UNIVERSITÄT PASSAU

Search-Based Software Engineering

Parameters, Adaptation, Hyper-Heuristics

Gordon Fraser
Lehrstuhl für Software Engineering II

Parameters in EAs

Qualitative Parameters

- Representation
- Recombination
- Mutation
- Parent selection
- Survivor selection
- …

Quantitative Parameters

- Mutation rate
- Crossover rate
- Selection bias
- Population size
- Migration frequency
- …

Default Values

Qualitative Parameters

- Representation
- Recombination (1-point)
- Mutation (Gaussian)
- Parent selection (Rank)
- Survivor selection
- …

Quantitative Parameters

- Mutation rate (<0.1)
- Crossover rate (0.6-0.9)
- Selection bias (1.7)
- Population size (50-100)
- Migration frequency
- …

No Free Lunch Theorem

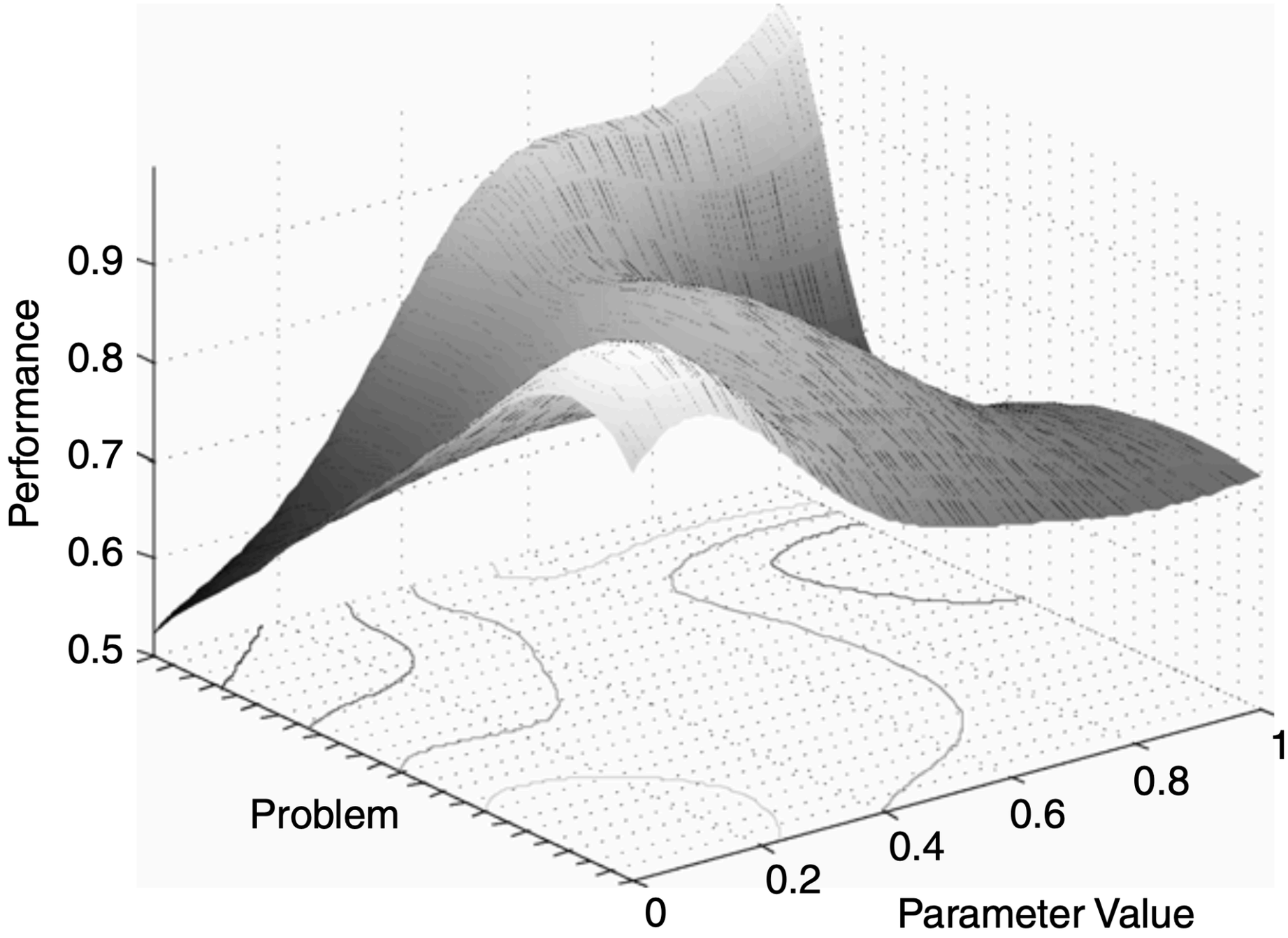
If an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems.

Wolpert & Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation, 1997.

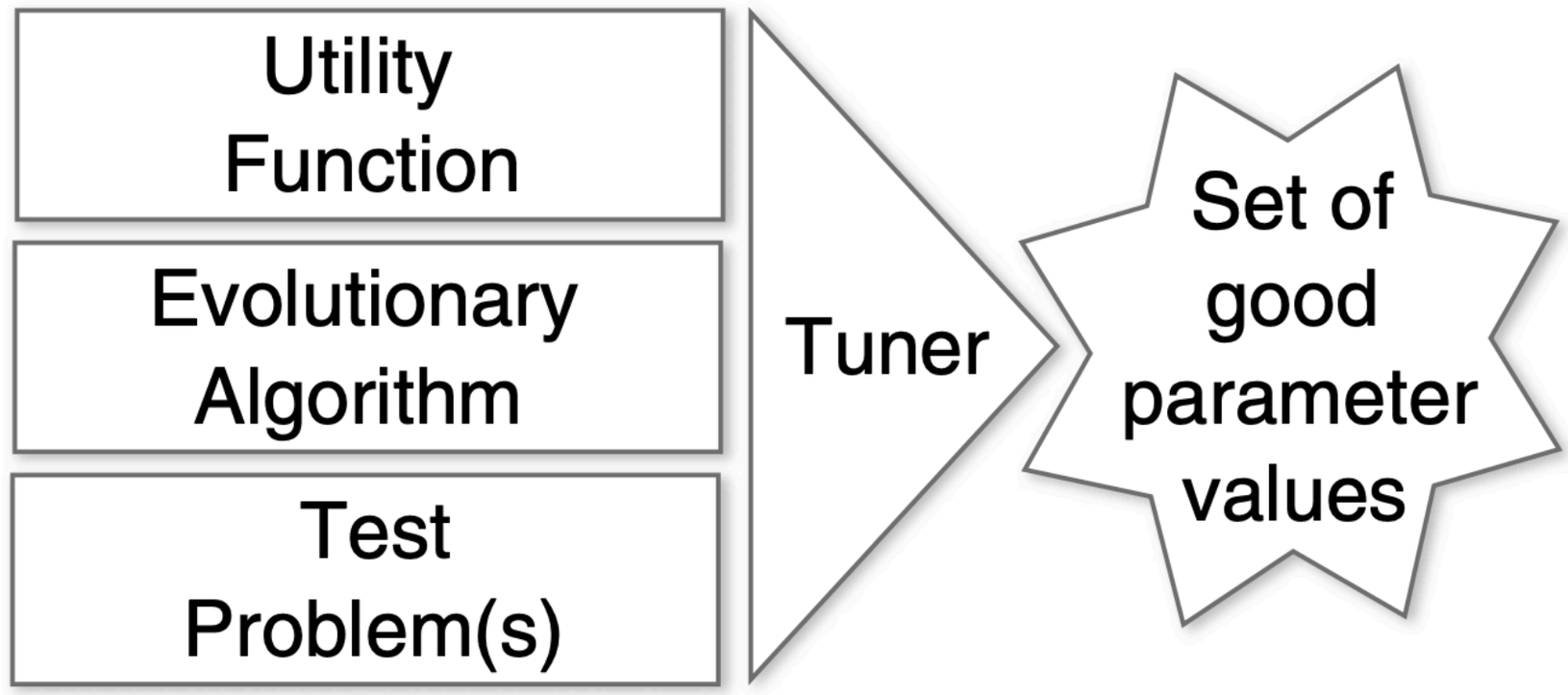
Parameter Tuning

- Structural tuning: takes place in the space of qualitative parameters
- Parametric tuning refers to searching through quantitative parameters
- Utility: Quality of parameter vector
- Utility landscape: abstract landscape where the locations are the parameter vectors of an EA and the height reflects utility
- Fitness values are most often deterministic—depending, of course, on the problem instance to be solved. However, the utility values are always stochastic, because they reflect the performance of an EA which is a stochastic search method

Parameter Tuning



Parameter Tuning



Performance Measures

- Solution quality is expressed by the fitness function
- As for algorithm speed, time or search effort needs to be measured
  - E.g. number of fitness evaluations, CPU time, wall-clock time, etc.
- There are different combinations of fitness and time that can be used to define algorithm performance in one single run. For instance:
  - Given a maximum running time (computational effort), algorithm performance is defined as the best fitness at termination.
  - Given a minimum fitness level, algorithm performance is defined as the running time (computational effort) needed to reach it.
  - Given a maximum running time (computational effort) and a minimum fitness level, algorithm performance is defined through the Boolean notion of success: a run succeeds if the given fitness is reached within the given time, otherwise it fails.

Robustness

- Robustness to changes in problem specification
  - If parameters are tuned on one instance of the problem, how well do the parameters work on other problems?
- Robustness to changes in parameter values
- Robustness to changes in random seeds
  - Ratio of runs ending with a good result above some threshold T
  - If the difference between the best and worst run is big, then we call this EA instance unstable.

Tuning Objectives

- To obtain an algorithm instance with high performance, given a performance measure or a combination of measures (on one or more problems).

- To obtain an algorithm instance that is robust to changes in problem specification.

- To indicate the robustness of the given algorithm to changes in parameter values.

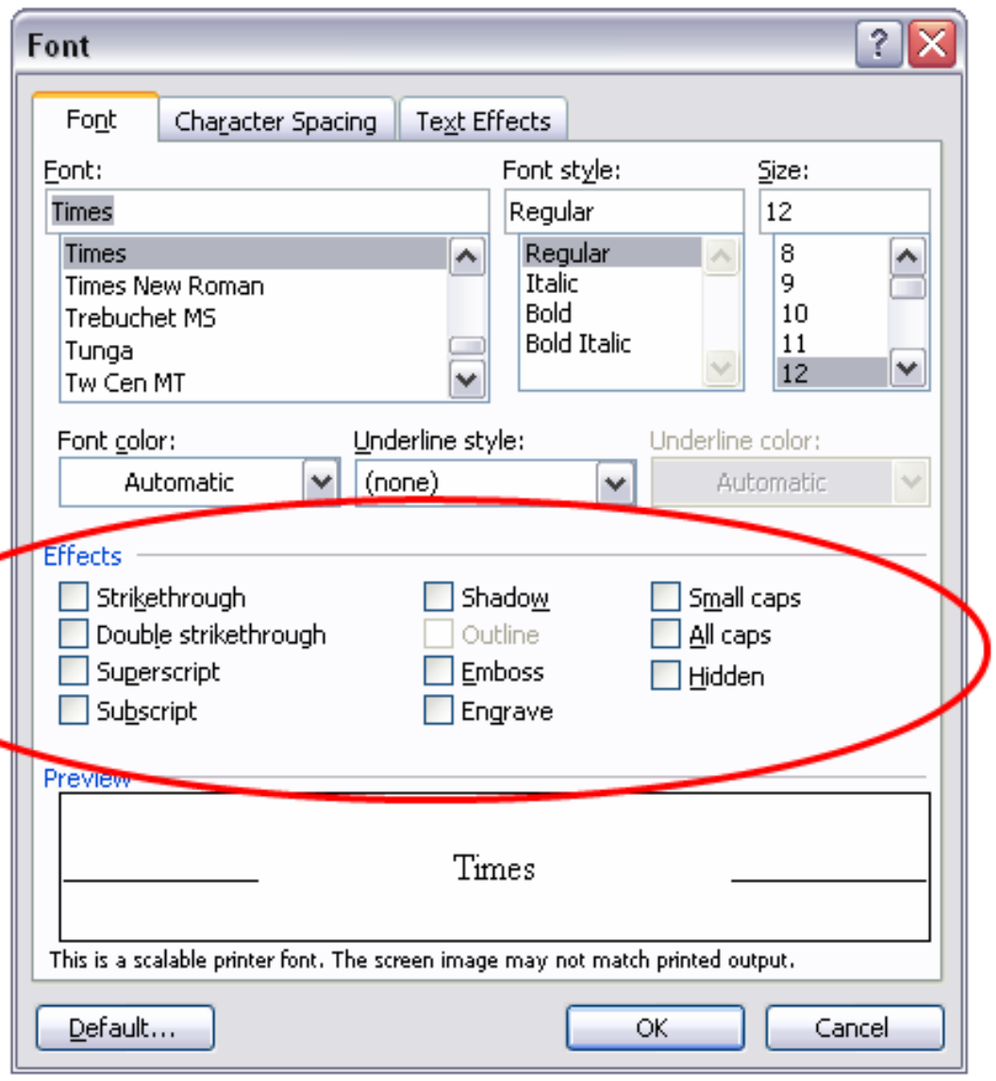
- To obtain an algorithm instance that is robust to random effects during execution.

Parameter Tuning

- Which parameters?
    - One at a time
    - Multiple parameters — which ones? All?
- Which levels?
- Which training set?

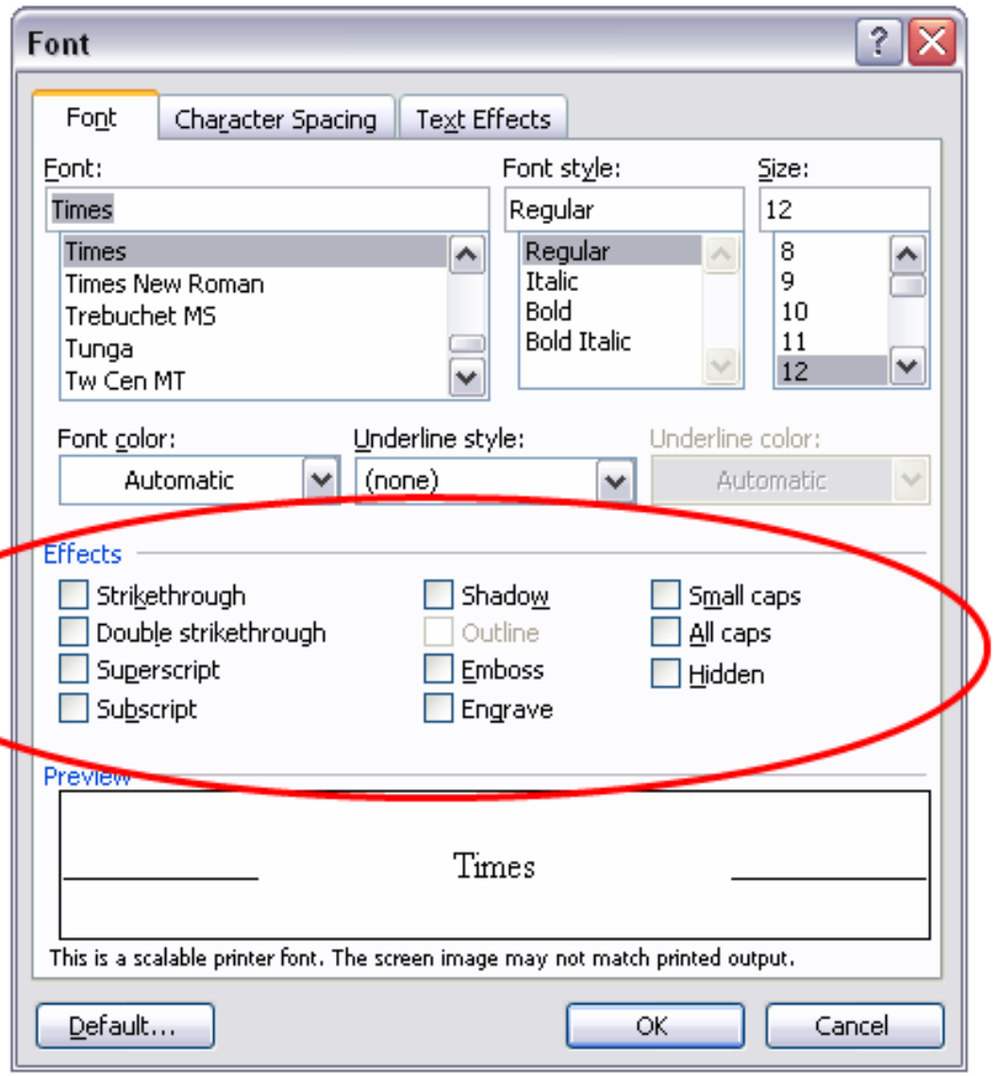
Example Problem: Combinatorial Interaction Testing

Combinatorial Interaction Testing

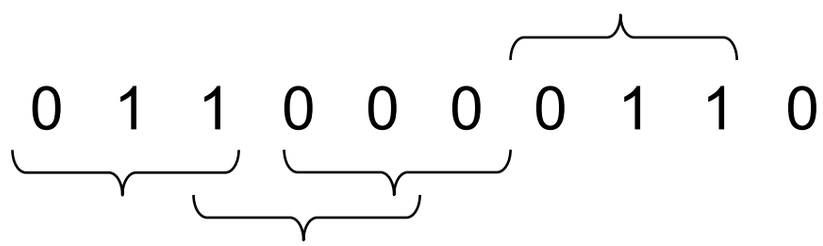


- 10 font effects, each can be on or off
- All combinations: 2^{10} = 1,024 tests

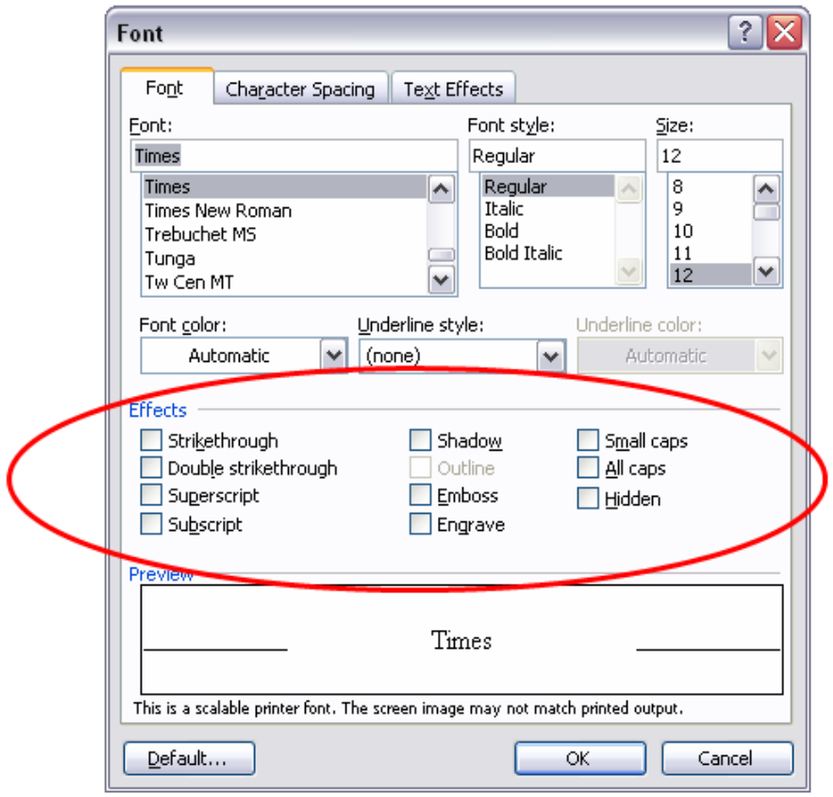
Combinatorial Interaction Testing

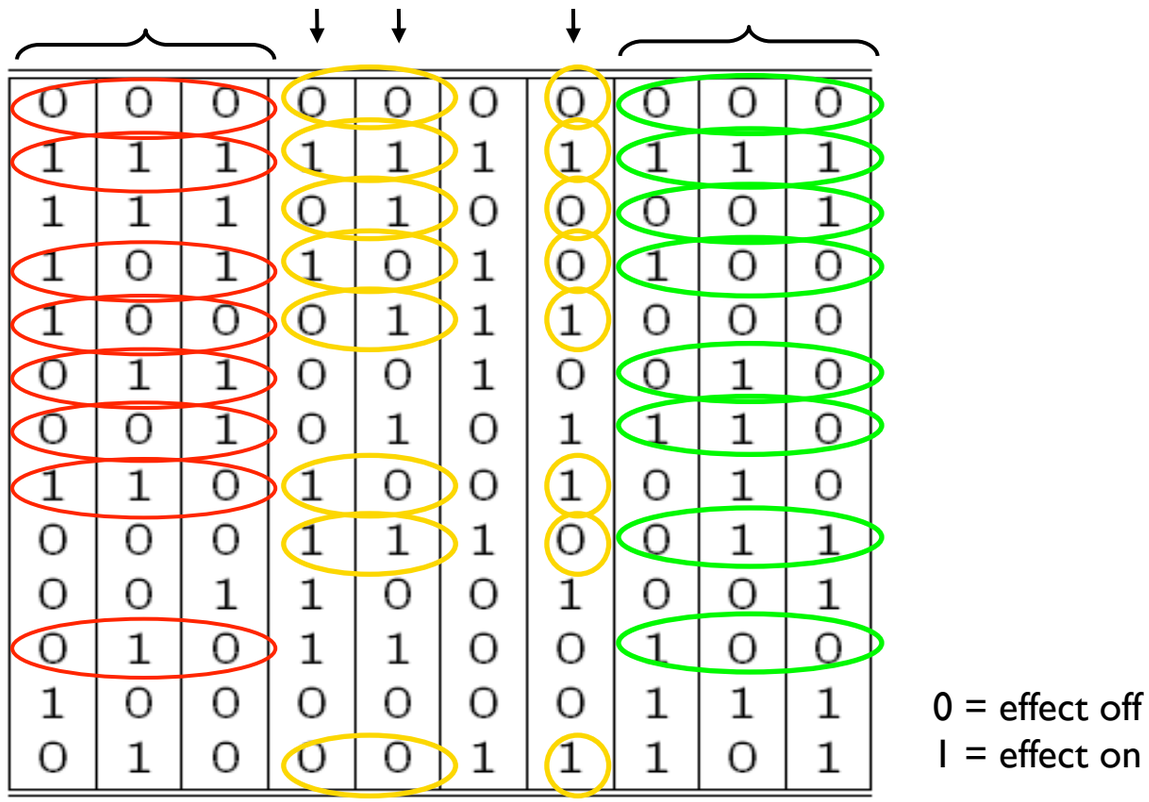


- There are $\binom{10}{3}$ =120 3-way interactions
- At least: 120 x 2^3 = 960 tests.
- Each test covers multiple triples:



Covering Array





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

0 = effect off
1 = effect on

- Each test covers 120 3-way combinations
- All 3-way combinations (960) with 13 tests
- Finding Covering Arrays is NP-hard

Orthogonal Arrays

An Orthogonal Array $OA_\lambda(N; t, k, v)$ is an $N \times k$ array with v symbols such that each $N \times t$ sub-array contains all ordered subsets of size t of v symbols exactly λ times

$OA_1(9, 2, 4, 3)$:

2	1	2	2
0	2	1	2
1	2	2	1
2	2	0	0
2	0	1	1
0	0	2	0
0	1	0	1
1	1	1	0
1	0	0	2

$\lambda = N/v^t$

$= 9/3^2 = 1$

λ = index of the OA

If omitted, then $\lambda = 1$

Latin Squares

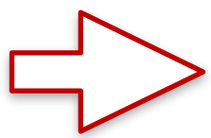
A	B	C
C	A	B
B	C	A

A latin square is an $n \times n$ array with n different symbols, each of which occurs exactly once in each row and column.

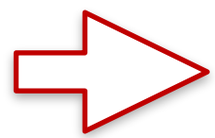
Pairwise Testing with Latin Squares

1. RAID0, RAID1, RAID5
2. XP, Linux, Novell
3. 64MB, 128MB, 256MB

0	1	2
1	2	0
2	0	1



Row	Column	Cell
0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	0
2	0	2
2	1	0
2	2	1



RAID 0	XP	64MB
RAID 0	Linux	128MB
RAID 0	Novell	256MB
RAID 1	XP	128MB
RAID 1	Linux	256MB
RAID 1	Novell	64MB
RAID 5	XP	256MB
RAID 5	Linux	64MB
RAID 5	Novell	128MB

Orthogonal Latin Squares

A	B	C
C	A	B
B	C	A

A	B	C
B	C	A
C	A	B

A,A	B,B	C,C
C,B	A,C	B,A
B,C	C,A	A,B

Two latin squares are orthogonal if all pairwise combinations occur exactly once in the combined square.

Pairwise Testing with Latin Squares

k orthogonals latin squares of size s are required to test k + 2 components with s values each.

1. RAID0, RAID1, RAID5
2. XP, Linux, Novell
3. 64MB, 128MB, 256MB
4. Ultra 320, Ultra 160-SCSI, Ultra 160-SATA

0	1	2
1	2	0
2	0	1

0	1	2
2	0	1
1	2	0

Row	Column	Cell 1	Cell 2
0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

Pairwise Testing with Latin Squares

Row	Column	Cell 1	Cell 2
0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

Factor 1	Factor 2	Factor 3	Factor 4
RAID 0	XP	64MB	Ultra 320
RAID 0	Linux	128MB	Ultra 160-SCSI
RAID 0	Novell	256MB	Ultra 160-SATA
RAID 1	XP	128MB	Ultra 160-SATA
RAID 1	Linux	256MB	Ultra 320
RAID 1	Novell	64MB	Ultra 160-SCSI
RAID 5	XP	256MB	Ultra 160-SCSI
RAID 5	Linux	64MB	Ultra 160-SATA
RAID 5	Novell	128MB	Ultra 320

Covering Arrays

- A Covering Array, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array of v symbols such that each $N \times t$ sub-array contains all ordered subsets of v symbols of size t *at least* λ times.
- If $\lambda = 1$ then $CA(N; t, k, v)$.

Component 1: {0, 1}
Component 2: {2, 3}
Component 3: {4, 5}
Component 4: {6, 7}

CA(5; 2, 4, 2):

1	2	5	6
1	3	4	7
0	2	5	7
0	3	5	6
0	2	4	6

Mixed Level Covering Arrays

- Covering Arrays (and OAs) require that all components have the same number of values. That is not always the case.
- A Mixed Level Covering Array $MCA_\lambda(N; t, k,(v_1, v_2, ..., v_k))$, is an $N \times k$ array with v symbols, where:

  1. Each column i ($1 \leq i \leq k$) contains only elements of S_i with size v_i
  2. The rows cover each t-tuple with values of the t at least λ times.

Mixed Level Covering Arrays

Component 1: {0, 1, 2, 3}
Component 2: {a, b, c}
Component 3: {4, 5, 6}
Component 4: {d, e}

MCA(12; 2, 4, $4^1 3^2 2^1$):

0	a	4	d
2	b	6	e
3	c	5	e
2	c	4	d
0	b	5	d
1	a	6	e
1	b	4	d
3	a	6	d
0	c	6	e
2	a	5	e
3	b	4	e
1	c	5	d

Covering Arrays as Search Problem

- Example system, 3 parameters, 3 values each:
  - A = a1,a2,a3
    B = b1,b2,b3
    C = c1,c2,c3
- Integer encoding:
  - 0 = a1
    1 = a2
    2 = a3
    3 = b1
    4 = b2
    …

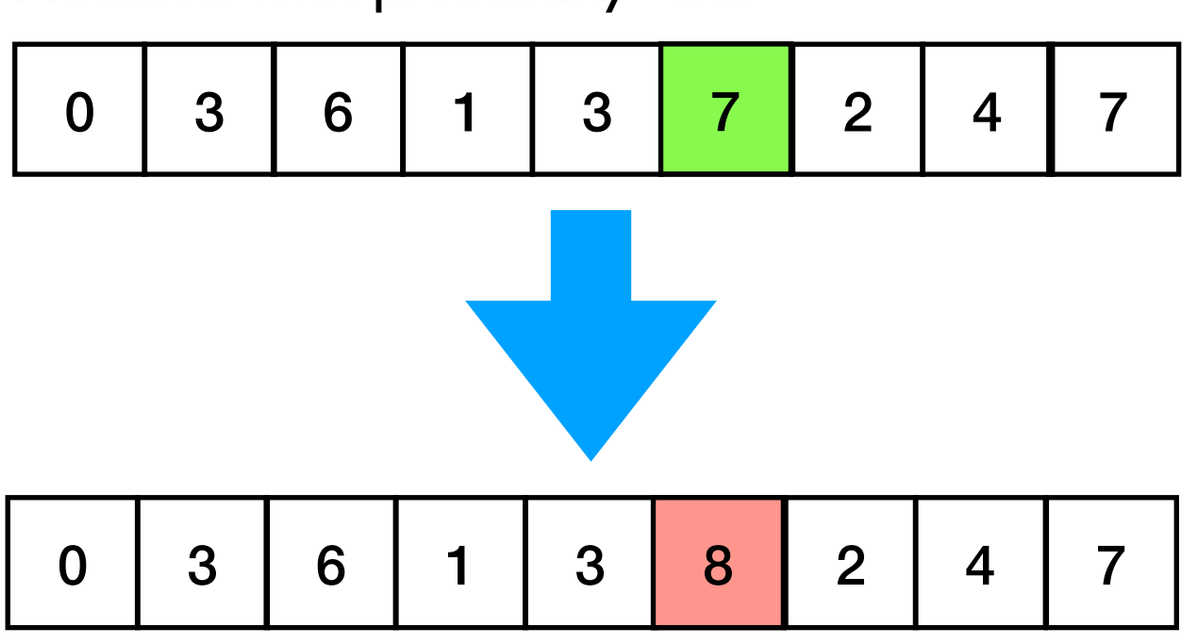
0	3	5

Encoding for multiple tests:

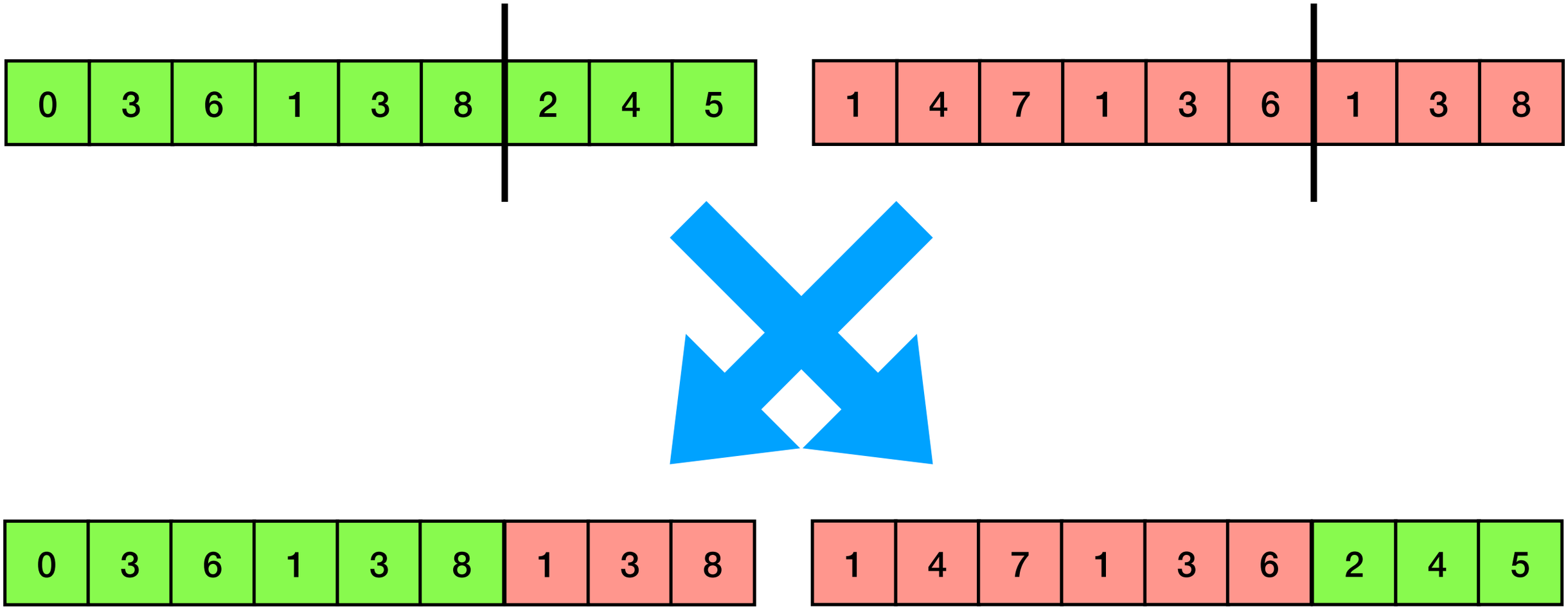
0	3	5	1	3	6	2	4	5

Covering Arrays as Search Problem

Mutation with probability 1/n:

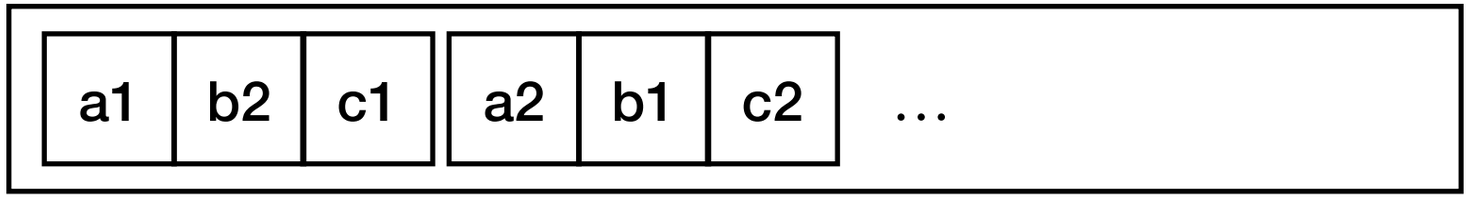


Covering Arrays as Search Problem



Covering Arrays as Search Problem

- Example system, 3 parameters, 3 values each:
  - A = a1,a2,a3
    B = b1,b2,b3
    C = c1,c2,c3
- We are not restricted to vector representation
  - Each gene can be a vector on its own (or any other datastructure)
  - Alleles can be any symbols, not just numbers/bits



Covering Arrays as Search Problem

- Crossover: Any regular vector-based xover function works
- Initial simple mutation:
  - Mutate each test with probability 1/#tests
    - Mutate each parameter in a mutated test with probability 1/#parameters
    - Replace parameter with another valid parameter
- Objective: Cover as many as possible t-way combinations
  - Fitness function is simple: Count number of combinations
- Search algorithm: Any.

Covering Arrays as Search Problem

- How to choose the number of tests *n*?
  - If we use too few, we cannot cover all pairs
  - If we use too many, we get redundant tests
- Iterative approach:
  - Run search with sufficiently large *n*
  - Run search again for *n* - 1
  - Run search again for *n* - 2
  - …
- Alternative:
  - Variable size representation

Variable Size Representations

- Mutation: Allow for increase/decrease of number of tests
  - Each test can be deleted with a certain probability
  - Mutation may insert new tests
- Potential Problem: Bloat
  - Individuals can grow with mutation
  - Individuals can grow with crossover
  - Adding tests will never reduce fitness, but will often increase fitness
  - Individuals in the population will grow uncontrollably
- Solution 1: Multi-Objective Search for coverage and size
- Solution 2: Bloat Control
  - Restrict growth in operators (crossover, mutation)
  - Selection: If two individuals have the same fitness, prefer the smaller one

Parameter Tuning

- Factorial Design: All combinations of all parameters
- Example: Considering four parameters and five values for each of them, one has to test $5^4 = 625$ different setups. Performing 100 independent runs with each setup, this implies 62,500 runs just to establish a good algorithm design
- Parameters are not independent, but trying all different combinations systematically is practically impossible.
- For a given problem the selected parameter values are not necessarily optimal, even if the effort made for setting them was significant.

Sampling Methods

- Iterative methods: Start with Graeco-Latin Squares, then do full factorial design on restricted parameter space
- Model-based methods: Construct a (regression) model of the utility landscape and use model estimates to reduce number of samples
- Iterative model-based: Learn model from factorial design, and determine path of steepest ascent. Then test this path until best solution has not changed for a number of iterations
- Meta-EA: Finding parameter vectors with a high utility is a complex optimization task with a nonlinear objective function, interacting variables, multiple local optima, noise, and a lack of analytic solvers. EAs are very good solvers for such problems…

Response Surface Methodology

- Given n parameters where each w_i represents their value, a parameter configuration would be defined by the vector $W = \{w_1, \ldots, w_n\}$.
- We choose k configurations and can calculate a response $r(W_\rho) = y$, i.e., a measure that quantifies how good the configuration W_ρ is at solving the problem at hand.
- Using the response $r(W_\rho) = y$ for each of them, we build a model f to capture the relations between the parameters w_i and response y:

$$f(W_\rho) = \beta_0 + \sum_{i=1}^{n} \beta_i w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \beta_{ij} w_i w_j$$

- Once the model f is built, we can optimise its variables (but keeping the βs as constant) to find the combination that obtains a response y as high as possible.

Response Surface Methodology

- Fractional Factorial Design: Subset of factorial design
- For each parameter we obtain five levels, i.e., five different values. In particular, we obtain $\{0, 1, -1, \alpha, -\alpha\}$, where α depends on several factors
- Central composite design:
  - Matrix F: A factorial (fractional) design in the factors studied, each having two levels (1, -1)
  - Matrix C: A set of center points, experimental runs whose values of each factor are the medians of the values used in the factorial portion.
  - Matrix E: A set of axial points, runs identical to the centre points except for one factor, which will take on values both below and above the median of the two factorial levels. All factors are varied in this way.

Response Surface Methodology

$$C = \begin{bmatrix} & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & & & \end{bmatrix}$$

Response Surface Methodology

$$E = \begin{bmatrix} \boldsymbol{\alpha} & 0 & \dots & \dots & \dots & 0 \\ \boldsymbol{-\alpha} & 0 & \dots & \dots & \dots & 0 \\ & \boldsymbol{\alpha} & 0 & \dots & \dots & \dots & 0 \\ & \boldsymbol{-\alpha} & 0 & \dots & \dots & \dots & 0 \\ \vdots & & & & & & \vdots \\ 0 & 0 & 0 & 0 & \dots & \dots & \boldsymbol{\alpha} \\ 0 & 0 & 0 & 0 & \dots & \dots & \boldsymbol{-\alpha} \end{bmatrix}$$

Response Surface Methodology

$$d = \begin{bmatrix} F \\ C \\ E \end{bmatrix}$$

Response Surface Methodology

Example parameters:

- Population Size: [5, 99]
- Rank Bias: [1.01, 1.99]
- Number of Mutations: [1, 10]
- Crossover Rate: [0.01, 0.99]

0 = (max+min)/2

1 = min + (x+1)/(2 x)*(max-min)

-1 = min + (x-1)/(2 x)*(max-min)

e.g. x = 3.13

	-α	-1	0	1	α
Population Size	5	37	52	67	99
Rank Bias	1.01	1.34	1.5	1.65	1.99
Mutations	1	4	5	7	10
Crossover Rate	0.01	0.34	0.5	0.65	0.99

Response Surface Methodology

$$f(W_\rho) = \beta_0 + \sum_{i=1}^{n} \beta_i w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \beta_{ij} w_i w_j$$

Andrea Arcuri and Gordon Fraser. "Parameter tuning or default values? An empirical investigation in search-based software engineering." *Empirical Software Engineering* 18.3 (2013): 594-623.

Parameter Tuning

- A general drawback of the parameter tuning approach, regardless of how the parameters are tuned, is based on the observation that a run of an EA is an intrinsically dynamic, adaptive process.
- The use of rigid parameters that do not change their values is thus in contrast to this spirit.
- Intuitively, different values of parameters might be optimal at different stages of the evolutionary process:
  - E.g. large mutation steps can be good in the early generations helping the exploration of the search space and small mutation steps might be needed in the late generations to help fine tuning the suboptimal chromosomes.
- This implies that the use of static parameters itself can lead to inferior algorithm performance.

Parameter Control

Deterministic Parameter Control

- Value of a strategy parameter is altered by some deterministic rule.
- This rule modifies the strategy parameter deterministically without using any feedback from the search.
- Usually, a time-varying schedule is used, i.e., the rule will be used when a set number of generations have elapsed since the last time the rule was activated

$$P_m(t) = \left(2 + \frac{n-2}{T-1} \cdot t\right)^{-1}$$

- T is the maximum number of generations
- t is the current generation number
- n is the size of the individual
- p0 = 1/2; pT-1 = 1/n

Adaptive Parameter Control

- Feedback from the search is used to determine the direction and/or magnitude of the change to the strategy parameter.
- Example: Progressive Rate Genetic Algorithm (PGRA)
- Adapts the mutation rate and crossover rate during the run of the genetic algorithm
- By calculating the average increase in fitness provided by both mutation and crossover, it is possible to find which function provides the greatest contribution to the fitness of the general population.
- p_m and p_c are mutation rate and crossover rate respectively. θ_1 and θ_2 are predefined constants, e.g. 0.01
- MP and CP are the average mutation and crossover performance
- Initial values: 0.5

$$
\begin{gathered}
p_c = p_c + \theta_1 \text{ if } \widetilde{CP} > \widetilde{MP}, \\
p_c = p_c - \theta_1 \text{ if } \widetilde{CP} < \widetilde{MP}, \\
\text{and} \\
p_m = p_m + \theta_2 \text{ if } \widetilde{CP} < \widetilde{MP}, \\
p_m = p_m - \theta_2 \text{ if } \widetilde{CP} > \widetilde{MP},
\end{gathered}
$$

Self-Adaptive Parameter Control

- Example: Rechenberg's 1/5 Rule
- The ratio of successful mutations to all mutations should be 1/5, hence if the ratio is greater than 1/5 then the step size should be increased, and if the ratio is less than 1/5, the step size should be decreased
- p_s is the relative frequency of successful mutations, measured over some number of generations and 0.817 <= c <= 1

$$
\begin{aligned}
&\textbf{if } (t \bmod n = 0) \textbf{ then} \\
&\qquad \sigma(t) := \begin{cases} \sigma(t-n)/c, & \textbf{if } p_s > 1/5 \\ \sigma(t-n) \cdot c, & \textbf{if } p_s < 1/5 \\ \sigma(t-n), & \textbf{if } p_s = 1/5 \end{cases} \\
&\textbf{else} \\
&\qquad \sigma(t) := \sigma(t-1); \\
&\textbf{fi}
\end{aligned}
$$

Self-Adaptive Parameter Control

- Each individual is given its own mutation and crossover rate,
- This value is mutated once per iteration if the individual is mutated.
- The mutation of these rates is carried out using Gaussian perturbation with a small standard deviation, e.g., 0.05.
- When deciding to apply mutation to an individual, the mutation rate defined by that individual is used rather than a global parameter.
- When applying crossover to two individuals, the average value of their two crossover rates is used as common crossover rate.
- In doing this, individuals which have a more suitable mutation and crossover rate for the current iteration are more likely to result in fitter offspring than others that do not, and are therefore more likely to survive.

Hyper Heuristics

- Ideally:

```
if (problemType(P) == p1 )
  apply(heuristic1, P);
else if (problemType(P) == p2)
  apply(heuristic2, P);
else ...
```

- We don't necessarily know all problem types, and the best heuristics to apply on each of them.

Hyper Heuristics

- There might not be a single optimal heuristic for a given problem type:
  - The strength of a heuristic often lies in its ability to make some good decisions *on the route* to fabricating an excellent solution.
- Idea of hyper heuristics is to apply different heuristics to different parts or phases of the solution process.
- A hyper-heuristic is a search method for selecting or generating heuristics to solve computational search problems.

Hyper Heuristics

1. Start with a set H of heuristic ingredients, each of which is applicable to a problem state and transforms it to a new problem state.

2. Let the initial problem state be S_0

3. If the problem state is S_i then find the ingredient that is in some sense most suitable for transforming that state. Apply it, to get a new state of the problem S_{i+1}

4. If the problem is solved, stop. Otherwise go to 3.

Heuristic Selection

- Methodologies for choosing or selecting existing heuristics
- Example:
  - Given different local search algorithms (low-level heuristics) for a problem (hill climbing, simulated annealing, tabu search, …)
  - Use tabu search to select a low-level heuristic with the highest rank not in the tabu list and apply it for one step
  - A single application of heuristic h is defined to be k iterations of h

Example: Combinatorial Interaction Testing

- Jia, Y., Cohen, M. B., Harman, M., & Petke, J. Learning combinatorial interaction test generation strategies using hyperheuristic search. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 540-550). IEEE Press, 2015

Example: Combinatorial Interaction Testing