# Implementation Exercise 2
## Heuristic Optimization

Jens Nevens - ID 500093

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Elsene, Belgium
jens.nevens@vub.ac.be

## Abstract

This report describes the design and implementation of two algorithms for solving the Set Covering Problem. Both of these algorithms are based on algorithms from the literature and will be empirically evaluated on a set of well known problem instances.

## Introduction

The Set Covering Problem (SCP) is a well-known and well-studied $NP$-hard optimization problem. An instance of the SCP consists of $m$ rows and $n$ columns, where each column $j$ has a cost $c_j$ associated with it. These rows and columns can be placed in an $m \times n$ 0-1 matrix $A$. Each element $a_{ij}$ of this matrix indicates whether column $j$ covers row $i$ (i.e. $a_{ij} = 1$) or does not (i.e. $a_{ij} = 0$). Let $I = \{1, \ldots, m\}$ and $J = \{1, \ldots, n\}$ be the set of rows and columns respectively.

The goal of the optimization task is to make sure the rows are covered by a subset of the columns, denoted $S$, which has minimal cost. In other words, assume $x_j = 1$ when $j \in S$ and $x_j = 0$ when $j \notin S$. Then, the SCP can be formally described as:

$$\text{Minimize } \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{subject to } \sum_{j=1}^{n} a_{ij} x_j \geq 1, \qquad i \in I \tag{2}$$

$$x_j \in \{0, 1\}, \qquad j \in J \tag{3}$$

Eq. 2 makes sure that each row is covered by at least one column. Additional commonly used notations are the following:

$$J_i = \{j \in J | a_{ij} = 1\} \tag{4}$$

the subset of columns covering row $i$

$$I_j = \{i \in I | a_{ij} = 1\} \tag{5}$$

the subset of rows covered by column $j$

In the following sections, two algorithms for solving the SCP will be described. The first one being an adaptation of the Ant Colony Optimization algorithm developed by Ren et al. (2010). This is followed by an algorithm based on the genetic algorithm presented in (Beasley and Chu, 1996). For each of these algorithms, the design and parameters considerations will also be documented and justified. Afterwards, several computational experiments that were conducted on the algorithms will be described, discussed an interpreted in detail. Finally, the performance of both algorithms will be compared.

## Ant Colony Optimization

### Overview

The technique of Ant Colony Optimization is based on observations of how real ant colonies communicate with each other whilst gathering food (Dorigo et al., 1996). In order to find food, the ants initially wander around randomly. When some ant has acquired food, it will return to the colony while depositing pheromones on its path. The other ants detect these pheromones and are likely to follow this path and in turn also deposit pheromones. This positive feedback leads to reinforcement of the pheromone trail. Over time, however, the pheromone evaporates. This makes the path less attractive for the other ants. Also, the longer the path is, the more time the pheromones have to evaporate. Thus the ants will prefer shorter paths over longer ones, since these shorter paths will have a higher density of pheromones and consequently be visited by more ants.

This observation can be translated directly to an algorithmic technique. Indeed, the ants wandering around to find food are represented by the artificial ants wandering around the search space to find a solution to the given problem. More specific, each ant in the colony will construct a solution. If a solution has been found, the artificial ants will deposit artificial pheromones on the solution components, to attract the other ants. A solution component in the context of the SCP is a column. These artificial pheromones are often combined with heuristic information that is problem-specific. So each column $j \in J$ has both an associated pheromone value $\tau_j$ and a heuristic value $\eta_j$. The pheromone value indicates the learned desirability of a column, while the heuristic value represents the prior desirability of a column. In order to avoid early convergence to local optima, the artificial pheromones must also evaporate at a certain rate. Again, as with the real ants, the shorter the path is, the higher the density of the pheromones will be and consequently, the more ants will visit the path. A short path in the SCP corresponds to a low cost solution. This procedure is repeated for a certain number of iterations in order to find an optimal solution. The stopping criteria can be the quality of the acquired solution or the amount of elapsed CPU-time.

This algorithmic technique is summarized in Algorithm 1. Each component of this algorithm will be discussed in detail in the next section.

**Algorithm 1:** Main ACO Algorithm

```
1  $S_o \leftarrow \emptyset$;
2  $\tau \leftarrow$ initPheromones();
3  $\mathcal{C} \leftarrow$ initColony($\tau$);
4  while runtime not over do
5      for ant $a_i$ in $\mathcal{C}$ do
6          $S_i \leftarrow$ constructSolution($a_i$);
7          localSearch($S_i$);
8      end
9      $S_o \leftarrow$ updateOptimal($\mathcal{C}$);
10     $\tau \leftarrow$ updatePheromones($\mathcal{C}$);
11 end
   Return: $S_o$
```

## Design

*Initialization* Before the algorithm can start, several constructs need to be initialized. First of all, a construct is created that will retain the best solution obtained thus far, denoted $S_o$. This is called the *Optimal* struct.

Afterwards, the pheromone associated with each solution component, i.e. each column, is initialized. This version of Ant Colony Optimization follows approximately the same notions as those introduced in Max-Min Ant System (MMAS) (Stützle and Hoos, 2000), namely that the value for the pheromones are bounded between $[\tau_{min}, \tau_{max}]$. All solution components receive as initial pheromone value $\tau_{max}$. The initial value for this parameter is calculated as follows:

$$\tau_{max} = \frac{1}{\left((1-\rho)\left(\sum_{j \in J} c_j\right)\right)} \tag{6}$$

where the parameter $\rho$ $(0 \leq \rho < 1)$ is the pheromone persistence. The lower bound, $\tau_{min}$, is initialized as defined by Eq. 7:

$$\tau_{min} = \varepsilon \tau_{max} \tag{7}$$

The parameter $\varepsilon$ $(0 < \varepsilon < 1)$ is called the ratio coefficient. Finally, the ant colony $\mathcal{C}$, comprised of $N_a$ individual ants, is constructed. Each ant represents a solution to the SCP in the form of the *Solution* struct.

*Solution Construction* The next step in the algorithm is for each ant $a_i \in \mathcal{C}$ to construct a solution. Such a constructive method will iteratively add columns to the solution until all rows are covered. It is important for the solution construction method to guarantee search diversification to some extent. In other words, the initially generated solutions should differ enough to span a wide area of the search space. The construction method used here tries to achieve this by first selecting an uncovered row $i \in I$ randomly. Next, a column needs to be chosen. In order to choose a column, the ants take into account both the earlier acquired preference for certain solution components via the pheromone trail $\tau_j$ and the prior preference for certain solution components via the heuristic information $\eta_j$. These values are usually weighted by the parameters $\alpha$ and $\beta$ respectively. Additionally, to reduce computational overhead, the column $j$ will be chosen from the set $J_i$ instead of all columns $J$. Indeed, the columns in this set will never be redundant, since they will always contain at least one uncovered row, namely $i$. The probability for a column $j \in J_i$ to be selected is defined by:

$$P(S_t = j | r_t = i, i \in R_{t-1}) = \begin{cases} \frac{\tau_j \eta_j^{\beta}}{\sum_{q \in J_i} \tau_q \eta_q^{\beta}}, & \text{if } j \in J_i \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

where $R_{t-1}$ is the set of uncovered rows at time step $t-1$, $r_t$ is the chosen row at time step $t$ and $S_t$ is the (partial) solution at time step $t$. As can be seen from Eq. 8 the pheromone values $\tau_j$ and $\tau_q$ are weighed by a factor $\alpha = 1$ and thus it is left out. The parameter $\beta$, to weigh the heuristic information, remains present. The heuristic information $\eta_j$ for a column $j$ used here is a kind of dynamic heuristic information, meaning that it will depend on the current partial solution. This also means that it has to be recomputed in every iteration. The formula for computing this heuristic information is defined by Eqs. 9 and 10:

$$\phi_j = |I_j \cap R_t| \quad (9)$$

$$\eta_j = \frac{\phi_j}{c_j} \quad (10)$$

where $R_t$ is again the set of uncovered rows at time step $t$. The value $\phi_j$ indicates the number of rows that will be additionally covered when choosing column $j$. This heuristic value enforces a preference towards columns that cover more rows and have lower costs.

The method described above is called the Single-row-oriented solution construction method (SROM) and can be summarized by Algorithm 2.

*Local Search* After constructing an initial solution $S_i$, each ant $a_i \in \mathcal{C}$ applies a local search procedure. The aim of this local search procedure is to improve the performance of the algorithm and the quality of the solutions. In fact, this implementation features two local search procedures: an iterative first improvement (FI) method combined with redundancy elimination (RE) and a method called Column Replacement (CR). Limited experimentation has shown that CR performs better than FI+RE. The CR method runs faster, can thus complete much more iterations in the same runtime and finds the optimal solution faster. For this reason, together with the fact that FI+RE was already extensively covered in the first implementation exercise, only the CR method will be covered here.

**Algorithm 2:** SROM

```
1  $t \leftarrow 0$;
2  $S_t \leftarrow \emptyset$;
3  $R_t \leftarrow I$;
4  while $\neg$ validSolution($S_t$) do
5      $t \leftarrow t + 1$;
6      $i \leftarrow$ pickRandom($R_t$);
       // This implements Eq. 8
7      probabilities $\leftarrow$ computeProbabilities($J_i, \tau$);
8      $j \leftarrow$ pickWithPDF(probabilities);
9      $S_t \leftarrow S_{t-1} \cup j$;
10     $R_t \leftarrow R_{t-1} \setminus I_j$;
11 end
   Return : $S_t$
```

The idea behind the Column Replacement local search method is the following: even though the solutions constructed by the SROM method are all valid solutions, it is still possible that it contains redundant columns. It is the goal of this method to find these columns and either remove them or replace them by lower cost columns. It is also important that the resulting solution remains valid.

It is important to note that the method described in this paragraph is executed for each ant $a_i$ individually. Since this method will try to remove high cost columns first, the initial step is to sort all columns $J$ in a descending fashion. In case the costs of two columns are equal, they will be sorted ascending based on the amount of rows they cover. The result of this is denoted $J'$. Next, for each column $j \in J'$ that is part of ant $a_i$'s solution $S_i$, the set $W_j$ is computed. This set contains the rows that are *only* covered by column $j$. More formally, $W_j = \{i \in I_j | o_i = 1\}$, where $o_i$ denotes the amount of columns covering row $i$. The ants keeps track of this value for every row[1].

In case $|W_j| = 0$, it can be concluded that column $j$ is redundant and can be removed. Indeed, when the set $W_j$ is empty for a certain column $j$, this means that no single row is covered by column $j$ alone. Since the columns are considered from most to least costly, removing $j$ removes the redundant column with the highest cost first.

On the other hand, when $|W_j| > 0$, the cost of column $j$ is compared to the total cost of all columns $low_i$ with

[1]In the code, this value is denoted `ncol_cover`

$i \in W_j$. A column $low_i$, for a row $i$, is the least cost column covering row $i$. If this total cost is lower than the cost of column $j$, this column can be replaced by all $low_i$'s. When $|W_j| = 1$, this is quite straightforward, since there is only one column $low_i$ to compare with. However, two cases can be distinguished when $|W_j| = 2$. This means that there are two rows, $i_1$ and $i_2$, which are only covered by column $j$. If the column $low_{i_1}$ is the same as the column $low_{i_2}$, but different from column $j$, column $j$ can be directly replaced by either of them. When the columns $low_{i_1}$ and $low_{i_2}$ are different, but their total cost is lower than the cost of column $j$, then column $j$ is replaced by both of them.

Algorithm 3 summarized the Column Replacement local search method in pseudo-code.

**Algorithm 3:** Column Replacement

```
Input  : $S_i$
1  $J' \leftarrow$ sortDescending($J$);
2  for $j \in J'$ do
3      if $j \in S_i$ then
4          $W_j \leftarrow$ computeWj($j$);
5          if $|W_j| = 0$ then
6              $S_i \leftarrow S_i \setminus j$;
7          else if $|W_j| = 1$ then
8              $low_i \leftarrow$ computeLow($W_j$);
9              $S_i \leftarrow S_i \setminus j \cup low_i$;
10         else if $|W_j| = 2$ then
11             $low_{i_1}, low_{i_2} \leftarrow$ computeLow($W_j$);
12             if $low_{i_1} = low_{i_2} \neq j$ then
13                 $S_i \leftarrow S_i \setminus j \cup low_{i_1}$
14             else if $low_{i_1} \neq low_{i_2} \neq j$ then
15                 if $c_{low_{i1}} + c_{low_{i2}} < c_j$ then
16                     $S_i \leftarrow S_i \setminus j \cup low_{i_1} \cup low_{i_2}$;
17                 end
18             end
19         end
20     end
21 end
Return: $S_i$
```

*Pheromone Update Rule* The final part of the algorithm is the updating of the pheromones. As was already mentioned earlier, the pheromone $\tau_j$ associated with a column $j$ represents the learned desirability of that column.

The pheromone update rule must work in such a way that a small fraction of the pheromone for each solution component evaporates, but the solution components often visited by the ants receive an increase in pheromone. There are two ways to define the often visited solution components: one way is to consider the solution components of the best solution obtained in the current iteration ($S_{ib}$), while the other considers the solution components of the best solution obtained thus far in the entire run-time of the algorithm ($S_{gb}$). Here, the latter will be used. This will allow for search intensification to occur faster, since the ants will focus on similar solutions across iterations. Search diversification is also important, but this is already covered by the constructive method discussed above. To avoid search stagnation, e.g. because the pheromone value associated with a certain column is excessively high or low, the pheromone values are bounded between $[\tau_{max}, \tau_{min}]$. The pheromone update rule can be formally summarized as follows:

$$\tau_j \leftarrow \rho\tau_j + \triangle\tau_j, \quad j \in J \tag{11}$$

$$\triangle\tau_j = \begin{cases} 1/\sum_{q\in S_{gb}} c_q, & \text{if } j \in S_{gb} \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

$$\text{if } \tau_j < \tau_{min}, \quad \tau_j \leftarrow \tau_{min} \tag{13}$$

$$\text{if } \tau_j > \tau_{max}, \quad \tau_j \leftarrow \tau_{max} \tag{14}$$

The parameter $\rho$ $(0 \leq \rho < 1)$, used in Eq. 11, is the pheromone persistence. This parameter will determine how much of the pheromone evaporates in each iteration. The amount of pheromone $\triangle\tau_j$ added to each solution component $j \in S_{gb}$ is inversely proportional to the cost of this optimal solution. The values for $\tau_{max}$ and $\tau_{min}$ are determined similarly to Eqs. 6 and 7, but now adjusted to the best solution obtained thus far:

$$\tau_{max} = \frac{1}{\left((1-\rho)\left(\sum_{q\in S_{gb}} c_q\right)\right)} \tag{15}$$

$$\tau_{min} = \varepsilon\tau_{max} \tag{16}$$

with $\varepsilon$ $(0 < \varepsilon < 1)$ the ration coefficient. Since these parameters depend on the value of $S_{gb}$, each time a new best solution is found, $\tau_{max}$ and $\tau_{min}$ need to be updated.

## Parameters

The parameter settings of any stochastic local search algorithm play a large part in the algorithm's performance and the quality of the solutions generated. The authors of the Ant Colony Optimization algorithm on which this implementation is based carefully tuned the parameters through experimentation. The parameter values tested for each of the parameters are:

$$\begin{aligned}\beta &\in \{1.0, 2.0, 5.0, 8.0\}\\ \rho &\in \{0.90, 0.95, 0.98, 0.99\}\\ \varepsilon &\in \{0.001, 0.002, 0.005, 0.010\}\\ N_a &\in \{10, 20, 50, 100\}\end{aligned}$$

They concluded the optimal configuration of parameters was $\beta = 5,\ \rho = 0.99,\ \varepsilon = 0.005$ and $N_a = 20$. Due to the limited time frame, this implementation did not base the parameter values on extensive experimentation. The base setup of this implementation used the parameters suggested by the original authors. An alternative could be to use the irace package (López-Ibánez et al., 2011) for automatic parameter configuration.

## Genetic Algorithm

### Overview

The idea behind a genetic algorithm is to mimic nature's process of biological evolution. A population arises and lives on according to the principles of natural selection and survival of the fittest. If an individual is better adapted to the natural environment, i.e. fit, it has a higher chance of both survival and reproduction. In other words, the genes from these fit individuals will be more likely to be passed on to the next generation of the population, so increasing the general fitness of the entire population. Following this principle, the population evolves in such a way that they are increasingly well adapted to the environment.

The genetic algorithm simulates this natural process by representing each solution to the SCP $S_i$ as an individual (or chromosome) $d_i$ of the population, where each individual consists of solution components (or genes) and has a fitness value associated with it. The fitness value in case of the SCP is simply the cost of the solution: $\sum_{j \in S_i} c_j$. A population consists of $N_d$ of these individual. Individuals with a low fitness value, meaning they are well adapted to their environment since the SCP is a minimization problem, are chosen for reproduction. This is done by combining the parents' genes in a crossover operator to produce new offspring. In order to introduce some randomness into the population, a small fraction of the offspring's genes are mutated before introducing them into the population. It is possible for the offspring to replace some less fit individuals while the overall population size remains the same (steady-state approach) or for it to replace the population entirely (generational approach). This is repeated until the termination criterion is satisfied with the aim of creating the fittest individual possible, i.e. the optimal solution. The termination criterion can again be the quality of the found solution or the amount of elapsed CPU-time, just like with the Ant Colony Optimization algorithm.

There is however a well known problem with the algorithmic technique described above. Often, these algorithms take a long time to converge towards the optimal solution. This is because there is not enough search intensification that can be guaranteed through the different genetic operators. That is why it is common to introduce an additional local search step after the initialization of the individual, the crossover operator and the mutation operator. When adding these additional steps to the algorithm described above, it is best described as a memetic algorithm instead of a genetic algorithm. The algorithm that will be described here is based upon the genetic algorithm presented by Beasley and Chu (1996), but adopted to become a memetic algorithm. The issue described above is exactly the cause of this change.

**Algorithm 4:** Main GEN Algorithm

```
1  $S_o \leftarrow \emptyset$;
2  **for** *individual* $d_i$ *in* $\mathcal{P}$ **do**
3      $S_i \leftarrow$ constructSolution($d_i$);
4      localSearch($S_i$)
5  **end**
6  **while** *runtime not over* **do**
7      $d_{p_1}, d_{p_2} \leftarrow$ selectParents($\mathcal{P}$);
8      $d_{child} \leftarrow$ crossover($d_{p_1}, d_{p_2}$);
9      $S_{child} \leftarrow$ makeValid($d_{child}$);
10     localSearch($S_{child}$);
11     mutate($d_{child}$);
12     $S_{child} \leftarrow$ makeValid($d_{child}$);
13     localSearch($S_{child}$);
14     $\mathcal{P} \leftarrow$ introduceChild($S_{child}$);
15     $S_o \leftarrow$ updateOptimal($\mathcal{P}$)
16 **end**
   **Return** : $S_o$
```

Algorithm 4 described the main loop of the implemented algorithm in pseudo-code. The next section will cover all components of this algorithm in detail.

## Design

*Representation* The first thing to consider when implementing a genetic algorithm is the representation. This is an important part of the algorithm since a suitable representation will facilitate the implementation of the crossover and mutation operators.

The solution representation used here is a binary representation of size $|J|$, where each bit represents a column. A value of 1 on position $j$ indicates that the $j$th column is present in the given individual $d_i$. With this representation, the fitness $f_i$ of an individual $d_i$ can be calculated by

$$f_i = \sum_{j \in J} x_{ij} c_j \tag{17}$$

where $x_{ij}$ is the value of the $j$th column in the $i$th individual and $c_j$ is the cost of the $j$th column. There is a downside to this representation, however. Namely the fact that the binary string representing the individual might become an invalid solution representation through application of the crossover and mutation operators. In order to overcome this, a solution validity method needs to be implemented that will complete a given invalid solution into a valid one, taking heuristic information into account. This validity method is crucial, since the local search method used here will only work properly on valid solutions. That is why, in Algorithm 4, it is applied before each local search step.

A solution can also be represented by a non-binary representation. Here, one uses a string of size $|I|$, where each element is a row $i$ of the SCP and the value on the $i$th position is a column $j$ covering that row. This representation eases the use of the crossover and mutation operators, since the solutions generally remain feasible, but the computation of the fitness value $f_i$ becomes more complex.

The implementation accompanying this report features the binary representation, since a binary string with the exact behaviour described above was already present in the previous implementation exercise[2].

*Solution Construction* Before the main loop of the genetic algorithm can start, the population $\mathcal{P}$ of individuals $d_i$ needs to be constructed. To construct a solutions $S_i$ associated with an individual $d_i$, the constructive method will iteratively add columns until the solution $S_i$ is valid. It is important for this constructive method to guarantee search diversification to some extent. This is done by introducing randomness in the constructive method, such that the solutions generated span a wide range of the search space. Just like with the ACO algorithm, the constructive method starts by picking an uncovered row $i \in I$ at random. Next, a column $j \in J_i$ is chosen randomly from the set of columns covering the chosen row $i$. This column will never be redundant, since it covers at least one uncovered row, namely $i$.

This constructive method is very simple, but allows for very diverse solutions to be generated. Because of this large diversity, the quality of these initial solutions is often quite poor. That is why an additional local search procedure is applied to each solution $S_i$ after it has been constructed. This can be seen in the first for-loop of Algorithm 4, specifying the population initialization. The local search procedures employed will be discussed later on in this report.

*Parent Selection* The first phase of the algorithm's main loop is the parent selection procedure. The aim of this procedure is to select parents that are suitable for passing on their genes to the next generation. This implementation features two such parent selection procedures.

The first procedure, called proportionate selection, calculates for each individual $d_i$ the probability for being selected as proportional to its fitness value $f_i$. This probability distribution is used to select parents $d_{p_1}$ and $d_{p_2}$. Tournament selection, on the other hand, works by creating two pools of size $T$ of individuals. These individuals are randomly chosen from the population. Then, from each of the two pools, the individuals with the best fitness value are chosen to be the parents. The optimal pool-size $T$ can be determined through experimentation.

In this implementation, the proportionate selection method is used, but limited experimentation has shown that the binary tournament selection (i.e. $T = 2$) returns comparable results.

*Crossover operators* After selecting two suitable parents, their genes are combined to generate offspring through the use of the crossover operator. Again, the implementation linked to this report contains two different crossover operators.

[2]Namely, each *Solution* struct has the array `x`

The simplest crossover operator is the uniform crossover operator. Each gene $j$ of the offspring is copied over from the corresponding gene of one of the parents. This is done by generating a random bit. The value of this bit, 0 or 1, determines which parent is chosen to copy the current gene $j$ from. Pseudo-code for this crossover operator can be seen in Algorithm 5. While this technique is very straightforward, it does not take into account the fitness of the parents in any way. It might be reasonable to weight the probability of choosing a gene from one parent or the other according to their relative fitness value.

**Algorithm 5:** Uniform Crossover Operator

```
Input  : Parents $d_{p_1}$ and $d_{p_2}$
1 $d_{child} \leftarrow \emptyset$;
2 for gene $j$ in $J$ do
3     bit $\leftarrow$ randomBit();
4     if bit = 0 then
5         $d_{child} \leftarrow d_{child} \cup d_{p_1}[j]$
6     else if bit = 1 then
7         $d_{child} \leftarrow d_{child} \cup d_{p_2}[j]$
8     end
9 end
Return : $d_{child}$
```

This is exactly what the fusion crossover operator aims to achieve. The idea behind the fusion crossover operator is that inheriting a gene from a more fit parent increases the likelihood of the offspring itself being fitter. Algorithm 6 illustrates how this is achieved. Since this crossover operator focuses on the differences of both parents, it is more capable of generating offspring that is different from the parents.

As was previously mentioned, the implemented algorithm is more a memetic algorithm rather than a genetic algorithm. That is why the generated offspring is passed on to a local search procedure. This procedure will be explained later on.

*Mutation operators* After the offspring is generated, it is passed on to the mutation operator. The aim of this operator is to alter a small fraction of the offspring's genes (columns) to introduce some randomness and thereby expanding the search space. Since a binary representation is used, the mutation operator will flip a small fraction of the offspring's bits.

**Algorithm 6:** Fusion Crossover Operator

```
Input  : Parents $d_{p_1}$ and $d_{p_2}$
1 $d_{child} \leftarrow \emptyset$;
2 $f_{p_1} \leftarrow$ fitness($d_{p_1}$);
3 $f_{p_2} \leftarrow$ fitness($d_{p_2}$);
4 for gene $j$ in $J$ do
5     if $d_{p_1}[j] = d_{p_2}[j]$ then
6         $d_{child} \leftarrow d_{child} \cup d_{p_1}[j]$ $(= d_{p_2}[j])$
7     else if $d_{p_1}[j] \neq d_{p_2}[j]$ then
8         $d_{child} \leftarrow d_{child} \cup d_{p_1}[j]$ with probability
          $p = f_{p_2}/(f_{p_1} + f_{p_2})$;
9         $d_{child} \leftarrow d_{child} \cup d_{p_2}[j]$ with probability
          $q = 1 - p$
10    end
11 end
Return : $d_{child}$
```

An important aspect of the mutation operator is therefore the calculation of how many genes should be mutated. This implementation features a variable mutation rate, where the amount of genes altered depends on the convergence of the genetic algorithm. At the start of the algorithm, the crossover operator is mostly responsible for the search process and thus the mutation operator alters only a minimal amount of genes. As the algorithm converges, the crossover operator will generate more similar solutions. Than, the mutation operator should take over by altering more genes of the offspring. As the genetic algorithm converges, so does the mutation rate converge to a stable amount of genes to alter.

This number of bits mutated, denoted $N_m$, can be expressed by the following equation:

$$N_m = \left\lceil \frac{m_f}{1 + \exp(-4m_g(t - m_c)/m_f)} \right\rceil \tag{18}$$

where $t$ is the number of offspring generated in the entire algorithm thus far, $m_f$ is the final stable mutation rate, $m_c$ specifies the number of offspring generated before the mutation schedule reaches half of the stable mutation rate ($m_f/2$) and $m_g$ is the gradient of the mutation function at $t = m_c$. Whilst the value for $m_f$ is user defined, the corresponding values for $m_c$ and $m_g$ should be determined according to the algorithm's convergence rate. This can be done by running the algorithm without the mutation operator and than manually altering $m_c$ and $m_g$ until the convergence of both is similar. This could also be done automatically, by using the irace package (López-Ibánez et al., 2011).

Finally, after the amount of genes or columns to alter is determined, the mutation should be applied to the offspring. In this implementation, the mutation operator chooses the $N_m$ columns to alter at random. Another way of doing this could be to construct a set of valuable columns and choose $N_m$ columns out of this set. The paper by Beasley and Chu (1996) constructs this set by taking the 5 least cost columns, for each row $i$, in $J_i$, the subset of columns covering row $i$.

As with the solution construction and crossover operator, so is an additional local search procedure applied after the mutation operator.

*Solution Validity* Both the crossover and mutation operator can have such an effect on the binary string of the individual that it is no longer a valid solution. Also, in order to properly apply the local search procedure, the solution passed to it should be a valid one. For these two reasons, a solution validity method should be implemented. Such a method will add columns to the solution until it is valid. Also, in order to retain solution quality, this validity method should take into account some form of heuristic information when adding columns to the invalid solution.

In order to do this, the validity method will first search for all uncovered rows $i \in I$. Afterwards, for each of these rows $i$, the column $j$ that minimizes the ratio

$$\frac{c_j}{|I_j \cap R_t|} \tag{19}$$

is added to the solution, where $c_j$ is the cost of the column and $R_t$ is the amount of uncovered rows at time step $t$. Thus $|I_j \cap R_t|$ is the amount of uncovered rows that column $j$ would additionally cover.

*Local Search* An additional local search method is added to the implementation in order to turn the genetic algorithm into a memetic algorithm. More specifically, the implementation features both the iterative first- and best-improvement algorithms where redundancy elimination is applied after each search step. The outline of these algorithms will be rather briefly explained, since they were extensively covered in the previous implementation exercise.

The redundancy elimination procedure works by first sorting all columns $j \in J$ in a descending fashion based on their cost. If two columns have the same cost, they are sorted ascending based on the amount of rows they cover. The set of sorted columns is referred to as $J'$. Then, for each column $j \in J'$, if it is part of the current individual's solution $S_i$, it is checked whether there exists a row $i$ such that column $j$ is the only column covering this row. If this is not the case, column $j$ can be removed. By sorting the columns beforehand, this procedure removes the most costly redundant columns first.

The idea behind iterative improvement methods is to consider the neighbourhood of the current solution $S_i$. The goal is to find neighbours of the current solution which are of a higher quality. The iterative first improvement method will move on to the next iteration as soon as a single improving neighbour has been found. It will proceed to consider any of the other neighbours. The best improvement method, on the other hand, will first consider all neighbours of the current solution and proceed with the best one in the following iteration. This is repeated until no more improvement is achieved.

*Population Replacement* The final step of the genetic Algorithm is to introduce the generated offspring into the population. Two general approaches can be considered for this operation. The steady-state approach replaces a less fit individual from the population with the generated offspring, while the generational approach replaces the population entirely with the generated offspring. In this implementation, the steady-state approach is used. A random individual, which has above average fitness value, is chosen to be replaced by the offspring. Nevertheless, care should be taken to not introduce duplicates into the population. Before an individual is replaced by the offspring, it is compared to all individuals in the population. If the offspring is identical to an already existing individual, the process of parent selection, crossover and mutation is repeated all over again.

## Parameters

As was the case with the Ant Colony Optimization algorithm, so does the genetic algorithm have different parameters that are crucial to the algorithm's performance and the solution quality. A first parameter to consider is the population size. The authors of the original genetic algorithm studied the effect of the solution construction method on the coverage of the search space via the density. They concluded a population size of $N_d = 100$ should be sufficient to provide adequate coverage of the search space.

When using the tournament selection procedure for parent selection, it is important to properly determine the pool-size $T$ used by this procedure. The authors of the original algorithm claim to achieve comparable solution quality when using either proportionate selection or tournament selection with pool-size $T = 2$. Limited experimentation has shown that this is indeed the case.

The final parameters to consider are those responsible for the variable mutation rate, namely $m_g, m_c$ and $m_f$. As was already mentioned in the section dedicated to the variable mutation rate, these parameters should actually be tuned for each instance, or group of instances, separately. The authors of the original paper further generalized by setting $m_f = 10, m_c = 200$ and $m_g = 2$.

Due to the limited time frame of this implementation exercise, no extensive parameter tuning was done. The parameter values as suggested by the authors of the original algorithm were used to perform the experiments.

## Computational Results

The algorithms presented in this paper were implemented in C, using XCode, and tested on a MacBook Pro Retina (mid 2012) with a 2.3 GHz Intel Core i7 processor and 8GB of 1600MHz RAM. The computational experiments were done on 62 SCP instances from the OR-library (Beasley, 1990).

### Relative Percentage Deviation

In order to determine the average relative percentage deviation for both algorithms on each instance, 5 trials of both algorithms were ran, each with a different random seed. The random seed used was the amount of seconds since the 1st of January 1970, also known as the Unix Epoch. Since each instance varies in size and complexity, adequate runtime should be provided. The amount of runtime provided is 100 times the amount of time to construct a random solution and perform the iterative first improvement algorithm on the instance. However, since an optimal solution can be found much sooner than the provided runtime, the elapsed time after the optimal solution was found is also measured. The relative percentage deviation (RPD) is computed by the following formula:

$$\text{RPD} = 100 \times \left( \frac{cost(S_o) - cost(S_{best})}{cost(S_{best})} \right) \qquad (20)$$

where $S_o$ is the optimal solution found by the algorithm and $S_{best}$ is the best known solution.

All computational results are summarized in Table 1. Table 1 contains, for each instance:

- The optimal solution or best known solution value.
- The total allowed runtime.
- The cost of the best solution found by both algorithms.
- The average runtime, over 5 runs, needed to find this solution.
- The average relative percentage deviation (ARPD) from the optimal or best known solution value.

It can be noted that only one cost is reported, even though the algorithm was ran for 5 times, each time with a different random seed. The reason for this is that both algorithms always found exactly the same best solution. From this it follows that the percentage deviation with respect to the best known solution will also always be the same. An exception to this is the ARPD indicated with a $*$. For this instance, the ACO algorithm found a worse solution in one of the five iterations. However, the runtimes for each of the 5 runs were slightly different. So, Table 1 contains the average of these 5 runtimes.

A first thing that can be noted from this table is that the Ant Colony Optimization algorithm and genetic algorithm do not find the optimal solution for the same instances. In fact, the ACO algorithm finds the optimal solution slightly more often (55% of the instances) than the genetic algorithm (47% of the instances). This is especially clear in instance set 5. Nevertheless, the average relative percentage deviations over all instances, seen at the bottom of the table, are quite similar. Also noteworthy is instance set B, where both algorithms always find the optimal solution.

For the instances where both algorithms fail to find the optimal solution, it can be seen that the genetic algorithm always has an equal or higher ARPD than the Ant Colony Optimization algorithm, except for NRG.4 and NRH.1 through NRH.4. This would indicate that the genetic algorithm has an edge over the ACO algorithm when it comes to solving very large instances.

When it comes to runtimes, there is no real pattern that can be distinguished. When both algorithms find the optimal solution, one is not always faster than the other or vice-versa.

| | | | ACO | | | GEN | | |
|---|---|---|---|---|---|---|---|---|
| Instance | Optimal | Runtime | Cost | Runtime | ARPD | Cost | Runtime | ARPD |
| 4.2 | 512 | 2.13 | *512* | 1.41 | | *512* | 0.56 | |
| 4.3 | 516 | 2.45 | *516* | 0.43 | | *516* | 1.95 | |
| 4.4 | 494 | 2.10 | 495 | 0.28 | 0.20 | 502 | 1.09 | 1.62 |
| 4.5 | 512 | 1.98 | 514 | 0.28 | 0.39 | *512* | 1.80 | |
| 4.6 | 560 | 2.46 | *560* | 1.70 | | 561 | 1.67 | 0.18 |
| 4.7 | 430 | 1.76 | 432 | 1.16 | 0.47 | *430* | 0.98 | |
| 4.8 | 492 | 1.88 | 493 | 0.13 | 0.20 | 493 | 0.31 | 0.20 |
| 4.9 | 641 | 2.74 | 645 | 1.09 | 0.62 | 654 | 1.15 | 2.03 |
| 5.1 | 253 | 2.76 | *253* | 0.70 | | *253* | 1.30 | |
| 5.2 | 302 | 5.00 | 306 | 2.76 | 1.32 | 308 | 1.99 | 1.99 |
| 5.3 | 226 | 3.14 | *226* | 0.80 | | 229 | 0.38 | 1.33 |
| 5.4 | 242 | 5.09 | *242* | 0.23 | | 244 | 0.60 | 0.83 |
| 5.5 | 211 | 2.92 | *211* | 0.30 | | 212 | 0.63 | 0.47 |
| 5.6 | 213 | 2.93 | *213* | 0.30 | | *213* | 0.77 | |
| 5.7 | 293 | 4.09 | *293* | 2.77 | | 294 | 0.93 | 0.34 |
| 5.8 | 288 | 4.14 | *288* | 0.08 | | 290 | 0.70 | 0.69 |
| 5.9 | 279 | 2.83 | *279* | 0.88 | | 281 | 1.35 | 0.72 |
| 6.1 | 138 | 2.40 | *138* | 0.62 | | 141 | 0.35 | 2.17 |
| 6.2 | 146 | 2.79 | *146* | 0.73 | | 148 | 0.58 | 1.37 |
| 6.3 | 145 | 2.35 | *145* | 1.86 | | *145* | 0.56 | |
| 6.4 | 131 | 1.97 | *131* | 0.21 | | *131* | 1.43 | |
| 6.5 | 161 | 2.54 | 164 | 0.83 | 1.86 | *161* | 0.52 | |
| A.1 | 253 | 8.74 | 254 | 4.11 | 0.40 | 254 | 2.07 | 0.40 |
| A.2 | 252 | 9.01 | *252* | 7.75 | | 256 | 3.85 | 1.59 |
| A.3 | 232 | 6.27 | 233 | 2.50 | 0.43 | 234 | 1.96 | 0.86 |
| A.4 | 234 | 8.75 | *234* | 2.00 | | *234* | 1.07 | |
| A.5 | 236 | 10.97 | *236* | 1.44 | | 237 | 1.44 | 0.42 |
| B.1 | 69 | 8.31 | *69* | 1.21 | | *69* | 2.09 | |
| B.2 | 76 | 13.97 | *76* | 0.85 | | *76* | 1.35 | |
| B.3 | 80 | 11.89 | *80* | 3.54 | | *80* | 1.49 | |
| B.4 | 79 | 12.42 | *79* | 1.29 | | *79* | 2.91 | |
| B.5 | 72 | 9.09 | *72* | 0.43 | | *72* | 1.07 | |
| C.1 | 227 | 17.02 | *227* | 7.25 | | 231 | 10.71 | 1.76 |
| C.2 | 219 | 16.41 | *219* | 1.77 | | *219* | 10.53 | |
| C.3 | 243 | 17.09 | *243* | 14.40 | 0.08* | 251 | 2.33 | 3.29 |
| C.4 | 219 | 16.37 | 221 | 6.34 | 0.91 | *219* | 4.10 | |
| C.5 | 215 | 13.21 | *215* | 2.85 | | *215* | 2.22 | |
| D.1 | 60 | 21.86 | 61 | 1.82 | 1.67 | *60* | 4.00 | |
| D.2 | 66 | 19.44 | 67 | 2.55 | 1.52 | 67 | 2.62 | 1.52 |
| D.3 | 72 | 21.06 | *72* | 5.18 | | 73 | 2.89 | 1.39 |
| D.4 | 62 | 15.05 | *62* | 1.54 | | *62* | 2.74 | |
| D.5 | 61 | 23.23 | *61* | 3.33 | | *61* | 7.91 | |

| Instance | Optimal | Runtime | ACO Cost | ACO Runtime | ACO ARPD | GEN Cost | GEN Runtime | GEN ARPD |
|---|---|---|---|---|---|---|---|---|
| NRE.1 | 29 | 41.24 | *29* | 2.23 | | *29* | 3.28 | |
| NRE.2 | 30 | 37.43 | 31 | 4.35 | 3.33 | *30* | 22.21 | |
| NRE.3 | 27 | 28.32 | 28 | 1.48 | 3.70 | 28 | 4.71 | 3.70 |
| NRE.4 | 28 | 29.60 | 29 | 2.95 | 3.57 | *28* | 22.31 | |
| NRE.5 | 28 | 41.29 | *28* | 5.70 | | *28* | 3.92 | |
| NRF.1 | 14 | 34.76 | *14* | 1.94 | | *14* | 12.70 | |
| NRF.2 | 15 | 54.29 | *15* | 1.02 | | *15* | 7.04 | |
| NRF.3 | 14 | 32.58 | 15 | 3.93 | 7.14 | *14* | 29.32 | |
| NRF.4 | 14 | 37.87 | 15 | 2.95 | 7.14 | 15 | 7.11 | 7.14 |
| NRF.5 | 13 | 43.28 | 14 | 2.85 | 7.69 | 14 | 11.89 | 7.69 |
| NRG.1 | 176 | 91.38 | 177 | 52.51 | 0.57 | 180 | 6.11 | 2.27 |
| NRG.2 | 154 | 122.52 | 156 | 67.37 | 1.30 | 158 | 6.55 | 2.60 |
| NRG.3 | 166 | 118.94 | 169 | 55.91 | 1.81 | 170 | 39.22 | 2.41 |
| NRG.4 | 168 | 121.44 | 174 | 87.04 | 3.57 | 172 | 24.62 | 2.38 |
| NRG.5 | 168 | 148.41 | 169 | 86.56 | 0.60 | *168* | 19.07 | |
| NRH.1 | 63 | 140.36 | 65 | 50.41 | 3.17 | 64 | 68.48 | 1.59 |
| NRH.2 | 63 | 144.39 | 66 | 115.01 | 4.76 | 64 | 11.73 | 1.59 |
| NRH.3 | 59 | 178.36 | 62 | 37.06 | 5.08 | 61 | 12.87 | 3.39 |
| NRH.4 | 58 | 142.06 | 61 | 77.74 | 5.17 | 59 | 16.66 | 1.72 |
| NRH.5 | 55 | 136.43 | *55* | 26.47 | | 57 | 23.11 | 3.64 |
| **Average** | | | | | **1.11** | | | **1.05** |

Table 1: Computational Results

The instances present in the OR-library (Beasley, 1990) are grouped together in instance sets, as can be derived from Table 1. The instances in an instance set are rather similar. More specifically, all instances in the same instance set have the same number of rows $m$ and columns $n$. So, it makes sense to study the performance of both the Ant Colony Optimization algorithm and the genetic algorithm on the scale of instance sets. Fig. 1 plots the ARPD values of both algorithms averages over all instances per instance set. This plot provides an easy way to compare both algorithms.

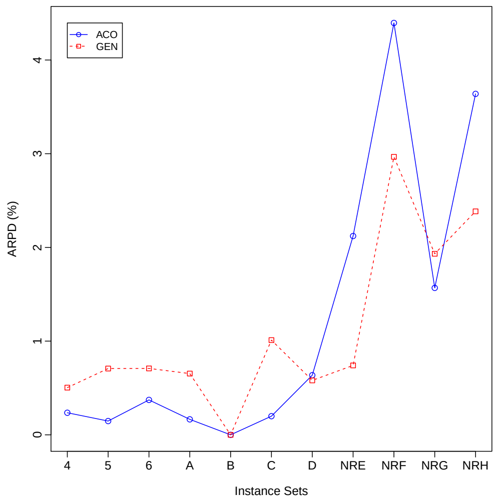

Figure 1: ARPD values for both algorithms, grouped per instance set

In fact, Fig. 1 confirms what was already found from the data in Table 1. More precisely, it can be seen from this plot that the genetic algorithm indeed does have an advantage over the ACO algorithm when it comes to solving large instances. This can be concluded since the ARPD values for the genetic algorithm are always lower than those for the ACO algorithm for the instance sets with large instances.

The same can be done for analysing the runtimes of both algorithms. Fig. 2 displays the runtimes of both algorithms, grouped per instance set. As was already concluded from the data in Table 1, there is no real pattern to distinguish for instance sets 4 through 6 and A through D. However, larger differences occur when looking at the instance sets with large instances. A difference in runtime of approximately 10 seconds in favour of the ACO algorithm can be noted for instance sets NRE and NRF, while a difference of respectively 50 and 30 seconds can be noted for instance sets NRG and NRH in favour of the genetic algorithm.

A possible explanation for this phenomenon could be that the runtime averaged over the instance sets is highly influenced by a single instance with a consistently poor runtime. Examples of this in Table 1 are for example instances NRE.2, NRE.4 and NRF.3 for the genetic algorithm and instance NRH.2 for the ACO algorithm. The high runtime value for instance set NRG seems appropriate for the ACO algorithm, since every instance in this set has a long runtime.

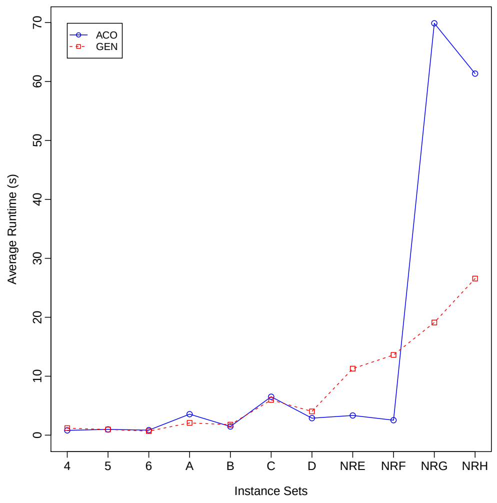

Figure 2: Average runtime for both algorithms, grouped per instance set

Finally, it is necessary to evaluate whether or not there is a statistical significant difference between the solution qualities generated by both algorithms. In order to do this, the Wilcoxon signed-rank test is used, with a significance level of $\alpha = 0.05$. This test is preferred over the Student's $t$-test, since the $t$-test assumes the underlying data to be normally distributed. The $p$-value obtained after executing the Wilcoxon test is $0.72$, meaning there is not enough evidence to conclude that the solution qualities generated by both algorithms are significantly different.

## Correlation

Another way of comparing the performance of the Ant Colony Optimization algorithm and genetic algorithm is to consider the ensemble of instances on which both algorithms have been tested. Since the instance ensemble is quite large (62 instances), a correlation plot is used for this purpose. Each point in Figure 3 corresponds to an instance, where the $x$ and $y$ values represent the relative percentage deviation for the ACO and genetic algorithm, respectively.

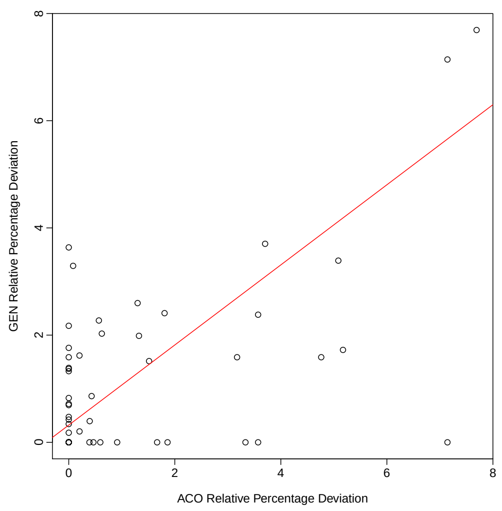

Figure 3: Correlation between RPD values of ACO and Genetic Algorithm (Spearman's $\rho = 0.41$).

A first thing that can be noted is that there are quite a few instances for which one of the algorithms finds the optimal solution, whilst the other does not. These instances have an $x$ value of 0, in case ACO found the optimal solution, or a $y$ value of 0, in case the genetic algorithm found the optimal solution.

To quantify the correlation between the relative percentage deviations of both algorithms, Spearman's $\rho$ is used. This returns a value of $\rho = 0.41$, meaning that the correlations are positively correlated. However, this correlation is not very strong nor very weak. Spearman's $\rho$ is preferred over Pearson's correlation coefficient, since the latter assumes a normal distribution. Also, Pearson's correlation coefficient requires a linear relationship, while Spearman's $\rho$ only requires a monotonic relationship. This is a less restrictive assumption. Furthermore, using Spearman's rank order test, with a significance level of $\alpha = 0.05$, one finds that this correlation is indeed significant ($p = 9.15 \times 10^{-4}$).

## Qualified Runtime Distributions

To further test and compare the performance of both implemented algorithms, several qualified runtime distributions (QRTD) are created. A QRTD can be formally described as follows: for optimization algorithms, the run-time (RT) and solution quality (SQ) are random variables. For a given algorithm $A$ and optimization problem $\Pi$:

- The success probability $P_s(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q)$ is the probability that algorithm $A$ finds a solution to instance $\pi \in \Pi$ of a quality $\leq q$ in time $\leq t$.
- The run-time distribution (RTD) of algorithm $A$ on $\pi$ is the probability distribution of the bivariate random variable $(RT_{A,\pi}, SQ_{A,\pi})$.
- The run-time distribution function $rtd : \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow [0,1]$, defined as $P_s(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q)$, completely characterises the RTD of $A$ on $\pi$.
- A qualified run-time distribution (QRTD) of an algorithm $A$ applied to a given problem instance $\pi$ for solution quality $q$ is a marginal distribution of the bivariate RTD $rtd(t,q)$ defined by $qrtd_{q'}(t) := rtd(t,q') := P_s(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q')$.

Often, the required solution quality $q$ is expressed as the relative solution quality.

| Instance | Runtime |
|---|---|
| A.1 | 87.41 |
| B.1 | 83.15 |
| C.1 | 170.18 |
| D.1 | 218.57 |

Table 2: Allowed runtimes for generating the QRTD

In order to create these QRTD, four instances are chosen. These are instance A.1, B.1, C.1 and D.1. Both algorithms are ran, for 25 repetitions, on these four instances with an allowed runtime of 10 times the runtime described in the previous experiment. These allowed runtimes are summarized in Table 2. The relative solution qualities required for these QRTD's are either the optimal solution or 0.5%, 1% or 2% deviation, depending on the algorithm's capabilities on the specified instance.

*Instance A.1* Figures 4 and 5 represent the qualified runtime distributions for the ACO algorithm and genetic algorithm, respectively, on instance A.1. Both algorithms achieve a relative percentage deviation of 0.5% on this instance. From these figures, it can be seen that the genetic algorithm needs at least 2.3 seconds to consistently reach the specified solution quality, while the ACO algorithm requires at least 4.6 seconds, meaning the genetic algorithm is twice as fast on this instance. In fact, the genetic algorithm completely dominates the ACO algorithm for instance A.1.

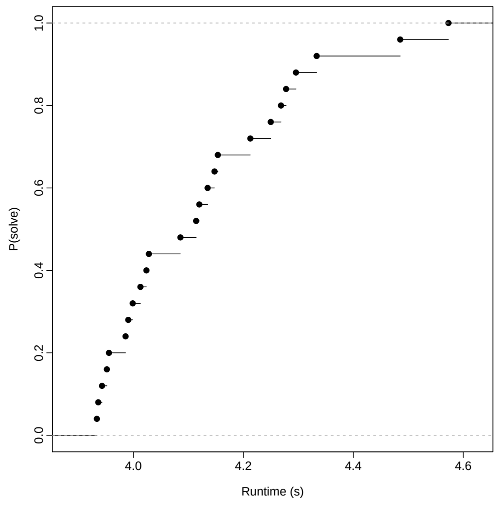

Figure 4: QRTD on instance A.1 for ACO with RPD of 0.5%

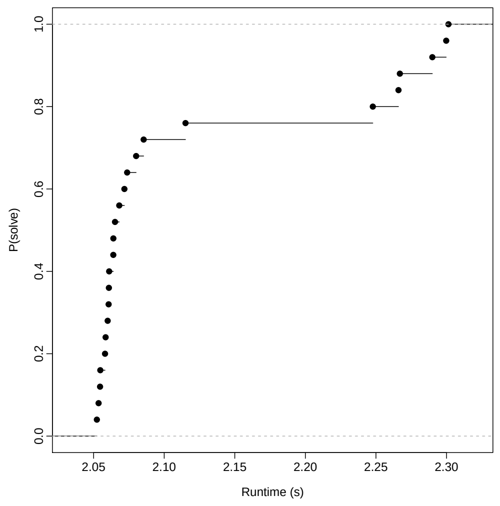

Figure 5: QRTD on instance A.1 for GEN with RPD of 0.5%

However, it should be noted that both algorithms require only a fraction of the allowed runtime (i.e. 87.41 seconds as can be seen from Table 2) to consistently achieve a 0.5% relative percentage deviation. Unfortunately, neither of the algorithms is able to find the optimal solution in the allowed runtime.

A final remark that can be made regarding the genetic algorithm is the following: there is quite a large gap between solving the instance in approximately 75% of the cases and approximately 80% of the cases. This can be seen by the long horizontal line on Figure 5. If one wants to solve instance A.1 with a probability of 75% or less, the genetic algorithm should only run for 2.1 seconds, while a runtime of at least 2.25 seconds is required for a higher probability of solving the instance. Of course, calling this time difference large only makes sense when comparing it relatively to the entire runtime distribution.

*Instance B.1* Figures 6 and 7 sketch a completely opposite image. On this instance, both algorithms arrive at the optimal solution in the allowed runtime, but the ACO algorithm completely dominates the genetic algorithm. In order to solve instance B.1 with probability 1, the ACO algorithm requires a mere 1.35 seconds, while the genetic algorithm has to run for at least 2.3 seconds.

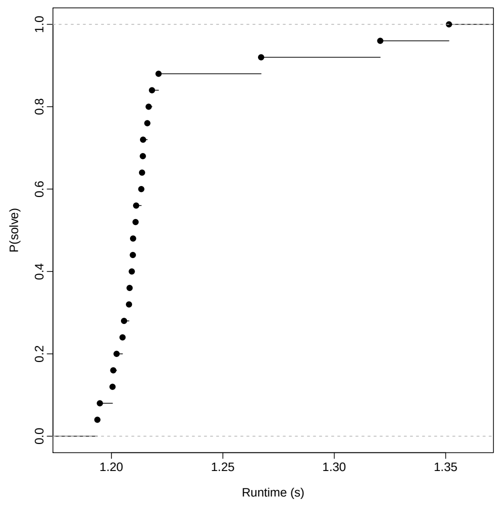

Figure 6: QRTD on instance B.1 for ACO with RPD of 0%

It can be noted from Figure 6 that the ACO algorithm's probability for solving this instance climbs very rapidly. While the instance is unsolvable in under 1.19 seconds, it is solvable in approximately 90% of the cases after 1.22 seconds. However, adding the final 10% to reach the optimal solution consistently requires a lot more time. The same trend is visible for the genetic algorithm, in Figure 7, but less pronounced. Going from

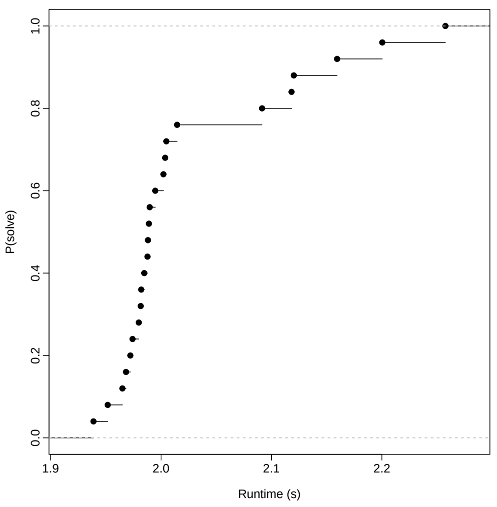

Figure 7: QRTD on instance B.1 for GEN with RPD of 0%

a probability of 0 to 0.75 only requires approximately 0.05 seconds, while adding the final 25% to consistently solve the instances requires an additional 0.3 seconds.

Finally, it can again be seen that the runtime required for both algorithms to solve the given instance with a probability of 1 (1.35 and 2.3 seconds respectively) is far less than the total allowed runtime of 83.15 seconds. (see Table 2).

*Instance C.1* Figures 8 and 9 represent the qualified runtime distributions for both algorithms on instance C.1. Both of these plots represent the QRTD with a relative percentage deviation of 1%, however, the ACO algorithm is able to consistently find the optimal solution on this instance. This can be seen in Figure 12 in the Appendix.

Again, from the plots, it is clear that one algorithm completely dominates the other for the given instance. The ACO algorithm can consistently achieve a 1% relative percentage deviation after 1.7 seconds, while the genetic algorithm needs at least 90 seconds to solve the instance (with a very low probability). This is a very large difference between the algorithms. Even when comparing the runtime distribution of the genetic algorithm, with 1% relative percentage deviation (Figure 9) to the distribution of the ACO algorithm, which finds the optimal solution (Figure 12), one can see there is a very large difference in required runtime even though finding the optimal solution is harder than finding a

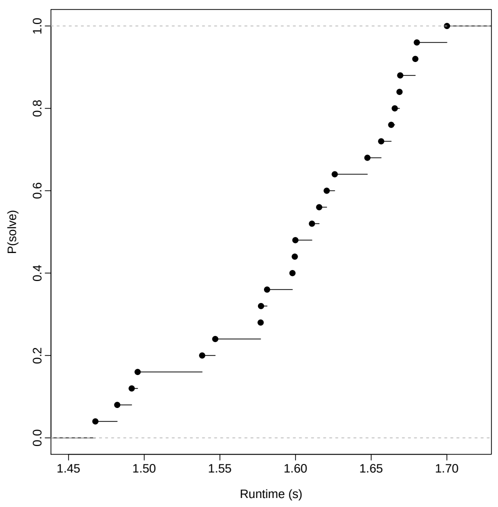

Figure 8: QRTD on instance C.1 for ACO with RPD of 1%

solution with a 1% relative percentage deviation.

Both algorithms have a quite steady runtime distribution. The only thing that can be noted, from Figure 9, is that approximately 5 additional seconds of runtime are needed to bring the probability of solving the instance from 90% to 100% certainty.

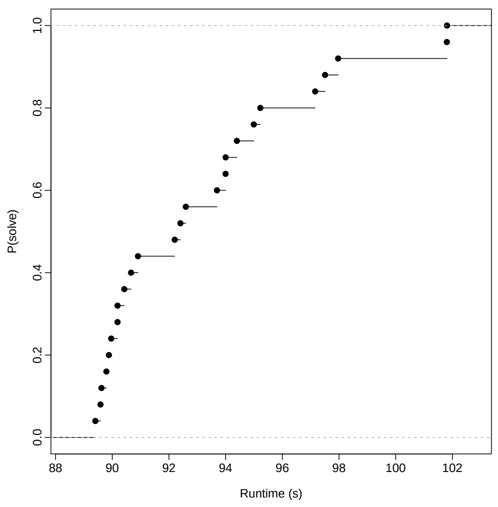

Figure 9: QRTD on instance C.1 for GEN with RPD of 1%

*Instance D.1* The final benchmark instance is instance D.1. Figures 10 and 11 represent the QRTD's for this instance, where a 2% relative percentage deviation is achieved. It is clear from these plots that the ACO algorithm dominates the genetic algorithm, when it comes to solving this instance. The ACO algorithm is able to solve the instance consistently when provided with 2.2 seconds runtime, while the genetic algorithm needs at least 3 seconds to solve the instance with a small probability.

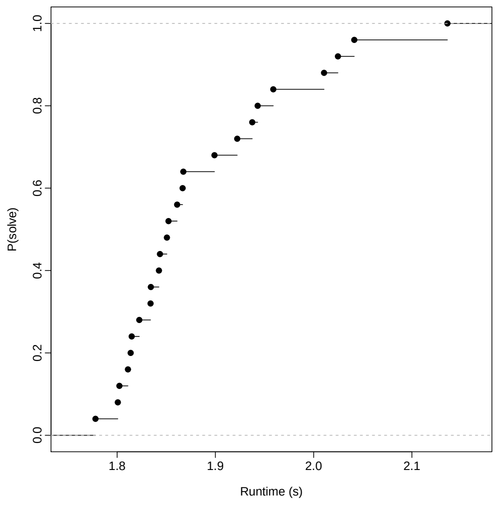

Figure 10: QRTD on instance D.1 for ACO with RPD of 2%

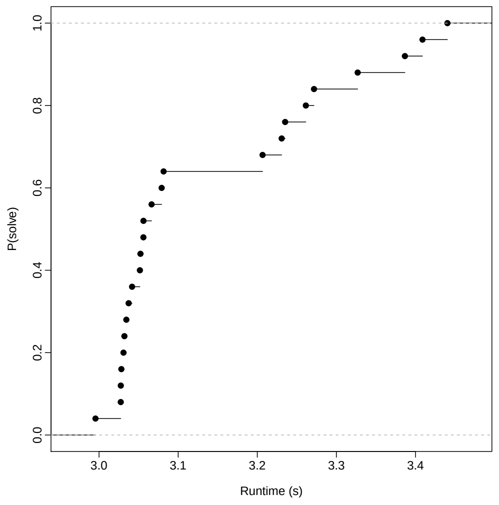

Figure 11: QRTD on instance D.1 for GEN with RPD of 2%

Another remark that can be made is that the runtime required to achieve a 2% relative percentage deviation is only a very small fraction of the total allowed runtime, which is 218.57 seconds (see Table 2). However, while the ACO algorithm is only able to solve this instance with a 2% relative percentage deviation, the genetic algorithm can find the optimal solution. This can be seen in the QRTD in Figure 13, present in the Appendix. The genetic algorithm takes slightly longer to reach the optimal solution than it does to reach a relative percentage deviation of 2%. It can be concluded that the genetic algorithm is better than the ACO algorithm for this instance, since it is able to find the optimal solution even though it takes twice as long as for the ACO algorithm to find the solution with 2% RPD.

## Conclusion

This paper has presented two algorithms for solving the Set Covering Problem. The first is an Ant Colony Optimization method based on the paper by Ren et al. (2010) and the second is a version of the genetic algorithm by Beasley and Chu (1996) modified to become a memetic algorithm. Both algorithms were empirically evaluated on a set of 62 instances from the OR library, together with statistical significance tests, correlation plots and qualified runtime distributions.

## Appendix: Running the Code

The code accompanying this report can be ran in two ways. Firstly, it can be ran manually from the command line. Since the same entry point is used for both implemented algorithms, there are several command line parameters to consider. In order to run the ACO algorithm, the following should be executed in the command line:

```
  ./lsscp.c --seed seed-value
--output /path/to/output-file
--instance /path/to/instance
--runtime #seconds
--aco
--ac colony-size
--beta beta-value
--ro rho-value
--epsilon epsilon-value
[--fi | --rep | --bi]
```

The last argument in the command above specifies which local search algorithm to use. FI stands for first improvement, BI for best improvement and REP stands for the column replacement method. A similar command is needed to run the genetic algorithm manually:

```
  ./lsscp.c --seed seed-value
--output /path/to/output-file
--instance /path/to/instance
--runtime #seconds
```

```
--ga
--pops population-size
--mf m_f-value
--mc m_c-value
--mg m_g-value
[--tour --pool pool-size | --prop]
[--fusion | --uniform]
[--fi | --bi]
```

The final three arguments in the command specified above indicate which parent selection operator, crossover operator and local search operator to use.

A simpler way to execute the attached code is to use provided the Bash script. This script will run both algorithms, five times, over all instances, each time with a different seed for the random number generator. Additionally, separate files for each run will be created automatically. This script requires 4 arguments and can be ran like so:

```
./run.sh /path/to/executable
/path/to/instances-folder
/path/to/output-folder
/path/to/runtimes-file
```

The first argument should be the build resulting from the compilation of the code. The second argument is the folder containing all instances. The third argument is the folder where the output files will be created and the final argument is a file specifying the allowed runtime for each instance.

## Appendix: Qualified Runtime Distributions

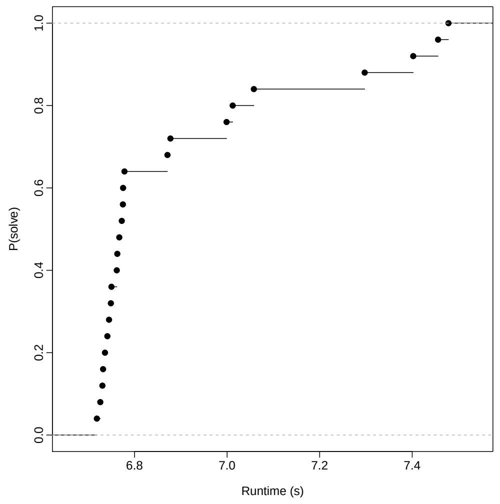

Figure 12: QRTD for ACO on instance C.1 (RPD = 0%)

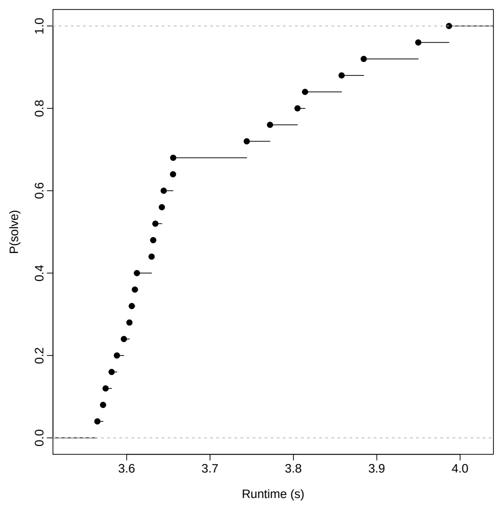

Figure 13: QRTD for GEN on instance D.1 (RPD = 0%)

## References

Beasley, J. E. (1990). A lagrangian heuristic for set-covering problems. *Naval Research Logistics (NRL)*, 37(1):151–164.

Beasley, J. E. and Chu, P. C. (1996). A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94(2):392–404.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41.

López-Ibánez, M., Dubois-Lacoste, J., Stützle, T., and Birattari, M. (2011). The irace package, iterated race for automatic algorithm configuration. Technical report, Citeseer.

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.

Stützle, T. and Hoos, H. H. (2000). Max–min ant system. *Future generation computer systems*, 16(8):889–914.