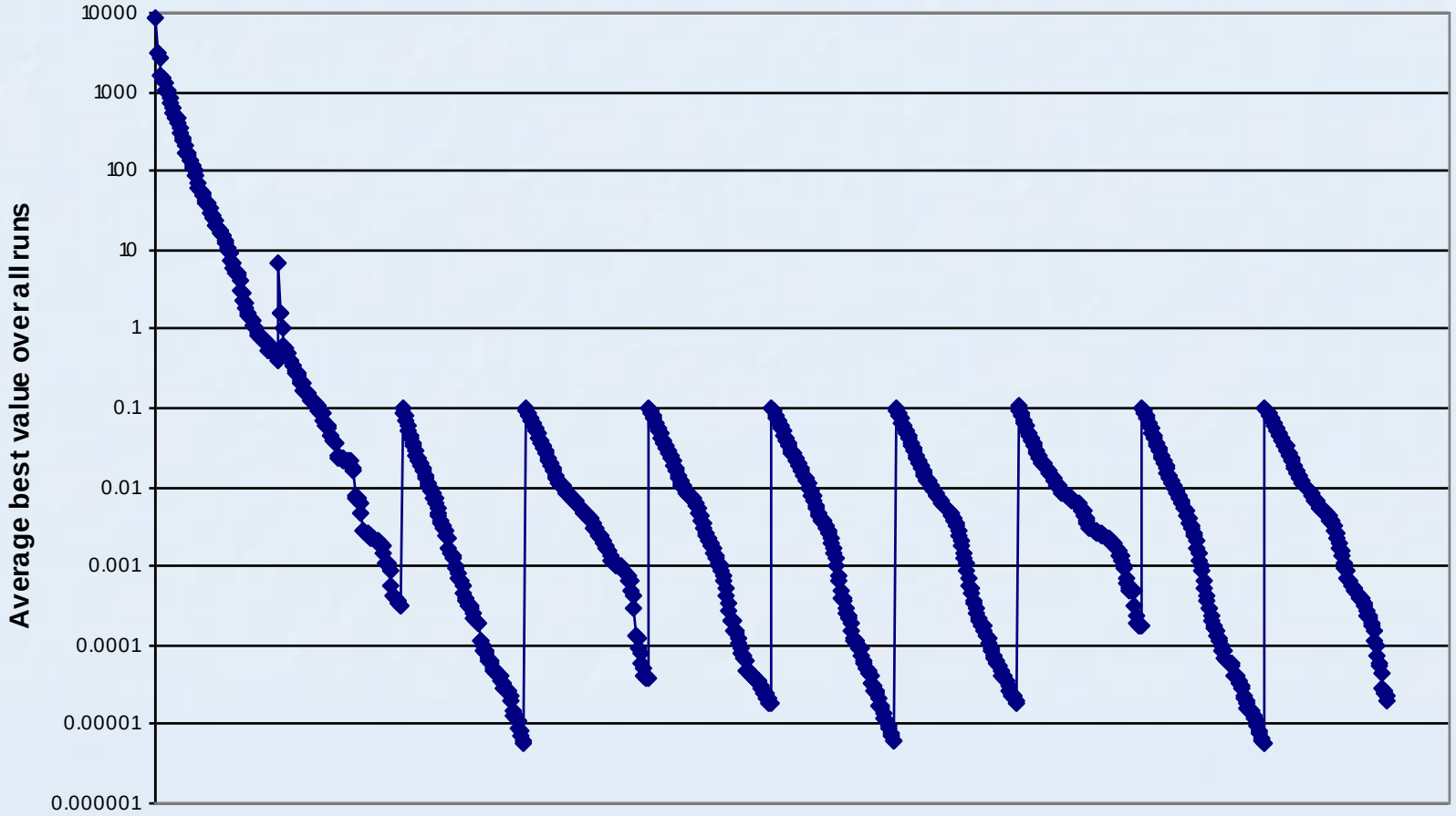
PSO average best over all runs
Severity = 0.5
Three dimensions



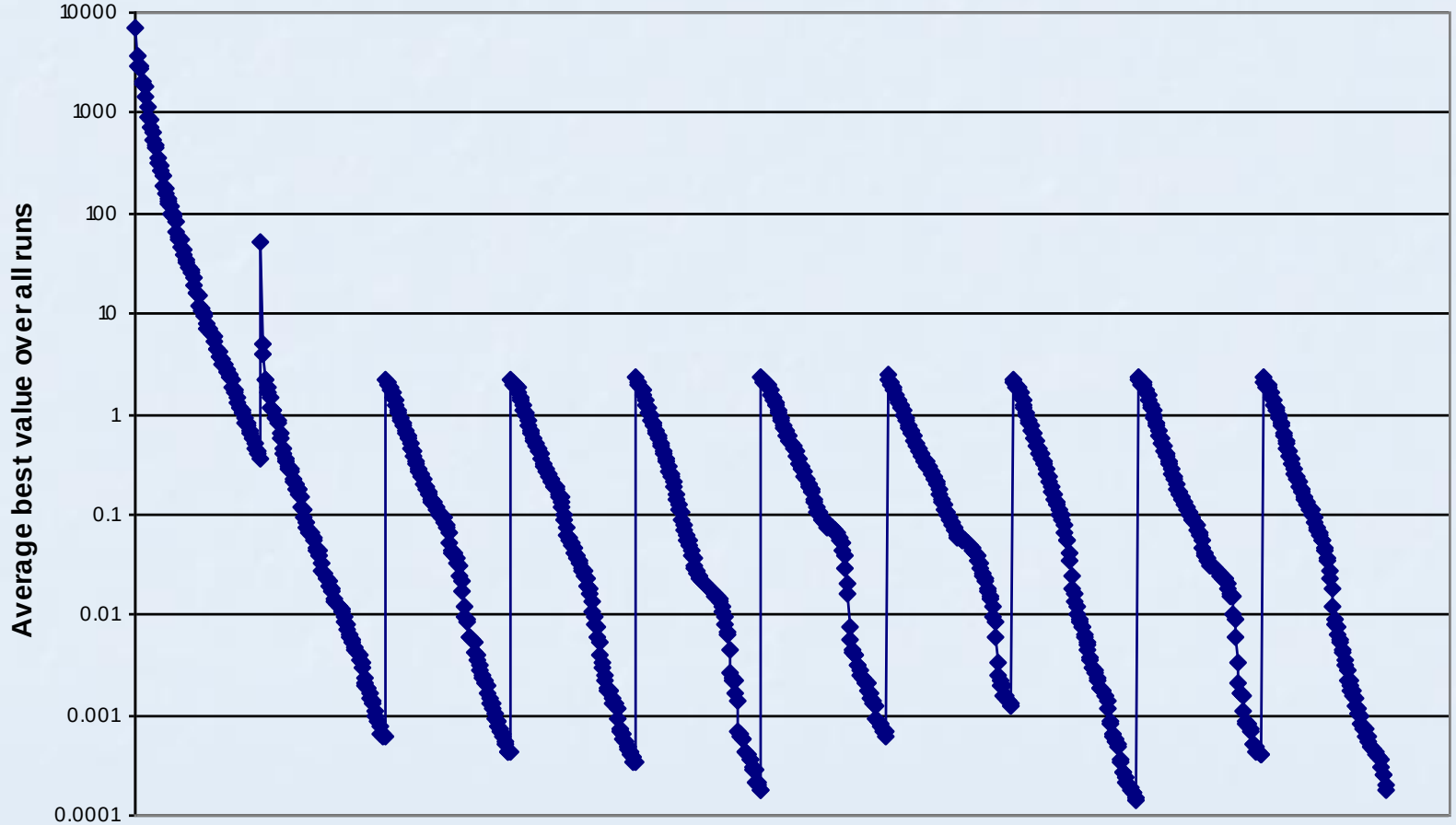
PSO average best over all runs
Severity = 0.1
Three dimensions



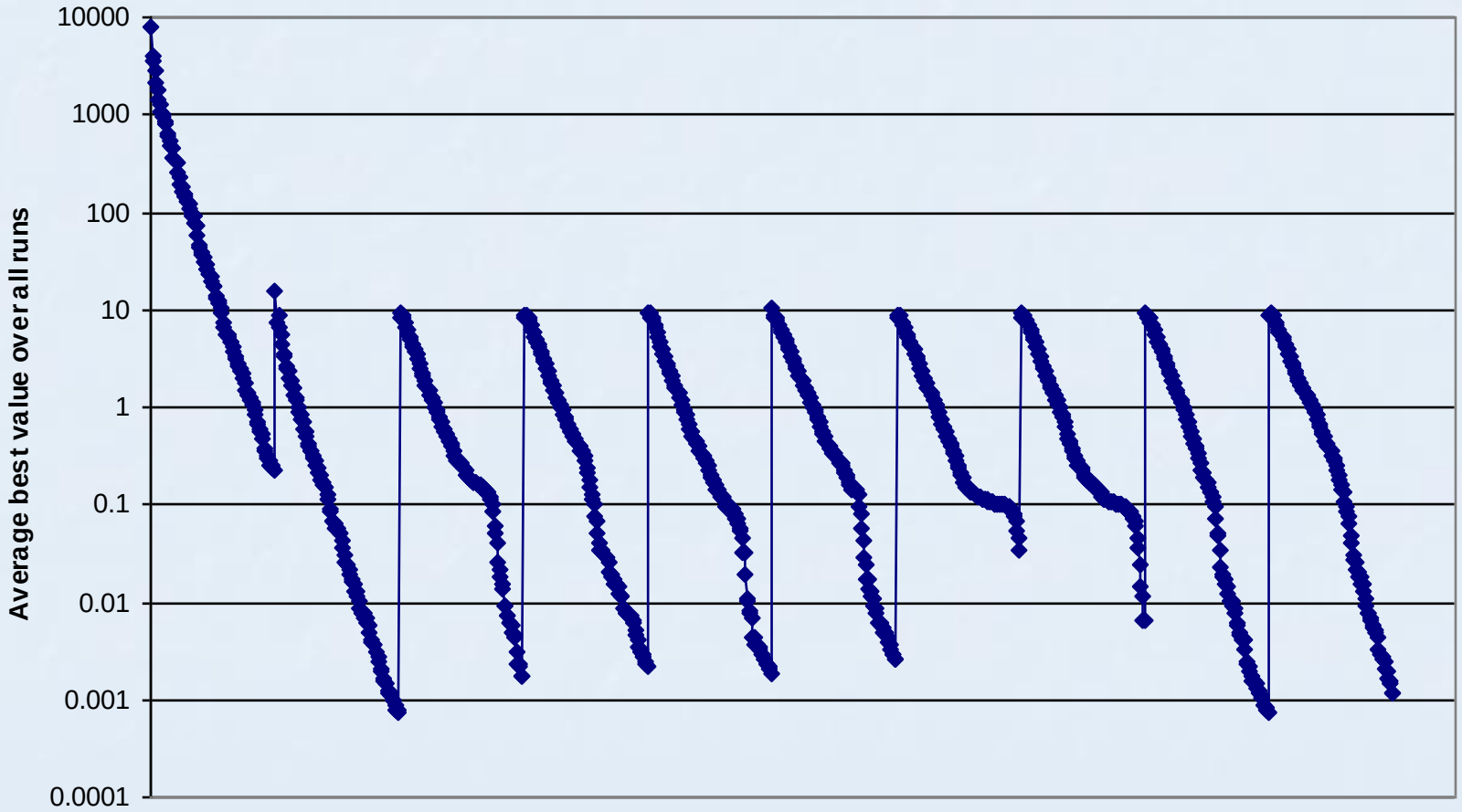
PSO average best over all runs
Severity = 0.1
10 dimensions



PSO average best over all runs
Severity = 0.5
10 dimensions



PSO average best over all runs
Severity = 1.0
10 dimensions



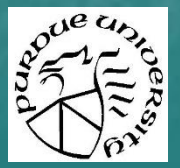
Comparison of Results: Error Values Obtained in 2000 Evaluations

	Severity 0.1	Severity 0.5
Angeline	$5x10^{-4} – 10^{-3}$	0.01-0.10
Bäck	$2x10^{-5}$	10^{-3}
Eberhart & Shi	$10^{-10} - 10^{-9}$	$10^{-9} – 10^{-8}$

IUPUI

Conclusions and Future Efforts

- Our results, including those in 10 dimensions and with severity = 1, are promising
- We are applying approach to other benchmark functions, and to practical logistics applications



IUPUI