A Germinal Centre Artificial Immune System for Software Test Suite Reduction

Lukas Rosenbauer[1], Anthony Stein[2] and Jörg Hähner[3]

[1] BSH Home Appliances, Im Gewerbepark B35, Regensburg, Germany
[2] Artificial Intelligence in Agricultural Engineering, University of Hohenheim, Garbenstr. 9, Stuttgart, Germany
[3] Organic Computing Group, University of Augsburg, Eichleitnerstr. 30, Augsburg, Germany
lukas.rosenbauer@bshg.com

Abstract

Testing is a crucial part in the development of a new product. If too little testing is done, customers might discover previously undetected failures. A common approach to avoid this is to define a test suite that contains at least one test for every requirement. As the execution of manual tests and the implementation of automated ones is timeintensive, it is a profitable goal to reduce the amount of tests during the specification of the test suite whilst still covering all requirements. In this work we provide an artificial immune system to detect redundant tests. Our new approach achieves optimal results for our industrial data sets and further we are able to reduce its runtime and memory usage drastically compared to the existing *germinal centre artificial immune system* (GCAIS).

Introduction

Testing is a timeintensive but nevertheless important part of product development. The verification of new products becomes even more essential as the complexity of software is increasing rapidly. Several studies confirm that the size of the specified test suite has a major impact on the total development cost (Fraser and Wotawa, 2007; Yu et al., 2008; Hsu and Orso, 2009). Thus it is a worthy goal to reduce the size of a test suite whilst maintaining its quality, such as its coverage or its capability to find errors.

Several approaches are already available to reduce the size of the test suite for certain stages of testing. For example Spieker et al. (2018) determine a test suite based on the history of individual tests (e. g. how often did a test fail or how long does its execution take) using a reinforcement learning agent. The agent selects tests that are more likely to fail and its suite has bounded execution time. Gotlieb and Marijan (2014) try to reduce the size of the test suite before any test is executed or implemented. In contrast to Spieker et al. (2018) they maintain the coverage of the original test suite because a testing history is not available yet. Each test covers a set of requirements and their goal is to determine the minimal set of tests that covers all requirements. In mathematics and computer science this problem is known as the *minimum set cover problem* (MSCP) (Williamson and Shmoys, 2011). The MSCP is a NP-hard optimization problem and thus an optimal solution is hard to find within a reasonable amount of time.

During this work we also intend to reduce the size of the test suite during its specification, similar to Gotlieb and Marijan (2014). However, Gotlieb and Marijan (2014) used a branch and bound approach that has worst case exponential runtime. On the other hand evolutionary algorithms tend to be computational lightweights that may not offer optimal solutions but approximations with reasonable quality and especially the MSCP has undergone heavy research from the evolutionary computation community (Li et al., 2009; Yu et al., 2010; Yu et al., 2014; Balaji and Revathi, 2016).

The immune system has been used an inspiration for both computational intelligence and rule based machine learning (Azuaje, 2003). The latter is closely related to *learning classifier systems* (LCS) which are frequently used in organic computing systems. *Organic computing* (OC) seeks to design systems that have self-x properties (Müller-Schloer and Tomforde, 2017) which can be found in LCS and in the immune system. The former has lead to a rather new evolutionary metaheuristic called *germinal centre artificial immune system* (GCAIS) (Joshi et al., 2014). The approach maintains a population that takes an analogy to self-reacting cells that create antibodies to eradicate pathogens. GCAIS has already been successfully applied to the MSCP on Beasley's OR library Joshi (2017) and often had optimal or close to optimal results. However, it turns out that GCAIS has a few downsides that we want to tackle in this paper.

The main contributions of this paper are:

- GCAIS maintains a population of non-dominated elements which is updated every iteration. The corresponding computation is also known as the calculation of the *skyline* (Börzsönyi et al., 2001). Several methods exist to calculate the skyline but they usually have higher than linear cost. We explicitly exploit the structure of GCAIS and the MSCP and provide an approach that has linear cost in terms of the population size and the problem size.

- The size of GCAIS' population may explode. We introduce simple population boundaries and a deletion mechanism similar to *genetic algorithms* (GA) to avoid this issue (Holland, 1992). In our experiments we show that this does not harm GCAIS' capability to find close to optimal solutions. In most cases our adapted version even is able to find an optimal one.

- We show on two industrial data sets that GCAIS can drastical reduce the size of the specified test suite. Further we can observe that the structure of test specifications differs from the more theoretical MSCP instances of Beasley's OR library.

In Section 2 we introduce the MSCP in a formal way, discuss its approximability, and briefly introduce related work. Afterwards we present GCAIS and show how to reduce its runtime and how we keep the population in bounds (Section 3). In Section 4 we perform experiments on two industrial data sets as well as on Beasley's OR library and examine memory usage, runtime and approximation quality. Further future work is discussed in Section 5. We close the paper with a conclusion (Section 6).

Minimum Set Cover Problem

Here we first intend to describe the MSCP in a more formal way. Let n be the number of sets (the test cases) and m be the number of elements (the requirements) to cover. We denote the sets as T_1, T_2,...,T_n. Thus the problem to be solved can be described as follows:

$$\begin{aligned} & min\,|\mathcal{TS}| \\ s.t. & \bigcup_{i\in\mathcal{TS}} T_i = \bigcup_{i=1}^{m} T_i \\ & \mathcal{TS} \subseteq \{1,2,...,m\} \\ & T_i \subseteq \{1,2,...,n\} \end{aligned} \tag{1}$$

Thus we want to determine the minmal number of tests that still cover all requirements. A set of tests is called *test suite* and thus the problem is coined *test suite reduction* if the undelying MSCP instance corresponds to tests (Gotlieb and Marijan, 2014). If redundant tests can already be identified during specification then their implementation as automated ones or their manual execution can be avoided.

From a mathematical perspective the MSCP is one of the more difficult NP-hard problems to solve as its worst case approximation ratio grows logarithmically in terms of the problem size for algorithms with polynomial runtime (Dinur and Steurer, 2014). However, as this result concerns the worst case there is still research ongoing to find a method that performs well on average.

Minotra (2008) gives an overview about genetic algorithms and simulated annealing methods. Balaji and Revathi (2016) designed a particle swarm optimization method, Yu et al. (2014) used chemical reaction optimization, and Ren et al. (2010) developed an algorithm based on ant colony optimization.

There are several pure mathematical approaches for solving the MSCP such as greedy algorithms, integer linear programs and rounding techniques which are guaranteed to converge (Williamson and Shmoys, 2011). Gotlieb and Marijan (2014) designed an algorithm called *FLOWER* that combines a branch and bound approach with flow networks. FLOWER always delivers an optimal solution, but on the other hand may have exponential runtime (depending on the problem instance).

Germinal Center Artificial Immune System

In this section we introduce the base version of GCAIS. Further, we identify its critical parts and show how the corresponding runtime and memory issues are avoided.

Base algorithm

GCAIS is a population-based, randomised search heuristic that is based on the immune system of vertebrates. The heuristic has been influenced by recent insights about germinal centre reaction (Joshi et al., 2014). *Germinal centres* (GC) are regions where the invading *antigen* (Ag) is presented to immune cells. If an invasion occurs, the cells produce *antibodies* (Ab) that try to bind the pathogen and eradicate it. The GCs start to grow and try to find the best Abs. GCs communicate with each other in order to exchange their Abs. The latter can be improved by proliferation, mutation and selection of immune cells.

We encode solutions as binary vectors of length m. If the entry i is one then T_i belongs to the solution and a zero indicates that T_i is not a part of the cover. Furthermore let $|\mathbf{x}|$ be the L1-norm of $\mathbf{x}$ (corresponds to the number of sets used).
The metaheuristic maintains a population of non-dominated solutions and also allows unfeasible solutions. A solution $\mathbf{x}$ is said to dominate another solution $\mathbf{y}$ if one of the following two condition holds:

i) $|\mathbf{x}| \le |\mathbf{y}| \land |\bigcup_{i\in\mathbf{x}} T_i| > |\bigcup_{i\in\mathbf{y}} T_i|$

ii) $|\mathbf{x}| < |\mathbf{y}| \land |\bigcup_{i\in\mathbf{x}} T_i| \ge |\bigcup_{i\in\mathbf{y}} T_i|$

We denote this relation as $\mathbf{x} >_p \mathbf{y}$. This relation is also known as *pareto dominance* (Börzsönyi et al., 2001).

The initial population P consists out of the zero vector $\mathbf{0}$ (no set at all is used). In every iteration the entire population is mutated (flipping individual bits with a probability of $\frac{1}{m}$) and merged with the original one. During the merge step every solution is eliminated that is either dominated by a mutated solution or a solution that is already in the population. This is repeated until a stopping criteria is met. We summarized the method in Algorithm 1.

The for loop (line 4-7) costs $O(|P|m)$ and can easily be parallelized using for example OpenMP or other standard parallelization libraries.

Algorithm 1: Germinal centre artificial immune system (GCAIS).

```
input  : $T_1, T_2, ..., T_n$, $m$
output: a solution
1  P = {$\mathbf{0}$}
2  while stopping criterion is not met do
3      P' = {}
4      for x in P do
5          y = mutate x
6          insert y to P'
7      end
8      Q = P ∪ P'
9      P = {$\mathbf{x} \in Q | \forall \mathbf{y} \in Q : \neg \mathbf{y} >_p \mathbf{x}$}
10 end
11 return best solution of P
```

We decided to go for such a population based approach as usually throughout a project the requirements (and thus the tests) may change (Nurmuliani et al., 2004). With previous approaches this would require a complete recalculation. However, GCAIS allows infeasible solutions in its population and would enable us to translate the previous population to the updated problem. We hope that this self-improving approach might reduce the runtime in the future.

GCAIS is similar to the *global simple evolutionary multi-objective optimiser* (GSEMO) (Giel and Wegener, 2003) which is another population-based approach. It differs from GCAIS as it has several populations and only mutates one solution per iteration and population instead of all. Further it sends a new solution with a probability p to all other populations. GSEMO's populations also consist out of non-dominated solutions.

Skyline

The for loop of Algorithm 1 is not the only costly step of an iteration. The recalculation of P is also computationally intense. In data engineering the calculation of the set of non-dominated solutions is also known as the computation of the *skyline* (Börzsönyi et al., 2001). It is coined skyline as in the two-dimensional case its solutions are "above" the others (see Figure 1). A side effect of this visualization is that it can be used to track the search process of GCAIS or GSEMO during runtime. The advantage is that a series of these plots displays the development of the population (in terms of its diversity) and the convergence behaviour. Note, that in other research areas the skyline is named *pareto-frontier*.

Several algorithms exist in order to calculate the skyline (complexities adapted to this use case):

- *Block nested loop* (BNL) is the straightforward approach that compares each solution $\mathbf{x}$ with all other solutions in order to determine the skyline. It is trivial to see that this costs $O(|P|^2)$ (Börzsönyi et al., 2001).

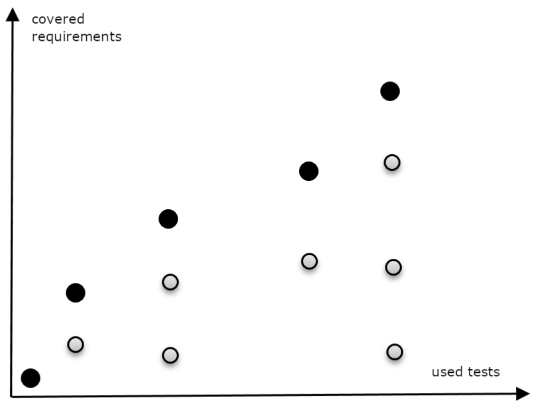


Figure 1: Example of a skyline. The elements of the skyline are marked black. The gray balls represent solutions found throughout the search that have been dominated.

- *Sort filter skyline* (SFS) sorts the considered solutions according to their entropy and exploits that subsequent solutions cannot dominate preceeding ones. The method has a cost of $O(|P|log(|P|))$ (Chomicki et al., 2003).
- *Divide and conquer* (D&Q) approaches split the solutions into chunks and calculate skyline recursively. This also costs $O(|P|log(|P|))$ (Börzsönyi et al., 2001).

There are further approaches especially designed to handle high-dimensional data or memory issues (Endres and Weichmann, 2017; Endres and Kießling, 2015; Endres et al., 2015); however, as this is out of the scope of this paper, we will not further discuss them.

The optimization problem that we try to solve is two-dimensional (covered requirements, used tests), both dimensions are discrete, and P is always a non-dominated set. We exploit these facts to provide a skyline calculation that costs $O(n + |P|)$.

We maintain a look-up table which holds an entry for every $i \in \{0, 1, 2, .., n\}$. The i-th entry holds all solutions of the population that use i tests. It further holds how many requirements are covered by the solutions (they all cover the same number of requirements, otherwise a solution would be dominated). Whenever we consider to insert a mutated solution $\mathbf{x}$ for insertion we check the entry $|\mathbf{x}|$. If the entry covers more elements then $\mathbf{x}$ is not inserted, if it covers the same amount of elements then we append the solution to the entry. If it covers more elements then we overwrite the entry with $\mathbf{x}$ and update the covered elements of the entry. Thus an insertion costs $O(1)$ and the insertion of the mutated population costs $O(|P|)$.

After the insertion of a solution the table may contain dominated solutions as only the entry of its cost is checked. An inserted solution could also dominate entries of higher cost. Thus we also need to introduce a repair method that is called at the end of every iteration of GCAIS. We traverse the table exactly once. We start by saving the index 1 into a variable i and check if the current entry to look at covers more entries. If so, we update i by its index and if not, we delete the entry and proceed. Thus the repair method costs $O(n)$ and the calculation of the skyline $O(n+|P|)$.

We describe our skyline algorithm in Algorithm 2. The variable table denotes the look-up table and table[i].cov is the number of requirements covered by the solutions using i tests. The number of requirements covered and the number of tests used by a solution **x** can easily be retrieved during the creation of the mutated population and thus does not affect the cost of that method. The table should be initialized before the main loop is entered and should be kept throughout the search.

Algorithm 2: Skyline procedure for GCAIS using a look-up table.

```
input : mutated solutions P'
1  // insertion
2  for x in P' do
3      covered_x = requirements covered by x
4      if table has no entry |x| then
5          table[|x|].cov = covered_x
6          set the solutions of table[|x|] to x
7      else if table[|x|].cov == covered_x then
8          append x to the solutions of table[|x|]
9      else if table[|x|].cov < covered_x then
10         table[|x|].cov = covered_x
11         set the solutions of table[|x|] to x
12 end
13 // repair the table
14 i = 1
15 for k in {2,...n} do
16     if table[k].cov ≤ table[i].cov then
17         delete table[k]
18     else if table[k].cov > table[i].cov then
19         i = k
20 end
```

Avoiding huge populations

A difference between GCAIS and for example genetic algorithms is that its population is unbounded. GCAIS keeps the non-dominated solutions it encounters throughout its search. The idea is that a mutated solution based on a non-dominated solution has a higher likelihood to be an optimal or close to optimal solution. However, this has the downside that the population may grow rapidly. For example if all tests cover the same amount of requirements and no requirement is covered by more than one test, then the population even grows exponentially.

In order to avoid an explosion of the population size, we introduce simple population boundaries for each entry of the look-up table. Whenever the capacity of an entry is exceeded then we delete a random solution from the entry to make space for a new one. Unlike the skyline computation, this may change the convergence behaviour of the algorithm which we examine in our experimental evaluation.

There is also another approach to keep GCAIS' population from growing too fast. Joshi et al. (2015) used ε-*dominance* instead of pareto-dominance. For the former, the space is separated in squares of side length ε (for spaces of higher dimensions it is separated into hypercubes). For two solutions from different squares the ε-dominace relation is the same as the pareto-dominance relation. If two solutions **x** and **y** are in the same square then **x** ε-dominates **y** if and only if:

$$|\mathbf{y}|-|\mathbf{x}|+|\bigcup_{i\in\mathbf{x}}T_i|-|\bigcup_{i\in\mathbf{y}}T_i|>0 \qquad (2)$$

If this approach is used then GCAIS keeps a population of non-ε-dominated solutions instead of non-pareto-dominated solutions. ε-dominance is more strict than pareto-dominance and thus it makes it harder for a solution to be inserted. However, this no guarantee that the population does not grow unrestricted.

Both approaches can easily be integrated into Algorithm 2. The boundary check and deletion of a random solution can be incorporated into the insertion part. In order to use ε-dominance we have to extend our repair method by incorporating a check if the table entries are in the same square and deleting ε-dominated entries. This updated version has the same worst case complexity as Algorithm 2. We describe the new repair method in Algorithm 3.

Algorithm 3: Updated repair method if epsilon dominance is used.

```
input : look-up table, ε
1  i = 1
2  for k in {2,...n} do
3      if ⌊(k−i)/ε⌋ == 0 then
4          // both entries are in the same square
5          d = k-i+table[i].cov - table[k].cov
6          if d > 0 then
7              // i dominates k
8              delete table[k]
9          else if d < 0 then
10             // k dominates i
11             delete table[k]
12             i = k
13     else if table[k].cov ≤ table[i].cov then
14         delete table[k]
15     else if table[k].cov > table[i].cov then
16         i = k
17 end
```

Evaluation

In our experiments we want to evaluate how much execution time we save due to our look-up table. Further, we investigate if the introduction of ε-dominance and population boundaries limits the capability of GCAIS to find close to optimal or optimal solutions. To our knowledge the former has yet only been applied to the multiobjective Knapsack problem (Joshi et al., 2015).

In our experiments we first focus on the main goal of this paper: the reduction of the amount of tests. For this we acquired two data sets from BSH Home Appliances which is a german company that develops and produces various home appliances such as ovens or dishwashers. Our data sets are for two different fridge projects. In our experiments we use cleaned versions of the data sets. We removed tests that exclusively cover a single requirement (these tests must be in a test suite that covers all requirements). We call these data sets Fridge-1 and Fridge-2.

As we deem two datasets as too little, we additionally perform evaluations on the scpe instances of Beasley's OR library which is frequently used for benchmarking MSCP algorithms (Balaji and Revathi, 2016; Joshi et al., 2014).

During our evaluation we consider the various variants of GCAIS next to the GSEMO algorithm. We follow the variant of Joshi et al. (2014) which was adapted to the MSCP. We examined several parameterizations for GSEMO and concluded that a population size of 30 and a send probability of $\frac{30}{nm}$ are suitable choices. We also performed a fine tuning of the hyperparameters of the artificial immune systems which we will not discuss here (due to space restrictions). We achieved reasonable results with a population boundary of 200. For ε we consider 0, 5, 10 and 15. The base variant of GCAIS (Algorithm 1) is parameterfree.

We repeat every experiment 100 times. Our implementation[1] is in Python and we used a Dell Precision 3520 for our experiments (Intel i7-6820HQ processor with 4 Cores and an individual clock rate of 2.7 Ghz, 32 GB RAM).

Every algorithm is given a time budget of ten minutes. Further, if there is no improvement in terms of the solutions quality for 100 iterations then we interpret this as convergence.

Quality Criteria

During our experiments we intend to measure the approximation quality, memory usage, and runtime. Thus we introduce the following *key performance indicators* (KPIs):

$$\text{mem\_save}(\text{alg}) = \frac{P(\text{GCAIS\_BASE})}{P(\text{alg})} \tag{3}$$

$$\text{speed\_up}(\text{alg}) = \frac{r(\text{GCAIS\_BASE})}{r(\text{alg})} \tag{4}$$

$$\text{approx\_rate}(\text{alg}) = \frac{o(\text{alg})}{\text{OPT}} \tag{5}$$

where *alg* denotes any considered algorithm (and its corresponding hyperparameters). GCAIS_BASE is the standard variant of GCAIS described in Algorithm 1 and for the calculation of its non-dominated population we use the BNL method. $P(\text{alg})$ is the maximum size of an algorithm's population during a run and $r(\text{alg})$ is its total runtime (until convergence is reached). OPT is the optimal value and $o(\text{alg})$ is the algorithm's output. The optimal values for the considered data sets of Beasley's OR library are known and for the industrial data sets we determined them via brute force.

The mem_save key performance indicator (Equation 3) measures the relative size of an algorithm's population with regards to GCAIS_BASE. We use the standard variant as all other GCAIS variants intend to either bound the population or to increase the likelihood of deleting solutions (e. g. ε-dominance). Thus we have a common baseline for all methods. Further, the population size is the main factor for memory usage of the considered algorithms.

Our speed up KPI (Equation 4) is an analogy to parallel computing. There the speed up is the quotient of a parallel program's runtime and the runtime of the sequential one (Kumar, 2002). Thus it measures how fast the parallel method is compared to the sequential one. Instead we evaluate how much faster an algorithm is compared to the standard GCAIS version.

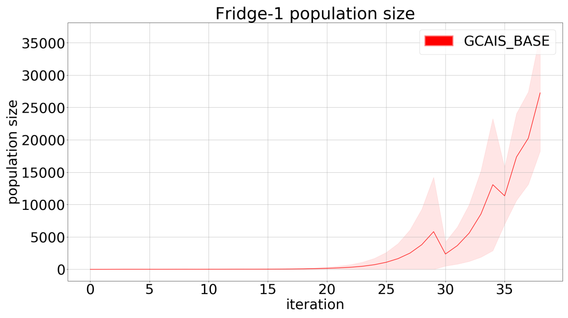
Our third KPI is the approximation rate (Equation 5) which indicates how close the produced solution is to being optimal and a value of one corresponds to an optimal solution.

All three KPIs should be seen in context to each other. For example a brute force search always leads to an optimal solution but will have an exponential runtime and on the other hand an algorithm that just takes all tests has the worst approximation ratio but the best speed up and memory usage. Hence the goal is to find an approach that leads to reasonable values in all three categories.

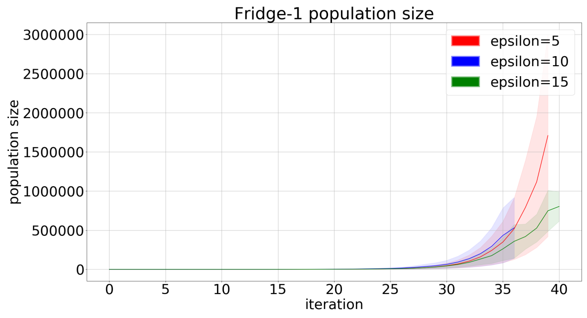
Industrial Datasets

The results of our experiments are displayed in Table 1. Our first dataset (Fridge-1) is rather easy to solve for the considered methods compared to our second one. The epsilon dominance variants, the bounded variant and the base variant of GCAIS always achieve optimal results. Also, GSEMO has close to optimal results. However the combination of a bounded population and epsilon dominance is rather detrimental as these versions produce worse solutions than the version without it. We can see certain differences in the memory usage and the speed up. The bounded version of GCAIS only uses about a tenth of the memory of its base variant and is about fourteen times faster. Yet GESMO is even faster but only achieves close to optimal results and requires more memory.

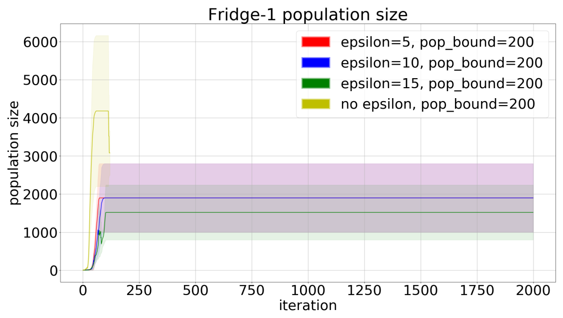
[1] Source code and data are available here: `https://github.com/LagLukas/gcais_test_suite_reduction`



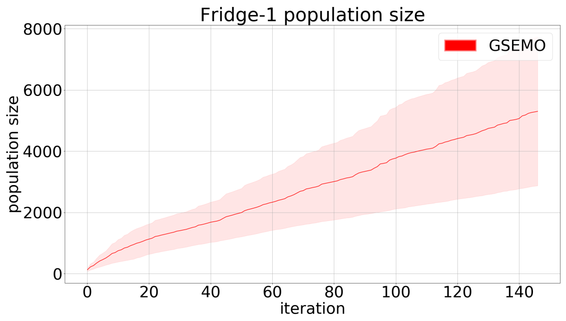
(a) GCAIS base variant



(b) GCAIS with epsilon dominance

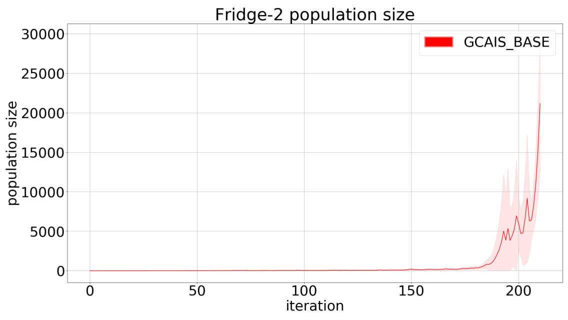


(c) GCAIS with bounded population

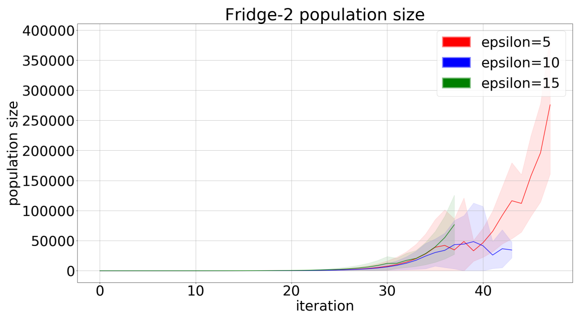


(d) GSEMO

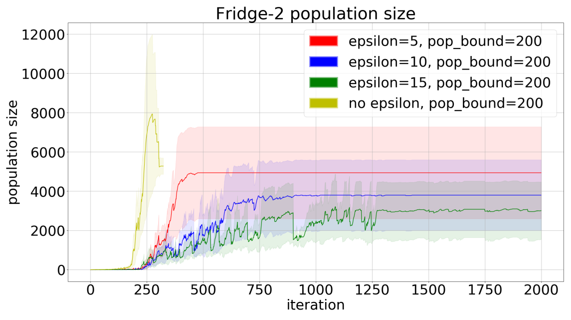
Figure 2: Population sizes $\pm\sigma$ for the Fridge-1 dataset.



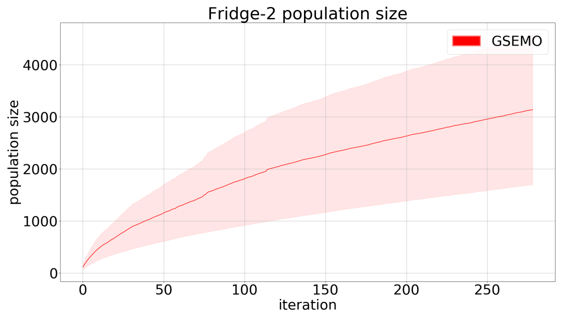
(a) GCAIS base variant



(b) GCAIS with epsilon dominance



(c) GCAIS with bounded population



(d) GSEMO

Figure 3: Population sizes $\pm\sigma$ for the Fridge-2 dataset.

Table 1: KPIs for the experimental results (averaged values ±σ). A - character indicates that the parameter was not used. We marked the best values of algorithms that always found optimal solutions bold. The best values of individual KPIs are marked in italics. The horizontal line separates our approaches to the ones we are comparing to.

Fridge-1	population boundary (per entry)	ε	mem_save	speed_up	approx_rate
GCAIS	200	5	22.77 ± 0.0	1.28 ± 0.19	1.29 ± 0.2
GCAIS	200	10	22.77 ± 0.0	1.54 ± 0.19	1.34 ± 0.17
GCAIS	200	15	*28.47 ± 0.0*	2.11 ± 0.16	1.38 ± 0.16
GCAIS	200	-	**10.35 ± 0.0**	**14.14 ± 0.02**	***1.0 ± 0.0***
GCAIS	-	5	0.03 ± 0.03	0.77 ± 0.16	***1.0 ± 0.0***
GCAIS	-	10	0.03 ± 8.95	0.75 ± 0.16	***1.0 ± 0.0***
GCAIS	-	15	0.03 ± 16.27	0.73 ± 0.15	***1.0 ± 0.0***
GCAIS_BASE	-	-	1.0	1.0	***1.0 ± 0.0***
GSEMO	-	-	1.66 ± 0.78	*16.81 ± 0.13*	1.03 ± 0.05

Fridge-2	population boundary (per entry)	ε	mem_save	speed_up	approx_rate
GCAIS	200	5	8.49 ± 0.0	0.32 ± 0.38	3.62 ± 0.46
GCAIS	200	10	10.9 ± 0.0	0.51 ± 0.48	3.82 ± 0.47
GCAIS	200	15	*12.49 ± 0.0*	0.84 ± 0.33	3.96 ± 0.3
GCAIS	200	-	**5.11 ± 0.01**	**3.58 ± 0.08**	***1.0 ± 0.0***
GCAIS	-	5	0.14 ± 3.4	0.88 ± 0.14	2.7 ± 0.13
GCAIS	-	10	0.18 ± 3.14	0.88 ± 0.14	2.66 ± 0.28
GCAIS	-	15	0.23 ± 2.47	0.96 ± 0.23	2.8 ± 0.19
GCAIS_BASE	-	-	1.0	1.0	1.0 ± 0.02
GSEMO	-	-	9.89 ± 0.01	*47.33 ± 0.01*	2.94 ± 0.15

Table 2: Experimental results for Beasley's OR library. The best values are marked bold. Each KPI is displayed ±σ.

KPI	algorithm	scpe1	scpe2	scpe3	scpe 4	scpe5
mem_save	GSEMO	1.62 ± 0.15	1.62 ± 0.13	1.39 ± 0.2	1.1 ± 0.3	1.91 ± 0.15
mem_save	bounded GCAIS	**4.23 ± 0.01**	**4.54 ± 0.01**	**4.68 ± 0.02**	**4.31 ± 0.02**	**4.54 ± 0.01**
speed_up	GSEMO	**2.56 ± 0.15**	**2.22 ± 0.15**	1.69 ± 0.28	1.26 ± 0.41	**3.37 ± 0.15**
speed_up	bounded GCAIS	1.73 ± 0.18	1.36 ± 0.24	**2.56 ± 0.14**	**2.04 ± 0.2**	1.87 ± 0.21
approx rate	GSEMO	1.46 ± 0.16	1.53 ± 0.1	1.49 ± 0.12	1.43 ± 0.14	1.53 ± 0.1
approx rate	bounded GCAIS	**1.06 ± 0.04**	**1.01 ± 0.03**	**1.09 ± 0.03**	**1.04 ± 0.05**	**1.08 ± 0.04**
approx rate	GCAIS_BASE	1.11 ± 0.04	1.08 ± 0.06	1.11 ± 0.05	1.07 ± 0.05	1.14 ± 0.06

Our other dataset (Fridge-2) is tougher to solve for the considered metaheuristics as only the bounded GCAIS variant without epsilon dominance always achieves optimal results. Once more the results show that this variant can drastically cut down memory usage and runtime. GSEMO has an even shorter runtime and memory usage but on the other hand only achieves approximation rates of about three. These differences between GSEMO and our bounded version of GCAIS are due to GSEMO's convergence to an inferior solution. GCAIS does not get stuck (as it always finds optimal solutions) and thus the population continues to grow as does the runtime.

On both datasets we could observe that in our case the epsilon dominance has a detrimental effect on the population size and therefore on the runtime. Combined with a bounded population these effects disappear but the method is unable to find optimal solutions. Hence we could not observe the same positive effects of the usage of epsilon dominance as Joshi et al. (2015) did for the Knapsack problem. The pure bounded version always achieved optimal results and achieved high values in our other KPIs as well.

Most of the observed differences can be explained by taking a look at the population growth and size which we visualized in Figures 2 and 3. The base variant and pure epsilon dominance variants of GCAIS show an exponential growth for Fridge-1 and on the other dataset we can observe a similar observation for epsilon equal to 5. For the other two variants the runtime ran out and thus we do not fully see an exponential growth. GSEMO's population size grows linearly for Fridge-1 and more or less logarithmically for Fridge-2. Our bounded GCAIS version has, as expected, a constant population size (after several iterations). The jumps in the graphs are due to newly found solutions that dominate other solutions in the population which get deleted. These different growths are one of the causes for the differences in speed up as all of the considered algorithms have a runtime which depends on this magnitude.

The growths in terms of population size can be explained by taking a look at the structure of our datasets and the problem itself. Our test specifications consist out of test cases that have similar sizes and only slightly overlap in terms of the requirements which they cover. Also, the two dimensions (covered requirements and used tests) are integers and there are only limited valid values. Thus there can be many solutions that cover the same amount of requirements and use the same tests and it is hard to find solutions which dominate large portions of the population. If the tests would highly differ in their size then it would be easier to find dominating solutions which would lead to smaller populations.

Next to our visual evaluation and the discussion of the raw values of Table 1, we perform additional statistical testing to confirm our observations. We test each KPI and each dataset individually. Our null hypothesis is that the algorithms do not differ on one dataset regarding one KPI. This can be verified using a Friedman test. On all different null hypotheses we observed p-values below 10^{-10} which we regard as significant. Thus we conclude that the algorithms differ in terms of the KPIs.

Overall we are able to reduce the size of original test suites by over 30 percent.

Beasley's OR Library

Due to the results on the industrial datasets and the spatial restrictions we focus solely on the base variant of GCAIS, the bounded variant without epsilon dominance, and GSEMO during this experiment. We evaluate the scpe1 to scpe5 instances of Beasley's OR library.

We displayed the experimental results in Table 2. Once more the population boundaries for GCAIS lead to a cut down in terms of memory and runtime. Further, they reveal that our adapted version of GCAIS did not lose its capability to find close to optimal or optimal solutions on these more theoretical instances. In all cases our version was even superior to the base variant. However, in three out of five cases GSEMO was once more faster than our bounded version as it once more converged towards a suboptimal solution.

We verified our observations about the bounded GCAIS' superiority in terms of memory usage and approximation quality using one-sided Wilcoxon signed-rank tests. The p-values were below 0.05 which we regard as significant.

Further, on these datasets the population of the base variant of GCAIS does not grow as much as during our evaluation of the industrial datasets. This explains why the memory savings are lower. The smaller populations thus lead to a smaller runtime which unfolds in smaller speed ups for the other algorithms. The reason for the smaller populations is that GCAIS detects dominating solutions more easily, which keep the population in bounds. Hence we think that the problem structure of test specifications differs from these more theoretical MSCP instances.

Future Work

From an engineering perspective we intend to roll out our version of GCAIS in the company. We further want to gather more datasets from different test levels to verify our approach. We take special interest in the evolution of the requirements and tests over the lifetime of a project. Thus we could examine if the reuse of past populations is an advantage.

Our next scientific goal is to apply the GCAIS approach to the *adaptive test case selection problem* (ATCS) (Spieker et al., 2018). Its goal is to find a test suite that maximizes a test metric such as coverage whilst maintaining a test suite that has a bounded duration (for its execution). In the case of coverage, the problem becomes a variant of the weighted MSCP and GCAIS could be applied. In this case the problem landscape varies even more over time as newly written test cases are being added and their duration might change over time (as the software to be tested might be changed). A reuse of GCAIS' population might lead in this case to a self-improving system as it is continuously adapted and optimised towards the new testing environment.

Conclusion

We introduced a test suite reduction problem which is a variant of *minimum set cover problem* (MSCP). Its goal is to find a test suite of minimal size that still covers all requirements. A state of the art approach for the MSCP is the *germinal centre artificial immune systems* (GCAIS) which has been heavily benchmarked on rather theoretical instances.

GCAIS maintains a population of non-dominated solutions whose calculation cost is quadratic. We apply a simple datastructure and an incremental update approach that allows us to reduce the cost to a linear one.

Our experiments revealed that on our test specifications, the populations of the standard variant of GCAIS explode which leads to a high memory consumption and longer runtimes. Thus we adapted GCAIS by applying fixed population capacities. Our improved variant could not only cut down runtime and memory usage compared to the standard variant, it was also able to find optimal or close to optimal solutions on our industrial data as well as on the more theoretical instances of Beasley's OR library.

Acknowledgement

We would like to thank Oliver Banf who helped us acquiring the industrial data.

References

Azuaje, F. (2003). Review of “artificial immune systems: A new computational intelligence approach” by l.n. de castro and j. timmis (eds) springer, london, 2002. *Neural Netw.*, 16(8):1229.

Balaji, S. and Revathi, N. (2016). A new approach for solving set covering problem using jumping particle swarm optimization method. 15(3):503–517.

Börzsönyi, S., Kossmann, D., and Stocker, K. (2001). The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, page 421–430, USA. IEEE Computer Society.

Chomicki, J., Godfrey, P., Gryz, J., and Liang, D. (2003). Skyline with presorting. pages 717– 719.

Dinur, I. and Steurer, D. (2014). Analytical approach to parallel repetition. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC ’14, pages 624–633, New York, NY, USA. ACM.

Endres, M. and Kießling, W. (2015). Parallel skyline computation exploiting the lattice structure. *Journal of Database Management*, 26:18–43.

Endres, M., Roocks, P., and Kießling, W. (2015). Scalagon: An efficient skyline algorithm for all seasons. In *Database Systems for Advanced Applications*, pages 292–308.

Endres, M. and Weichmann, F. (2017). Index structures for preference database queries. In *Flexible Query Answering Systems*, pages 137–149.

Fraser, G. and Wotawa, F. (2007). Redundancy based test-suite reduction. In Dwyer, M. B. and Lopes, A., editors, *Fundamental Approaches to Software Engineering*, pages 291–305, Berlin, Heidelberg. Springer Berlin Heidelberg.

Giel, O. and Wegener, I. (2003). Evolutionary algorithms and the maximum matching problem. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, STACS ’03, page 415–426, Berlin, Heidelberg. Springer-Verlag.

Gotlieb, A. and Marijan, D. (2014). Flower: Optimal test suite reduction as a network maximum flow. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 171–180, New York, NY, USA. ACM.

Holland, J. H. (1992). Genetic algorithms. *Scientific American*, 267(1):66–73.

Hsu, H. and Orso, A. (2009). Mints: A general framework and tool for supporting test-suite minimization. In *2009 IEEE 31st International Conference on Software Engineering*, pages 419–429.

Joshi, A. (2017). The germinal centre artificial immune system. University of Birmingham.

Joshi, A., Rowe, J. E., and Zarges, C. (2014). An immune-inspired algorithm for the set cover problem. In Bartz-Beielstein, T., Branke, J., Filipič, B., and Smith, J., editors, *Parallel Problem Solving from Nature – PPSN XIII*, pages 243–251, Cham. Springer International Publishing.

Joshi, A., Rowe, J. E., and Zarges, C. (2015). Improving the performance of the germinal center artificial immune system using epsilon-dominance: A multi-objective knapsack problem case study. In Ochoa, G. and Chicano, F., editors, *Evolutionary Computation in Combinatorial Optimization*, pages 114–125, Cham. Springer International Publishing.

Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition.

Li, Y., Hu, X., and Zhang, J. (2009). A new genetic algorithm for the set k-cover problem in wireless sensor networks. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, pages 1405–1410.

Minotra, D. (2008). A study of heuristic-algorithms for set-covering problems.

Müller-Schloer, C. and Tomforde, S. (2017). Organic computing – technical systems for survival in the real world. In *Autonomic Systems*.

Nurmuliani, N., Zowghi, D., and Powell, S. (2004). Analysis of requirements volatility during software development life cycle. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 28–37.

Ren, Z.-G., Feng, Z.-R., Ke, L.-J., and Zhang, Z.-J. (2010). New ideas for applying ant colony optimization to the set covering problem. *Computers & Industrial Engineering*, 58(4):774 – 784.

Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. (2018). Reinforcement learning for automatic test case prioritization and selection in continuous integration. *CoRR*, abs/1811.04122.

Williamson, D. and Shmoys, D. (2011). The design of approximation algorithms. *The Design of Approximation Algorithms.*

Yu, J. J. Q., Lam, A. Y. S., and Li, V. O. K. (2014). Chemical reaction optimization for the set covering problem. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 512–519.

Yu, Y., Jones, J. A., and Harrold, M. J. (2008). An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE ’08, page 201–210, New York, NY, USA. Association for Computing Machinery.

Yu, Y., Yao, X., and Zhou, Z.-H. (2010). On the approximation ability of evolutionary optimization with application to minimum set cover. *Artificial Intelligence*, s 180–181.