# Machine Learning (course 1DT071)
# Uppsala University – Spring 2016
# Project Report by Team 17
## Solving the Square Packing Problem using Population Methods

Alexander Ek Sebastian Rautila Mikael Östlund

13th June 2016

## Contents

## 1 Introduction

The Square Packing Problem[1] (SPP) is a combinatorial optimisation problem where $n$ increasingly sized squares are to be placed in a $s \times s$ enclosing square. More formally, given a number $n > 1$, the SPP consists of finding a placement for $n$ increasingly sized squares $(1 \times 1, 2 \times 2, \ldots, n \times n)$ inside an enclosing square of size $s \times s$. The squares are placed at whole number coordinates such that their boundaries do not lie outside of the enclosing square, and none of the squares overlap. (Touching boundaries does not imply overlap.) We call such a placement of squares a *packing*, and an *incorrect packing* otherwise. The size of the enclosing square is to be minimised. A packing with a minimal $s$ is called a *minimal packing*. In Figure 1, a packing, in particular a minimal packing, of 5 squares can be visualised, where the numbers in the squares denote the size of that square.

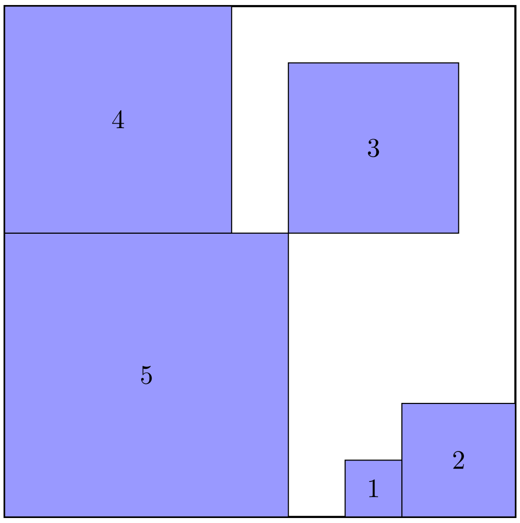


Figure 1: A visualisation of a packing of 5 squares.

In this report we measure and examine the performance of two population methods in the realm of machine learning in context of SPP. Namely Genetic Algorithms (GA) and Particle Swarm Optimisation (PSO). For each method we measure average execution time for each $n$ as well as maximal $n$ reached given a timeout. The implementation of both were done using Python in combination with the `DEAP` (*Distributed Evolutionary Algorithms*) library [1].

Section 2 introduces the two population methods used, both GA and PSO. In Section 3, the chosen representation of SPP in both GA and PSO is presented as well as different fitness functions (that evaluate a placement) that are used in the experiments. Following this, details about different mutation and crossover operators used in the context of genetic algorithms are presented in Section 3.3. Following this, in Section 3.4 we go through the problem specific optimisations used in the experiments. The results of the experiments are then presented in

[1]A variant of the Rectangle Packing Problem [8].

Section 4 followed by a discussion of these in Section 5. Thereafter, in Section 6, we introduce and discuss some of the possible future work and questions this project raises. The report is then concluded in Section 7.

## 2 Background

In this section a brief explanation of both Genetic Algorithms and Particle Swarm Optimisation is provided in order to ease the understanding of this report.

In Genetic Algorithms (GA), a *population* of *genomes* is used to find an optimum for some function or problem [6]. Each genome represents a candidate solution to the problem (SPP). Genomes can be combined through crossover operators, that is, different properties of a pair of genomes are in some pre-defined way mixed to create a new genome. Genomes can also be mutated through pre-defined mutation operators. Some simply mutate a genome in a stochastic way while other may be user-defined, optimised for a specific problem.

In Particle Swarm Optimisation (PSO) a swarm of particles is constructed and used to find a solution to some defined problem [7]. The swarm is a set of particles where each particle represents a candidate solution to the problem (SPP). Each candidate solution is then updated in a stepwise manner, where each complete iteration of the update loop is called a *generation*.

The solution represented by a particle is called the particles *position* and is for regular PSO defined as a $\mathbb{R}$ vector where $d$ is the number of dimensions of the problem. In our case each dimension represents the $x$- or $y$-coordinate of a square so for SPP with $n$ squares the position will have $2n - 2$ dimensions. A positions quality is measured by a fitness function which takes the position as an argument and returns a real valued measurement of its quality.

For each generation the particles moves through the search space. For the implementation used by us [7], there are two factors which affects the movement of each particle. Each particle in the swarm moves towards it's own, across all generations, best known position. This is known as the particles *personal best* position. All particles in the swarm also moves toward the best known position, across all generations, for all particles in the swarm. This is known as the *global best* position.

The movement of a particle is based on its velocity of $i$ in dimension $d$ and defined as follows:

$$v_{id}(t) = v_{id}(t-1) + \phi_1 r_1 (p_{id} - x_{id}(t-1)) + \phi_2 r_2 (p_{\gamma d} - x_{id}(t-1)),$$

where:

- $r_1$ and $r_2$ are random numbers sampled from two (possibly different) uniformly distributed stochastic variables.
- $t$ is a moment in time.
- $\gamma$ is the index of the global best particle.
- $\phi_1$ is the cognitive factor used in the swarm, that is, how much emphasis should be put on each swarm particle's personal best in the velocity calculation.

- $\phi_2$ is the social factor used, this configures how much emphasis that is put on the social component, such as personal best, local best or global best heuristic.
- $x_{id}$ is particle $i$'s position in dimension $d$ at iteration $t-1$.

## 3 Representation and Design

In this section we describe different design decisions such as choice of fitness function(s) and we present the representation chosen for SPP in both GA and PSO.

### 3.1 Representation

For the square packing problem proposed we have one square of each size from 1 to $n$ which is represented by a pair of coordinates $(x, y)$ which is the bottom left coordinate of the square. Each square will be referenced to as $S_i$ where $S_i = (x_i, y_i)$ is the bottom left coordinate of the square of size $i$.

$$\boxed{x_1}\boxed{y_1}\underbrace{\boxed{x_2}\boxed{y_2}}_{S_2}\boxed{\dots}\boxed{x_n}\boxed{y_n}$$

Figure 2: Representation of a genome (particle) in our GA (PSO) implementation. The coordinates for square 2 ($S_2$) are underbraced.

A genome/particle is represented as $[x_1, y_1, \dots, x_n, y_n]$ as illustrated in Figure 2, where $(x_1, y_1)$ is the position of the $1 \times 1$ square $S_1$, $(x_2, y_2)$ is the position of the $2 \times 2$ square $S_2$, etc. We will call this an *individual* and for any individual $S$, $|S|$ will denote the number of squares to be packed. Mathematically, $S$ will be resresented as $S = [S_1, \dots S_n]$, and every $S_i \in S$ is represented as $S_i = (x_i, y_i)$.

### 3.2 Fitness Functions

We consider two approaches when minimising the size of the enclosing square $s$: successively decreasing $s$, and considering $s$ during fitness calculation. First, the algorithm is run on a fixed $s$ and tries to find a packing. If a packing is found, then the algorithm is executed once again but with $s - 1$. Secondly, we will use a fitness function which considers the size of the minimal possible enclosing square and the how far from a packing the placement is. These will both be used, in combination, to simultaneously find a minimal $s$ and a feasible solution for that $s$. The first approach may be better at finding a solution since $s$ is fixed.

The term *overflow* is used to denote that some part of a square lies outside of the enclosing square, and the term *overlap* is used to denote that some area of the enclosing square is occupied by more than one square (i.e. two squares overlap).

**Fixed $s$.** The first approach is to fix $s$ and simply calculate the total overflow and total overlap of the placement. In this approach, $s$ would be iteratively decreased each time a packing is found, and the hope is that a minimal packing would be found this way. This fitness function can be defined as follows:

$$\mathsf{fitness_F}(S, s) := \sum_{i=1}^{|S|} \left( \mathsf{overflow}(S_i, s) + \sum_{j=i+1}^{|S|} \mathsf{overlap}(S_i, S_j) \right),$$

where $s$ is the size of the enclosing square. Furthermore, $\mathsf{overflow}$ denotes the cost of the current overflow of a square, and $\mathsf{overlap}$ denotes the cost of the current overlap between two squares. How the functions $\mathsf{overflow}$ and $\mathsf{overlap}$ are defined will be discussed in the following sections.

**Dynamic $s$.** Instead of a using a fixed $s$, the minimal possible $s$ will be calculated from a placement itself. This approach, then, calculates the fitness of a placement using the overlap as well as how big the smallest possible enclosing square is. The fitness is calculated as follows:

$$\mathsf{fitness_D}(S) := \mathsf{fitness_F}(S, \mathsf{size}(S)) \cdot (\mathsf{void_{max}}(|S|) + 1) + \mathsf{void}(S),$$

where $\mathsf{size}(S)$ denotes the smallest possible enclosing square of the placement $S$, and is defined as:

$$\mathsf{size}(S) := \max\left( \max_{i=1}^{n} x_i,\ \max_{i=1}^{n} y_i \right).$$

Furthermore, $\mathsf{void}(S)$ denotes the area (or the cost) of the unoccupied space in placement $S$, and $\mathsf{void_{max}}(S)$ denotes the maximal value of $\mathsf{void}(S')$ such that $|S| = |S'|$. These are defined as:

$$\begin{aligned} \mathsf{void}(S) &:= \mathsf{size}(S)^2 - \mathsf{squarea}(S) + \mathsf{fitness_F}(S, \mathsf{size}(S)), \\ \mathsf{void_{max}}(S) &:= \mathsf{size}(S)^2 - \mathsf{squarea}(S) + \frac{n(n+1)}{2}, \end{aligned}$$

where $\mathsf{squarea}(S)$ denotes the total area of $|S|$ increasingly sized squares, and is defined as:

$$\mathsf{squarea}(S) := \left\lceil \sqrt{\frac{|S|(|S|+1)(2|S|+1)}{6}} \right\rceil.$$

### 3.2.1 Overlap and Overflow

In this section, different definitions of $\mathsf{overlap}$ and $\mathsf{overflow}$ will be established. These definitions will be used in combinations with the different fitness functions described in Section 3.2, effectively create multiple fitness functions. Note, that for each of these definitions, we will simply refer to its name when it is used with $\mathsf{fitness_F}$ and with an iteratively decreasing $s$, and its name with the word 'Dynamic' in front of it when referring to it used with $\mathsf{fitness_D}$.

**Area Overlap.** Starting with the most intuitive which is simply the total area of the overlap between all $n$ squares. In this case, $\mathsf{overlap}$ and $\mathsf{overflow}$ determines the area of overlap between two squares and the area of the overflow, respectively, and is defined as:

$$\mathsf{overlap}(S_i, S_j) := \mathsf{overlap}'(x_i, i, x_j, j) \cdot \mathsf{overlap}'(y_i, i, y_j, j),$$

where $S_i = (x_i, y_i)$ and is of size $i \times i$, analogous for $S_j$. And were:

$$\mathsf{overlap}'(x_i, i, x_j, j) := \max\left(0, \min(x_i + i, x_j + j) - \max(x_i, x_j)\right).$$

Furthermore, overflow is a function which determines the area of the part of the square that is outside the enclosing square, and is defined as:

$$\mathsf{overflow}(S_i, s) := i^2 - \mathsf{overlap}'(x_i, i, 0, s) \cdot \mathsf{overlap}'(y_i, i, 0, s),$$

where, again, $S_i = (x_i, y_i)$ and is of size $i \times i$.

**Enumerative Overlap.** With enumerative overlap, the number of overlaps and the number of overflows determine the cost of a placement of a square. In this case, $\mathsf{overlap}$ determines whether the there is an overlap between two squares, and $\mathsf{overflow}$ whether a square is outside of the enclosing square. The function $\mathsf{overlap}$ is defined as follows:

$$\mathsf{overlap}(S_i, S_j) := \mathsf{overlap}'(x_i, i, x_j, j) \cdot \mathsf{overlap}'(y_i, i, y_j, j),$$

where $S_i = (x_i, y_i)$ and is of size $i \times i$, analogous for $S_j$. And were:

$$\mathsf{overlap}'(x_i, i, x_j, j) := \begin{cases} 1 & \text{if } \min(x_i + i, x_j + j) - \max(x_i, x_j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, $\mathsf{overflow}$ is defined as:

$$\mathsf{overflow}(S_i, s) := i^2 - \mathsf{overlap}'(x_i, i, 0, s) \cdot \mathsf{overlap}'(y_i, i, 0, s),$$

where, again, $S_i = (x_i, y_i)$ and is of size $i \times i$.

**Big Expensive.** Big Expensive works very similarly to Area Overlap. The main difference is that every overflow is also multiplied by the size of the square that overflows and every overlap is multiplied by the the size of the biggest of the two squares. Making an overlap containing a bigger square more expensive compared to an overlap containing a small square.

### 3.2.2 Distance Fit

This function is an extension of $\mathsf{fitness}_\mathsf{F}$ with Area Overlap that attempts to give a slope to the function for more effective usage, primarily in PSO. As Area Overlap, Distance Fit calculates a weighted sum of the total area of the overlap of all squares and the total area of the overflow of the squares and the enclosing square. The new feature added to the weighted sum in Distance

Fit is, for each pair of squares that overlap, the shortest distance they need to be moved from each other to not overlap. Of course, moving all overlapping squares away from each other does not guarantee a packing. The intention is simply to add a way to favour particles which are more likely to be close to a packing, e.g. where only some squares need to be moved a very short distance to form a packing.

The shortest distance for each pair is calculated by examining the $x$- and $y$-coordinates of the pairs separately and choosing the shortest possible distance in both axes which separates the squares. For each axis, the movement distance needed is defined by the type of the overlap between the squares and is one of the three cases illustrated in Figure 3, 4 and 5 where the dotted line illustrates the movement needed. As seen in Figure 5, there are two cases when the smaller square is fully enclosed by the larger one. In that case the distance is simply the shorter of the two. The final distance used by the function is then chosen as the shortest of the distances for each axis.

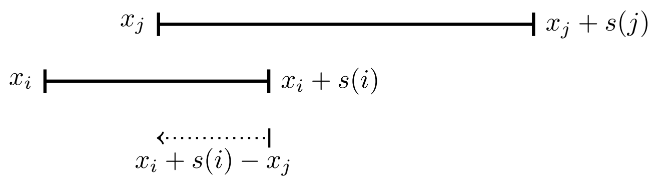


Figure 3: Smaller square is overlapping with the left of larger square

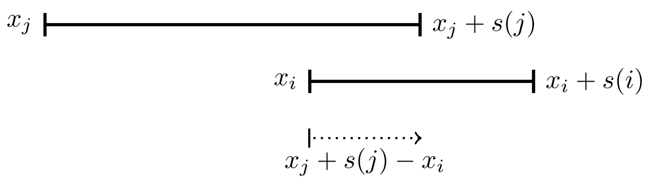


Figure 4: Smaller square is overlapping with the right of larger square

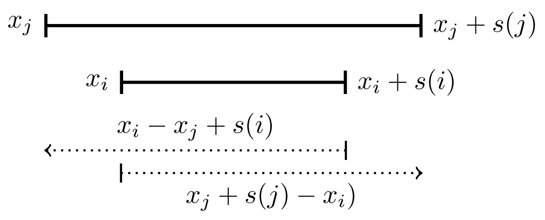


Figure 5: Smaller square is inside larger square

From the three cases we can define a mathematical helper function taking two coordinates of squares of size $i$ and $j$ where $i < j$:

$$\mathsf{distance}'(x_i, x_j) := \begin{cases} x_i + i - x_j & \text{if } x_i < x_j \\ x_j + j - x_i & \text{if } x_j + j < x_i + i \\ \min(x_i - x_j + i, x_j + j - x_i) & \text{otherwise} \end{cases} \tag{1}$$

Using this we can define the final shortest distance for a pair of squares $S_i := (x_i, y_i)$ and $S_j := (x_j, y_j)$ as:

$$\mathsf{distance}(S_i, S_j) := \begin{cases} \min(\mathsf{distance}'(x_i, x_j), \mathsf{distance}'(y_i, y_j)) & \text{if } \mathsf{overlap}(S_i, S_j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

which gives the final fitness of a candidate solution with Distance Fit as the weighted sum:

$$\sum_{i=1}^{|S|} \left( 3 \cdot \mathsf{overflow}(S_i, s) + \left( \sum_{j=i+1}^{|S|} 2 \cdot \mathsf{overlap}(S_i, S_j) + \mathsf{distances}(S_i, S_j) \right) \right)$$

### 3.2.3 Grid Fit

Grid Fit is another extension of $\mathsf{fitness}_\mathsf{F}$ with Area Overlap, also used to construct a slope in fitness towards better solutions. For a candidate solution, Grid Fit constructs an $s \times s$ grid and fills all spots which are occupied by a square. For each square which is overlapping with some other larger square, the shortest distance in a straight line from the smaller square to a free spot in the grid is calculated.

In order to reduce the amount of computations needed for each overlap, the distances are calculated by using the middlemost spot of each edge of the smaller square and then stepping in the grid in the corresponding direction until until a empty spot is found. For example, in the right direction we start at the right edge of the square, on the spot in the grid which is closest to the middle of the square (rounded up) and move in a straight line until we find a free spot in the grid.

The python implementation of for Grid Fit's distance calculation is seen in Appendix A. The sum of all minimal distances calculated is then used in a weighted sum for the final fitness together with overlap and overflow, as defined in Area Overlap in Section 3.2.1, which carry double and triple weight respectively over the sum of distances.

In comparison with Distance Fit which examines all pairs of squares in a candidate solution, Grid Fit only examines each square once. However, since a grid needs to be constructed, Grid Fit is still a costly fitness function to run. Grid fit also only covers the case from Distance Fit where the smaller square is totally enclosed by the larger. If a square is right next to an empty slot in the grid the distance will be 1 even though the square needs to be moved further to not overlap. This is intentional as overlap will then instead be the main factor in evaluating the final fitness of a candidate solution (e.g. if a square is moved so that the overlap is smaller the fitness will still be smaller).

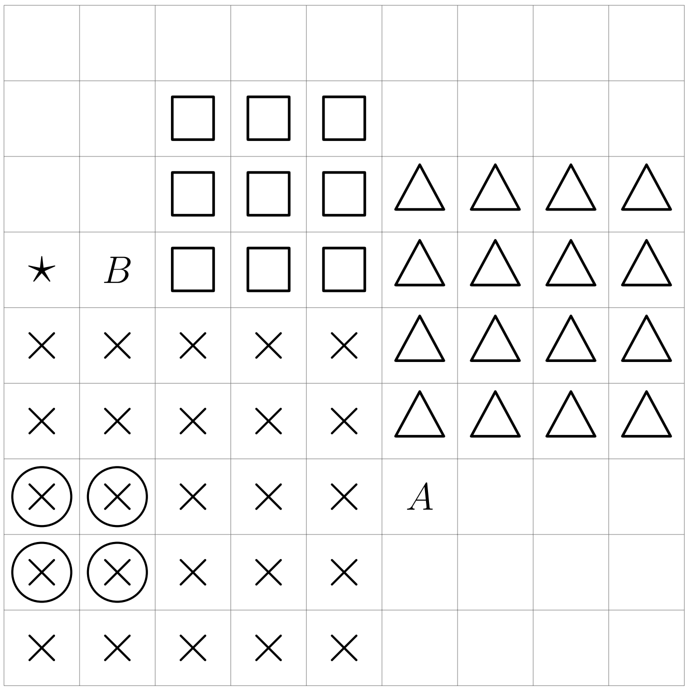


Figure 6: Example grid constructed by Grid Fit

An example grid is given in Figure 6. Let $C$ be the $2 \times 2$ square symbolised by circles. Since $C$ is the only square which overlaps, the distance sum calculated by Grid Fit will only depend on $C$. When calculating the distance for $C$, Grid Fit will not find any way out in the left or down direction of the grid. In the right direction, the approximate middle of $C$ will be the upper right corner. Grid Fit will find that the distance to the first empty spot $A$ is 4 steps. In the up direction Grid Fit will find $B$ with a distance of 3 steps. So even though moving the square upwards will require as many steps as in the right direction for the square to not overlap, Grid Fit will favour the up direction. This is a trade-off for avoiding to calculate the distances multiple times. The intention of Grid Fit is merely to give a slope to the fitness function towards what might be a better placement, not to present an actual solution to the SPP.

## 3.3 Genetic Algorithms

In contrast to gbest PSO, for which the behaviour can only be altered via the choice of fitness function, cognitive factor and social factor, GA can be altered to a much higher degree. In this section we present the different mutation and crossover operators explored in the context of this report.

### 3.3.1 Mutations

Two mutation operators are explored in the experiments, one custom made as well as one already included in DEAP. All mutations explored in this report uses some stochastic elements, these are modified through the indpb parameter.

**Local Gaussian Mutation.** This custom-made mutation operator moves each coordinate of an individual in a locally gaussian way. More exactly, each coordinate in an individual is moved in such a way that the expected value (or *mean value*) of the gaussian function is the coordinate's current value. The standard deviation is configurable via input argument but is usually set to $\frac{s}{4}$, where $s$ is the side length of the enclosing square.

This mutation operator also consider the borders of the enclosing square so that all square coordinates are always inside the enclosing square, as other placements are guaranteed to be infeasible.

Each coordinate has a probability of being moved in this way, this probability is configurable as an input argument called indpb and is set to 0.15 as default.

**Uniform Integer Mutation.** This mutation[2] is included in the DEAP package and will, for each coordinate, assign it a value uniformly chosen between low and up. Each coordinate has a probability of indpb of being replaced.

**Shuffle Indexes** The shuffle mutation[3] is included in the DEAP package and shuffles the attributes of a genome. Where each attribute has a indpb probability of being moved.

### 3.3.2 Crossovers

Apart from mutations, a couple of crossover operators were also explored, some custom made as well as some already included in DEAP.

**Two Point Crossover.** This crossover operator[4] is included in DEAP and chooses two cutoff positions stochastically, creating an interval inside the two participating genomes. The values inside said interval in one of the genomes is then swapped with the values within the interval of the other genome, rendering two children.

**One Point Crossover.** This crossover operator[5] is included in DEAP and chooses one cutoff position stochastically, creating a cutoff point inside the two participating genomes. The values beyond the cutoff point in one of the genomes is then swapped with the values within the interval of the other genome, rendering two children.

[2]http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.mutUniformInt (Accessed: 2016-05-25).

[3]http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.mutShuffleIndexes (Accessed: 2016-05-25).

[4]http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.cxTwoPoint (Accessed: 2016-05-25).

[5]http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.cxOnePoint (Accessed: 2016-05-25).

**Size Matters Crossover.** This is one of our custom made crossover operators which swaps the positions of specific squares between two individuals with a square size based probability. It is stochastically decided, with a probability of fifty percent, whether this probability is highest for large squares and smallest for small squares or highest for small squares and smallest for big squares.

**Weighted Average Crossover.** This crossover operation will calculate the two average individuals: when counting the first parent twice, and when counting the second parent twice. The individual when counting the $k$th parent twice, there is a fifty percent probability, for each element $i$, that the $i$th element copied directly from $i$th element of the $k$th parent.

**Quantitative Crossover.** Given two individuals $S$ and $S'$, two new individuals are generated. The first one is constructed by taking the $i \times i$ square, for each $0 < i \leq n$, from whichever individual the $i \times i$ square overlaps with the fewest squares. The other one is constructed by choosing the squares that are left (i.e. the squares with the most overlaps). Tie-break by choosing the square from $S$.

### 3.3.3 Selections

In order to create offspring with a crossover operation, which individuals to use the crossover operation on has to be decided. This is where selections is used. A selection is simply a rule on how to choose individuals to then use a crossover operation on.

**Tournament Selection.** First, a uniformly random subset of the individuals of the current generation is generated. This subset will have $k$ individuals, where $k$ is a predefined parameter. Then, the two best individuals (i.e. lowest fitness) of this subset are selected to be used with the crossover operation.[6]

**Random Selection.** This is a selection mechanism that is called `selRandom` in `DEAP`.[7] Random Selection simply selects two individuals at random.

### 3.3.4 Escaping Local Minima

One common issue with genetic algorithms is that they often converge to a local minimum before arriving at the global minimum, thus it's common to restart genetic algorithms according to some given criterion [2]. We implemented a variation of random restarts by checking if the current best local minimum has been constant for a number of generations, defaulting to 10 generations. However, we do not perform random restarts as such, but rather random *shakes* by modifying the behaviour of the individuals. This is done by a change to the behaviour of the genomes by adjusting the mutation function as well as the selection mechanism. By default, the mutation

[6]`http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.selTournament` (Accessed: 2016-05-26).

[7]`http://deap.readthedocs.io/en/master/api/tools.html#deap.tools.selRandom` (Accessed: 2016-05-25).

is changed to the uniform integer mutation with `low` set to 0 and `up` set to $s-2$, where $s$ is the side length of the enclosing square. The selection mechanism is by default changed to Random Selection. When using the default setting, the system runs 15 generations using this mutation and selection configuration before returning to the original configuration. We have then performed a shake in the evolution.

This choice of mutation technique causes big changes to the square positions of the individuals and the selection mechanism causes individuals that previously survived for having a low fitness may no longer survive the evolution. Thus escaping the current local minimum with the hope of finding the global minimum after a shake has been performed.

Throughout the report this feature is denoted as *escaper*.

### 3.4 Problem Specific Optimisation

In this section we present problem specific optimisations to SPP that we have explored with in our experiments.

**Fixing Largest Square.** This optimisation is done by fixing the largest square to position $(x, y) = (0, 0)$. By doing this we hope to reduce the search space of mostly suboptimal packings and to some degree incorrect packings. Since having the largest square on any position such that it does not touch a corner, will yield a minimal enclosing square larger than (or in some cases equal to) the general minimal enclosing square size. The justification for this follows from the reasoning by Simonis and O'Sullivan [8].

**Ignoring Smallest Square.** This optimisation involves ignoring the smallest square in placements. Again, the justification for this follows from the reasoning by Simonis and O'Sullivan [8]. That is, we will only place $n-1$ squares instead of $n$, but the smallest is size of $2 \times 2$ instead of $1 \times 1$. The $1 \times 1$ square can be placed arbitrarily on any free slot in a packing. We will ensure that the $1 \times 1$ square will be placeable (i.e. there is always at least one free $1 \times 1$ slot) by using the combined area of all $n$ squares in calculations. This will hopefully speed up the optimisation process as well as reduce the search space with mostly incorrect packings.

## 4 Results

In this section investigations of different combinations of the aforementioned configurations and parameters for GA and PSO are performed.

All tests for PSO were run under Xubuntu 15.10 (64 bit) Linux on an Intel Core i5-4200U of 1.60 GHz with an 512 kB L2 cache and a 4 GB RAM. All tests for GA were run under Ubuntu 14.04 LTS (3.16.0-30-generic) using a Intel Core i5 2500K (4 cores, 6MB L3 Cache) processor running at 4 Ghz with 16GB of RAM.

## 4.1 Upper Bound of Enclosing Square Size

In the case where a fixed $s$ is used, there needs to be a starting point for the size $s$. This is also used with a dynamic $s$, where it is used to set an upper bound of the $x$ and $y$ coordinates of the squares to be placed.

The placement shown in Figure 7 a placement of $n$ squares that will be relatively small. The number in each square denotes its size.

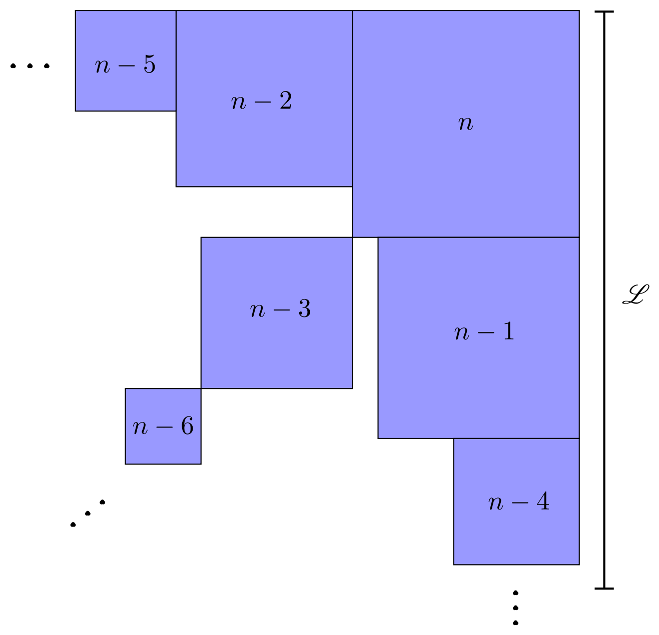


Figure 7: Illustration of a square placement to find a reasonable upper bound ($\mathscr{L}$) on $s$ for a given $n$

The side length $\mathscr{L}$ in Figure 7 is a longest side for all $n \leq 1$. This is obvious since each square that adds length to $\mathscr{L}$ is one length unit greater than the square added to the other side. Hence, one can safely set $s = \mathscr{L}$ to start with. The value of $\mathscr{L}$ is calculated as follows:

$$\mathsf{maximal_s}(n) := \begin{cases} 0 & \text{if } n < 1 \\ n + \mathsf{maximal'_s}(n-1) & \text{otherwise} \end{cases}$$

where $\mathsf{maximal'_s}$ is defined as follows:

$$\mathsf{maximal'_s}(n) := \begin{cases} 0 & \text{if } n < 1 \\ n + \mathsf{maximal'_s}(n-3) & \text{otherwise} \end{cases}$$

## 4.2 Known Minima

Known minimal enclosing square sizes have been previously computed with an exact optimisation algorithm using constraint programming in Gecode [3]. The minimal values for $s$ when solving

SPP for $2 \leq n \leq 23$ can be found in Table 1. These values will be used to evaluate performance

| $n$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $s$ | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 43 | 47 | 50 | 54 | 58 | 62 | 66 |

Table 1: The minimal enclosing square size $s$ for packing $n$ squares.

of different configurations of GA and PSO. Furthermore, these values will be used to stop the search when finding a packing with this size of $s$, since seeing how fast a configuration can determine that a minimal packing has be found is deemed uninteresting by the authors. In some experiments, the size of $s$ will be fixed to this minimal size instead of using $\mathsf{maximal_s}$ as the implementations may not always find a packing for that $s$, it is interesting to see how high the success rate is for finding one.

## 4.3 Finding Optimal Parameters

For population-based search methods, finding optimal parameters is very important as can be seen in the following sections. We examined GA and PSO separately as the parameters vary for the two algorithms.

### 4.3.1 PSO

Our first step when running PSO was to find good values for the social and cognitive factors. This was done by running tests on PSO using 7 squares, as this was the crossing point whereafter PSO had a hard time finding an minimal packing. Also a fixed value for $s$ was used, namely the minimal $s = 13$ for $n = 7$, derived from Table 1. The tests were run over three different fitness functions: Grid Fit, Distance Fit and Area Overlap.

| cognitive \social | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0.20 | 0.06 | 0.26 | 0.20 | 0.20 | 0.14 | 0.20 | 0.12 | 0.20 | 0.26 | 0.20 |
| 1.1 | 0.10 | 0.14 | 0.10 | 0.16 | 0.16 | 0.24 | 0.26 | 0.20 | 0.22 | 0.28 | 0.20 |
| 1.2 | 0.22 | 0.22 | 0.28 | 0.28 | 0.20 | 0.16 | 0.10 | 0.26 | 0.18 | 0.32 | 0.28 |
| 1.3 | 0.14 | 0.18 | 0.22 | 0.12 | 0.24 | 0.24 | 0.16 | 0.20 | 0.28 | 0.28 | 0.26 |
| 1.4 | 0.28 | 0.26 | 0.14 | 0.22 | 0.32 | 0.22 | 0.28 | 0.26 | 0.28 | 0.30 | 0.22 |
| 1.5 | 0.24 | 0.16 | 0.18 | 0.24 | 0.18 | 0.16 | 0.26 | 0.18 | 0.30 | 0.24 | 0.24 |
| 1.6 | 0.16 | 0.16 | 0.18 | 0.18 | 0.26 | 0.26 | 0.32 | 0.22 | 0.20 | 0.20 | 0.34 |
| 1.7 | 0.20 | 0.32 | 0.20 | 0.20 | 0.26 | 0.22 | 0.38 | 0.28 | 0.22 | 0.34 | 0.30 |
| 1.8 | 0.18 | 0.34 | 0.14 | 0.14 | 0.26 | 0.20 | 0.22 | 0.22 | 0.32 | 0.32 | 0.34 |
| 1.9 | 0.22 | 0.22 | 0.18 | 0.28 | 0.22 | 0.30 | 0.28 | 0.24 | 0.34 | 0.24 | 0.30 |
| 2.0 | 0.24 | 0.26 | 0.20 | 0.28 | 0.22 | 0.28 | 0.30 | 0.38 | 0.32 | 0.36 | 0.32 |

Table 2: Sample probability of finding a minimal packing over 50 runs with $n = 7$ and minimal $s$ using PSO, for each combination of cognitive and social factors, using Area Overlap.

| cognitive \social | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0.16 | 0.12 | 0.22 | 0.28 | 0.16 | 0.18 | 0.28 | 0.22 | 0.22 | 0.28 | 0.20 |
| 1.1 | 0.16 | 0.28 | 0.18 | 0.18 | 0.28 | 0.34 | 0.18 | 0.30 | 0.26 | 0.16 | 0.26 |
| 1.2 | 0.16 | 0.20 | 0.18 | 0.14 | 0.22 | 0.14 | 0.10 | 0.32 | 0.32 | 0.28 | 0.16 |
| 1.3 | 0.14 | 0.20 | 0.14 | 0.22 | 0.18 | 0.24 | 0.18 | 0.24 | 0.18 | 0.14 | 0.30 |
| 1.4 | 0.12 | 0.22 | 0.18 | 0.20 | 0.26 | 0.26 | 0.28 | 0.20 | 0.22 | 0.28 | 0.16 |
| 1.5 | 0.26 | 0.24 | 0.28 | 0.22 | 0.28 | 0.22 | 0.22 | 0.30 | 0.30 | 0.22 | 0.30 |
| 1.6 | 0.22 | 0.26 | 0.24 | 0.26 | 0.24 | 0.32 | 0.28 | 0.28 | 0.26 | 0.30 | 0.24 |
| 1.7 | 0.18 | 0.22 | 0.26 | 0.26 | 0.32 | 0.30 | 0.30 | 0.38 | 0.30 | 0.36 | 0.20 |
| 1.8 | 0.16 | 0.18 | 0.24 | 0.30 | 0.18 | 0.28 | 0.34 | 0.28 | 0.20 | 0.18 | 0.42 |
| 1.9 | 0.18 | 0.12 | 0.30 | 0.28 | 0.18 | 0.24 | 0.38 | 0.42 | 0.48 | 0.38 | 0.36 |
| 2.0 | 0.24 | 0.18 | 0.32 | 0.26 | 0.40 | 0.30 | 0.24 | 0.28 | 0.30 | 0.34 | 0.44 |

Table 3: Sample probability of finding a minimal packing over 50 runs with $n = 7$ and minimal $s$, for each combination of cognitive and social factors using PSO, using the Distance Fit fitness function.

| cognitive \ social | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0.18 | 0.14 | 0.20 | 0.12 | 0.28 | 0.18 | 0.18 | 0.18 | 0.18 | 0.32 | 0.22 |
| 1.1 | 0.12 | 0.20 | 0.26 | 0.16 | 0.20 | 0.30 | 0.26 | 0.24 | 0.28 | 0.18 | 0.32 |
| 1.2 | 0.16 | 0.28 | 0.20 | 0.30 | 0.24 | 0.26 | 0.34 | 0.24 | 0.36 | 0.32 | 0.28 |
| 1.3 | 0.22 | 0.14 | 0.24 | 0.34 | 0.30 | 0.34 | 0.36 | 0.32 | 0.26 | 0.24 | 0.22 |
| 1.4 | 0.36 | 0.42 | 0.30 | 0.30 | 0.24 | 0.38 | 0.24 | 0.36 | 0.36 | 0.40 | 0.28 |
| 1.5 | 0.26 | 0.20 | 0.28 | 0.26 | 0.22 | 0.44 | 0.26 | 0.32 | 0.24 | 0.42 | 0.28 |
| 1.6 | 0.26 | 0.30 | 0.20 | 0.20 | 0.28 | 0.22 | 0.34 | 0.40 | 0.36 | 0.26 | 0.38 |
| 1.7 | 0.18 | 0.24 | 0.22 | 0.24 | 0.26 | 0.26 | 0.34 | 0.32 | 0.18 | 0.32 | 0.28 |
| 1.8 | 0.22 | 0.26 | 0.22 | 0.36 | 0.34 | 0.34 | 0.30 | 0.34 | 0.30 | 0.40 | 0.34 |
| 1.9 | 0.20 | 0.28 | 0.28 | 0.22 | 0.30 | 0.30 | 0.38 | 0.34 | 0.38 | 0.44 | 0.42 |
| 2.0 | 0.20 | 0.32 | 0.22 | 0.26 | 0.30 | 0.46 | 0.34 | 0.30 | 0.40 | 0.30 | 0.50 |

Table 4: Sample probability of finding a minimal packing over 50 runs with $n = 7$ and minimal $s$ using PSO, for each combination of cognitive and social factors, using the Grid Fit fitness function.

The results can be seen in Tables 4, 3 and 2, where each cell represents the percentage of 50 runs that found a minimal packing. The best results are marked as grey cells.

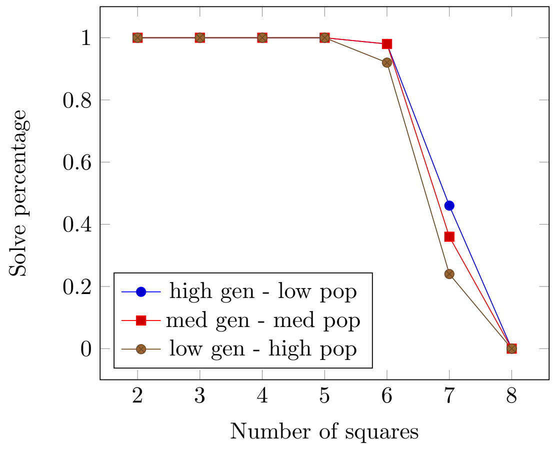
The next step was to examine how population size and number of generations affected the results. Due to time constraints, only the best parameters for the cognitive and social factors where used. For each of three fitness functions, Grid Fit, Distance Fit and Area Overlap, three configurations of generations and population size where used:

- 2500 generations and a population of 20 - High gen and low pop

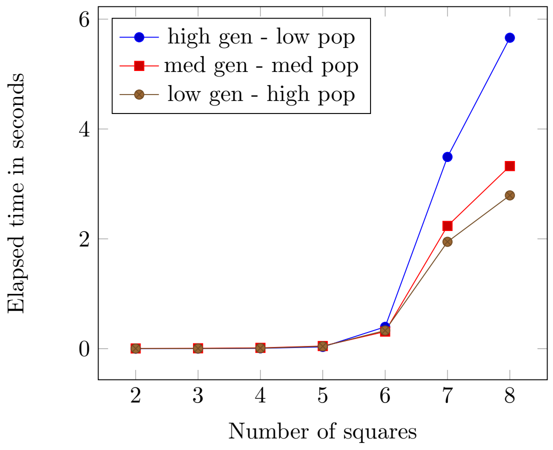
- 500 generations and a population of 50 - Mid gen and mid pop
- 300 generations and a population of 100 - low gen and medium pop

Again, the known minimal $s$ was used but now with the number of squares ranging from 2 to 8. The average results over 50 runs can be viewed in Figure 8. As can be seen, it may be worthwhile to either have a large population or many generations. The middle ground seems to be the worst even when accounting for run times. A high number of generations clearly increased the solve percentages, most notably for Distance Fit. However, a higher number of generations also lead to longer run times so this is a trade-off. Two more things we can note from this run is that none of the configurations led to a solution for $n = 8$ in any run and that Grid Fit seems to be the best fitness function as it had higher solve percentage for high population - low generations for $n = 7$ with 52% over 44% and 46% for Distance Fit and Area Overlap respectively.

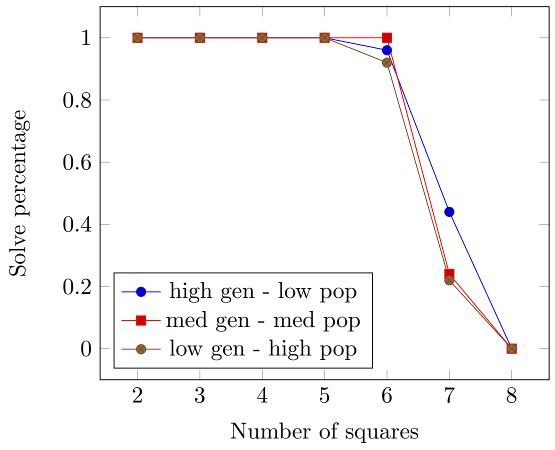
All the top results for $n = 7$ where achieved with a high generation count and low population but as can be seen in Table 8 (f) the run time was close to double so the comparison might be unfair and for this reason it is hard to determine which of the combinations is better. Therefore two configurations where run for the final evaluation which can be seen in Section 4.5.



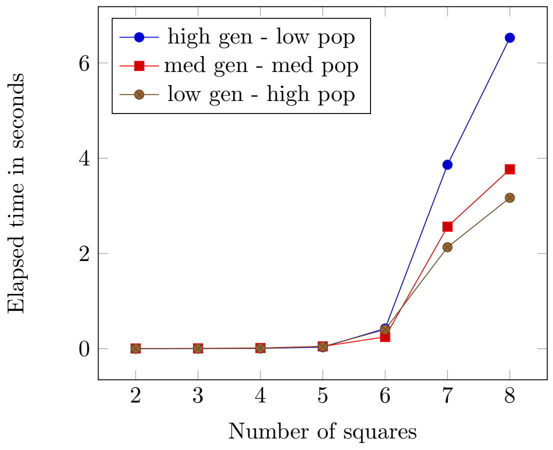
(a) Solve percentage for Area Overlap.



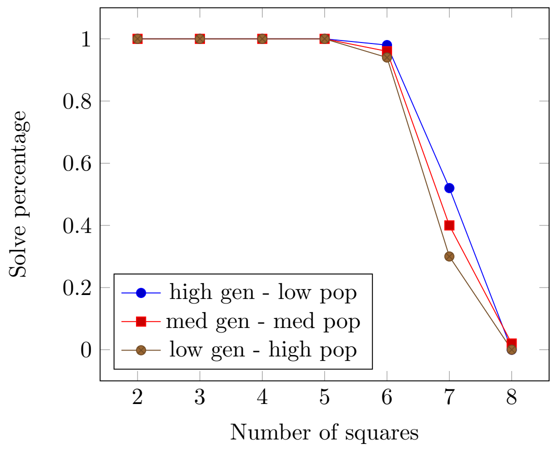
(b) Run times for Area Overlap.



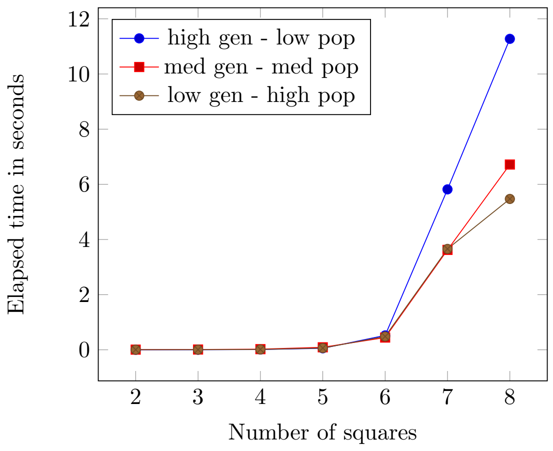
(c) Solve percentage for Distance Fit.



(d) Run times for Distance Fit.



(e) Solve percentage for Grid Fit.



(f) Run times for Grid Fit.

Figure 8: Performance plots for PSO with fixed optimal s

### 4.3.2 GA

Tests for parameter turning were also run on the GA implementation. Here we investigated the effects of different mutation probability (mutpb) and crossover probability (cxpb) configurations over four different fitness functions, namely Area Overlap, Enumerative Overlap, Big Expensive and Grid Fit. All with $s$ fixed to the minimal. The results of these runs can be found in Tables 5, 7, 6 and 8 respectively. For these measurements the selection function used was *tournament selection* with a tournament size of 3. The mutation function *local gaussian* was used with a `indpb` setting of 0.15.

Note that when testing GA here, a greater value of $n$ is used compared to PSO. This is because of the fact that GA generally performs better than PSO from what we've noticed during development of both implementations. $n = 7$ is a trivial case for GA and almost all of the instances would always reach a packing if $n = 7$ would have been used in the tests, rendering the results useless. For reference, see Figure 15 where GA solves all $n = 7$ with a 100% solve percentage.

| mutpb \ cxpb | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.16 | 0.12 | 0.28 | 0.16 | 0.20 | 0.24 | 0.08 | 0.16 | 0.28 | 0.28 | 0.24 |
| 0.2 | 0.32 | 0.16 | 0.24 | 0.32 | 0.20 | 0.40 | 0.28 | 0.40 | 0.64 | 0.40 | 0.52 |
| 0.3 | 0.36 | 0.28 | 0.28 | 0.36 | 0.64 | 0.44 | 0.48 | 0.40 | 0.68 | 0.60 | 0.64 |
| 0.4 | 0.24 | 0.28 | 0.40 | 0.40 | 0.64 | 0.56 | 0.40 | 0.80 | 0.76 | 0.68 | 0.68 |
| 0.5 | 0.36 | 0.44 | 0.20 | 0.60 | 0.48 | 0.68 | 0.68 | 0.52 | 0.76 | 0.60 | 0.68 |
| 0.6 | 0.40 | 0.40 | 0.32 | 0.52 | 0.60 | 0.64 | 0.68 | 0.64 | 0.72 | 0.76 | 0.76 |
| 0.7 | 0.44 | 0.44 | 0.40 | 0.64 | 0.64 | 0.64 | 0.68 | 0.48 | 0.68 | 0.76 | 0.84 |
| 0.8 | 0.44 | 0.56 | 0.52 | 0.64 | 0.72 | 0.60 | 0.52 | 0.80 | 0.80 | 0.64 | 0.68 |
| 0.9 | 0.16 | 0.24 | 0.32 | 0.44 | 0.40 | 0.32 | 0.64 | 0.52 | 0.56 | 0.64 | 0.52 |
| 1.0 | 0.00 | 0.00 | 0.04 | 0.00 | 0.04 | 0.04 | 0.08 | 0.08 | 0.04 | 0.04 | 0.08 |

Table 5: Sample probability of finding a minimal packing over 25 runs with $n = 11$ and minimal $s$ using GA, for each combination of mutation and crossover probabilities, using Area Overlap.

As seen in Table 7, Enumerative Overlap did not solve any of the instances for any of those configurations, we therefore consider Enumerative Overlap uninteresting for this problem, and hence will not investigate it further.

When inspecting the results in Tables 5 and 6, it can be seen that the performance of the Area Overlap fitness function greatly surpasses the performance of the Big Expensive fitness function. Where the best measured solve percentage was 84% for Area Overlap but only 68% for Big Expensive.

From Table 8 we can observe that Grid Fit actually outperform all the other fitness functions when using GA. With a staggering 92% solve percentage edges the performance of Area Overlap with a 4% increase. Do remember that due to time constraints we only had time to run 25 runs on the GA configurations, since it takes a lot of time (roughly 30 *seconds*) to solve one $n = 11$

| mutpb \ cxpb | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.16 | 0.16 | 0.16 | 0.20 | 0.20 | 0.12 | 0.24 | 0.24 | 0.32 | 0.04 | 0.16 |
| 0.2 | 0.20 | 0.16 | 0.24 | 0.20 | 0.28 | 0.24 | 0.40 | 0.20 | 0.40 | 0.28 | 0.52 |
| 0.3 | 0.28 | 0.24 | 0.12 | 0.28 | 0.32 | 0.40 | 0.40 | 0.40 | 0.40 | 0.36 | 0.40 |
| 0.4 | 0.28 | 0.32 | 0.40 | 0.28 | 0.40 | 0.56 | 0.48 | 0.40 | 0.44 | 0.36 | 0.64 |
| 0.5 | 0.28 | 0.48 | 0.52 | 0.40 | 0.24 | 0.44 | 0.64 | 0.48 | 0.44 | 0.52 | 0.56 |
| 0.6 | 0.20 | 0.44 | 0.64 | 0.44 | 0.52 | 0.28 | 0.44 | 0.60 | 0.60 | 0.68 | 0.60 |
| 0.7 | 0.36 | 0.52 | 0.44 | 0.52 | 0.48 | 0.56 | 0.44 | 0.64 | 0.36 | 0.36 | 0.52 |
| 0.8 | 0.32 | 0.48 | 0.60 | 0.52 | 0.24 | 0.52 | 0.28 | 0.36 | 0.36 | 0.32 | 0.36 |
| 0.9 | 0.08 | 0.20 | 0.24 | 0.08 | 0.12 | 0.24 | 0.04 | 0.04 | 0.16 | 0.12 | 0.12 |
| 1.0 | 0.00 | 0.00 | 0.00 | 0.04 | 0.04 | 0.00 | 0.00 | 0.00 | 0.04 | 0.00 | 0.00 |

Table 6: Sample probability of finding a minimal packing over 25 runs with $n = 11$ and minimal $s$ using GA, for each combination of mutation and crossover probabilities, using Big Expensive.

| mutpb \ cxpb | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.2 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.4 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.5 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.6 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.7 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.8 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.9 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

Table 7: Sample probability of finding a minimal packing over 25 runs with $n = 11$ and minimal $s$ using GA, for each combination of mutation and crossover probabilities, using Enumerative Overlap. (Yes, this table is filled with actual data.)

instance with GA.

We decided to use Area Overlap as a basis for further parameter tuning since at the time of conducting the mutation, escaper, population and crossover and other experiments the result of Grid Fit was unknown to us at the time. We also concluded that the optimal mutation and crossover configuration was $mutpb = 0.7$ and $cxpb = 1.0$ based on the Area Overlap measurement data. This is also to be viewed as the default mutation and crossover configuration unless otherwise specified in the report.

We then experimented with the effect of the problem specific optimisations discussed in Section 3.4. We ran four different configurations, one where we only fixed the largest square (denoted as *force placement*) and one where we only ignored the smallest square (denoted as *ignore smal-*

| mutpb \ cxpb | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.08 | 0.20 | 0.08 | 0.32 | 0.00 | 0.16 | 0.24 | 0.20 | 0.16 | 0.36 | 0.24 |
| 0.2 | 0.08 | 0.08 | 0.28 | 0.20 | 0.28 | 0.28 | 0.32 | 0.32 | 0.48 | 0.44 | 0.44 |
| 0.3 | 0.24 | 0.24 | 0.28 | 0.64 | 0.32 | 0.56 | 0.52 | 0.60 | 0.52 | 0.48 | 0.48 |
| 0.4 | 0.12 | 0.32 | 0.48 | 0.40 | 0.48 | 0.24 | 0.64 | 0.72 | 0.52 | 0.64 | 0.56 |
| 0.5 | 0.32 | 0.48 | 0.44 | 0.48 | 0.48 | 0.56 | 0.64 | 0.64 | 0.60 | 0.76 | 0.76 |
| 0.6 | 0.32 | 0.32 | 0.44 | 0.44 | 0.72 | 0.68 | 0.64 | 0.64 | 0.64 | 0.64 | 0.64 |
| 0.7 | 0.40 | 0.36 | 0.48 | 0.56 | 0.60 | 0.60 | 0.72 | 0.76 | 0.88 | 0.68 | 0.72 |
| 0.8 | 0.56 | 0.56 | 0.68 | 0.64 | 0.60 | 0.72 | 0.68 | 0.68 | 0.92 | 0.72 | 0.64 |
| 0.9 | 0.28 | 0.32 | 0.36 | 0.44 | 0.40 | 0.64 | 0.72 | 0.72 | 0.56 | 0.68 | 0.80 |
| 1 | 0.00 | 0.00 | 0.04 | 0.00 | 0.04 | 0.00 | 0.00 | 0.00 | 0.20 | 0.16 | 0.16 |

Table 8: Sample probability of finding a minimal packing over 25 runs with $n = 11$ and minimal $s$ using GA, for each combination of mutation and crossover probabilities, with Grid Fit.

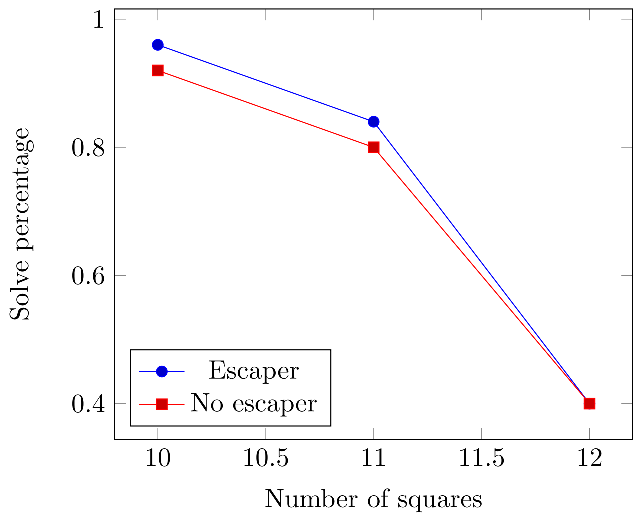


Figure 9: Solve percentage for escaper and no escaper with $mutpb = 0.7, cxpb = 1.0$, running the Area Overlap fitness function.

*lest*). We also included both in combination as a configuration (denoted as *forced + ignore*), and finally a configuration with no optimisations. The results of running these configurations can be found in Figure 10. Not surprisingly, the configuration with no optimisations performs the worst while ignore smallest is slightly better. Most improvement is seen in the fixing the position of the largest square (force placement) but combining both ignore smallest and force placement performs best. The latter is also the default parameters in further tests in this report unless otherwise stated.

For testing out our implementation of escaping local mimima, as discussed in Section 3.3.4, we ran experiments using the previously best known parameter configurations for $10 \leq n \leq 12$.

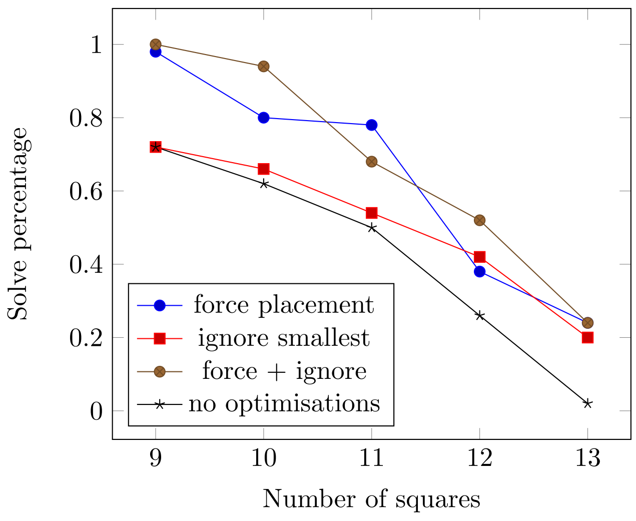


Figure 10: Solve percentage for forcing placement of the biggest square, ignoring the smallest square, combining both optimisations and no optimisations using GA with $mutpb = 0.7, cxpb = 1.0$. Running the Area Overlap fitness function.

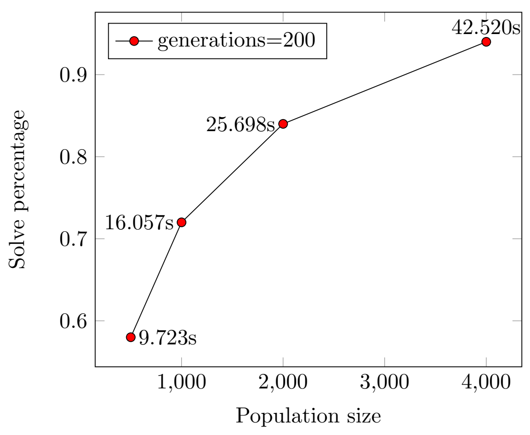
| configuration \ n | 10 | 11 | 12 |
|---|---|---|---|
| Escaper | 0.96 | 0.84 | 0.40 |
| No escaper | 0.92 | 0.80 | 0.40 |

Table 9: Solve percentage for escaper and no escaper using $mutpb = 0.7, cxpb = 1.0$ and Area Overlap over different configurations of $n$.

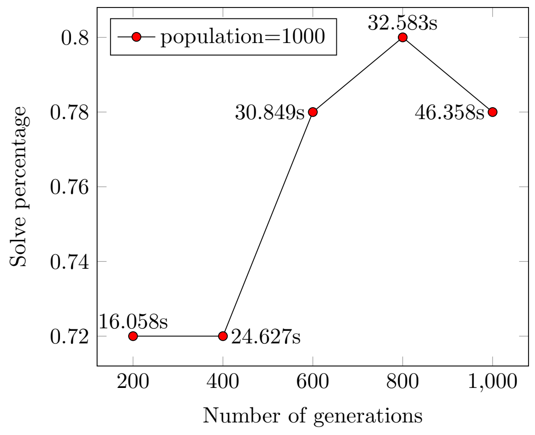
The measurements of these tests can be found in Figure 9. We only conducted experiments using the default parameters of escaper and reached no significant performance improvements by doing so, as seen in more detail in Table 9. We decided to not include escaper in further tests since we did not have time to further adjust it's internal parameters, which may have lead to bigger performance increases.

When investigating the effects of number of generations and population size in GA we conducted experiments which results are found in Figures 11 and 12. In these measurements it can be observed that it is much better to increase the population size rather than the number of generations on this specific mutation and crossover configuration. It should be noted that, as seen in Figure 12, using a very large population size but a small number of generations is not beneficial, as is the case if 10 generations and 20000 population size, which didn't manage to solve a single $n = 11$ instance. Due to the time penalties of increasing population size seen in Figure 11, we decided against increasing the population size for our final evaluation tests and continued with a 200 generations and a population size of 1000.

Using the previously deduced optimal parameters, we wanted to test if changing either the crossover operation or the mutation operation would have a positive effect. As seen in Figures 14



(a) Solve percentage when keeping generations fixed and gradually increasing population size.



(b) Solve percentage when keeping population size fixed and gradually increasing number of generations.

Figure 11: Performance plots for GA averaged over 50 runs when adjusting number of generations and population size.

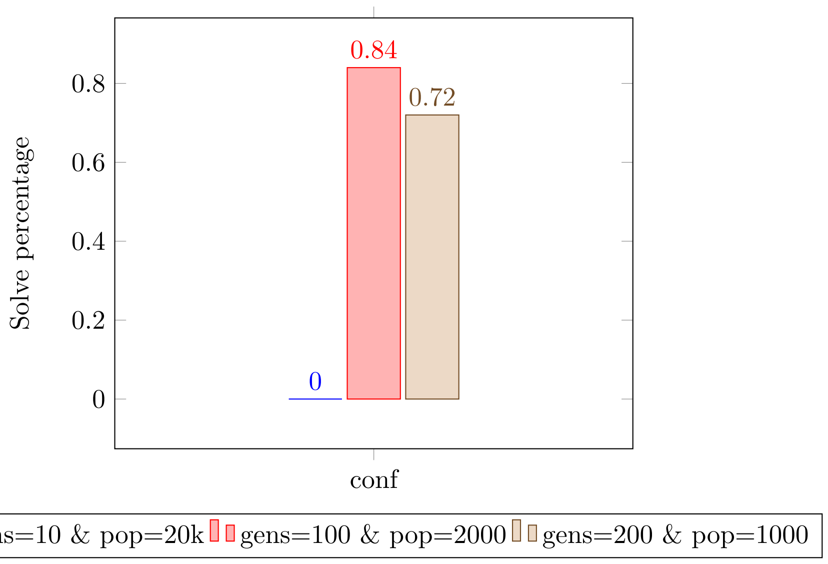


Figure 12: Solve percentage for different (generation, population) combinations in the GA implementation. A mutation rate $mutpb = 0.7$ and a crossover rate of $cxpb = 1.0$ was used in combination with the fitness function Area Overlap with $n = 11$.

and 13, Local Gaussian Mutation and Two Point Crossover seems to perform the best for these parameters. Hence, we will use them in future tests and evaluations.

We wrapped up the parameter tuning with testing different tournament sizes when using the tournament selection for the three best combinations of mutation and crossover probabilities that

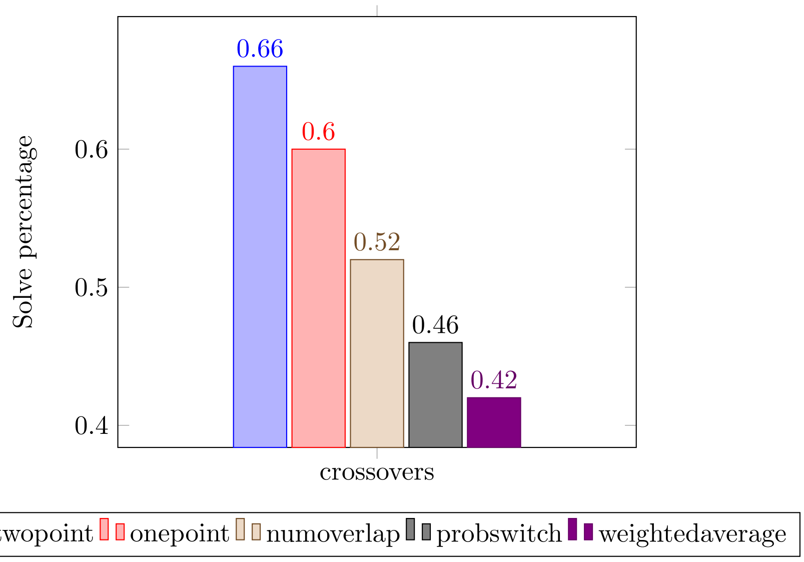


Figure 13: Solve percentage for different crossover functions in the GA implementation. A mutation rate $mutpb = 0.7$ and a crossover rate of $cxpb = 1.0$ was used in combination with the fitness function Area Overlap with $n = 11$. Local Gaussian Mutation was used.

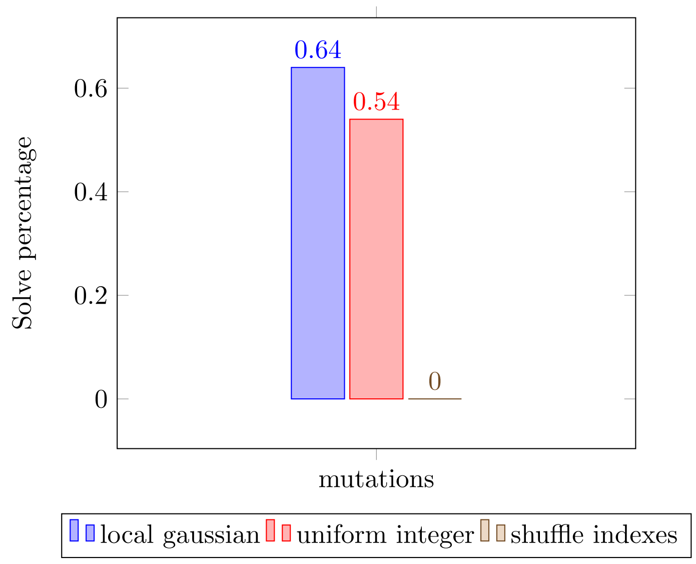
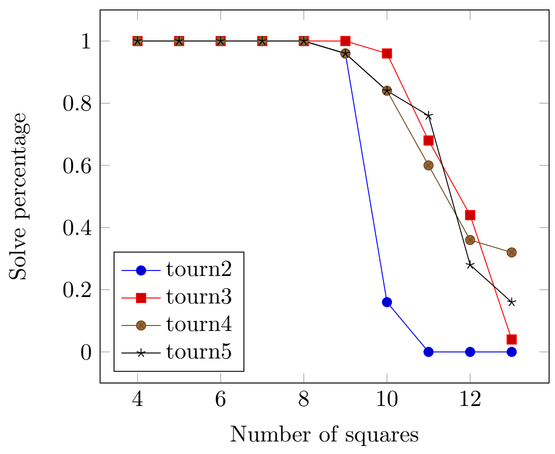
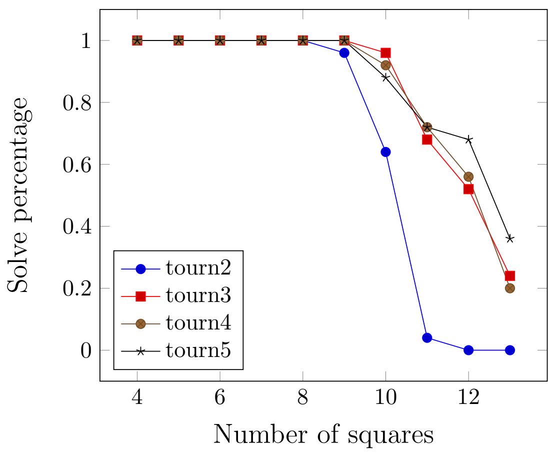


Figure 14: Solve percentage for different mutation functions in the GA implementation. A mutation rate $mutpb = 0.7$ and a crossover rate of $cxpb = 1.0$ was used in combination with the fitness function Area Overlap with $n = 11$. Two Point Crossover was used.

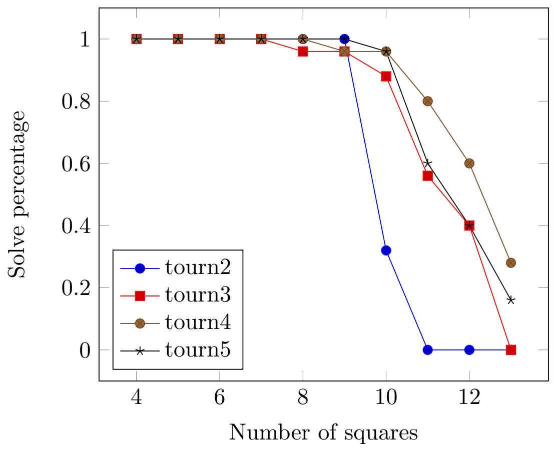
we found. The results from this experiment can be seen in Figure 15. We did not conduct much experiments on different selection functions other than tournament selection however. As seen



(a) $mutpb = 0.8, cxpb = 0.7$



(b) $mutpb = 0.7, cxpb = 1.0$



(c) $mutpb = 0.8, cxpb = 0.8$

Figure 15: Measurements of the solve percentage for different tournament selection sizes using GA.

in the measurements, a tournament size of 5 performs best when using the optimal mutation and crossover configuration ($mutpb = 0.7, cxpb = 1.0$) and this was picked out as the optimal tournament size.

## 4.4 Final evaluation

For the final evaluation, solution percentage is no longer the only interesting performance metric, but what $s$ is reached on average and how much time a configuration takes are also interesting. Hence, the Dynamic $s$ fitness functions ($\mathsf{fitness_D}$) and the Fixed $s$ fitness functions without using the already known minima, will be used for each $n$ the authors deem interesting.

## 4.5 PSO

For PSO, $s$ started at $\mathsf{maximal_s}(n)$ (i.e. the upper bound on the enclosing square size for $n$ squares) and was decreased iteratively as long as a solution was found or the minimal $s$ was reached. Since the parameter tuning did not clearly favour a high number of generations or a high population, two configurations where run for the three fitness functions: Grid Fit, Distance Fit and $\mathsf{fitness_F}$ with Area Overlap. The configurations run where:

- Lowpop: 2500 generations and 20 population
- Highpop: 500 generations and 100 population

Following this we also used the following parameters for all configurations:

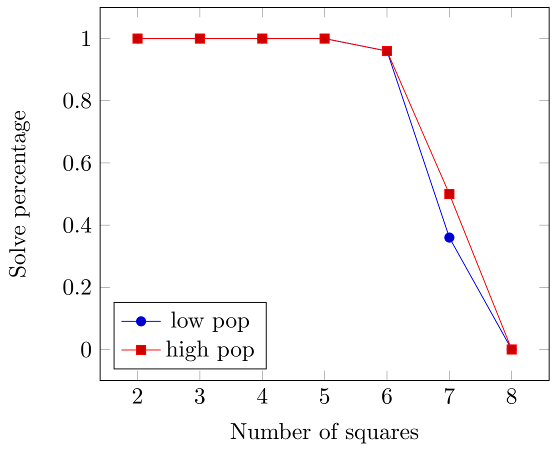
- ignore smallest square

The average results over 50 runs for the two configurations are seen in Figure 16. Also a bar chart with the solve percentages for $n = 7$ for the different fitness functions and configurations are seen in figure 17 As can be seen Highpop performed better all across the board with a significant improvement for Area Overlap, beating out Distance Fit for $n = 7$. Adding another 200 generations to Highpop evened out the run times, leading to a more fair comparison. So, it seems that a high population increases the probability of finding a solution, at least for the challenging $n = 7$ case.

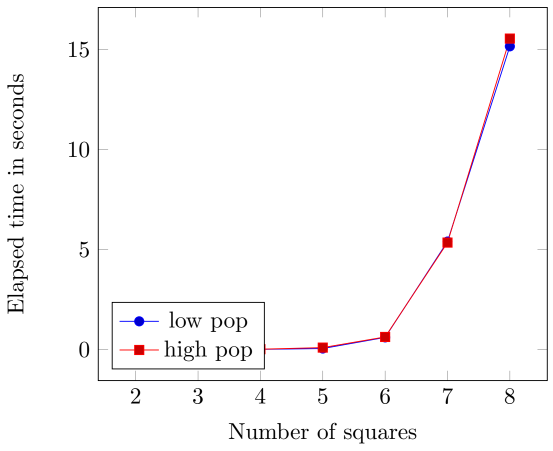
The smaller differences for Distance Fit and Grid Fit can be understood by considering that any small change of the placement of the squares will affect the fitness (unlike for Area Overlap where the actual overlap needs to decrease) so running for more generations is more likely to have a good result for the more complex fitness functions.

We note that decreasing $s$ instead of having it fixed did not impact the solution percentages negatively. However, the only case which gave any meaningful data for this was $n = 7$ so this conclusion is not definitive.

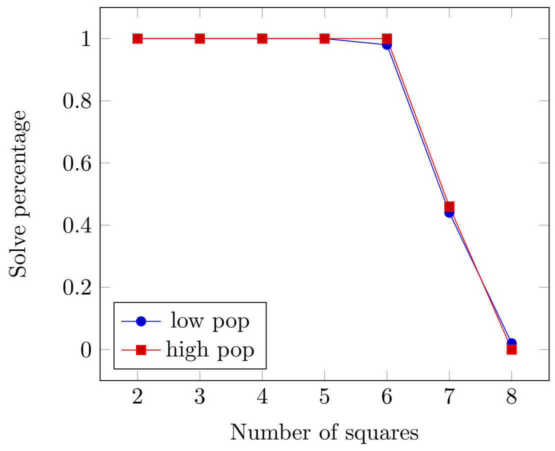
Figure 18 shows the difference between the best packing found and the best packing known (see Section 4.2) for the three fitness functions, Area Overlap, Distance Fit and Grid Fit, averaged over 50 runs. As can be seen Highpop has a slightly better average $s$ and Grid Fit performs best for increasing $n$. However the difference between the three fitness functions is small for large $s$ with the Lowpop configuration.



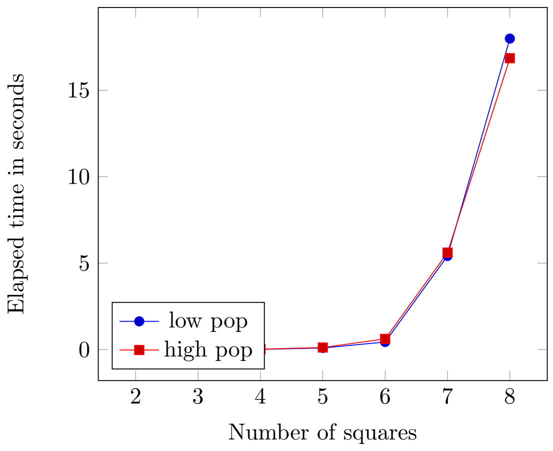
(a) Solve percentage for Area Overlap.



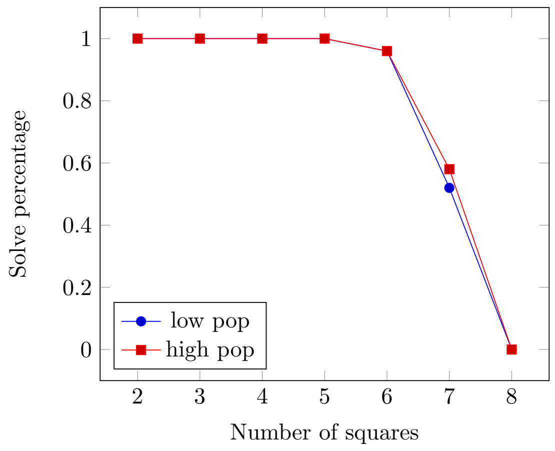
(b) Run times for Area Overlap.



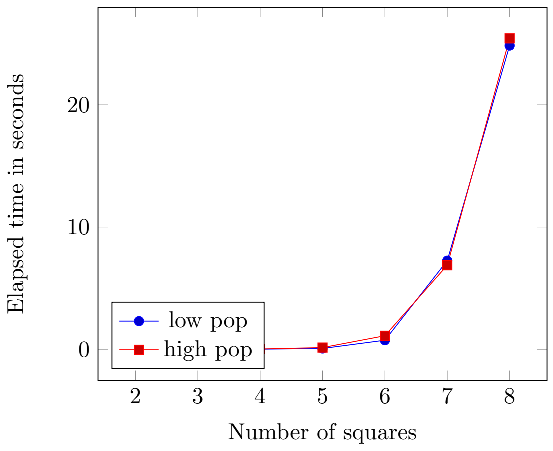
(c) Solve percentage for Distance Fit.



(d) Run times for Distance Fit.



(e) Solve percentage for Grid Fit.



(f) Run times for Grid Fit.

Figure 16: Performance plots for PSO with decreasing s.

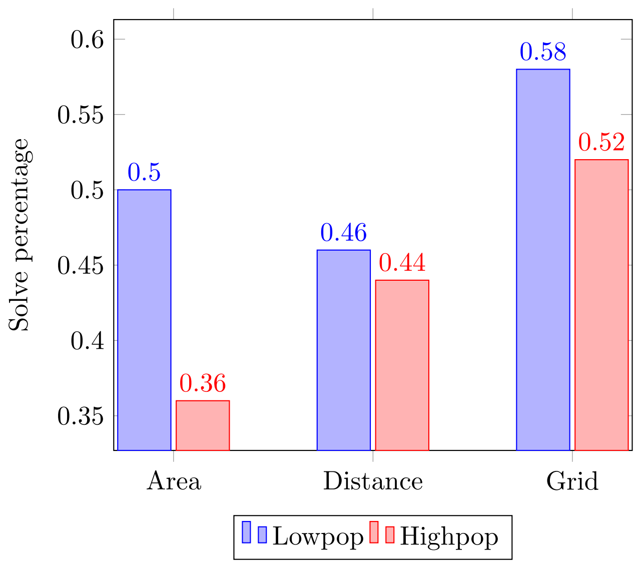
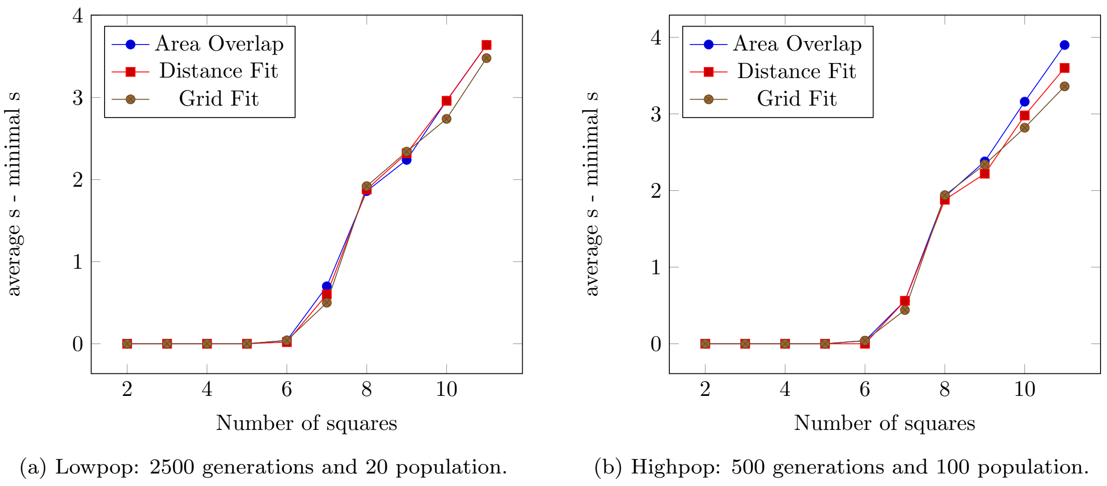


Figure 17: Solve percentages for $n = 7$, using PSO.



(a) Lowpop: 2500 generations and 20 population.

(b) Highpop: 500 generations and 100 population.

Figure 18: Average difference over 50 runs between smallest s found and known minimal $s$, using PSO.

## 4.6 GA

For GA, Area Overlap and Dynamic Area Overlap was tested and compared using the most interesting parameters and configurations found in Section 4.3.2. When using a fixed $s$, the method of finding a minimal packing is as described in Section 4.5.

For each of the two fitness functions, the following parameters will be used:

- The Local Gaussian Mutation operation.

- Mutation probability (`mutpb`) of 0.7.
- The Two Point Crossover opertaion.
- Crossover probability (`cxpb`) of 1.0.
- Tournament Selection with subsets of size 5 (i.e. $k = 5$).
- 200 generations.
- A population of 1000 indviduals.
- Ignoring the smallest square.
- Forcing the largest sqaure.

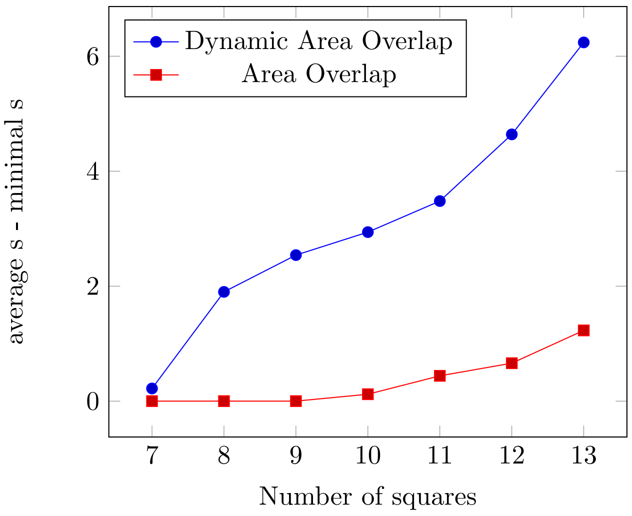


Figure 19: Average difference over 50 runs between smallest s found and known minimal $s$, using GA with dynamic area overlap and area overlap with $mutpb = 0.7, cxpb = 1.0$ over different configurations of $n$.

The average results over 50 runs for the two configurations are seen in Figure 19. The difference between the measured average $s$ and the known minimal $s$ is to be viewed as an accuracy measure of how good each configuration was at approximating the minimal $s$ size for a given number of squares ($n$). It can be seen that Area Overlap does a good job until $n = 10$ where it starts approximating the minimal $s$ value slightly too high. It does perform a lot better than the Dynamic Area Overlap fitness function however, which does not approximate the minimal $s$ value good at all. In fact, it's around 20% of the mark ($averages = 36$ whereas $minimals = 30$).

Area Overlap outperforming Dynamic Area Overlap is to be expected since the search space of the latter is a lot bigger. Also, Dynamic Area Overlap has a few problems that might cause this (described in Section 5).

## 5 Discussion

In this section, analysis and discussion of the results seen in Section 4 will be presented.

The performance for the integer rounding PSO was quite bad compared to GA as PSO never reached above $n = 7$ while GA was up at $n = 11$ with the same fitness functions. It would have been interesting to compare the runtimes between PSO and GA, but since the tests were run on different systems it does not necessarily give that much to compare them. However, based on the results in Section 4, we consider GA to be better for the SPP.

Using PSO efficiently for SPP would probably need a more intelligent representation of the problem and/or not using the ordinary gbest PSO-implementation as it is intended for real valued problems while we used it with rounding on a discrete problem. Redefining PSO has been done for other discrete problems such as the Travelling Salesman Problem [4] and a general binary implementation [5]. Such a redefinition would probably require a new definition of velocity, as rounding leads to small velocities maybe not creating new solutions at all (the movement was not large enough to get to the next coordinate with rounding). However, the coordinate representation was intuitive for constructing fitness functions and with another representation some different fitness functions would need to be invented. The conclusion would thus be that in order to get an effective PSO-implementation of SPP, everything would have to be redone from scratch in some other way.

As seen in Section 4.6, Area Overlap heavily outperformed Dynamic Area Overlap. This might be down to the simple fact that the parameters used for Dynamic Area Overlap was deduced from tests using Area Overlap, and not the dynamic $s$ version. Due to time limitations, these tests could not be performed and we had to assume that the parameters that were optimal for Area overlap was optimal for Dynamic Area Overlap as well. Furthermore, the Dynamic $s$ fitness functions punishes overlap unreasonably much, it could be lowered while still maintaining the invariant that decreasing the unoccupied area but increasing the overlap would never result in a fitness decrease.

For both GA and PSO it was better to spend the run time on having a larger population than more generations. We assume this is because more initial placements leads to a better exploration of the search space, as GA and PSO can only do so much.

Another note for both GA and PSO was that inventing more elaborate fitness function gave improvements. So when tackling problems with no clear fitness function, effort should be put into conceiving a good one.

## 6 Future Work

The current implementation of PSO might be improved if some other implementation of regular PSO such as local best PSO. However, considering the bad results with PSO for SPP this is not recommended.

Another thing that could be done is to create a better Dynamic $s$ fitness function. In its current state, it punishes overlap unreasonably much. Improving this fitness function could increase the viability of the dynamic $s$ approach.

Exploring other selection mechanism for GA could also be interesting. In this project, only the Tournament Selection mechanism (and to some degree Random Selection) were used with different parameters. However, different selection mechanisms may impact the performance of GA to a huge degree, as different parameters for Tournament Selection did.

It would also be interesting to adjust mutation probability inside mutation functions (by adjusing `indpb` parameter). This could potentially help with performance, but we assumed that the effect was minimal and didn't put time into investigating this much.

Moreover, putting more effort into finding a good fitness function is also a bit of work that might be beneficial, and could potentially be the biggest performance factor. We theorise that this could be a project on its own, and a very interesting one.

Escaper was perhaps a good idea, but we didn't have time to investigate the effects in detail. It is likely that having the functionality of escaping local minimum is beneficial for this kind of problem. However, to really see these potential benefits of Escaper, we would need to adjust it's two settings. Namely the amount of shakes performed as well as how long Escaper waits before escaping local minimum. Future work involves investigating these settings to examine potential performance improvements.

## 7 Conclusion

Neither GA nor PSO is good for this problem from what we have found. By comparison, using an exact search algorithm with constraint programming, we would be able to always find a minimal packing for $n = 23$ in just over 4 minutes and $n = 13$ in about three milliseconds, whilst it took PSO and GA several minutes (or even hours) for $n = 13$ with a low probability of finding a minimal packing. Overall, however, GA was better than PSO for this problem with the settings we explored. There is, also, a lot of work left to be done and it is not really possible to draw a conclusion with this few data.

## References

[1] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[2] Alex S. Fukunaga. Restart scheduling for genetic algorithms. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, PPSN V, pages 357–366, London, UK, UK, 1998. Springer-Verlag.

[3] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[4] Lan Huang, Kang-ping WANG, Chun-guang ZHOU, Wei PANG, Long-jiang DONG, and Li PENG. Particle swarm optimization for traveling salesman problems [j]. *Acta Scientiarium Naturalium Universitatis Jilinensis*, 4:012, 2003.

[5] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 5, pages 4104–4108 vol.5, Oct 1997.

[6] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1996.

[7] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[8] Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In *Principles and Practice of Constraint Programming*, pages 52–66. Springer, 2008.

# Appendices

## A Grid Fit's distance calculation

The overlapping squares are found when calculating the total area of the overlap (as described in Section 3.2.1). For each two squares that overlap, only the smaller is considered. $x_i$ and $y_i$ are the coordinates for the bottom left corner of the smaller square and `grid[][]` is the 2D-grid as described.

```
# sum costs for getting out of overlap
for i in overlapping_squares:
        x_i = individual[i]
        y_i = individual[i+1]
        distances = [s]

        ## left direction
        pos = x_i
        y_middle = y_i + square_Size(i)/2
        steps = 0
        # Check if we are outside the grid
        if not (pos < 0 or pos >= s or y_middle < 0 or y_middle >= s):
                while (pos >= 0):
                        if grid[pos][y_middle] == 0:
                                break
                        pos -= 1
                        steps += 1
                # make sure we stayed in the square
                if not (pos < 0):
                        distances.append(steps)

        ## right direction
        pos = x_i + square_Size(i) - 1
        y_middle = y_i + square_Size(i)/2
        steps = 0
        if not (pos < 0 or pos >= s or y_middle < 0 or y_middle >= s):
                while (pos <= s-1):
                        if grid[pos][y_middle] == 0:
                                break
                        pos += 1
                        steps += 1
                # make sure we stayed in the square
                if not (pos > s-1):
                        distances.append(steps)

        ## down direction
        pos = y_i
        x_middle = x_i + square_Size(i)/2
        steps = 0
        if not (pos < 0 or pos >= s or x_middle < 0 or x_middle >= s):
                while (pos >= 0):
                        if grid[x_middle][pos] == 0:
                                break
                        pos -= 1
                        steps += 1
                # make sure we stayed in the square
                if not (pos < 0):
                        distances.append(steps)

        # up direction
        pos = y_i + square_Size(i) - 1
        x_middle = x_i + square_Size(i)/2
        steps = 0
        if not (pos < 0 or pos >= s or x_middle < 0 or x_middle >= s):
                while (pos <= s-1):
                        if grid[x_middle][pos] == 0:
                                break
                        pos += 1
                        steps += 1
                # make sure we stayed in the square
                if not (pos > s-1):
                        distances.append(steps)

        overlap_sum += min(distances)
```