# Metaheuristics for the Minimum Set Cover Problem: A Comparison

Lukas Rosenbauer[1], Anthony Stein[2], Helena Stegherr[3] 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*

Keywords: Metaheuristics, Combinatorial Optimization, Empirical Study.

Abstract: The minimum set cover problem (MSCP) is one of the first NP-hard optimization problems discovered. Theoretically it has a bad worst case approximation ratio. As the MSCP turns out to appear in several real world problems, various approaches exist where evolutionary algorithms and metaheuristics are utilized in order to achieve good average case results. This work is intended to revisit and compare current results regarding the application of metaheuristics for the MSCP. Therefore, a recapitulation of the MSCP and its classification into the class of NP-hard optimization problems are provided first. After an overview of notable approximation methods, the focus is shifted towards a brief review of existing metaheuristics which were adapted for the MSCP. In order to allow for a targeted comparison of the existing algorithms, the theoretical worst case complexities in terms of the big O-notation are derived first. This is followed by an empirical study where the identified metaheuristics are examined. Here we use Steiner triple systems, Beasley's OR library, and introduce a new class of instances. Several of the considered approaches achieve close to optimal results. However, our analysis reveals significant differences in terms of runtime and shows that some approaches may even have exponential runtime.

## 1 INTRODUCTION

The *set cover problem* is one of the first NP-complete decision problems found by Karp (1972). Thus, a solution is easy to verify but difficult to find. The corresponding optimization problem is NP-hard. The goal of the decision problem is to answer the question if there are $k$ sets among a given list of sets $S_1, S_2, ..., S_m$ that cover all the elements of $\bigcup_{i=1}^{m} S_i$. Hence the objective of the optimization problem is to find the minimal $k$ to achieve this. The *minimum set cover problem* (MSCP) can be described formally as follows:

$$\begin{aligned} & min\,|M| \\ s.t. & \bigcup_{S\in M} S = \bigcup_{i=1}^{m} S_i \\ & S_i \subseteq \{1,2,...,n\} \\ & M \subseteq \{S_1, S_2, ..., S_m\} \end{aligned} \tag{1}$$

The MSCP occurs in many different domains. Cormode et al. (2010) have identified use cases in information retrieval, operations research, machine learning as well as planning and data mining. The objective of Gotlieb and Marijan (2014) is to find a minimal test suite at specification level.

Within this work we want to give a didactic overview about the MSCP, corresponding approximation algorithms and we conduct a comparison about some of the state of the art metaheuristics. For the latter we derive the worst case iteration cost. We can show that there can be problematic edge cases for certain algorithms that lead to an exponential runtime. Further, we compare some of the introduced metaheuristics on three different benchmark suites.

In Section 2 we give an overview about related work. In Section 3 we introduce various metaheuristics for the MSCP and in the following section we intensively benchmark the aforementioned approaches (Section 4). We close the paper with a summary and conclusion (Section 5).

## 2 RELATED WORK

In complexity theory NP-hard optimization problems have been segregated in various subclasses. No-

Rosenbauer, L., Stein, A., Stegherr, H. and Hähner, J.
Metaheuristics for the Minimum Set Cover Problem: A Comparison.
DOI: 10.5220/0010019901230130
In *Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020)*, pages 123-130
ISBN: 978-989-758-475-6

table are *approximable* (APX), *polynomial-time approximation schemes* (PTAS) and *fully polynomial-time approximation schemes* (FPTAS) (Williamson and Shmoys, 2011).
For these classes the relationship $FPTAS \subseteq PTAS \subseteq APX$ holds if $P \neq NP$. Not every NP-hard optimization problem is one of these classes, such as the minimum set cover problem (Alon et al., 2006). Hence we cannot improve the approximations quality arbitrary whilst still having a polynomial runtime like in PTAS or FPTAS and we do not have a fixed approximation ratio like in APX. Thus it is one of the more difficult NP-hard problems. Further, Dinur and Steurer (2014) proofed that the MSCP cannot be approximated to $(1-o(n))ln(n)$ given $P \neq NP$. Thus for every approximation algorithm for the MSCP there exists a problem instance such that the calculated solution is at least about $ln(n)$ times the optimal value, given $P \neq NP$. As these results are about worst case approximation ratios there is still research for an approximation algorithm that might have good results on the average.

There exists a greedy algorithm for the MSCP. It starts with an empty collection of sets and always takes the set next that covers the most elements which are not yet covered by the greedy's collection (Williamson and Shmoys, 2011). The MSCP can also be approximated using an ILP (Williamson and Shmoys, 2011). Mannino and Sassano (1995) have designed a branch and bound algorithm that can find optimal solutions but may have an exponential runtime.

## 3 METAHEURISTICS

Metaheuristic approaches like evolutionary algorithms have the advantage that they are not limited to a certain set of problems. Hence there have been several attempts to create approximation algorithms for the MSCP using such approaches.

We also want to analyse the runtime of each considered algorithm during this paper. For this various ways are possible such as running the algorithm for a fixed number of iterations (and measuring the the actual duration) (Yu et al., 2014) or deriving the runtime in terms of the Landau notation (until convergence) (Jungnickel, 2007). As usually there is no guarantee that a metaheuristic converges and a reasonable number of publications prefer to run an algorithm for a fixed number of iterations, we decided to analyse the worst case cost of an iteration in terms of the Landau notation. However, we additionally examine the actual runtime later on.

### 3.1 SEIP

*Simple evolutionary algorithms with isolated population* (SEIP) are a rather new family of algorithms. It is a population based approach that starts with an empty one. At the beginning an initial solution such as the zero vector $\mathbf{0}$ is chosen and inserted into the population. After that a loop is entered. In each iteration an instance of the population is drawn at random. Then it is copied and mutated. Next, the mutated instance is compared to the ones in the population. If there is an instance superior to the mutated one, it is not inserted. Otherwise the mutated instance is inserted and all instances inferior to the mutated one are deleted. The superior relation is specific to the problem. The loop is exited if a stopping criterion is met, e. g. no iterations are left (Yu et al., 2010).

We encode solutions as binary vectors of dimension $m$ where the i-th entry corresponds to $S_i$ and a one indicates that the solution uses the set. For the MSCP a solution $\mathbf{x}$ is superior to $\mathbf{y}$ if and only if it covers the same number of elements but uses fewer sets. In our implementation we use a bitwise mutation. Hence we flip each bit with a probability of $\frac{1}{m}$ and we stop after a certain number of iterations. The cost of one SEIP iteration is $O(m+|P|)$ where $P$ denotes the population. $P$ has a maximum size of $n+1$. Thus, the worst case runtime of an iteration is $O(n+m)$.

### 3.2 Artificial Immune Systems

*Artificial immune systems* (AIS) are population based heuristics inspired by the immune systems of vertebrates. The population is extended from time to time and the AIS tries to identify bad solutions among the population and deletes them (as an immune system tries to eliminate pathogens).

Joshi et al. (2014) designed an AIS called *germinal centre artificial immune system* (GCAIS) for the MSCP. GCAIS allows infeasible solutions in its population. The initial population consists of the zero vector $\mathbf{0}$. In every iteration all members of the population are mutated (flipping individual bits with a probability of $\frac{1}{m}$). Afterwards the mutated population and the original one are merged. Then all elements of the new population that are *dominated* by other solutions of it are deleted. A solution is said to dominate another one if and only if it either covers more elements and costs the same (or less) or if it covers the same (or more) amount of elements and costs less. The mutation of the population can also be parallelized in order to reduce the runtime.

The mutation of single element costs $O(m)$ and thus the mutation of the population costs $O(|P|m)$.

The size of the population can get large on certain problem instances. Rosenbauer et al. Rosenbauer et al. (2020) showed that the population size might grow exponentially and that the merge step can be computed in $O(|P|+n)$. The initialization of the algorithm costs $O(m)$ and in summary the cost of an iteration is $O(|P|m+n)$. Hence the population size has a major impact on the runtime.

### 3.3 GSEMO

The *global simple evolutionary multi-objective optimiser* (Giel and Wegener, 2003) is similar to GCAIS as it maintains a population of non-dominated solutions during each iteration. In contrast to GCAIS it only mutates one member of the population instead of all. This solution is drawn uniformly at random and mutated the same way. If it is dominated by any solution of the population then it is not inserted. Upon insertion of a non-dominated solution, the population is searched for dominated solutions and these are removed. Thus GSEMO also is similar to SEIP as they only differ in the way they regard a solution as superior.

GSEMO may also be parallelized. In order to do so $\mu$ populations are introduced. On each population an instance of GSEMO is run. Whenever an instance encounters a solution to be inserted to its population, it decides with a probability $p$ if it sends the found solution to all the other populations. The recipients then also update their populations according to the received solution. For the sake of simplicity we call this variant *GSEMO*.

GSEMO is also similiar in terms of its runtime to GCAIS. The initialization cost is $O(\mu m)$, the mutation cost is $O(m)$ per solution, and the insertion costs $O(m)$ if the same table approach is used. Thus an iteration costs worst case $O(\mu^2 m)$ (if every population sends a mutated solution).

### 3.4 Genetic Algorithms

*Genetic algoritms* (GAs) are a framework of population based algorithms that roughly consist of the three operators: selection, crossover and mutation (Holland, 1992). Solutions are interpreted as chromosomes of an individual. The selection operator chooses a solution from the population. The GA uses it to choose two individuals. Via the crossover operator the two individuals are combined to get two new ones called the children. The children are changed probabilistically using the mutation operator. Afterwards, the GA tries to insert the children into the population but its capacity is limited. Hence sometimes individuals must be removed from the population. This approach is repeated until a stopping criterion is met and the best solution found is returned.

Beasley and Chu (1996) used a binary tournament selection for their GA which draws two individuals from the population at random and takes the one with the least used sets. They further use a one-point crossover operator. It creates a new child by drawing a random integer $i$ from $\{1,2,..,m\}$. For the first child the first $i$ entries of $\mathbf{x}$ and the last $m-i$ of $\mathbf{y}$ are used and for the second the first $i$ entries of $\mathbf{y}$ and the last $m-i$ of $\mathbf{x}$ are used.

The mutation operator of Beasley and Chu (1996) inverts bits with a certain probability that is inverse monotone to the iteration. As a GA is not guaranteed to produce a feasible solution, they also introduced a greedy heuristic to make infeasible solutions valid. If the population becomes too big, random solutions with a cost above average are deleted to keep it in bounds. The selection costs constant time, the crossover and mutation $O(m)$. The greedy repair operation costs $O(n^2m)$ and the deletion costs $O(|P|)$. Hence one iteration costs $O(n^2m+|P|)$.

Beasley and Chu (1996) also developed a method to initialize the population in order to only generate feasible solutions. They iterate over every element to be covered and choose one set at random from the sets that can cover the element. This costs worst case $O(n)$ if one saves a list for every element that contains the indexes of the sets that cover it. The newly created solution is feasible, but may contain redundant sets. In order to delete redundant sets from the solution, they draw each used set exactly once uniformly at random and check if its elements are covered by other sets of the solution. If so, then the set is deleted from it. This check costs worst case $O(nm^2)$. Hence, overall the creation of one element costs $O(n+nm^2)=O(nm^2)$ time and the creation of the population $O(|P|nm^2)$.

### 3.5 Simulated Annealing

*Simulated annealing* (SA) is an analogy to physics which unlike the previous methods is not population based and not an evolutionary algorithm. The analogy reproduces a hot material that is cooling down. During the cooling the metal atoms have enough time to get into an optimal state by ordering and getting in a stable structure. This process is translated into a local search method. Therefore, a temperature function $T(\cdot)$ is created that decreases over time. The method starts at an initial solution and then tries to jump to another solution in its neighbourhood. The new solution is accepted if it is better (in terms of its cost). If not, it is only accepted with a probability based on the temperature function (Kruse et al., 2015).

We use the SA approach of (Minotra, 2008). The initial solution can be created by the greedy algorithm and there are some degrees of freedom for the neighbourhood operator. The neighbourhood operator that we use drops sets at random. This might lead to an infeasible solution that can be repaired using a greedy algorithm. Afterwards a small search for redundant sets is done to keep the solution as small as possible. The neighbourhood operator costs $O(m)$ and the repair of the solution $O(n^2m)$. So overall one iteration costs $O(n^2m)$ in the worst case.

Minotra (2008) used the following temperature function:

$$T(t) = \gamma^t T_{initial} \tag{2}$$

where $T_{initial}$ is the starting temperature and $\gamma$ is a real number between zero and one.

## 3.6 Particle Swarm Optimization

*Particle swarm optimization* is a population based metaheuristic originally developed to solve continuous problems and not discrete ones such as the MSCP. Balaji and Revathi (2016) designed one for the latter which they call *JPSO*. It creates the initial solutions of its particles the same way as the genetic algorithm of Beasley and Chu (1996), which costs as mentioned before $O(|P|nm^2)$.

In contrast to the traditional PSO the JPSO algorithm does not calculate a direction based on several solutions. A particle $p_i$ decides at random if it either moves its current solution $\mathbf{v}_i$ towards a random solution $\mathbf{x}$, the best solution of its neighbourhood $\mathbf{L}_i$, the global best solution $\mathbf{g}$ or towards its own best solution $\mathbf{b}_i$. The target solution is called the attractor. These moves have an analogy to jumps of frogs, thus are called jumps and hence the name *jump particle swarm optimization*. Each jump has the same probability of 0.25. The current solution is merged with the attractor. If the newly generated solution is superior to its own best solution, the best solution of the neighbourhood, or the global one then these are updated.

The merge operator first draws a random number $r$ between 0 and the number of used sets by the particle's current solution $\mathbf{v}_i$. Then we either delete a random set from $\mathbf{v}_i$ or add a random set from the attractor. Each event has the same probability of 0.5. This is repeated $r$ times and costs $O(m)$. During this operation the solution might lose its feasibility. Hence it is repaired greedily if necessary which additionally costs us $O(n^2m)$. After this greedy repair the solution might contain redundant sets. Thus, the repair method further iterates over all used sets and checks if its elements are covered by other sets and removes it if so. This can be done in $O(nm)$. Thus the cost of a jump is $O(n^2m)$. The overall update of a particle depends on the used neighbourhood operator and random solution generator. Hence one iteration costs worst case $O(|P|(n^2m + max\{rand, neigh\}))$, where *rand* is the cost of the generation of a new solution and *neigh* is the cost of the neighbourhood operator.

It is worth to mention that Balaji and Revathi (2016) do not state how they generate a new random attractor. For our later implementation we assumed that they take the same method as for their population generator. Furthermore they do not describe how to choose the neighbourhood of a solution.

## 3.7 Chemical Reaction Optimization

*Chemical reaction optimization* (CRO) is a rather new metaheuristic that is still topic of ongoing research in terms of run time efficiency (Stegherr et al., 2019) and on what problems it performs well (Lam and Li, 2012). Similar to GAs it can be seen as a framework of algorithms that has to be adjusted to the specific problem. There already exists a CRO version for the MSCP which has been successfully tested on the benchmark suite of Beasley (Yu et al., 2014) such as the algorithm of Balaji and Revathi (2016).

CRO is a population based approach and each member of the population represents a molecule. Each molecule has a potential energy and a kinetic energy. The former is the cost of the solution the molecule holds and the latter describes its willingness to change to a worse solution. The method also holds a central buffer of energy that can exchange energy with the molecules. The entire process resembles more or less a chemical reaction in a box. A molecule can collide with the boxes' wall and might change its structure. This on-wall collision searches the neighbourhood of the molecule's solution for a new one and changes it according to a certain probability that depends on the molecule and its buffer's energy. This introduces a form of random search to CRO. Molecules can also decompose into two new molecules that are partially based on the previous one which also represents a form of search.

Based on the quality of the two new solutions and their energy levels, they are either accepted into the population or not. If so, the original molecule gets destroyed. Furthermore two molecules can collide and form a new one. This process is called synthesis and has a similar objective like the crossover step in a GA. It combines two solutions in order to get an improved one. Similar to the preceding operators, the acceptance of the new molecule depends on the solution's quality and the energy levels. CRO takes another analogy to chemical reactions as two molecules can collide and bounce away. This is achieved once more by using a neighbourhood operator to perform local search. This operation is called inter-molecular collision. The four operators are called in a loop until a stopping criterion is met. For example, Yu et al. (2014) used a fixed amount of iterations. We keep it at this abstract level as a detailed description can be found in Lam and Li (2012) and as we focus on the MSCP and not CRO in general in this paper. We focus more on the adaptations of the neighbourhood operator, how the population is initialized, and how infeasible solutions are turned into feasible ones (Yu et al., 2014).

Yu et al. (2014) use vectors of dimension $n$ to encode a solution. They enumerate all elements and there the i-th entry indicates by which set the i-th element is covered. A new solution is created by iteration over every element that must be covered. If the i-th element shall be covered then it is detected by which sets it can be covered. One out of these is drawn at random using a uniform distribution. A generation of a new solution can thus be achieved in $O(nm)$.

The neighbourhood operator first deletes the set from the cover which has the worst efficiency among all used ones. The efficiency of a set in a solution is the number of occurrences in the solution, which can be calculated in $O(n)$. After the set of worst efficiency is removed the solution might become unfeasible as it can contain uncovered elements, which can be verified in worst case $O(n)$. In order to regain feasibility the operator chooses new sets to cover those elements. The i-th set is drawn with the following probability:

$$\frac{s_i}{\sum_{k=1}^{m} s_k} \tag{3}$$

where $s_i$ is the number of elements that the i-th set can cover among the yet uncovered elements. The distribution can be calculated in $O(nm)$. Such a repair attempt has to be done in the worst case $O(n)$ times. When a set is drawn the solution's entries are updated by it. This approach is repeated until the solution is feasible again. So in the worst case the neighbourhood operator costs $O(n^2m)$.

The CRO framework has a decomposition operator that tries to create two molecules from one. This is done by copying the original molecule twice and performing the neighbourhood operator on each copy ten times. Thus the decomposition also costs $O(n^2m)$. The synthesis operator takes two solutions $\mathbf{x}$, $\mathbf{y}$ and combines them into a new solution $\mathbf{z}$. Here they are combined by choosing entries from each one uniformly at random.

This combination costs $O(n)$ and, due to the representation, no repair method is necessary. For the weighted case the probability distribution is once more adapted to the set's cost. The collision operators do not need to be discussed further as they only rely on the neighbourhood operator. The complexity of both is dominated by the neighbourhood operator. Hence both cost $O(n^2m)$. Overall, an iteration of CRO costs worst case $O(n^2m)$.

## 4 EVALUATION

The introduced GA, JPSO, GSEMO, GCAIS and CRO algorithms have all been tested on Beasleys benchmark suite by their creators. They all performed at least reasonably well, but to our knowledge some of them have never been compared with each other. Yet when the MSCP is studied, one should always keep the theoretical bad worst approximation rate in mind. Hence we perform our experiments also on other MSCP datasets in order to see if the mentioned algorithms create as good results on those as well or if their algorithmic design is overfitted to Beasley's data sets[1].

Even though we already examined the introduced metaheuristics using the big O notation, we also measure the actual runtime of each algorithm and use it to compare the methods. Thus, we also take a closer look at the actual runtime of the algorithms which is sometimes not considered during the experimental evaluation of metaheuristics for MSCP (Yu et al., 2014; Joshi et al., 2014; Yu et al., 2010). We think that this is of major importance as the goal of approximating a NP-hard problem is to find a feasible solution of reasonable quality in a reasonable amount of time. Another approach is to measure the number of iterations until an algorithm converges (Joshi et al., 2014; Balaji and Revathi, 2016) which we measure as well.

### 4.1 Experimental Setup

Each experiment is run a hundred times. We stop the execution of an algorithm if either a time budget of one hour is expired or the the algorithm's best solution does not improve other 2000 iterations.

In our JPSO version we use thirty particles and use the 5 nearest neighbours for all neighbourhood operations. As metric we use the $L1$ norm. For the SA we set $\gamma$ to 0.975 and start with an initial temperature of 256. The GA has a population size of 200. For the remaining parameters of CRO we used the setting of Yu et al. (2014). SEIP and GCAIS do not need any additional parameters. We parameterize GSEMO similar

[1]https://https://github.com/LagLukas/mscp

to Joshi et al. (2014) and use 30 populations and a send probability of $\frac{30}{nm}$. We confirmed the quality of the aforementioned parameters in a preliminary study that we leave out due to spatial restrictions.

## 4.2 Steiner Triple Systems

*Steiner triple systems* are regarded as tough instances of the MSCP (Fulkerson et al., 2009). Hence there have also been various publications about approximation algorithms that were tested on this class of instances (Mannino and Sassano, 1995; Wen-Chih Huang et al., 1994). Fulkerson et al. (2009) stated that Steiner triple systems of size 27 and 45 are hard thus we consider these instances[2].

The results of each introduced algorithm are displayed in Table 1. The tables show the average, best approximation ratio, number of iterations and the actual runtime of each considered method. Here GCAIS and the GA produced optimal solutions. The other methods vary in their results and are achieving approximation ratios between 1.1 and 1.6. JPSO cannot improve the initial solutions of its particles for the Steiner triple system of size 27 (as it already converges after 2001 iterations).

The runtime results expose, similar to the theoretical examination, huge differences in terms of runtime. JPSO has by far the highest runtime and SEIP the lowest. The experiment also reveals that a pure evaluation of the number of iterations until an algorithm converges is ambiguous as JPSO has the smallest amount of used iterations, but the highest runtime.

Further, the results show the weakness of GCAIS in terms of its runtime that we discussed in our theoretical analysis: if the sets are rather disjoint and of equal size, then the population can become rather huge which leads to a high runtime. The latter is the case for Steiner triple systems which explains that GCAIS has such a high runtime compared to for example CRO or the GA.

Additionally we performed statistical tests to verify if the considered algorithms vary in terms of their iterations, durations and approximation ratios. Our null hypotheses are that they all behave the same way for these magnitudes. We applied Friedman tests to test these hypotheses which all had p-values lower than $10^{-9}$ which we regard as significant. Thus statistical tests state that the algorithms differ regarding the aforementioned magnitudes.

[2]The experiment's data is from here: http://mauricio.resende.info/data/index.html

## 4.3 Bad Case for the Greedy Algorithm

When the set cover problem is studied, often the greedy algorithm is also introduced. There is a known class of MSCP instances that are tough for the greedy algorithm. In those the method has an approximation ratio of $log_2(n)/2$ which is more or less the worst case scenario. As many of the introduced approximation methods have a greedy component we also use those examples.

The example can be constructed as follows. Let $k$ be an integer bigger than zero. We construct sets $S_1$,...,$S_k$ that are pairwise disjoint. $S_k$ holds $2^k$ elements. $S_j$ holds the elements $\{2^j+1, 2^j+2, ..., 2^{j+1}\}$ for $j > 1$ and $S_1 = \{1,2\}$. We construct two additional sets $M_1$ and $M_2$. $M_1$ contains all even numbers and $M_2$ all odd numbers. Hence we want to cover a total number of $2^{k+1}-2$ elements and $M_1$ and $M_2$ are the biggest sets. Further, half the elements of every $S_j$ are in $M_1$ and the other half in $M_2$. The greedy algorithm first takes either $M_1$ or $M_2$ and then $S_j$, $S_{j-1}$,..., $S_1$ instead of the optimal solution consisting of $M_1$ and $M_2$.

For our benchmarking we create several such systems for one instance. To create one instance we draw a $k$ from $\{2,3,4,5\}$ at random and create the corresponding set system. We repeat this 5 times and each system is disjoint to get our overall instance. Thus, we make it hard for greedy based approaches to break out of a bad solution to the known optimal one.

Table 2 shows the results on those randomly created instances (summarized as random instance). We put all in one table as the problems share the same inner structure and all have the same optimal value of 10. In this case only the GA is capable of finding optimal solutions and the optimal solutions were already in its population from the start. GCAIS, GSEMO, SEIP, and SA also seem to have rather good solutions (their best solutions use 12 instead of 10 sets). Algorithms such as CRO and JPSO that are relying on greedy heuristics achieve rather worse solutions compared to the aforementioned algorithms (due to the design of the problem class). JPSO is also not able to improve the initial solutions of its particles.

Once more JPSO has the highest runtime and SEIP the lowest. The runtime of GCAIS differs a lot on this problem class compared to the Steiner triple systems. In this problem class the available sets highly differ in size which may lead to a smaller population (as it is easier to find a solution that dominates another one).

Here we also performed Friedman tests to verify if the algorithms differ in terms of their iterations, duration and approximation ratios. Once more we could

Table 1: Experimental results for Steiner triple systems of size 27 and 45. It contains average values $\pm\sigma$ for the iterations, approximation ratios and runtime (in seconds). The best values of each column are marked bold and the worst are in italics.

| Steiner 27 | iterations | duration | avg. iteration duration | avg. approx. ratio | best |
|---|---|---|---|---|---|
| CRO | 2126.0 ± 53.0 | 1.048 ± 0.16 | 0.00049 ± 7e-05 | *1.644 ± 0.07* | *1.556* |
| GA | 2558.0 ± 478.0 | 4.342 ± 1.109 | 0.00172 ± 0.00305 | 1.078 ± 0.091 | **1.0** |
| GCAIS | 2102.0 ± 22.0 | 474.271 ± 86.936 | 0.22574 ± 0.0563 | **1.0 ± 0.0** | **1.0** |
| GSEMO | 3608.0 ± 1073.0 | 20.952 ± 8.107 | 0.00586 ± 0.00077 | 1.111 ± 0.091 | **1.0** |
| JPSO | **2001.0 ± 0.0** | *701.123 ± 130.386* | *0.35039 ± 0.08644* | 1.433 ± 0.082 | 1.333 |
| SA | 2649.0 ± 839.0 | 14.047 ± 5.01 | 0.00537 ± 0.02241 | 1.433 ± 0.063 | 1.333 |
| SEIP | *4768.0 ± 1250.0* | **0.666 ± 0.242** | **0.00014 ± 3e-05** | 1.167 ± 0.131 | **1.0** |
| **Steiner 45** | **iterations** | **avg. duration** | **avg. iteration duration** | **avg. approx. ratio** | **best** |
| CRO | 2166.0 ± 41.0 | 2.485 ± 0.62 | 0.00115 ± 0.00038 | *1.727 ± 0.119* | *1.6* |
| GA | 2847.0 ± 767.0 | 25.979 ± 5.508 | 0.00943 ± 0.01905 | 1.2 ± 0.144 | **1.0** |
| GCAIS | 2631.0 ± 454.0 | 1951.188 ± 320.956 | 0.74636 ± 0.23683 | **1.033 ± 0.035** | **1.0** |
| GSEMO | 4810.0 ± 2115.0 | 53.366 ± 28.564 | 0.01106 ± 0.00182 | 1.28 ± 0.076 | 1.133 |
| JPSO | **1833.0 ± 121.0** | *3594.182 ± 15.282* | *1.9693 ± 0.63859* | 1.64 ± 0.064 | 1.533 |
| SA | 3678.0 ± 1176.0 | 88.132 ± 37.294 | 0.02327 ± 0.00612 | 2.16 ± 0.126 | 2.0 |
| SEIP | *7197.0 ± 1473.0* | **2.362 ± 0.706** | **0.00033 ± 3e-05** | 1.247 ± 0.063 | 1.133 |

Table 2: Experimental results for the bad cases for the greedy algorithm. It contains average values $\pm\sigma$ for the iterations, approximation ratios and runtime (in seconds). The best values of each column are marked bold and the worst are in italics.

| rand | iterations | avg. duration | avg. iteration duration | avg. approx. ratio | best |
|---|---|---|---|---|---|
| CRO | 2076.0 ± 53.0 | 1.084 ± 0.405 | 0.00052 ± 0.00021 | *2.0 ± 0.0* | *2.0* |
| GA | **2001.0 ± 0.0** | 2.635 ± 0.919 | 0.00132 ± 0.00346 | **1.163 ± 0.084** | **1.0** |
| GCAIS | 2302.0 ± 252.0 | 5.346 ± 1.357 | 0.00236 ± 0.00027 | 1.288 ± 0.06 | 1.25 |
| GSEMO | 2648.0 ± 200.0 | 9.208 ± 2.573 | 0.0035 ± 0.00021 | 1.288 ± 0.06 | 1.25 |
| JPSO | **2001.0 ± 0.0** | *492.657 ± 113.087* | *0.24621 ± 0.06516* | 2.013 ± 0.092 | 1.875 |
| SA | 2297.0 ± 514.0 | 9.221 ± 4.08 | 0.00403 ± 0.00175 | 1.3 ± 0.065 | 1.25 |
| SEIP | *5813.0 ± 1258.0* | **1.049 ± 0.388** | **0.00018 ± 4e-05** | 1.45 ± 0.222 | 1.25 |

observe p-values lower than $10^{-9}$. Thus the statistical tests support the claim that the algorithms differ in the aforementioned magnitudes.

## 4.4 Beasley's OR Library

Beasley's OR library is another source for MSCP instances that has been used to benchmark approximation algorithms (Balaji and Revathi, 2016; Yu et al., 2014; Joshi et al., 2014). Thus we also took a look at some of its MSCP instances (the scpe1 to scpe5 instances and scpclr10 to scplr13 instances). Due to the spatial limitations of this paper we only briefly summarize (Table 3) and discuss our results.

The scpclr instances have rather small and disjoint sets. Thus there we could observe the same problems in terms of runtime for GCAIS as once more its population explodes. On the other hand the scpe class does not have this structure and thus GCAIS has a much lower runtime. JPSO once more had the highest runtimes and SEIP the lowest runtimes on both problem classes. In terms of their approximation ratios we could verify the results of Balaji and Revathi (2016); Joshi et al. (2014) that often had optimal or close to optimal solutions. SA fails to produce solutions of reasonable quality for the scpclr instances.

We also performed Friedman tests which were once more significant (p-values lower than $10^{-9}$) and thus the algorithms behave differently on these instances.

## 5 CONCLUSION

In this work we gave a didactic overview about the *minimum set cover problem* (MSCP) from various perspectives. We took a special focus on metaheursitics. We discussed the worst case iterational cost and evaluated the approaches on various benchmarking problems.

We could observe that some approaches have a hard time finding high quality solutions outside of Beasley's well-known OR library. Further we could observe high runtimes and in our theoretical evaluation we could identify edge cases where algorithms might even have an exponential one. Thus we recommend to also perform a formal analysis of an algorithm's runtime in order to detect bottlenecks.

Table 3: Aggregated results for the scpe and scpclr classes (averaged durations and approximation ratios $\pm\sigma$).

| scpe | duration | approx. ratio |
|---|---|---|
| CRO | **3.9 ± 1.0** | *1.66 ± 0.0* |
| GA | 13.0 ± 17.2 | 1.19 ± 0.25 |
| GCAIS | 18.8 ± 36.9 | **1.03 ± 0.1** |
| GSEMO | 24.4 ± 22.9 | 1.09 ± 0.19 |
| JPSO | *3564.6 ± 64.5* | 1.31 ± 0.33 |
| SA | 52.0 ± 64.3 | 1.09 ± 0.18 |
| SEIP | 11.0 ± 29.6 | 1.38 ± 0.62 |

| scpclr | duration | approx. ratio |
|---|---|---|
| CRO | **5.6 ± 2.3** | 1.02 ± 0.16 |
| GA | 99.2 ± 97.4 | **1.0 ± 0.0** |
| GCAIS | 508.1 ± 375.3 | **1.0 ± 0.0** |
| GSEMO | 98.1 ± 98.9 | 1.31 ± 0.58 |
| JPSO | *3581.5 ± 29.2* | 1.38 ± 0.87 |
| SA | 809.8 ± 1065.6 | *34.82 ± 5.13* |
| SEIP | 12.0 ± 19.9 | 2.91 ± 2.29 |

## REFERENCES

Alon, N., Moshkovitz, D., and Safra, S. (2006). Algorithmic Construction of Sets for K-restrictions. *ACM Trans. Algorithms*, 2(2):153–177.

Balaji, S. and Revathi, N. (2016). A New Approach for Solving Set Covering Problem Using Jumping Particle Swarm Optimization Method. 15(3):503–517.

Beasley, J. and Chu, P. (1996). "A Genetic Algorithm for the Set Covering Problem". *European Journal of Operational Research*, 94(2):392 – 404.

Cormode, G., Karloff, H., and Wirth, A. (2010). Set Cover Algorithms for Very Large Datasets. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 479–488, New York, NY, USA. ACM.

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.

Fulkerson, D., Nemhauser, G., and Trotter, L. (2009). *Two Computationally Difficult Set Covering Problems that arise in Computing the 1-width of Incidence Matrices of Steiner Triple Systems*, volume 2, pages 72–81.

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.

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.

Jungnickel, D. (2007). *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd edition.

Karp, R. (1972). Reducibility Among Combinatorial Problems. volume 40, pages 85–103.

Kruse, R., Borgelt, C., Braune, C., Klawonn, F., Moewes, C., and Steinbrecher, M. (2015). *Computational Intelligence Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Computational Intelligence. Springer Vieweg, Wiesbaden, 2 edition.

Lam, A. and Li, V. (2012). Chemical Reaction Optimization: A tutorial. *Memetic Computing*, 4.

Mannino, C. and Sassano, A. (1995). "Solving Hard Set Covering Problems". *Operations Research Letters*, 18(1):1 – 5.

Minotra, D. (2008). A Study of Heuristic-Algorithms for Set-Covering Problems.

Rosenbauer, L., Stein, A., and Hähner, J. (2020). A Germinal Centre Artificial Immune System for Software Test Suite Reduction. *Artificial Life*.

Stegherr, H., Stein, A., and Haehner, J. (2019). Parallel Chemical Reaction Optimisation for the Utilisation in Intelligent RNA Prediction Systems. In *ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems*, pages 1–8.

Wen-Chih Huang, Cheng-Yan Kao, and Jorng-Tzong Horng (1994). A Genetic Algorithm Approach for Set Covering Problems. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 569–574 vol.2.

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., 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.