EMPIRICALLY IDENTIFYING THE BEST GENETIC ALGORITHM FOR COVERING ARRAY GENERATION

Liang Yalan[1], Changhai Nie[1], Jonathan M. Kauffman[2], Gregory M. Kapfhammer[2], Hareton Leung[3]

[1]Department of Computer Science and Technology, Nanjing University
[2]Department of Computer Science, Allegheny College
[3]Department of Computing, Hong Kong Polytechnic University

3rd International Symposium on Search Based Software Engineering
Szeged, Hungary
September 10-12, 2011

Combinatorial Testing

Modern software systems are highly configurable and involve many interacting parameters

Combinatorial testing is a widely used and practical technique for detecting failures caused by the parameter interactions

One of the key challenges in combinatorial testing is covering array generation, which is an noteworthy area of research

From Kuhn et al., 70% of failures can be detected by 2-way interactions of the software system's parameters

Covering arrays can save testing time while still detecting many important software faults

Combinatorial Testing

Modern software systems are highly configurable and involve many interacting parameters

Combinatorial testing is a widely used and practical technique for detecting failures caused by the parameter interactions

One of the key challenges in combinatorial testing is covering array generation, which is an noteworthy area of research

From Kuhn et al., 70% of failures can be detected by 2-way interactions of the software system's parameters

Covering arrays can save testing time while still detecting many important software faults

Combinatorial Testing

- Modern software systems are highly configurable and involve many interacting parameters
- Combinatorial testing is a widely used and practical technique for detecting failures caused by the parameter interactions
- One of the key challenges in combinatorial testing is covering array generation, which is an noteworthy area of research
- From Kuhn et al., 70% of failures can be detected by 2-way interactions of the software system's parameters
- Covering arrays can save testing time while still detecting many important software faults

Combinatorial Testing

Modern software systems are highly configurable and involve many interacting parameters

Combinatorial testing is a widely used and practical technique for detecting failures caused by the parameter interactions

One of the key challenges in combinatorial testing is covering array generation, which is an noteworthy area of research

From Kuhn et al., 70% of failures can be detected by 2-way interactions of the software system's parameters

Covering arrays can save testing time while still detecting many important software faults

Combinatorial Testing

Modern software systems are highly configurable and involve many interacting parameters

Combinatorial testing is a widely used and practical technique for detecting failures caused by the parameter interactions

One of the key challenges in combinatorial testing is covering array generation, which is an noteworthy area of research

From Kuhn et al., 70% of failures can be detected by 2-way interactions of the software system's parameters

Covering arrays can save testing time while still detecting many important software faults

2-way Covering Arrays

Suppose there are 4 parameters (pa_1, pa_2, pa_3, and pa_4) in a system under test (SUT), each with 3 values (0, 1, 2)

If we want to cover all 54 pair-wise interactions between every 2 parameters in the SUT, then only 9 test cases are needed

What is the most efficient and effective method for **generating** covering arrays?

Table 1. Covering array of the SUT.

pa_1	pa_2	pa_3	pa_4
0	0	0	0
1	0	2	1
2	1	2	0
2	0	1	2
1	1	0	2
0	1	1	1
2	2	0	1
1	2	1	0
0	2	2	2

2-way Covering Arrays

Suppose there are 4 parameters (*pa_1*, *pa_2*, *pa_3*, and *pa_4*) in a system under test (SUT), each with 3 values (0, 1, 2)

If we want to cover all 54 pair-wise interactions between every 2 parameters in the SUT, then only 9 test cases are needed

What is the most efficient and effective method for **generating** covering arrays?

Table 1. Covering array of the SUT.

pa_1	pa_2	pa_3	pa_4
0	0	0	0
1	0	2	1
2	1	2	0
2	0	1	2
1	1	0	2
0	1	1	1
2	2	0	1
1	2	1	0
0	2	2	2

Covering Array Generation

Mathematical and Greedy Methods

- OFOT: One Factor One Time Method
- AETG: Automatic Efficient Tests Generator

Evolutionary Search Techniques

- Particle swarm optimization
- Simulated annealing
- Ant colony optimization

Covering Array Generation

Mathematical and Greedy Methods

- OFOT: One Factor One Time Method
- AETG: Automatic Efficient Tests Generator

Evolutionary Search Techniques

- Particle swarm optimization
- Simulated annealing
- Ant colony optimization

Covering Array Generation

Mathematical and Greedy Methods

- OFOT: One Factor One Time Method
- AETG: Automatic Efficient Tests Generator

Evolutionary Search Techniques

- Particle swarm optimization
- Simulated annealing
- Ant colony optimization

This paper studies and improves genetic algorithms for covering array generation

Genetic Algorithm Phases

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

Genetic algorithms solve complex problems

Genetic Algorithm Phases

1 Initialize
- Accept input parameters
- Generate population

2 Fitness
- Higher is better
- Pair-wise interactions

3 Crossover
- Create new individuals
- Combine two parents

4 Mutation
- Modify individuals
- Increase diversity

5 Selection
- Choose individuals
- Many variants

Genetic algorithms are hard to configure

Genetic Algorithm Parameters

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

System under test (SUT) description (e.g., 3^{13})

Genetic Algorithm Parameters

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

Number of uncovered pair-wise interactions

Genetic Algorithm Parameters

1

Initialize
- Accept input parameters
- Generate population

2

Fitness
- Higher is better
- Pair-wise interactions

3

Crossover
- Create new individuals
- Combine two parents

4

Mutation
- Modify individuals
- Increase diversity

5

Selection
- Choose individuals
- Many variants

P_c controls the probability of crossover

Genetic Algorithm Parameters

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

If P_c is too high, then break good individuals

Genetic Algorithm Parameters

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

If P_c is too low, then miss good solutions

Genetic Algorithm Parameters

1. **Initialize**
   - Accept input parameters
   - Generate population

2. **Fitness**
   - Higher is better
   - Pair-wise interactions

3. **Crossover**
   - Create new individuals
   - Combine two parents

4. **Mutation**
   - Modify individuals
   - Increase diversity

5. **Selection**
   - Choose individuals
   - Many variants

P_m controls the probability of mutation

Genetic Algorithm Parameters

1

Initialize
- Accept input parameters
- Generate population

2

Fitness
- Higher is better
- Pair-wise interactions

3

Crossover
- Create new individuals
- Combine two parents

4

Mutation
- Modify individuals
- Increase diversity

5

Selection
- Choose individuals
- Many variants

If P_m is too small, then cannot escape minima

Genetic Algorithm Parameters

1. **Initialize**
   - Accept input parameters
   - Generate population
2. **Fitness**
   - Higher is better
   - Pair-wise interactions
3. **Crossover**
   - Create new individuals
   - Combine two parents
4. **Mutation**
   - Modify individuals
   - Increase diversity
5. **Selection**
   - Choose individuals
   - Many variants

If P_m is too large, then degrade into random

Genetic Algorithm Parameters

1. Initialize
- Accept input parameters
- Generate population

2. Fitness
- Higher is better
- Pair-wise interactions

3. Crossover
- Create new individuals
- Combine two parents

4. Mutation
- Modify individuals
- Increase diversity

5. Selection
- Choose individuals
- Many variants

Standard GA: Select the superior individuals

Genetic Algorithm Parameters

1

Initialize
- Accept input parameters
- Generate population

2

Fitness
- Higher is better
- Pair-wise interactions

3

Crossover
- Create new individuals
- Combine two parents

4

Mutation
- Modify individuals
- Increase diversity

5

Selection
- Choose individuals
- Many variants

GA-: Select the inferior individuals

Genetic Algorithm Parameters

1 Initialize

- Accept input parameters
- Generate population

2 Fitness

- Higher is better
- Pair-wise interactions

3 Crossover

- Create new individuals
- Combine two parents

4 Mutation

- Modify individuals
- Increase diversity

5 Selection

- Choose individuals
- Many variants

GAr: Randomly select the individuals

Genetic Algorithm Parameters

1 Initialize
- Accept input parameters
- Generate population

2 Fitness
- Higher is better
- Pair-wise interactions

3 Crossover
- Create new individuals
- Combine two parents

4 Mutation
- Modify individuals
- Increase diversity

5 Selection
- Choose individuals
- Many variants

GA climb: Use elitism to keep best individual

Genetic Algorithm Parameters

1

Initialize

- Accept input parameters
- Generate population

2

Fitness

- Higher is better
- Pair-wise interactions

3

Crossover

- Create new individuals
- Combine two parents

4

Mutation

- Modify individuals
- Increase diversity

5

Selection

- Choose individuals
- Many variants

GA, GA-, GAr, GA climb, GA- climb, GAr climb

Experimental Design

- Population Size
- Number of Generations
- Crossover Probability
- Mutation Probability

Experimental Design

Population Size	• 100, 2100, • 4100, 6100
Number of Generations	
Crossover Probability	
Mutation Probability	

Experimental Design

Population Size	
Number of Generations	• 100, 600, 1100
Crossover Probability	
Mutation Probability	

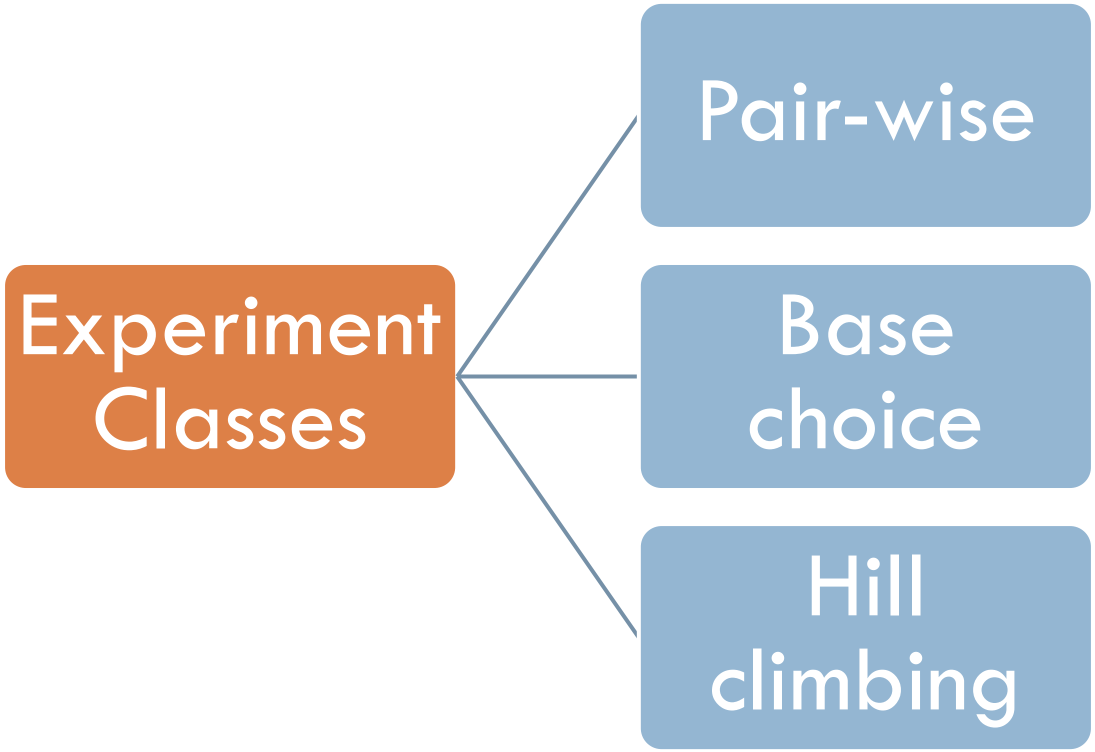
Experimental Design

Population Size	
Number of Generations	
Crossover Probability	• 0.2, 0.4, 0.6, 0.8, 1.0
Mutation Probability	

Experimental Design

Population Size	
Number of Generations	
Crossover Probability	
Mutation Probability	• 0.2, 0.4, 0.6, 0.8, 1.0

Experimental Design



Is there an improved configuration of genetic algorithm for a particular pair-wise SUT?

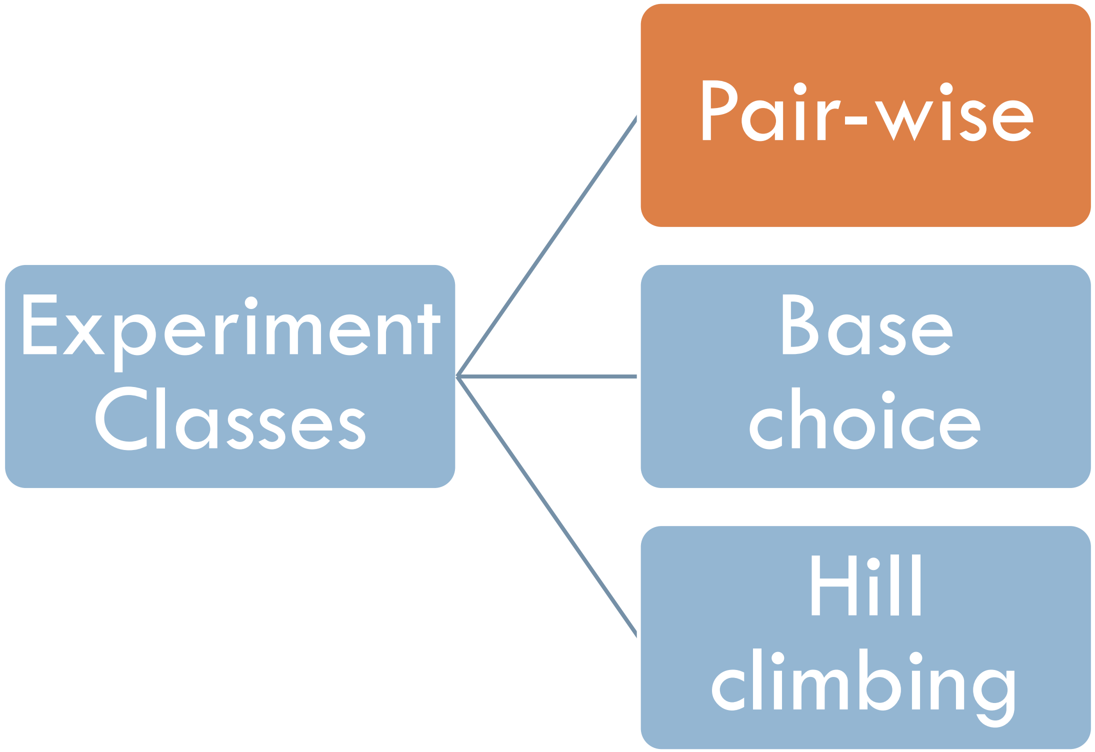
Experimental Design

Experiment Classes

- Pair-wise
- Base choice
- Hill climbing

Is there a common improved configuration for all pair-wise SUTs?

Experimental Design



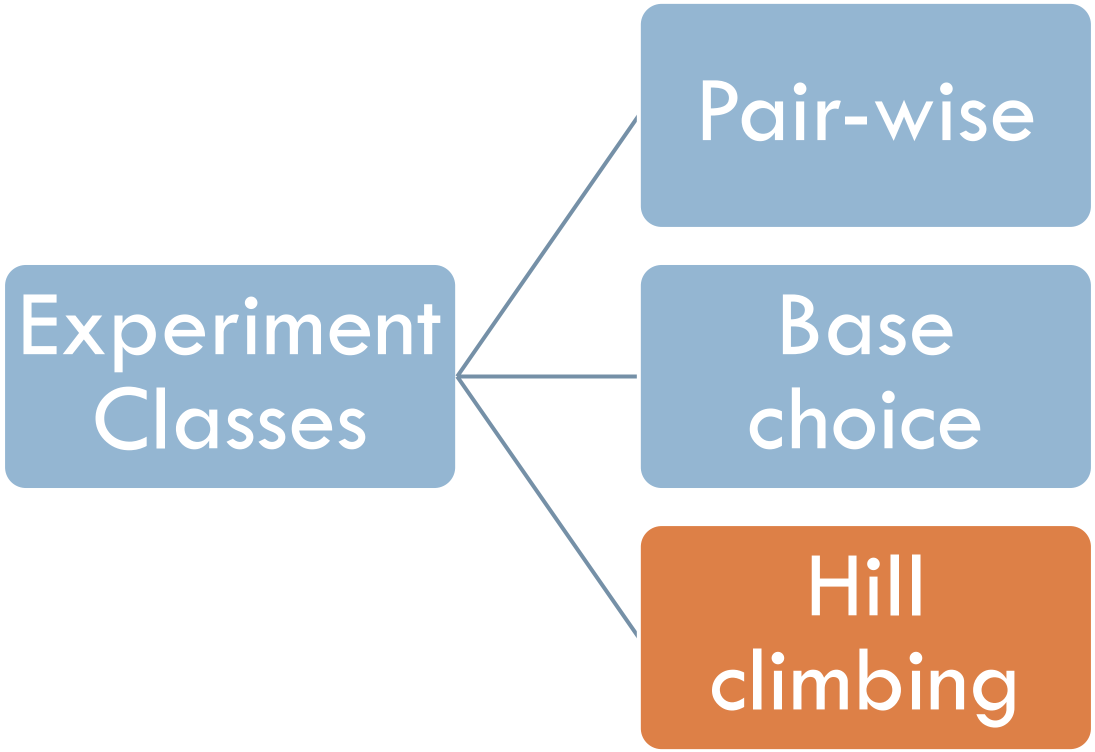
Produce a 2-way covering array with 34 configurations and input into the next phases

Experimental Design

- Experiment Classes
  - Pair-wise
  - Base choice
  - Hill climbing

Create configurations by changing the value of one parameter and not modifying others

Experimental Design



Iteratively refine the configurations in order to find the best one for each SUT

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time	SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^{6}6^{7}5^{6}$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^{2}7^{2}6^{2}5^{2}$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^{1}5^{1}4^{6}3^{8}2^{3}$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^{4}	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^{1}3^{8}2^{2}$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

For the chosen SUTs, there is no single genetic algorithm configuration that is the best

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	VGA	m	G	P_c	P_m	CA Size	Run Time	SUT	VGA	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^{6}6^{7}5^{6}$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^{2}7^{2}6^{2}5^{2}$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^{1}5^{1}4^{6}3^{8}2^{3}$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^{4}	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^{1}3^{8}2^{2}$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

Different values for the effectiveness of the genetic algorithm (e.g., CA size)

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time	SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

Different values for the efficiency of the genetic algorithm (e.g., run time)

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time	SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^{6}6^{7}5^{6}$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^{2}7^{2}6^{2}5^{2}$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^{1}5^{1}4^{6}3^{8}2^{3}$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^{4}	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^{1}3^{8}2^{2}$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

The VGAs of all the improved configurations all use a climbing genetic algorithm

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time	SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

GA- and GAr yield the best configuration for CA generation in 10 out of 15 SUTs

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time	SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

For all SUTs, a lengthier evolutionary process improves CA generation

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	VGA	m	G	P_c	P_m	CA Size	Run Time	SUT	VGA	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

In 13 out of 15 SUTS, creating fewer mutated individuals leads to better CAs

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time	SUT	*VGA*	m	G	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

There is no common best value of P_c or m for the chosen SUTs

Experimental Results

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time	SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

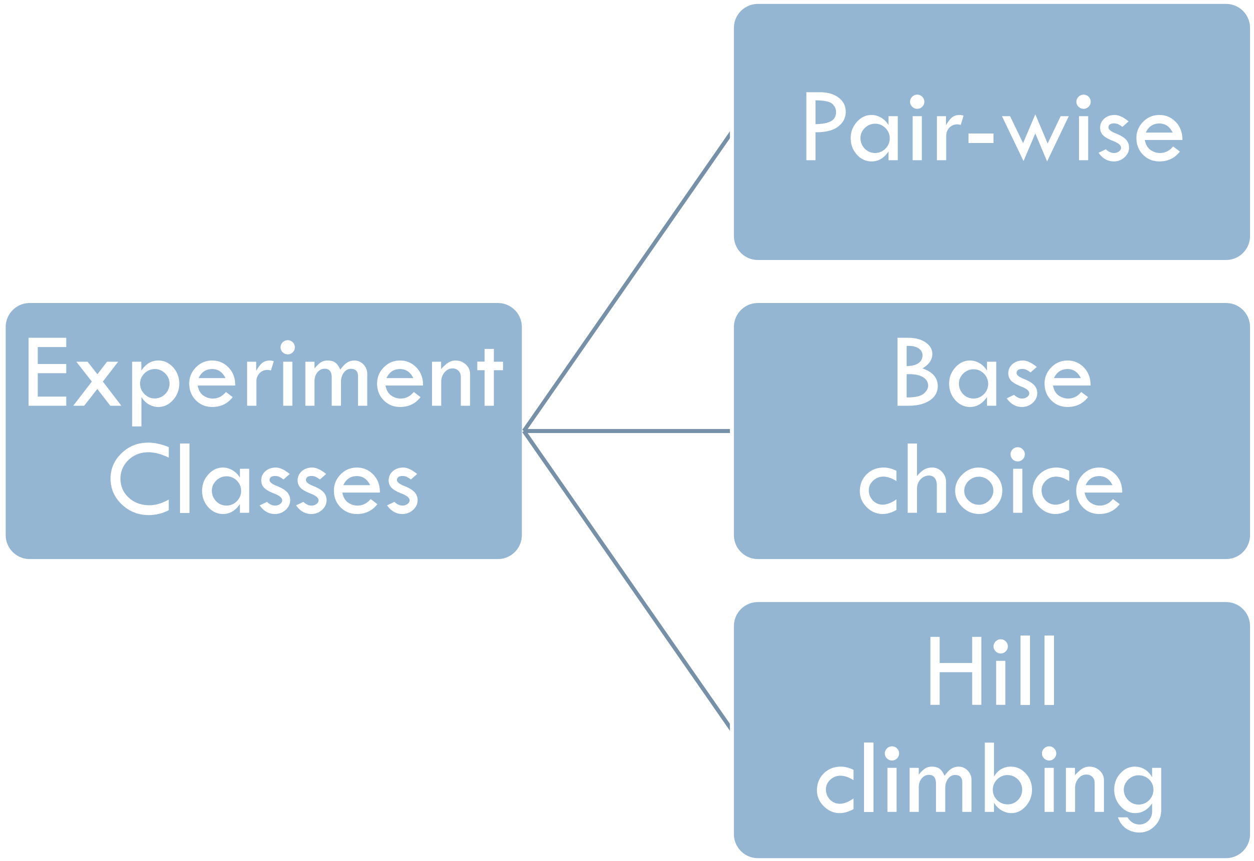
Please see the paper for additional insights concerning the experimental results

Conclusions

1. **Initialize**
   - Accept input parameters
   - Generate population
2. **Fitness**
   - Higher is better
   - Pair-wise interactions
3. **Crossover**
   - Create new individuals
   - Combine two parents
4. **Mutation**
   - Modify individuals
   - Increase diversity
5. **Selection**
   - Choose individuals
   - Many variants

Genetic algorithms for covering array generation

Conclusions



Systematic study on the impact of GA parameters

Conclusions

Counter-intuitive selection strategies

+

Elitist selection methods

=

Efficient and effective genetic algorithm for covering array generation

Fundamental insights into the genetic algorithm

Conclusions

Counter-intuitive selection strategies

+

Elitist selection methods

=

Efficient and effective genetic algorithm for covering array generation

Fundamental insights into the genetic algorithm

Conclusions

Counter-intuitive selection strategies

+

Elitist selection methods

=

Efficient and effective genetic algorithm for covering array generation

Fundamental insights into the genetic algorithm

Conclusions

Counter-intuitive selection strategies

+

Elitist selection methods

=

Efficient and effective genetic algorithm for covering array generation

Fundamental insights into the genetic algorithm

Conclusions

Counter-intuitive selection strategies

+

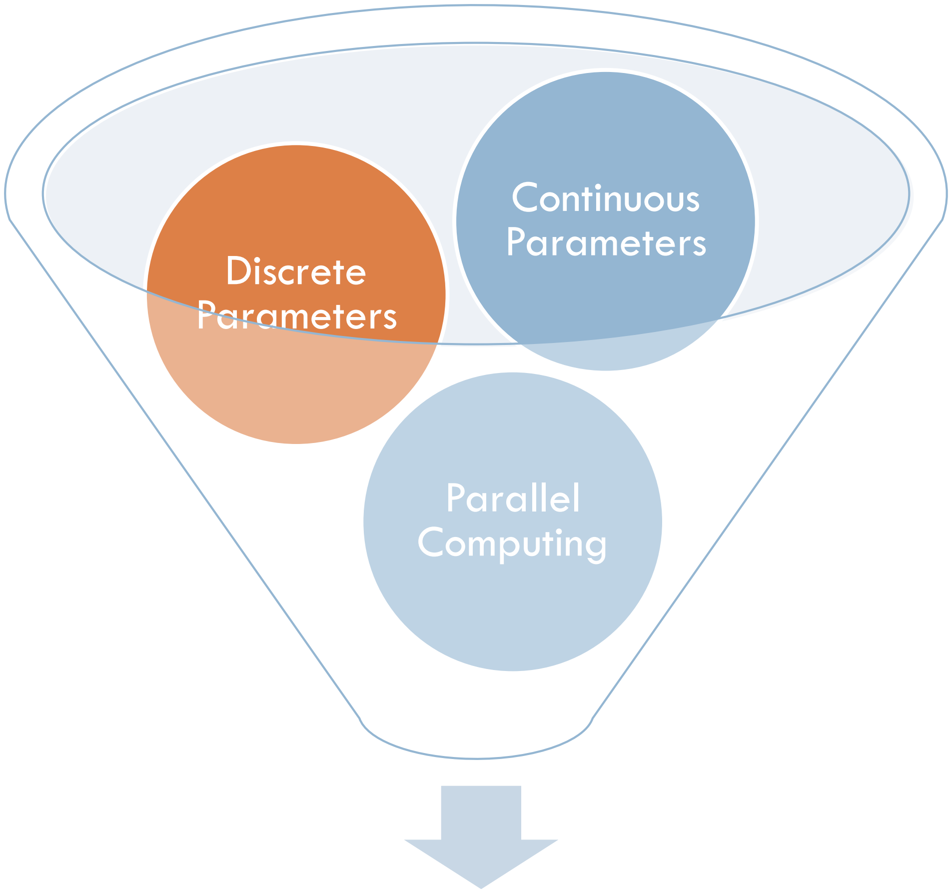
Elitist selection methods

=

Efficient and effective genetic algorithm for covering array generation

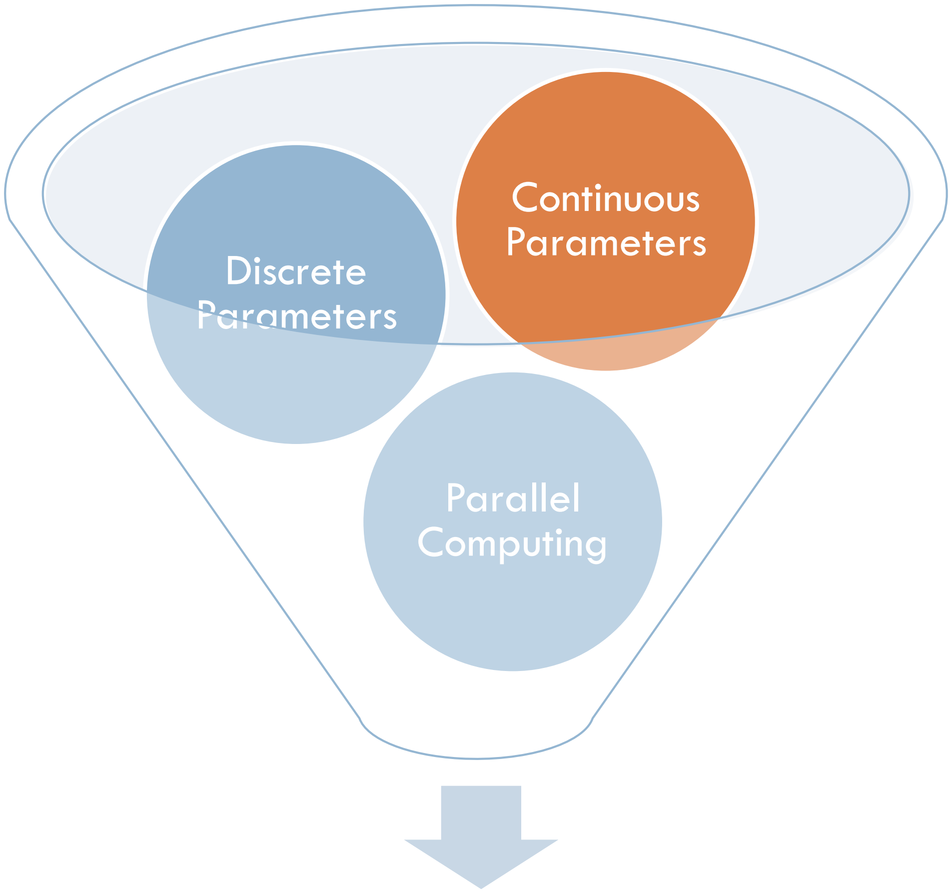
Fundamental insights into the genetic algorithm

Future Work



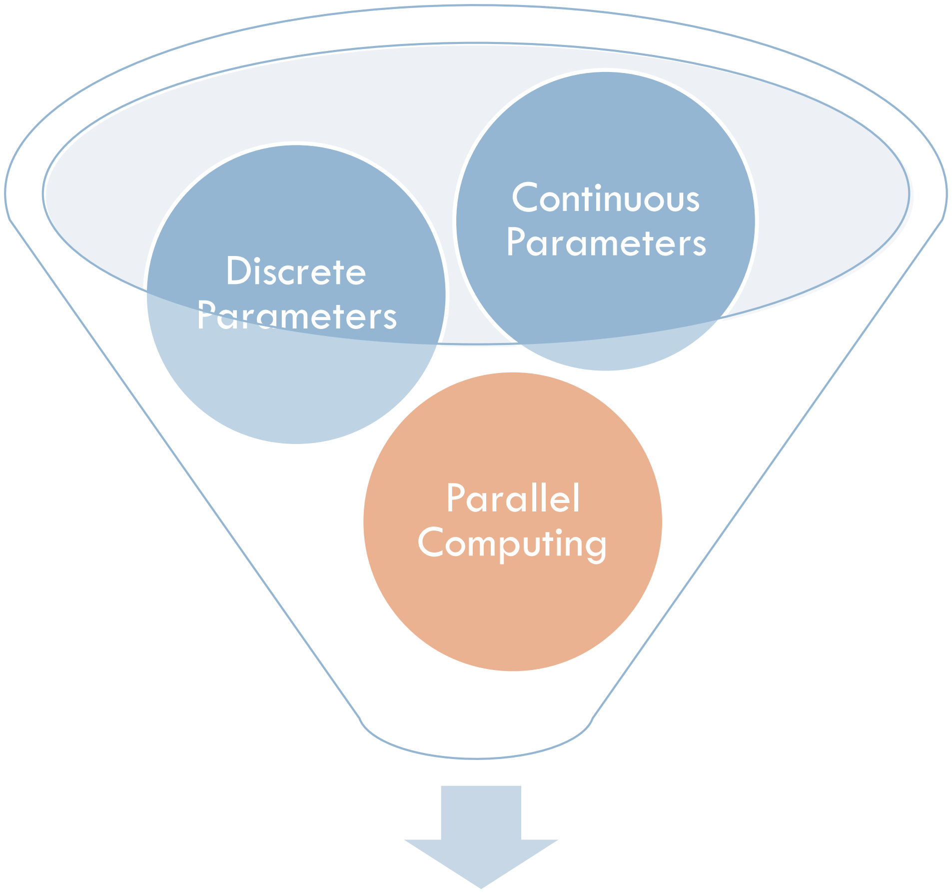
Improved Understanding of Parameters

Future Work



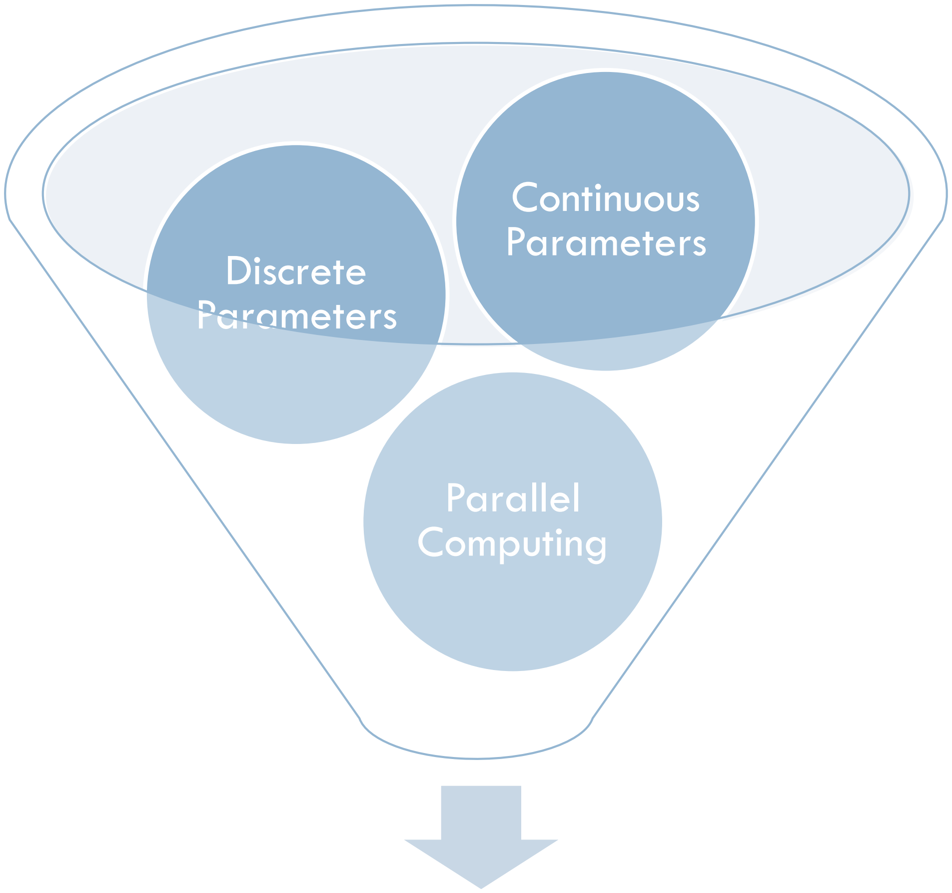
Improved Understanding of Parameters

Future Work



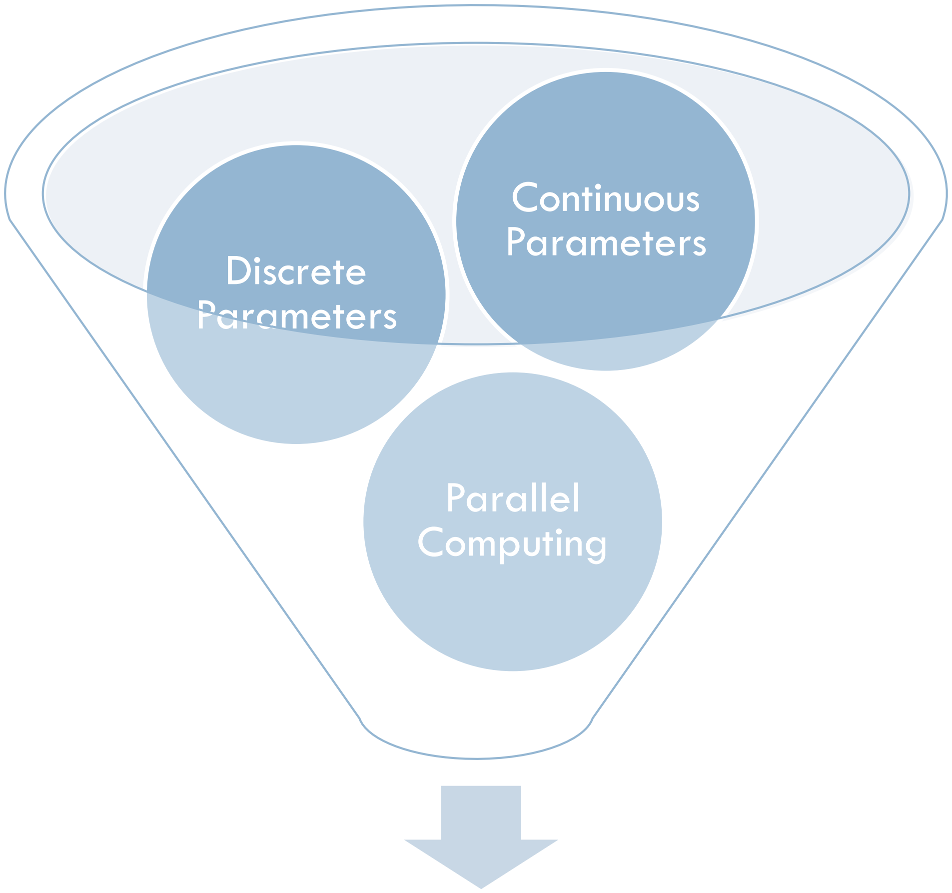
Improved Understanding of Parameters

Future Work



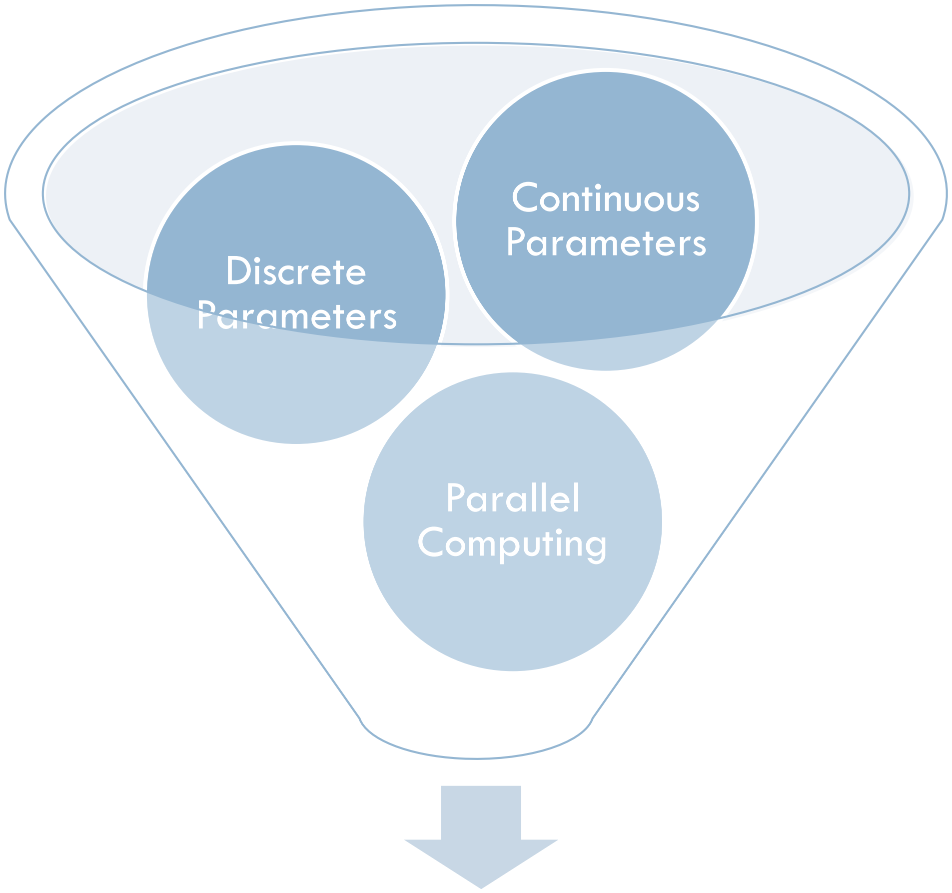
Improved Understanding of Parameters

Future Work



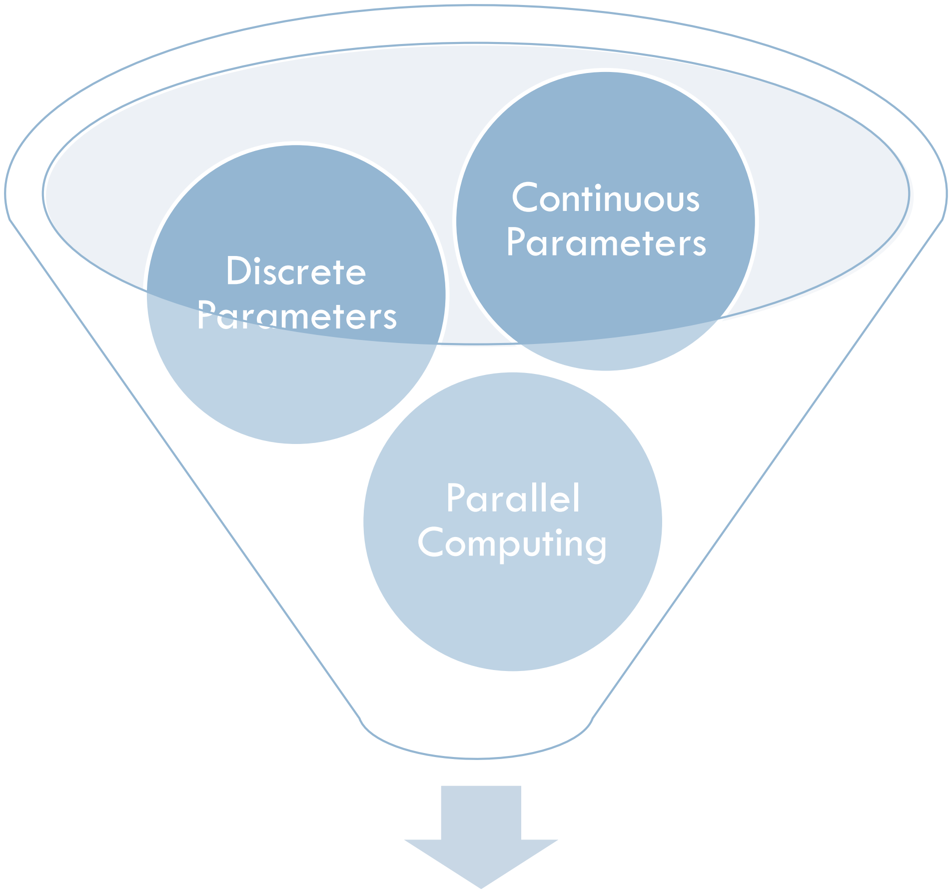
Efficient and Effective Genetic Algorithms

Future Work



Enhanced Covering Array Generators

Future Work



Better Tested and Higher Quality Software

EMPIRICALLY IDENTIFYING THE BEST GENETIC ALGORITHM FOR COVERING ARRAY GENERATION

Liang Yalan, Changhai Nie, Jonathan M. Kauffman,
Gregory M. Kapfhammer, Hareton Leung

3rd International Symposium on Search Based Software Engineering
Szeged, Hungary
September 10-12, 2011

QUESTIONS OR COMMENTS?

Thank you for your attention!