Laboratory 1

Introduction to Artificial Intelligence

Genetic Algorithms

by Ernesto Vieira Manzanera. Student id: 251663

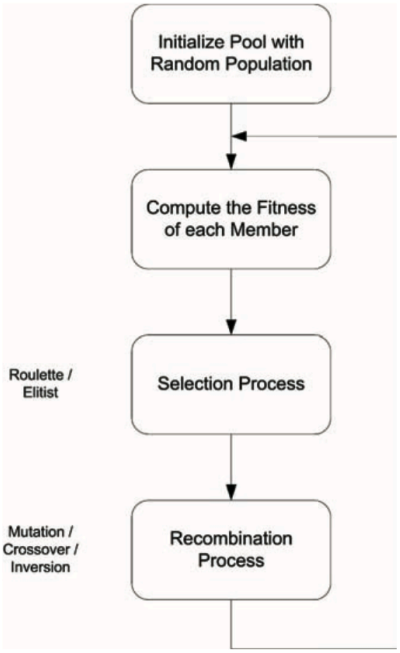
1. Goal of the project

The aim of this research is firstly design and implement a genetic algorithm that retrieves an optimal solution to the Knapsack Problem; and consequently perform an analysis on the outcomes of the simulations as a function of the parameters supplied to the algorithm:

- Crossover Probability.
- Mutation rate.
- Tournament size.
- Population size.

2. Design and implementation of the algorithm

A Genetic Algorithm must be composed of 4 essential stages, depicted in the following figure. To this basic schema, we can add features that are potentially able to improve the performance of our implementation under certain conditions.



2.1 Simple Flow of the genetic algorithm[1]

[1] *M. Tim Jones. Artificial Intelligence: A Systems Approach*

To this process we add two functionalities:

- Generation of a task: an object that represents the parameters of the problem to be solved. In this case, the number of items (`n_items`), maximum capacity (S) of the knapsack and maximum weight (W) allowed. Additionally, a list of `n_items` is generated, with a random set of individual weights, volumes and values (w_i, s_i, c_i). This object is represented by a .csv file.
- Load of a task: reading of a properly formatted .csv file and storing the values that define the knapsack problem to be optimised as a Task type object.

The implementation of the algorithm is carried out by the creation of the so-called "knapsack" module in Python. The module itself makes use of the python modules *csv*, *numpy*, and *matplotlib* to define 4 new Types of objects that support the simulation of the knapsack problem:

- Session object: stores all the values needed and generated in the runtime of the simulation. Its methods define the different stages of the genetic algorithm.

```
knapsack.Session (population_size, tournament_size,
crossover_rate, mutation_rate, iterations)
```

- `Session.population: Population`
- `Session.task: Task`
- `Session.number_of_items: int`
- `Session.population_size: int`
- `Session.tournament_size: int`
- `Session.crossover_rate: float`
- `Session.mutation_rate: float`
- `Session.iterations: int`
- `Session.result: Individual`
- `Session.stats: Stats`

- `Session.generator(n: int, w: int, s: int, output_file: string):` Generates n items with random values in the range, and outputs them into output_file.
- `Session.read_task(input_file: string):` Loads the items from input_file and creates a Task object stored in Session.task.
- `Session.init_population():` Creates a Population object with Session.population_size number of random individuals.
- `Session.tournament():` returns an Individual object with the highest fitness from a random sample of Session.tournament_size individuals from the current population.
- `Session.generate_new_population():` creates a Population object from the individuals in Session.population, as a result of consecutive crossover and mutation process, and stores it in Session.population.
- `Session.crossover(parent1: Individual, parent2: Individual):` Performs crossover between parent1 and parent2 with a probability of Session.crossover_rate. Returns an Individual.
- `Session.mutate(individual: Individual):` performs mutation of Session.number_of_items*Sesssion.mutation_rate genes over individual, and returns a mutated copy.
- `Session.best():` returns the Individual with the highest fitness of Session.population.
- `Session.execute():` performs the genetic algorithm in accordance with the parameters supplied.

- Task: represents the Knapsack problem to be analysed.

`knapsack.Task(n, w, s)`

- `Task.itemCounter: int`
- `Task.n: int`
- `Task.w: int`
- `Task.s: int`
- `Task.items: ndarray(n, 3)`

- `Task.add_item(w: float, s: float, c: float)`: adds the item represented by the tuple (w, s, c) to the items list Task.items.

- Population: stores all the individuals of the population currently being analysed.

`knapsack.Population(size: int)`

- `Population.no_individuals: int`
- `Population.max_size; int`
- `Population.population: ndarray(Population.max_size)`
- `Population.no_of_zeros: int`

- `Population.addindividual(individual: Individual):` add the individual passed as argument to the population.
- `Population.evaluate:` performs fitness evaluation on every element of the population.

- Individual: represents the individual - Knapsack - given by the tuple (wi, si, ci) and its fitness value. A Knapsack is represented as an array (*ndarray* object) of

Session.number_of_items booleans. Each position represents one item, and the value stored establishes wether such item is contained in the knapsack (True) or not (False).

(W, S)
$x_1(w_1, s_1, c_1)$
$x_2(w_2, s_2, c_2)$
$x_5(w_5, s_5, c_5)$
...
$x_n(w_n, s_n, c_n)$

x_1	x_2	x_3	x_4	x_5	...	x_n
`True`	`True`	`False`	`False`	`True`	...	`True`

2.2 Representation of an Individual - Knapsack - as an array of booleans.

`knapsack.Individual(n_items: int, representation = numpy.empty(0): ndarray(n_items))`

- `Individual.representation: ndarray(n_items)`
- `Individual.fitness: value`
- `Individual.value: float`
- `Individual.weight: float`
- `Individual.load: float`

- `Individual.evaluate(task: Task)`: performs evaluation of the individual with respect to the items defined in the task object, and assigns a fitness value to the individual.

3. The fitness function

The aim of the simulation is to find the optimal individual in a space of possible solutions. This space is made up of the 2^n knapsacks that can be created combining all the n possible items. The term "optimal individual" in this problem refers to the individual that:

(I) Has the maximum value, defined as the sum of the values of its items:

$$C = \sum_{i=1}^{n} c_i x_i,\ x_i \in \{0,1\}$$

and

(II) Does not exceed the knapsack constraints:

$$W \geq \sum_{i=1}^{n} w_i x_i \qquad\qquad S \geq \sum_{i=1}^{n} s_i x_i$$

In such position, we evaluate each individual assigning it a fitness value result of the evaluation with a fitness function that will be defined as the following:

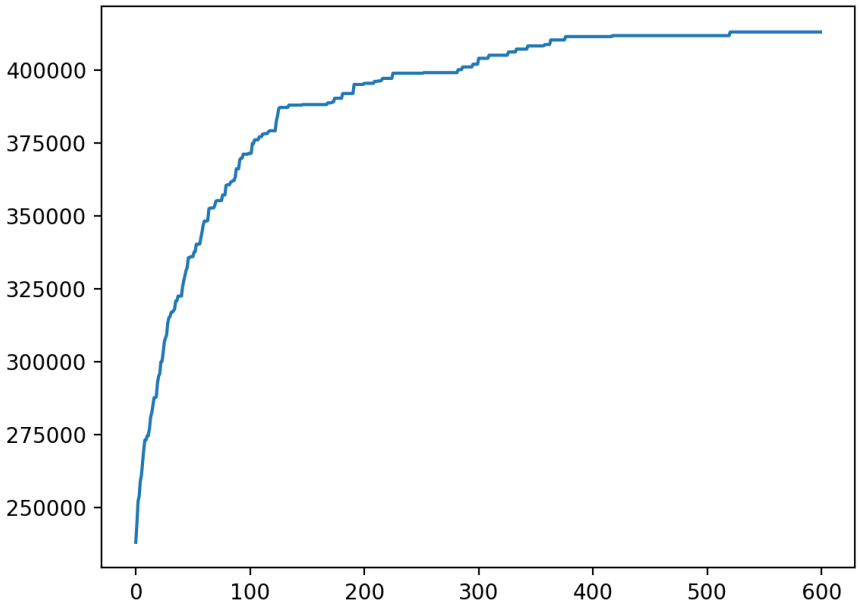
```
fitness = C if (II) is fulfilled
fitness = 0.0 otherwise
```

This fitness function will divide the set of individuals into two subsets: the accepted solutions set, whose members have a fitness value higher than 0.0, and the non-accepted solutions set, made up of those individuals that not fulfil the constraints. In the next point we will describe a tool for visualising both sets.

4. Visualisation of the simulation

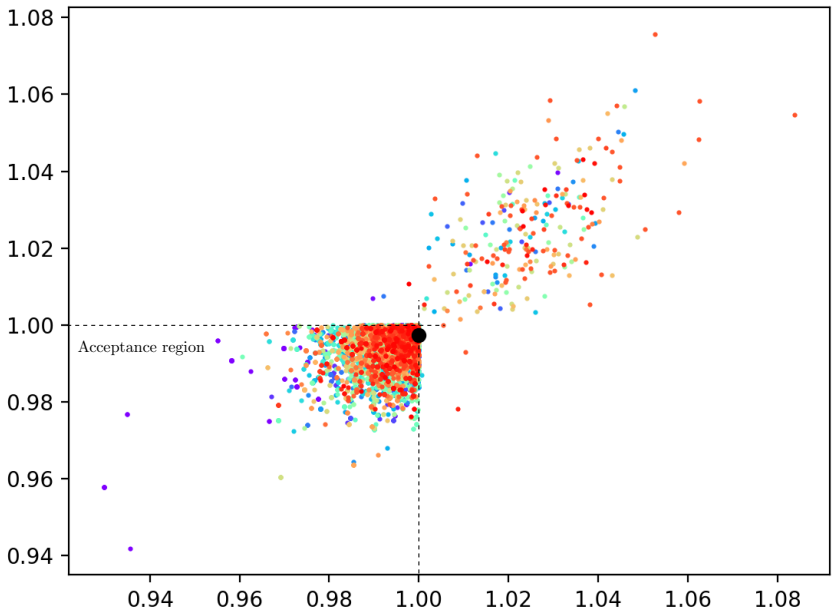
Due to the enormous size of the solutions space, and the discontinuities that exist in the fitness function, a direct representation of the function cannot be achieved. However, we can visualise the performance of the simulation at each iteration with two tools:

- Fitness Value of best individual: in a 2D axis system we can plot the value of the fitness of the best individual at each generation. The growth of the highest fitness over time measured as iterations represents the efficiency of the algorithm and to what extent the simulation has reached an optimal point.



4.1 Fitness plot of the best individual in each iteration

- Relative Weight - Size Map (W/S map): For each generation, we yield a random sample of m individuals, and calculate the relative weight and size with respect to the total knapsack weight and load. Every individual with a normalised weight or load over 1.0 will not be a correct solution, its fitness will automatically be 0.0 and it will fall out of the acceptance region That way, we can visualise all valid individuals inside a 1x1 region, and not valid ones outside that region; as well as the tendency of the individuals towards certain points. A colour key is introduced to indicate the iteration or generation in which the individual was generated: warmer colours indicate latest generations, while blue ones code for the early ones.



4.2 W/S map of the individuals of a simulation

5. Running of the simulation

With the proposed implementation, a simulation would be initialised and run in the following way:

```
1. session = knapsack.Session(pop_size, tournament_size, crossover,
   mutation, iterations)
2. session.generator(N, W, S, ‘‘data.csv’’)
3. session.read_task(‘‘data.csv’’)
4. session.init_population()
5. session.stats = knapsack.Stats(session.task, session.iterations, 50)
6. session.execute()
7. session.stats.plot_fitness("PS = " + str(session.population_size))
8. session.stats.plot_scatter()
9. session.show_stats()
```

Figure 5.1: General setting up end execution of a knapsack simulation

6. Simulation and Analysis

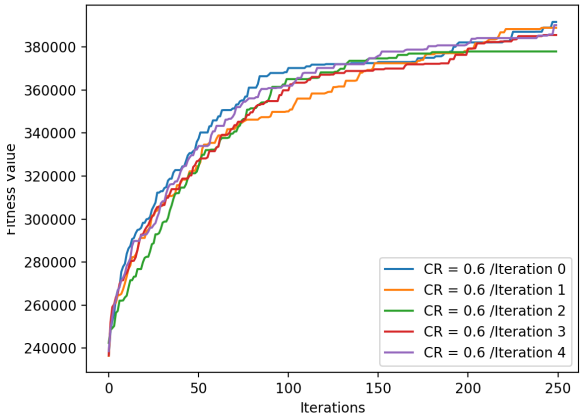
A systematic analysis of the impact of the parameters on the efficiency of the algorithm is presented. The general parameters will be

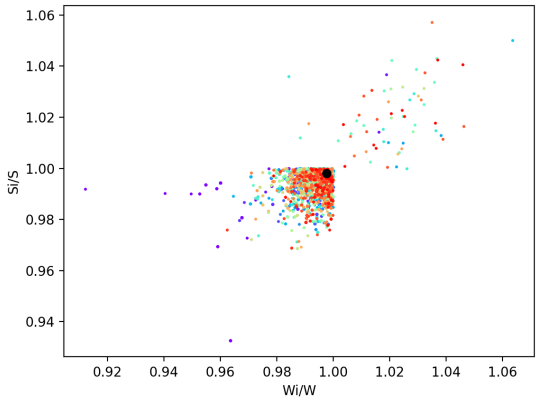
$$(PS, TS, CR, MR, Iterations) = (300,150,0.3,0.01,250)$$

Additionally, an elitism number of $\epsilon = 1$ is included in every iteration, in order to have at least one individual inside the acceptance region. All these parameters will be changed in convenience with the experiment being carried out.

A. Analysis of the impact of the crossover probability

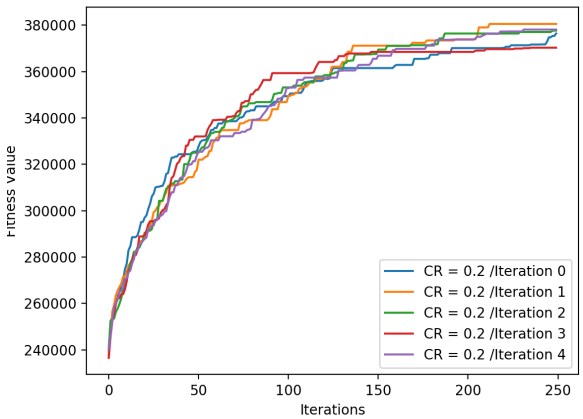
CR = 0.6

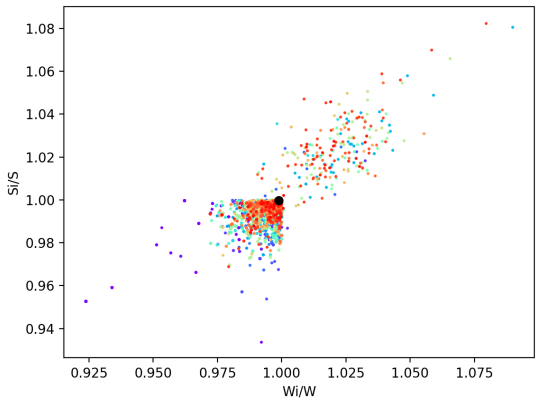





```
ID #0 Fitness: 391685.38421374967
ID #0 Fitness: 388957.9101348827
ID #138 Fitness: 378158.7954870302
ID #264 Fitness: 385933.45350070833
ID #0 Fitness: 390178.59468489414

POPULATION SIZE TEST (CR = 0.6) Average fitness after 5 iterations: 386982.827604253
```

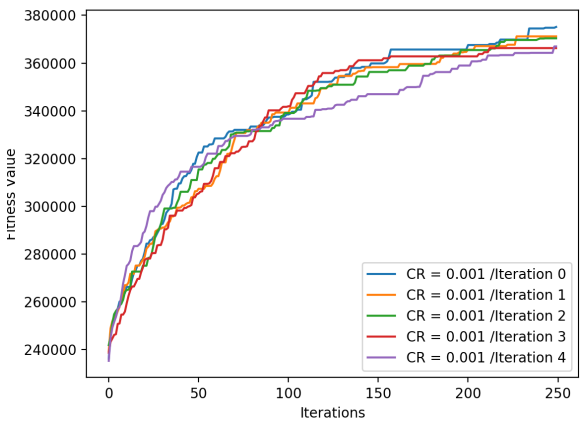
CR = 0.2

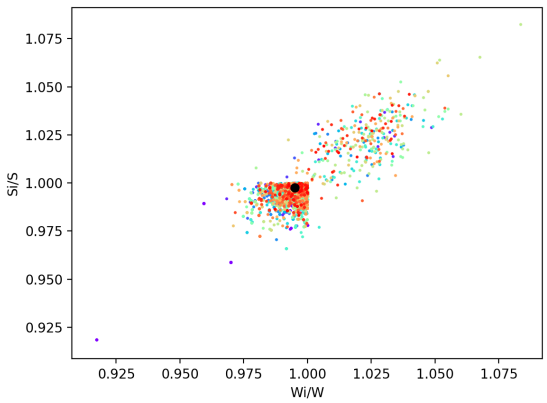





```
ID #0 Fitness: 376371.4766082363
ID #0 Fitness: 380507.4285115071
ID #0 Fitness: 377626.24159964756
ID #0 Fitness: 370269.884546485
ID #0 Fitness: 378075.0621188835

POPULATION SIZE TEST (CR = 0.2) Average fitness after 5 iterations: 376570.0186769519
```

CR = 0.001





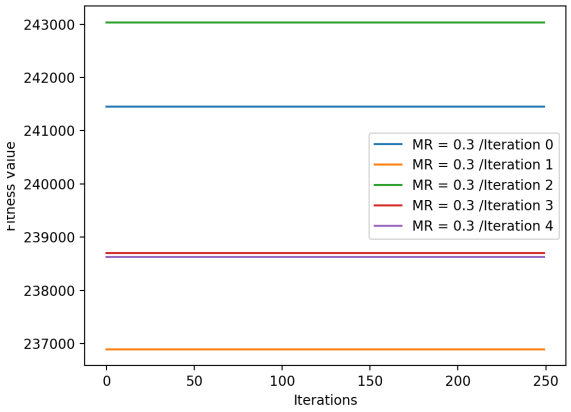

```
ID #0 Fitness: 375150.7089969189
ID #0 Fitness: 371197.7512871732
ID #0 Fitness: 370383.1559236993
ID #0 Fitness: 366290.27422729204
ID #0 Fitness: 366987.2671059361

POPULATION SIZE TEST (CR = 0.001) Average fitness after 5 iterations: 370001.83150820393
```

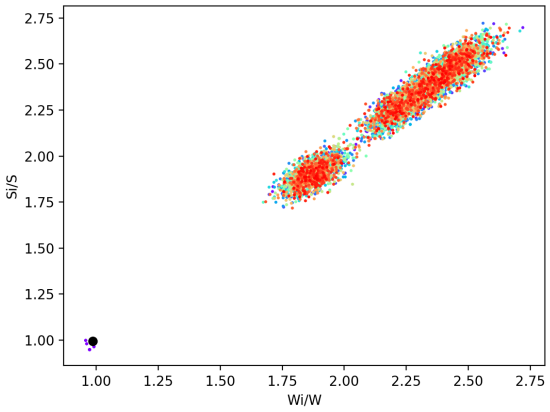
Observing the fitness chart over the iterations, we can visualise the effect of the crossover operator on the generation of new individuals. As we increase the CR, the fitness curve increases its steepness in the early iterations., and provokes a flattening of the curve in the latest ones. On the other hand, looking at the W/S map we observe a smaller number of points in the outer region, which means less uncertainty, throughout the simulation.

This can be due to the fact that 'crossover' is a convergence operator, which aims to find local maximums and pull the population towards it. Thus we are talking about exploration, in order to quickly identify regions were potential solutions might lie. As we firstly want to find those regions, higher crossovers up to certain quantities allow for quicker growth towards potential regions. However, if the CR is too high, populations might be pulled towards a local maximum as soon as we get enough well fitted individuals, not exploring other potential areas of the solution space and thus neglect possible optimal solutions to the problem at hand.

B. Analysis of the impact of the mutation probability

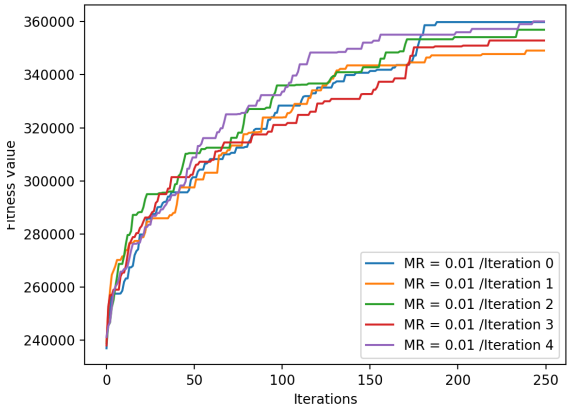
MR = 0.3

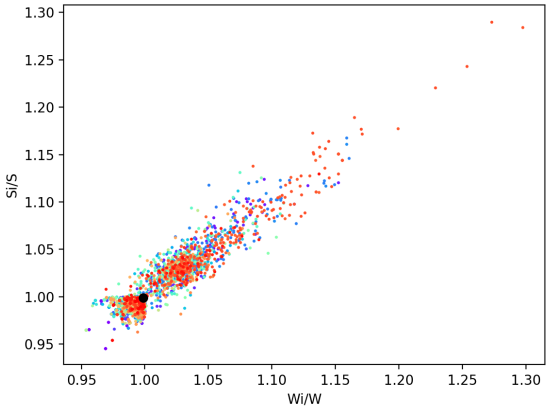





```
ID #0 Fitness: 241447.15175807188
ID #0 Fitness: 236894.35780740177
ID #0 Fitness: 243033.89823351245
ID #0 Fitness: 238697.10882191933
ID #0 Fitness: 238624.84264069324

MUTATION RATE TEST (MR = 0.3) Average fitness after 5 iterations: 239739.47185231972
```

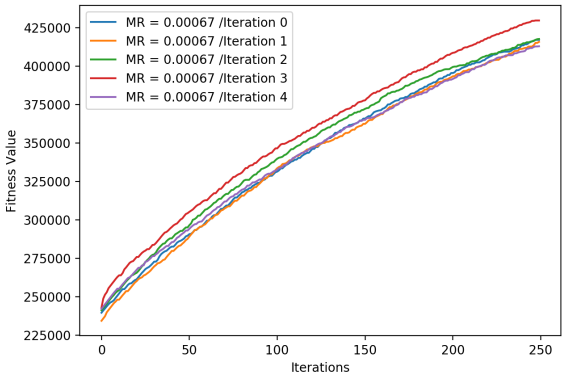
MR = 0.01

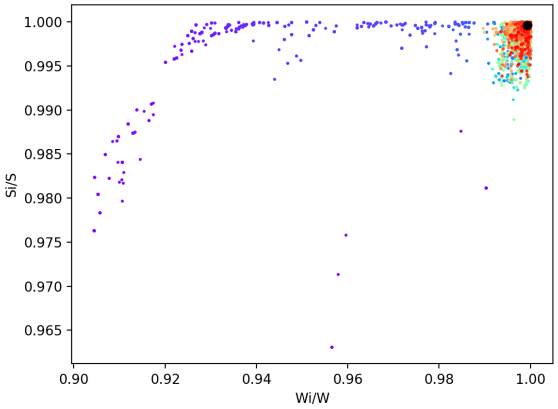





```
ID #0 Fitness: 359827.550276359
ID #0 Fitness: 349099.00133107195
ID #0 Fitness: 356947.0993973025
ID #0 Fitness: 352885.9922730614
ID #0 Fitness: 360089.414866521

MUTATION RATE TEST (MR = 0.01) Average fitness after 5 iterations: 355769.8116288632
```

MR = 0.00067






```
ID #0 Fitness: 417151.29607004824
ID #95 Fitness: 416367.1647584552
ID #67 Fitness: 418262.71205623256
ID #206 Fitness: 430322.683044989
ID #0 Fitness: 412994.61741610995

MUTATION RATE TEST (MR = 0.00067) Average fitness after 5 iterations: 419019.6946691669
```

As we can firstly observe a relatively high mutation rate (0.3) provokes that the initial population quickly jumps from the acceptance region to its outbounds, which avoids any possibility of improvement of the individuals throughout the simulation.

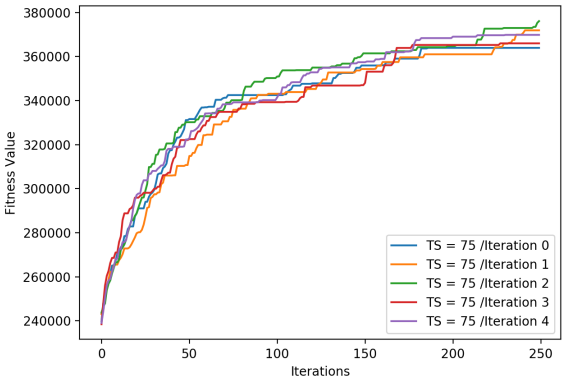
If we decrease the mutation rate to 0.01, we can observe in the W/S Map that the simulation has reached the points around the solution faster and has continued analysing the surroundings, both inside and outside the acceptance region. This solution has great acceptance if we want to minimise the number of iterations, as a good solution is reached in few generations, as we can see in the fitness plot. However, due to the existence of points out of the outside region, many of the individuals in the next generation are going to fall out of the acceptance region and thus not provide any insight of the accepted solutions space.

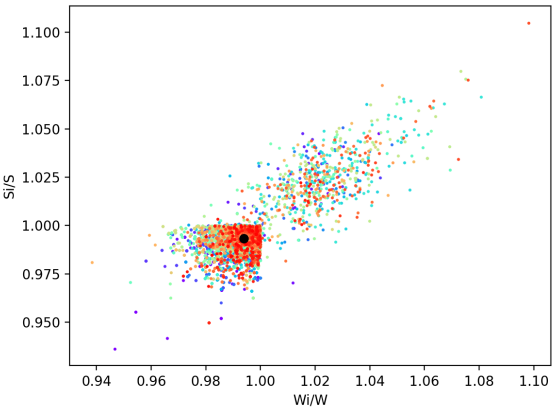
Lastly we observe the diagrams obtained with the 0.00067 simulation. Such a number means that in every mutation process, 0.00067 * 1500 = 1,005 = 1 gene is mutated. If we were to decrease the mutation rate to 0,00066, the number of genes would be 0.99 = 0, which would lead us to a situation like the first chart. As we can see in the W/S map, the points evolve towards the (1.0, 1.0) corner, moving slowly throughout the acceptance region, and never falling off the outer bounds. This supposes an advantage when we can afford multiple iterations, as we are reaching the optimal solution making small changes (1 gene) at each step. This is as well due to the non-existence of individuals outside the 1x1 square, which means that in every new generation, almost every point is going to have a fitness different from 0.0, which gives more opportunities to probe the accepted-solutions space. As shown in the fitness plot, at almost each iteration the algorithm is able to provide a better solution, though not very different than the previous one.

In contrast with the crossover operator, the 'mutation' operator provides a method for exploring the surroundings of the individuals in a smaller scale. This is an advantage when we have reached somewhere near the possible maximum, and we want to find which individual is the most fitted one. Thus, we are talking about exploitation, a characteristic of GAs to measure the capability of the populations to tend towards optimal solution by producing controlled changes into the population.

C. Analysis of the impact of the tournament size

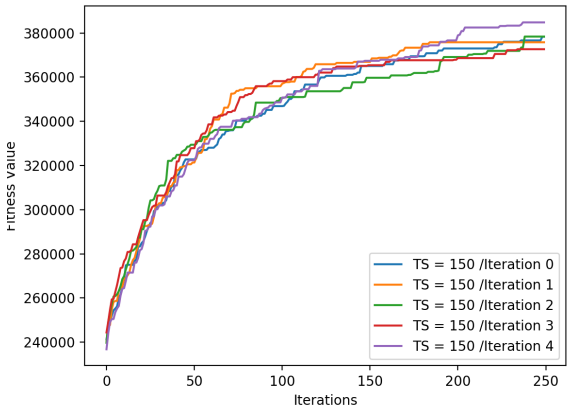
TS = 75 (25% of the population size)

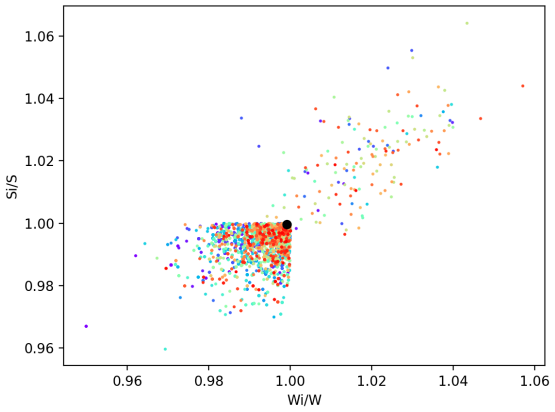





```
ID #0 Fitness: 363912.604439425
ID #0 Fitness: 371867.83205118665
ID #0 Fitness: 376056.515530038
ID #0 Fitness: 365963.3348707468
ID #0 Fitness: 369812.7785760494

TOURNAMENT SIZE TEST (TS = 75) Average fitness after 5 iterations: 369522.61309348914
```

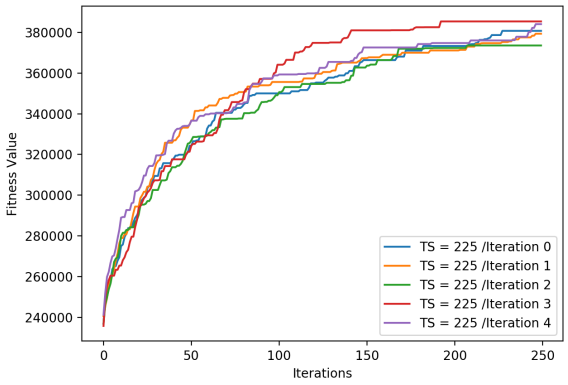
TS = 150 (50% of the population size)

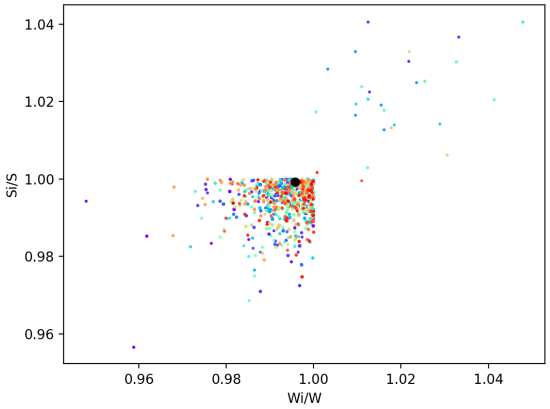





```
ID #285 Fitness: 378572.1429717954
ID #0 Fitness: 375800.60537872557
ID #0 Fitness: 378379.73773107026
ID #0 Fitness: 372651.96083601133
ID #0 Fitness: 384803.9062932265

TOURNAMENT SIZE TEST (TS = 150) Average fitness after 5 iterations: 378041.6706421658
```

TS = 225 (75% of the population size)





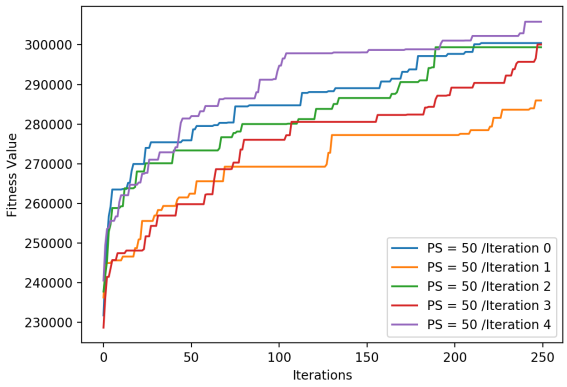

```
ID #0 Fitness: 380781.6628889923
ID #0 Fitness: 379375.4649313101
ID #0 Fitness: 373588.0316188643
ID #0 Fitness: 385413.3633939588
ID #0 Fitness: 384087.527009961

TOURNAMENT SIZE TEST (TS = 225) Average fitness after 5 iterations: 380649.20996861724
```

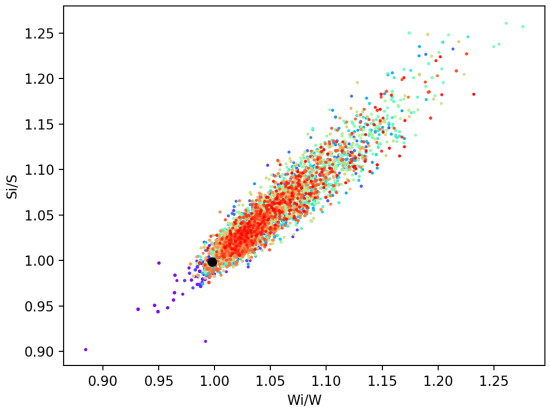
Even though the average of results is quite similar among the 50% and 75% cases, we can observe an improvement from the 25% case. Noticing a less density of points in the outer region of the W/S map of the last figure, we can state that the higher the tournament size is, the smaller the quantity of individuals out of the acceptance region is. This is at first glance a good result, and reasonable since the higher the TS is, the higher are the probabilities that the best-fitted individuals pass on to the next generation and affect it in a higher degree.

However, numbers show that the last fitness, although quite high, has not reached the optimal values that were shown in the 0.00067 Mutation Rate case. Looking at the W/S map, we observe that around the (1.0, 1.0) corner, the density of points from the last iterations (the ones in red) is not that high as in other cases. This is due to the high TS, as we are choosing and passing on the best individuals of the population, which means that almost in every tournament the same individuals end up chosen to generate the next generation. This derives in a lack of diversity when the fitness value is increasing and when we are getting close to the optimal solutions, which is reflected in the fitness plot as a difficult to generate better individuals at higher generations (horizontal plots). In this situation higher mutation rates are needed to provide enough variability and increase the exploitation of the implementation.

D. Analysis of the impact of the population size

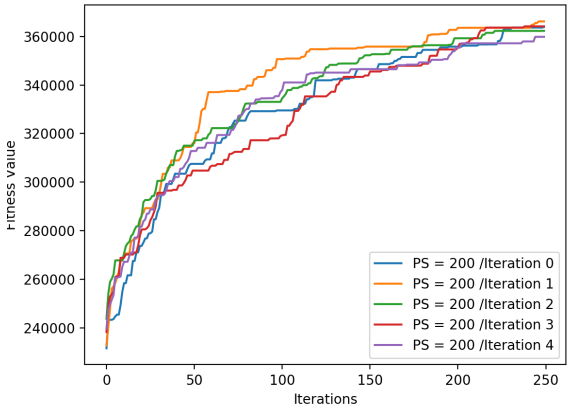
PS = 50

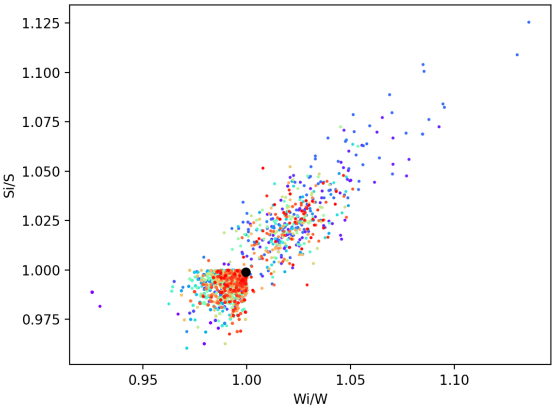





```
ID #0 Fitness: 300437.9714154634
ID #0 Fitness: 285960.1095344483
ID #0 Fitness: 299388.58956907777
ID #0 Fitness: 300074.3259761788
ID #0 Fitness: 305821.1803864066

POPULATION SIZE TEST (PS = 50) Average fitness after 5 iterations: 298336.43537631497
```

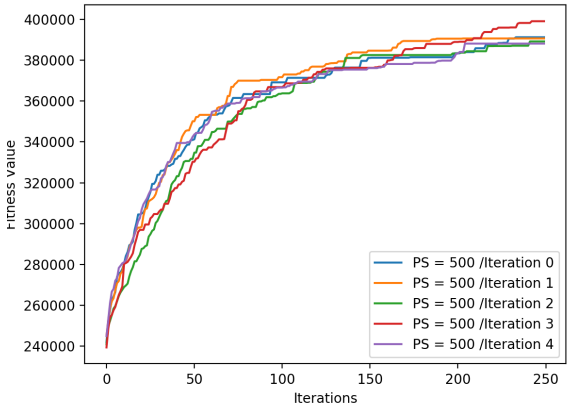
PS = 200

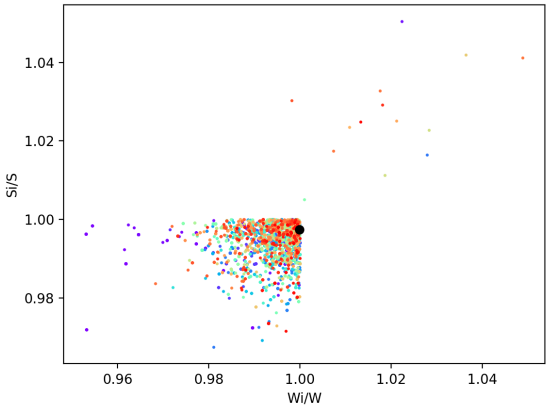





```
ID #20 Fitness: 364236.0494142443
ID #0 Fitness: 366225.18864185794
ID #0 Fitness: 362309.3134609363
ID #0 Fitness: 364180.7739657719
ID #0 Fitness: 359857.75077066687

POPULATION SIZE TEST (PS = 200) Average fitness after 5 iterations: 363361.81525069545
```

PS = 500






```
ID #0 Fitness: 391180.49837430916
ID #0 Fitness: 390589.18295103323
ID #0 Fitness: 389133.12772642495
ID #0 Fitness: 399023.0761757212
ID #0 Fitness: 388081.010393726

POPULATION SIZE TEST (PS = 500) Average fitness after 5 iterations: 391601.3791242429
```

The effect of the population size can be perfectly observed in both plots. Starting with the fitness over iterations graph, we notice that the higher the PS is, the higher the tendency of the simulations to follow the same path is. Now, regarding the plotted individuals in the W/S map, we observe that the population size affects somehow to the distribution of the individuals among the acceptance and non-acceptance region: a higher population size make the individuals tend towards the inbounds of the 1x1 area, and aggregate the points around the top-right corner.

As we have seen in previous points, the higher the number of points in the non-acceptance region and further from the (1.0, 1.0) points, the higher is the uncertainty that faces the generation. This could justify why the fitness curves present such different behaviours from one iteration to another.

It can be observed that smaller number of individuals per population increases the uncertainty of the algorithm. This could be due to the existence of less variability and less probability of the appearance of well-fitted individuals. In order to alleviate the downsides of small populations, as bigger numbers suppose more resources, the crossover rate could be increased, so as to get the exploration higher, or increase the elitism number and select more well-fitted individuals that will be automatically passed on to the next generation.

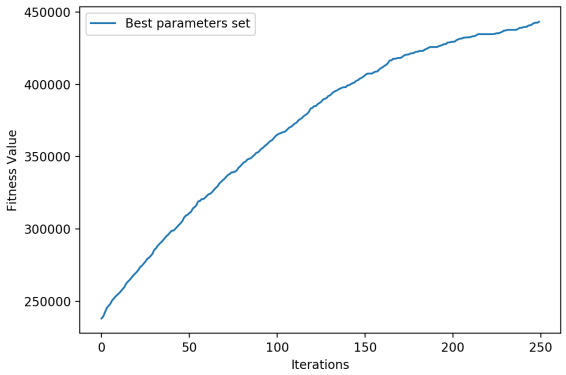
7. Comparison of the solution with non-evolutionary methods

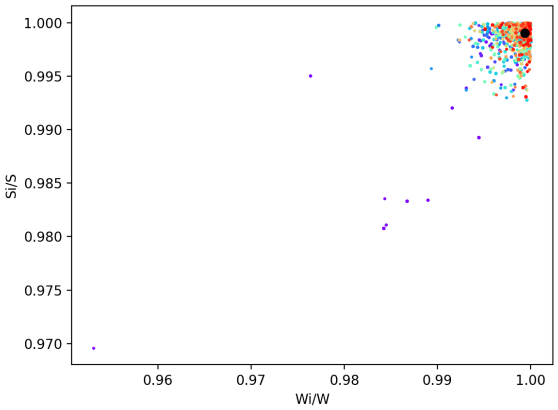
In order to compare the efficiency of the implementation provided in this document, a simple recursive program has been prepared for solving the knapsack problem with two constraints. Unfortunately, the program is not able to calculate a solution in a reasonable

amount of time. Nonetheless, the results of a final simulation with the correct parameters has been run and its result are attached in the following images:

$(PS, TS, CR, MR, Iterations) = (500,400,0.6,0.00067,250)$

```
knapsack.Session(500, 400, 0.6, 0.00067, 250)
```





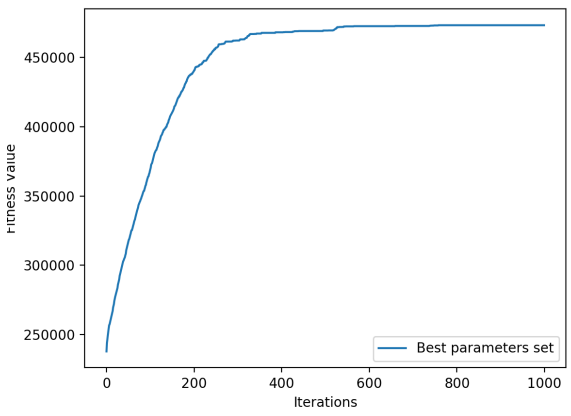

```
ID #0 Fitness: 443251.21946097154

612.2113628387451 seconds.
```

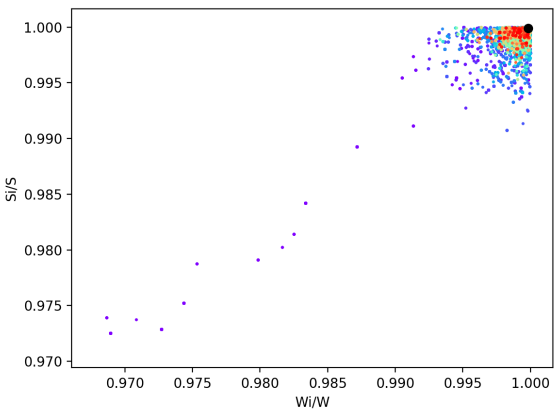
A final simulation with optimal parameters shows a fitness plot that represents an always-growing curve, which means that the simulation finds a better solution at each iteration. Also, due to the small mutation rate, the algorithm probes the accepted solution space slowly, but grows faster towards the local maximum in the first iterations due to the high crossover rate.

The result obtained is the best one among all the obtained with the previous simulations, though the time it took is slightly superior. In order to obtain the best result we can get from the given set of items, a last simulation is going to be run with an iterations number of 1000.

$(PS, TS, CR, MR, Iterations) = (500,400,0.6,0.00067,1000)$

```
knapsack.Session(500, 400, 0.6, 0.00067, 1000)
```





```
ID #462 Fitness: 473369.6931750156

4672.14421415329 seconds.
```

As expected, the simulation finds the best value we have seen in the whole project. It can be seen that the fitness plot shows a very relatively steep curve during the first 250/300 iterations, and it is suddenly flattened for the rest of the simulation.