Population Based Search

- Ant Colony Optimization
- Evolutionary Algorithms
  - Evolutionary Strategies
  - Genetic Algorithms
  - Genetic Programming

Ant Colony Optimization (ACO) [Dorigo 1992]

- constructive meta-heuristic that shares limited information between multiple search paths
- Inspiration: ants follow pheromone paths of other ants to find food. Over time, the best paths have strongest smell, luring more ants.
- Characteristics:
  - multi-agent/solution exploration (population based),
  - stochastic,
  - memory (individual and colony),
  - adaptive to problem,
  - implicit solution evaluation via "who gets there first"

Core Pseudo-Code for Each Ant

1. initialize
2. until reach solution
   1. read local routing table
   2. compute P(neighbors | routing-table, memory, constraints, heuristics)
   3. select move based on Ps and constraints
   4. update memory for move (pheromone globally and visited arc locally)
3. die

ACO Parameters

- when to terminate
- how many ants
- how to schedule ant activities (ant generation and optionals)
- what to keep in memory locally
- how to select a next state
- how to update memory (e.g., amount as function of solution quality)
- optionals:
  - delayed updating of pheromones on trail?
  - allow pheromone trail evaporation?
  - permit off-line pheromone updates (daemon actions)?

Evolutionary Algorithms Overview

- Evolutionary algorithms search a population representing different sample points in the search space.
- Each sample point is represented as a string which can be recombined with other strings to generate new sample points in the space.
- Algorithms are based on biological analogies with "population genetics" and "simulated evolution".

Why Evolutionary Algorithms?

- *No Gradient Information Is Needed.* These algorithms do not search along the contours of the function, but rather by hyperplane sampling in Hamming space.
- *The Resulting Search is Global.* Since they do not hill-climb, they avoid local optima and so can be applied to multimodal functions.
- *Potential for Massive Parallelism.* Can effectively exploit thousands of processors in parallel.
- *They Can Be Hybridized* with conventional optimization methods.

Issues: Representation

- Evolutionary model
  - Genotype: represents information stored in chromosomes
  - Phenotype: describes how the individual appears
- Approaches
  - Indirect: standard data structure, genotype is mapped to phenotype
  - Direct or natural: specialized representation using domain specific search operators

Issues: Representation

Binary: genotype is encoded as bit strings of length l

- Grey codes: similar values have similar representations
- Binary numbers: may be some issue of precision in encoded values

Issues: Representation

Nonbinary (integer, continuous, permutation): larger alphabets, real-valued encodings, more natural

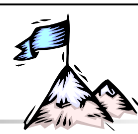
Arguments against:

- tends to have larger search space
- there will be fewer explicit hyperplane partitions
- the alphabetic characters will not be as well represented in a finite population.

Issues: Fitness Function

- Based on the objective function
- Allows comparison of different solutions
- Domain specific to goals of problem
- Single value output: multi-objective must be combined into single function

Issues: Fitness Function



- Modifications
  - Must be fast! May need to be executed hundreds of thousands of times
  - Sometimes approximate to achieve speed
  - Smoothing: replacing fitness with average of neighbors (useful for plateaus)
  - Penalty functions: to relax feasibility requirements when infeasible solutions cannot be removed

Issues: Initializing Population

Tension between good solutions and diversity (low diversity can lead to quick stagnation or large distance from optimum)

Random: generate n strings uniform randomly, within encoding requirements.

Domain specific: use heuristic method to generate "ok"(greedy) solutions that can be refined.

Evolution Strategies

- Search method like local search but can be population based
- Vector of continuous variables
- Mutates each variable by incrementing value using randomly generated change with zero mean and set standard distribution… Mutation only
- Survival of fittest between new and previous vectors

(1+1)-Evolution Strategy Algorithm

1. Create initial solution x
2. while termination criterion is not met do
   1. for $i = 1$ to n do
      $x_i' = x_i + \sigma N_i(0,1)$
   2. If $f(x') \geq f(x)$ then
      $x := x'$
   3. Update σ

N(0,1) indicates a normal distribution with 0 mean and 1 standard deviation.

(1+1)-ES Update

How to update σ?

- Convergence proofs on two simple problems define a 1/5 rule

$$\frac{\#\ \text{search steps that find a better solution}}{\text{all or last } t \text{ steps}}$$

  - If ratio is > 1/5, increase σ, else decrease
- Keep it fixed, which is better for escaping local optima

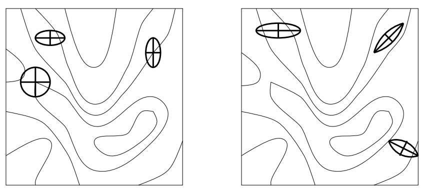
(μ+λ)-ES

- μ solutions in population
- Generate λ new solutions
- Choose the best μ from superset at each iteration
- Can add recombination before mutation
- Can have different σ per variable

Evolution Strategies (cont.)

Self adaptive mutation and rotation

- Log-normal distribution for mutation
- Adaptive through strategy (σ) and rotation angle (α) parameters added to chromosome



$$\langle x_1, x_2, \sigma_1, \sigma_2, \alpha_{1,2} \rangle$$

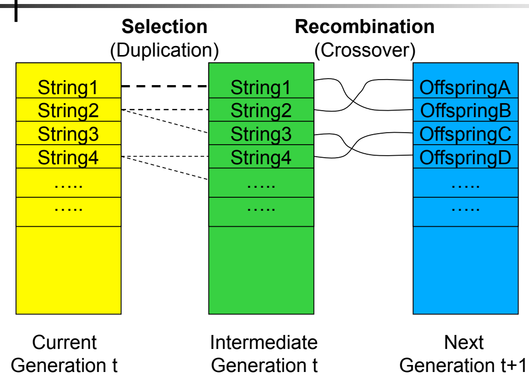
Simple Mutations — Correlated Mutation via Rotation

Figures courtesy of D. Whitley

Simple Genetic Algorithm (Alg 7 in Rothlauf)

1. Create initial population P with N solutions $x^i (i \in \{1,\ldots,N\})$
2. for $i = 1$ to N do
   1. Calculate $f(x^i)$
3. while (termination criterion is not met) and (population has not yet converged) do
   1. Create M with N individuals selected from P
   2. $ind = 1$
   3. repeat
      1. if $random(0,1) \leq p_c$ then recombine $x^{ind} \in M$ and $x^{ind+1} \in M$ and place the offspring in P'
      2. Else copy $x^{ind} \in M$ and $x^{ind+1} \in M$ to P'
      3. $ind = ind + 2$
   4. until $ind > N$
   5. for $i = 1$ to N do
      1. for $j = 1$ to l do
         1. if $random(0,1) \leq p_m$ then $mutate(x^i_j)$ where $x^i \in P'$
      2. Calculate $f(x^i)$ where $x^i \in P'$
   6. $P = P'$

Genetic Algorithm Process



Proportionate Selection

- Population is evaluated according to a fitness function.
- Parents are selected for reproduction by ranking according to their relative fitness

$$\frac{f_i}{\bar{f}} \qquad \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Proportionate Selection Process

- Stochastic sampling with replacement
  - Map individuals to space on a roulette wheel, more fit individuals are allocated proportionally more space.
  - Spin wheel repeatedly until desired population size is achieved

Population Example, *Stochastic Sampling w/Replacement*

String	Fit	Space	copies
001000000	2.0	.095	
101010101	1.9	.186	
111110011	1.8	.271	
010001100	1.7	.352	
111100000	1.6	.429	
101000110	1.5	.5	
011001110	1.4	.567	
001111000	1.3	.629	
000110100	1.2	.686	
100100011	1.1	.738	
010000111	1.0	.786	

String	Fit	Space	copies
011001111	0.9	.829	
000100110	0.8	.867	
110001101	0.7	.9	
110000111	0.6	.929	
100100100	0.5	.952	
011011011	0.4	.971	
000111000	0.3	.986	
001100100	0.2	.995	
100111010	0.1	1.0	
010010011	0.0	--	

Random #s: .93, .65, .02, .51, .20, .93, .20, .37, .79, .28, .13, .70, .80, .51, .76, .45, .61, .07, .76, .86, .29

Other Selection Methods

- **Tournament Selection**: randomly select two strings, place the best into the new population, repeat until intermediate population is full
- **Ranking**: order individuals by rank rather than fitness value

Another Fitness Selection Method

- Remainder stochastic sampling
  - Again map to roulette wheel, but this time add outer wheel with N evenly spaced pointers.
  - Spin once to determine all population members

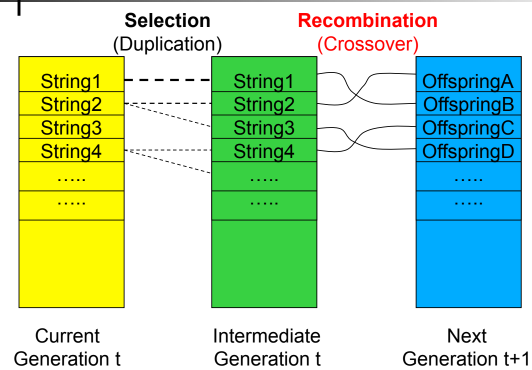
Population Example, *Remainder Stochastic Sampling*

String	Fitness	Random	copies
001000000	2.0	--	2
101010101	1.9	0.93	2
111110011	1.8	0.65	2
010001100	1.7	0.02	1
111100000	1.6	0.51	2
101000110	1.5	0.20	1
011001110	1.4	0.93	2
001111000	1.3	0.20	1
000110100	1.2	0.37	1
100100011	1.1	0.79	1
010000111	1.0	--	1

String	Fitness	Random	copies
011001111	0.9	0.28	1
000100110	0.8	0.13	0
110001101	0.7	0.70	1
110000111	0.6	0.80	1
100100100	0.5	0.51	1
011011011	0.4	0.76	1
000111000	0.3	0.45	0
001100100	0.2	0.61	0
100111010	0.1	0.07	0
010010011	0.0	--	0

Random #s: .93, .65, .02, .51, .20, .93, .20, .37, .79, .28, .13, .70, .80, .51, .76, .45, .61, .07, .76, .86, .29

Genetic Algorithm Process



Reproduction: Recombination/ Crossover

Two parents: binary strings representing an encoding of 5 parameters that are used in some optimization problems.

```
  10010101   11011001  01110100  10100101  10000101
 xyxyyyxx   xyyyyxyx  xyyxyyxy  yyxyxxxy  yxyxyxyx
```

Recombination occurs as follows:

```
10010 \ / 10111011001011101 \ / 001010010110000101
xyxyy / \ yxxxyyyyxyxxyyxy  / \ xyyyxyxxxyyxyxyxyx
```

Producing the following *offspring*:

```
10010yxxxyyyyxyxxyyxyy001010010110000101
xyxyy10111011001011101xyyyxyxxxyyxyxyxyx
```

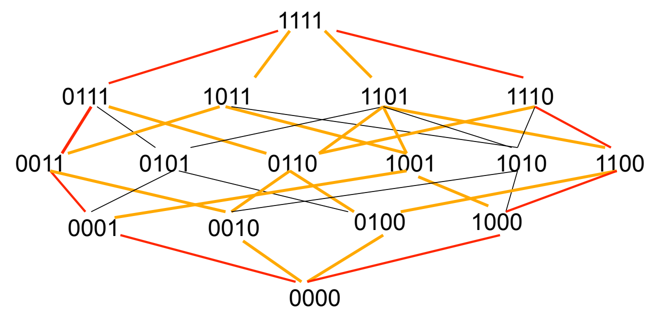
Desirable Characteristics of Recombination Operators

- Respect: any commonalities of the parents are inherited by the offspring
- Transmission: all components of the offspring must have come from a parent
- Assortment: all components of the offspring must be compatible/feasible
- Ergodicity: can reach any combination from all possible starting points

Issues: Combination Operators

- **1 point crossover**: pick single crossover point, split strings at this point, recombine
- **2 point crossover**: pick two crossover points, split strings at these points, recombine (think of ring for string)
- **Uniform crossover**: randomly pick each element from one of the two parents

Crossover and Hypercube Paths



Other Combination Operators

HUX: exactly half of the differing bits are swapped

Given parents:

10011010111100010110

10110001110000110001

a new individual is:

10010011111000110010

More Combination Operators

Reduced Surrogate: Crossover points chosen within differing bits only

10011010111100010110

10110001110000110001

may become:

10011011110000110001

Domain specific: operations designed to match demands of the domain (e.g., reorder portions for scheduling application)

Why Might Reduced Surrogate Be Important

- Closely related to HUX
- Key idea: look only at portions of strings that *differ*

  *0001**11**11011**1**010011**1***

  *0001**00**11010**0**010010**0***
- How does probability of new string change with reduced surrogate versus 1-point crossover?

Mutation

- For each bit in population, mutate with probability p_m
- p_m should be low, typically < .01
- can mean randomly select a value for the bit or flip the bit

Issues: Which Strings in New Generation

- Replace with offspring
  - Assumption of canonical GA
- Best of offspring and parents
  - Alternative view which guarantees always keep best and puts intensive pressure on population for improvement

Issues: Termination Criteria

- Quality of solution: best individual passes some pre-set threshold
- Time: certain number of generations have been created and tested
- Diminishing returns: improvement over each generation does not exceed some threshold

Alternatives to the Simple GA Model

- Genitor
- CHC

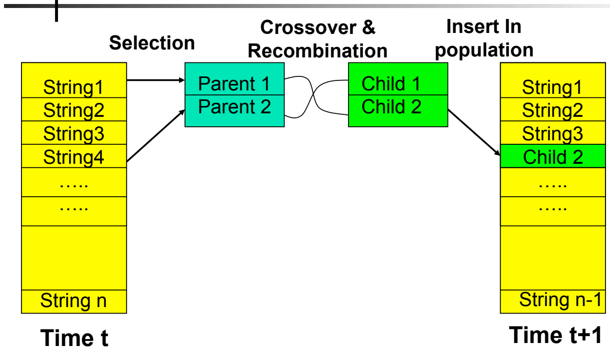
Genitor (Whitley et al.)

Differences with canonical genetic algorithm:

- Reproduction proceeds one individual at a time.
- Worst individual in population is replaced by new offspring.
- Fitness is assigned by rank.

- Steady State GA

Genitor Algorithm (Whitley)



CHC (Eshelman)

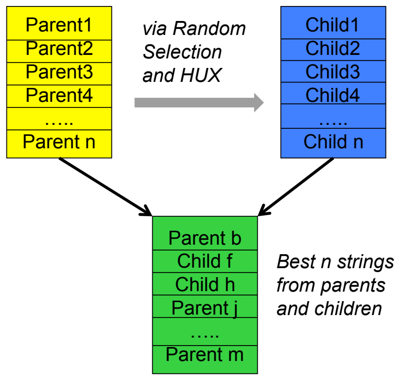
Crossover using generation elitist selection

Heterogeneous recombination by incest prevention

- Origin of HUX terminology

Cataclysmic mutation, when population starts to converge

CHC Process



Genetic Algorithms for Scheduling

- Represent solution as permutation of tasks
- Requires a schedule builder to convert solution into schedule and assess objective function. [Indirect search]
- Syswerda's permutation crossover was used for scheduling.

GA applied to TSP

- Applet showing simple GA for TSP

http://www.obitko.com/tutorials/genetic-algorithms/tsp-example.php

Genetic Programming

- Combine and/or parameterize code blocks (functions and terminals) to produce program to solve some problem.
- Follows GA functionality
- Solution is often a parse tree of functions and terminals.
  - Interior nodes are functions (e.g., "+", "or"), control structures (e.g., "if", "while") or functions with side effects (e.g., "read", "print").
  - Terminals are variables and constants.

GP Steps

1. Initialize population of programs
2. Loop
   1. Assess fitness
   2. Construct new generation through selection and
      - Reproduction,
      - Mutation or
      - Crossover

GP initialization

- Grow: start with empty tree and iteratively assign nodes to be function or terminal. All nodes at depth k_{max} are terminals.
- Full: start with empty tree, randomly add functions up to depth k_{max-1}, terminals at depth k_{max}.
- Ramped-half-and-half: divide population into (k_{max}-1) parts, half of each is created by grow and other by full, depth of nodes in ith part is varied from 2 to k_{max}

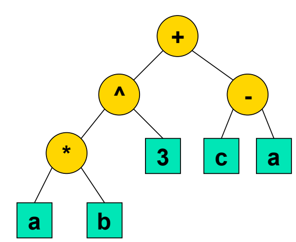
GP: Assessing Fitness

- Use program to solve problem
  - Quality of solution
  - Efficiency of solution
- May require simulation, e.g., UAV

GP: Representation

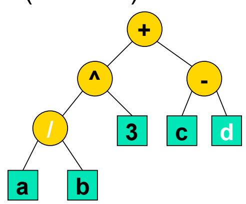
Domain Dependent!

- E.g., Simple Lisp functions (math, cons, list, append…)

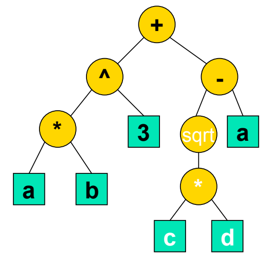


GP: Mutation

- Substitute one function (terminal) for another

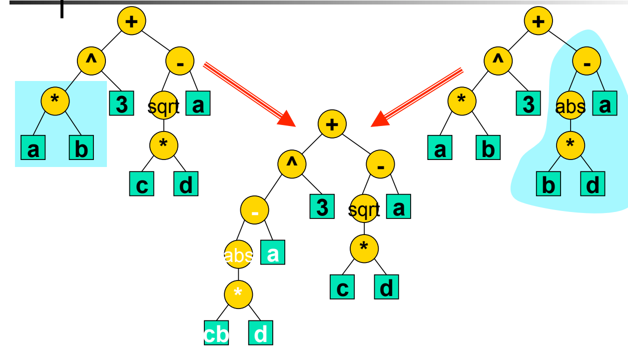


- Substitute one subtree for another



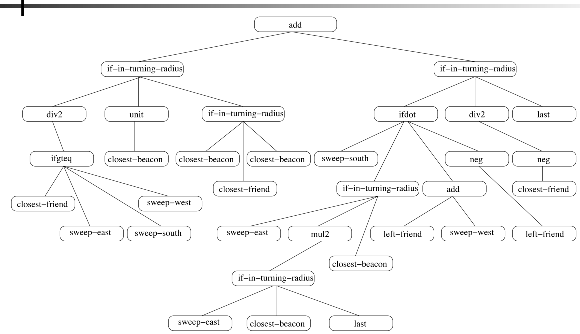
Animation: http://www.genetic-programming.com/mutation.gif

GP: Crossover



Animation: http://www.genetic-programming.com/crossover.gif

GP Representation for UAVs I



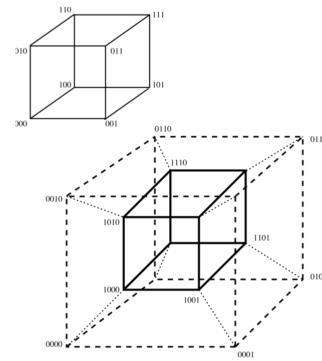
Project of Whitley, Beveridge, Richards, Mytkowicz

GP Representation for UAVs II

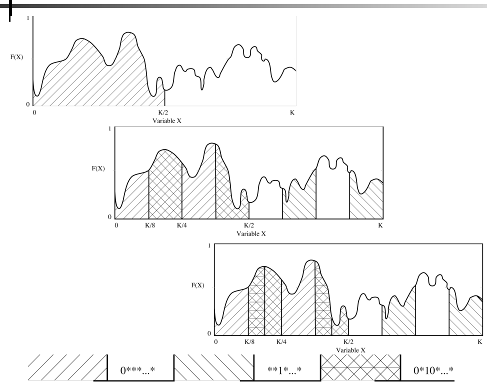
```
(add
  (if-in-turning-radius
  (div2 (ifgteq (closest-friend) (sweep-east)
                (sweep-south) (sweep-west)))
  (unit (closest-beacon))
  (if-in-turning-radius (closest-beacon)
                (closest-friend) (closest-beacon)))
  (if-in-turning-radius
  (ifdot (sweep-south)
           (if-in-turning-radius (sweep-east)
                (mul2 (if-in-turning-radius (sweep-east)
                      (closest-beacon) (last)))
                (closest-beacon))
           (add (left-friend) (sweep-west))
           (neg (left-friend)))
  (div2 (neg (closest-friend)))
  (last)))
```

UAV Movies

Underlying Theory: Hyperplane Sampling



Another View of Hyperplane Sampling



Population Example for Hyperplane Sampling

String	Fitness	Random	copies
001##...##	2.0	--	2
101##...##	1.9	0.93	2
111##...##	1.8	0.65	2
010##...##	1.7	0.02	1
111##...##	1.6	0.51	2
101##...##	1.5	0.20	1
011##...##	1.4	0.93	2
001##…##	1.3	0.20	1
000##…##	1.2	0.37	1
100##…##	1.1	0.79	1
010##…##	1.0	--	1

String	Fitness	Random	copies
011##…##	0.9	0.28	1
000##…##	0.8	0.13	0
110##…##	0.7	0.70	1
110##…##	0.6	0.80	1
100##…##	0.5	0.51	1
011##…##	0.4	0.76	1
000##…##	0.3	0.45	0
001##…##	0.2	0.61	0
100##…##	0.1	0.07	0
010##…##	0.0	--	0

Some Schemata and Fitness Values

Schema	Mean	Count	Expect	Observe
101**...*	1.70	2	3.4	3
111**...*	1.70	2	3.4	4
1*1**...*	1.70	4	6.8	7
**01*...*	1.38	5	6.9	6
***1*...*	1.30	10	13.0	14
**11*...*	1.22	5	6.1	8
11***...*	1.175	4	4.7	6
001**...*	1.166	3	3.5	3
1****...*	1.089	9	9.8	11
0*1**...*	1.033	6	6.2	7
10***...*	1.020	5	5.1	5
1...*	1.010	10	10.1	12
*****...*	1.000	21	21.0	21

Hyperplane Deception

Since genetic algorithms are driven by hyperplane sampling a misleading problem can be constructed as follows.

f(0**) > f(1**)
f(*0*) > f(*1*)
f(**0) > f(**1)
f(00*) > f(01*), f(10*), f(11*) f(0*0) > f(0*1), f(1*0), f(1*1)
f(*00) > f(*01), f(*10), f(*11)

BUT f(111) > f(000)
where f(x) gives the average fitness of all strings in the hyperplane slice represented by x.

Fitness Landscape Analysis

- A fitness landscape *(X,f,d)* of a problem instance consists of a set of solutions *X*, an objective function *f* and a distance measure *d*.
- We can analyze problems based on characteristics derived from this definition and determine (roughly) problem difficulty.

Locality

- How well distances correspond to differences in fitness between solutions
- High locality if nearby solutions have similar fitness
- Low locality problems are difficult for local search

Fitness-Distance Correlation

- Metric of locality

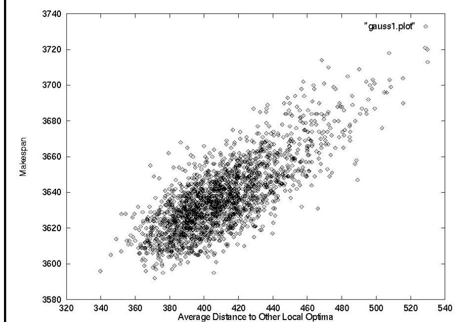
$$\rho_{FDC} = \frac{c_{fd}}{\sigma(f)\sigma(d_{opt})}$$

where

$$c_{fd} = \frac{1}{m}\sum_{i=1}^{m}\left(f_i - \langle f\rangle\right)\left(d_{i,opt} - \langle d_{opt}\rangle\right)$$

- c is the covariance, <> are means.
- Positive correlation is easy, uncorrelated is difficult, negative correlation is misleading.

Big Valley Topology



Original observation from (Boese, Kahng & Muddu 1994)

Figure: benchmark FlowShop scheduling problem from Watson et al. 1999

Ruggedness

- Statistical property on relation of objective values to intra-solution distances
- Random walk correlation between solutions separated by s steps

$$r(s) = \frac{\langle f(x_i)f(x_{i+s})\rangle - \langle f\rangle^2}{\langle f^2\rangle - \langle f\rangle^2}$$

x_i is the solution at step i

- Correlation length…high means smooth, easy for guided search

$$l_{corr} = -\frac{1}{\ln\left(\left|r(1)\right|\right)}$$

No Free Lunch…

- Wolpert and Macready, 1997

  "For any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class."

- Proven for finite optimization problems solved with deterministic "black-box" algorithms

NFL Theorem I

For any pair of algorithms a_1 and a_2:

$$\sum_f P(d_m^y \mid f, m, a_1) = \sum_f P(d_m^y \mid f, m, a_2)$$

f is an optimization problem, m is number of distinct points sampled by the algorithm, y is the cost value, d is the best cost value over the sample.

Meaning… *average* performance for an algorithm over *all* problems is the same as for any other algorithm, assuming equal sampling size.

Implications of NFL

- Emphasizes need for different algorithms
- Does not preclude superior performance on some interesting class(es) of problem!

$$P(d_m^y \mid m, a) = \sum_f P(d_m^y \mid f, m, a) P(f)$$

- However, if no problem knowledge is used, then *P(f)* is essentially uniform.

Implications (cont.)

- Structure of problem must be understood and used to inform choice of search.
- Knowledge of cost function properties is folded into P(f) or $\vec{p}$

$$P(d_m^y \mid m, a) = \vec{v}_{d_m^y, m, a} \bullet \vec{p}$$

- "performance of an algorithm is determined by … how aligned $\vec{v}_{d_m^y, m, a}$ is with the problems $\vec{p}$ "

Comparing Performance

- Assume we can construct a histogram of cost values produced by a run of an algorithm $\vec{c}$, then use these to gauge performance:
  - Over all cost functions:
    - Average of $P(\min(\vec{c}) > \varepsilon \mid f, m, a)$
    - $P(\min(\vec{c}) > \varepsilon \mid f, m, a)$ for the random algorithm
  - Given f and m, % of algorithms with $\min(\vec{c} > \varepsilon)$

Minimax & Fixed f

- NFL focuses on *average* performance.
  - One algorithm may be *much* better than another on a subset of f (the other is a little better on remaining to even out average).[Minimax]
  - May occur when samples intersect
- If f is fixed, observing performance until time *t* does not indicate what will happen later.
  - Intelligent choice requires knowledge of both f and a

Phase Transitions

Cheeseman, Kanefsky & Taylor (1991) observed that:

- The hardest problems in an NP-complete class (e.g., TSP, SAT) were at the boundary between feasibility and infeasibility.
- Problems that are clearly feasible (or not) are easy to solve (or easy to recognize there's no solution).
- "*phase transition*" depends on identifying a key scale up parameter (e.g., constraints per variable, standard deviation of costs)