MAX-SAT Solution Using an Advanced Iterative Method

DAMIAN MALARCZYK

23 JANUARY 2017

1. Problem Statement

Given a Boolean formula `F` having `n` variables (x) and `n` weights, one for each of the variables, solution has to be found such that chosen variables assignment will result in `F(x) = 1`, and that the summary weight of variables set to 1 is maximized.

2. Algorithm

2.1. GENETIC ALGORITHM

Based on the idea of evolution, where an individual represents a sample solution, through individuals selection, their crossover and mutation in order to create new ones and iterative generations of whole population some solutions can be found. Finding optimal solution isn't guaranteed.

To solve the problem I will be using the mentioned genetic algorithm. Focusing on first part of the task the *fitness function* of individuals will be equal to the ratio of satisfied clauses multiplied by `100`. During evolution of the population all individuals which satisfy the formula are being collected, so that no solution is lost and can be used for the second part of the task - maximization of weights assignment. Out of all the collected individuals one with the highest weight will be chosen as a solution.

2.2. DATA STRUCTURE

To increase performance of the code I've decided to use a bit more dedicated approach rather than builtin data structures. Already in the previous homework I was using a simple binary representation of knapsack problem solution, by treating `Integer` type as binary vector storage. This time I've extended this approach, as on a 64-bit machine that would allow me to encoded only 64 variables, thus I've implemented specialized binary vector which under the hood is using an array of integers. Thanks to this there's no limit on amount of variables and it gives a significant performance increase as I will later show in gathered evaluation.

2.3. TESTING MACHINE

Tests were run on MacBook Air with macOS 10.12.1, Intel Core i5 4th gen 1.4 GHz CPU, 4 GB RAM 1600 Mhz DDR3, SSD hard disk model: APPLE SSD SD0128F.

3. Heuristic paramaters

3.1. NOTE

Heuristic parameters evaluation tests are done solely on the genetic algorithm solving problem of satisfiability, neither weights nor adaptive mechanisms are included. All instances come from `DIMACS SAT instances` data.

3.2. RANDOMIZED NATURE

In order to minimize the impact of randomized nature of the genetic algorithm all tests were run multiple (100) times, details regarding repetition will be included in specific test description.

3.3. ITERATIVE POWER

```
Mutation probability: 0.1
```

```
Crossover probability: 0.9
```

```
Crossover method: uniform 3
```

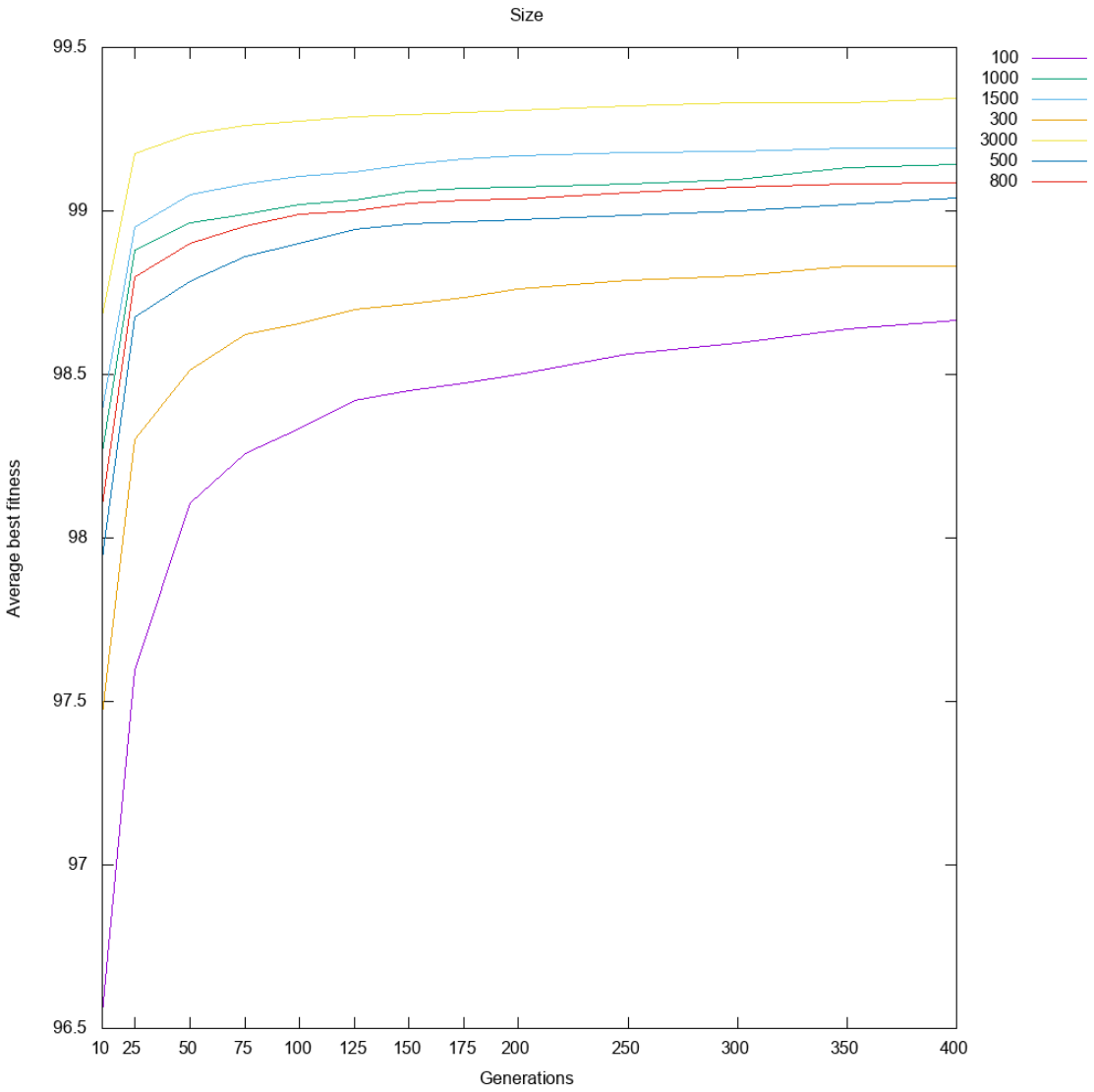
```
Selection method: tournament 5
```

```
Number of variables: 50
```

```
Number of clauses: 218
```

```
Number of instances: 1
```

For this test I've used instance with smaller number of variables because of the tested population sizes, having chosen instances with more variables and testing big population size, final results could be highly affected by the random initial population.



From the graph we can see that the size of a population has higher impact on the end result than number of iterations.

3.4. FURTHER EVALUATION

All of the further tests will have such default parameter values:

`Size of population: 50`

`Mutation probability: 0.1`

`Crossover probability: 0.9`

`Crossover method: uniform 3`

`Selection method: tournament 5`

`Elitism: 1`

`Generations: 75`

(unless stated otherwise - testing given parameter)

Tests will be conducted on such data:

`Number of variables: 250`

`Number of clauses: 1065`

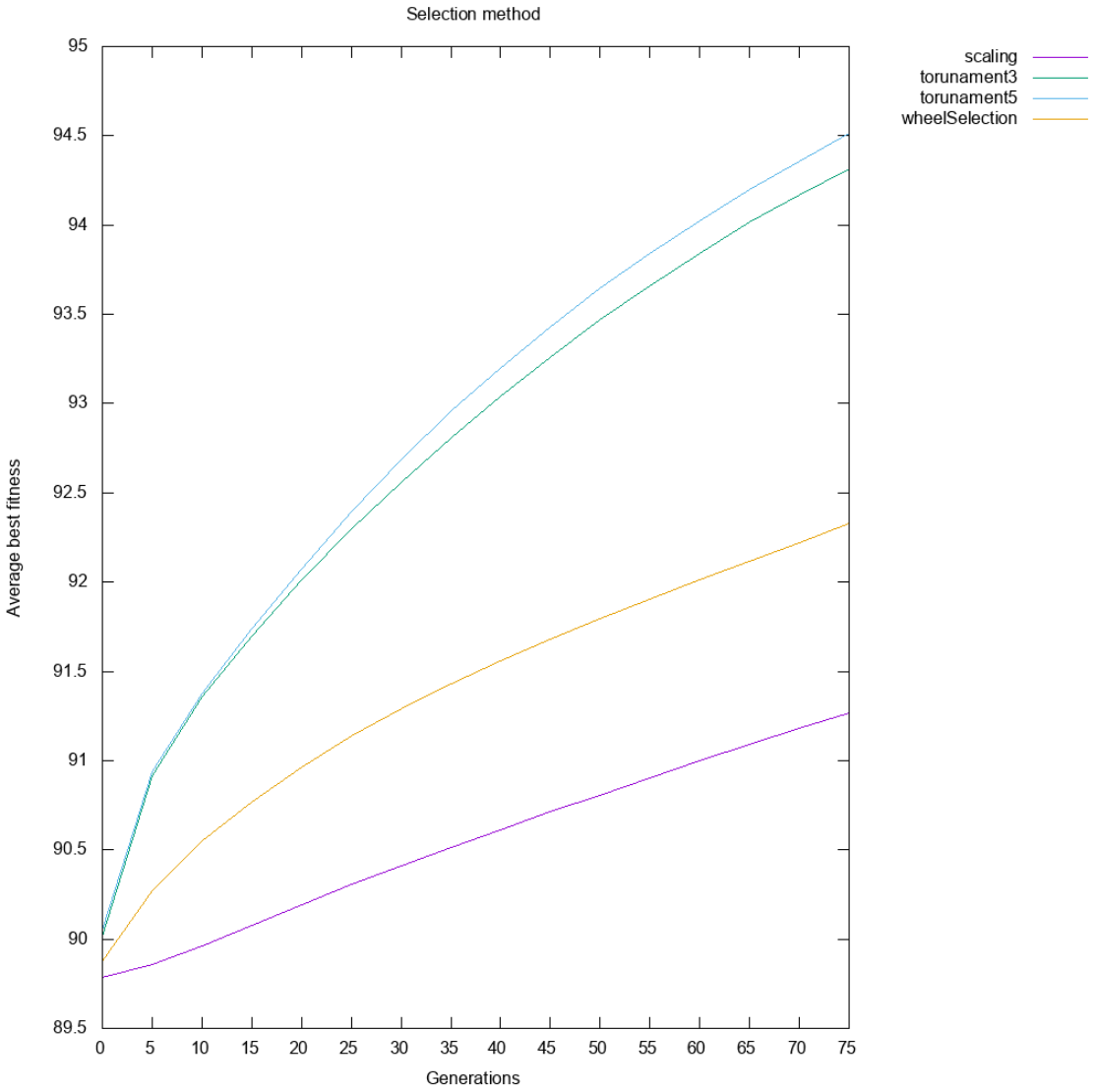
`Number of tested instances: 10`

All of the evaluations (excluding mutation, crossover probability), are made in a following manner: `100` repetitions for each possible configuration, up to given number of generations collecting gathered results every `5` generations. The graphs present average fitness value of best individual within all repetitions for given generations.

3.5. SELECTION METHODS

Tested methods:

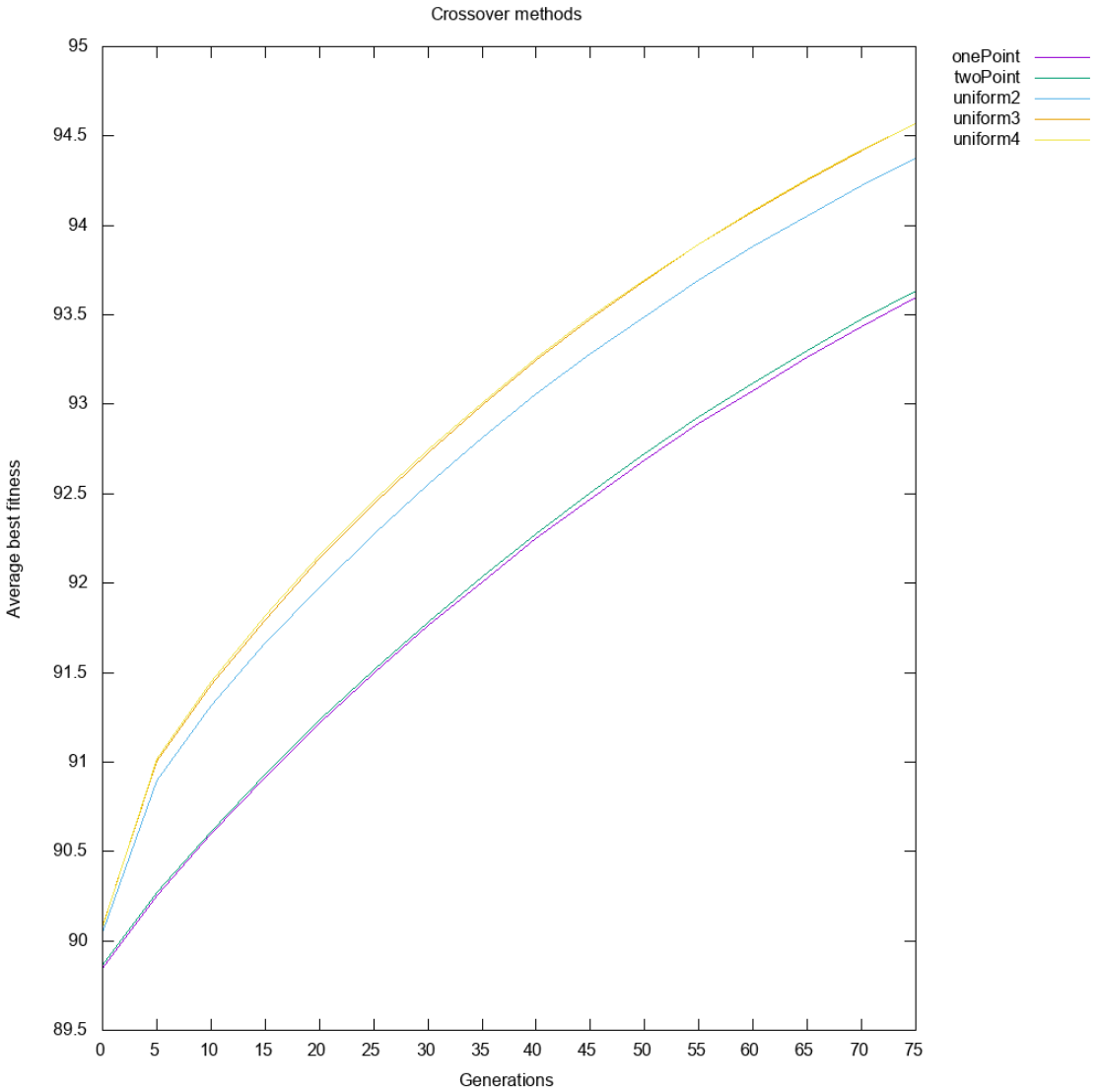
- Wheel selection
- Tournament of size 3 and 5
- Scaling - probability of an individual being chosen is ratio of it's fitness to fitness of the best individual



3.6. CROSSOVER METHODS

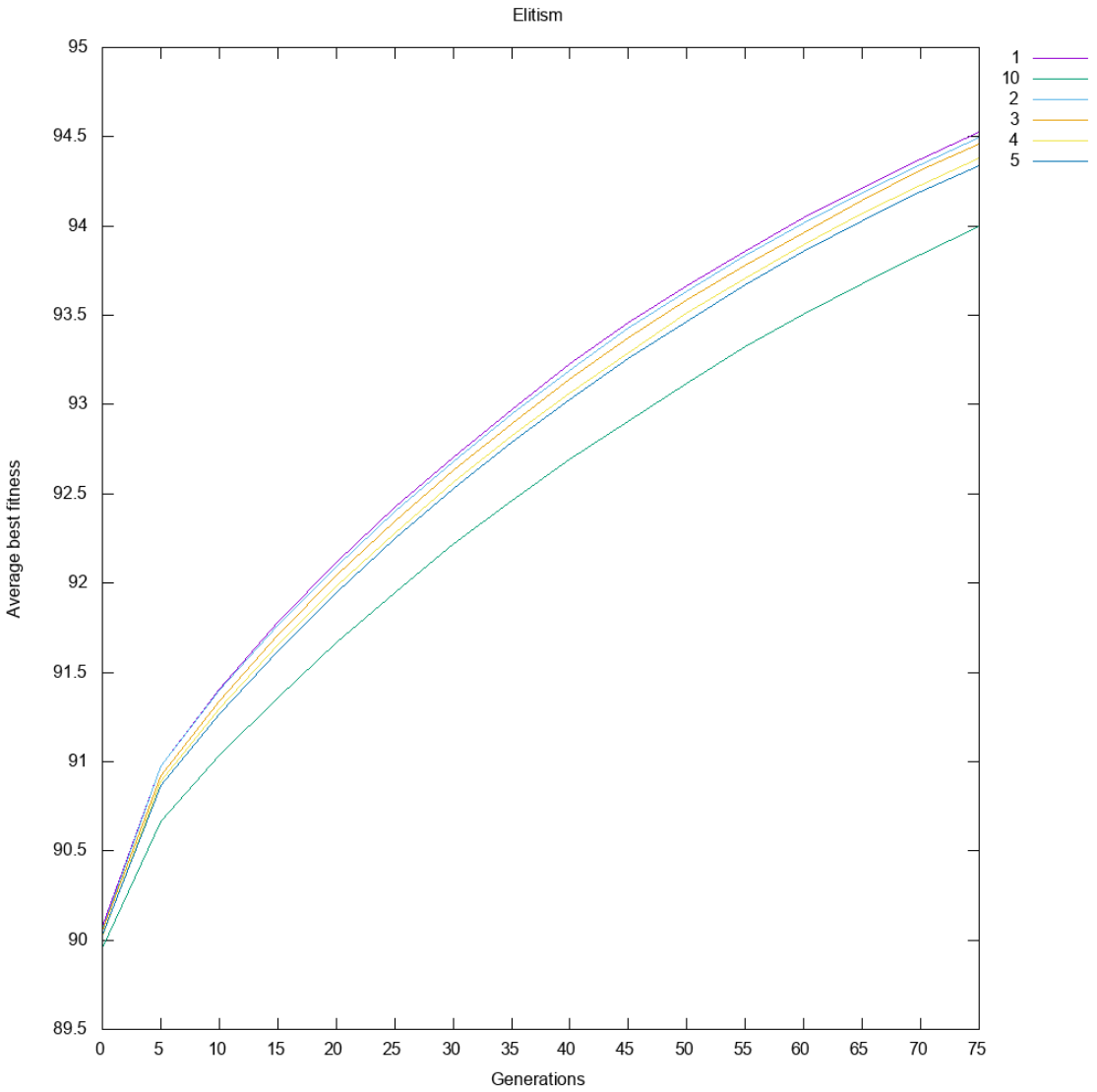
Tested methods:

- One point
- Two point
- Uniform points, with x - 2, 3, 4 where `number of variables / x` is maximum number of crossover points



As the instances have quite a few variables it was possible to predict that uniform crossover would have better results, one, two points are simply not enough for this problem.

3.7. ELITISM

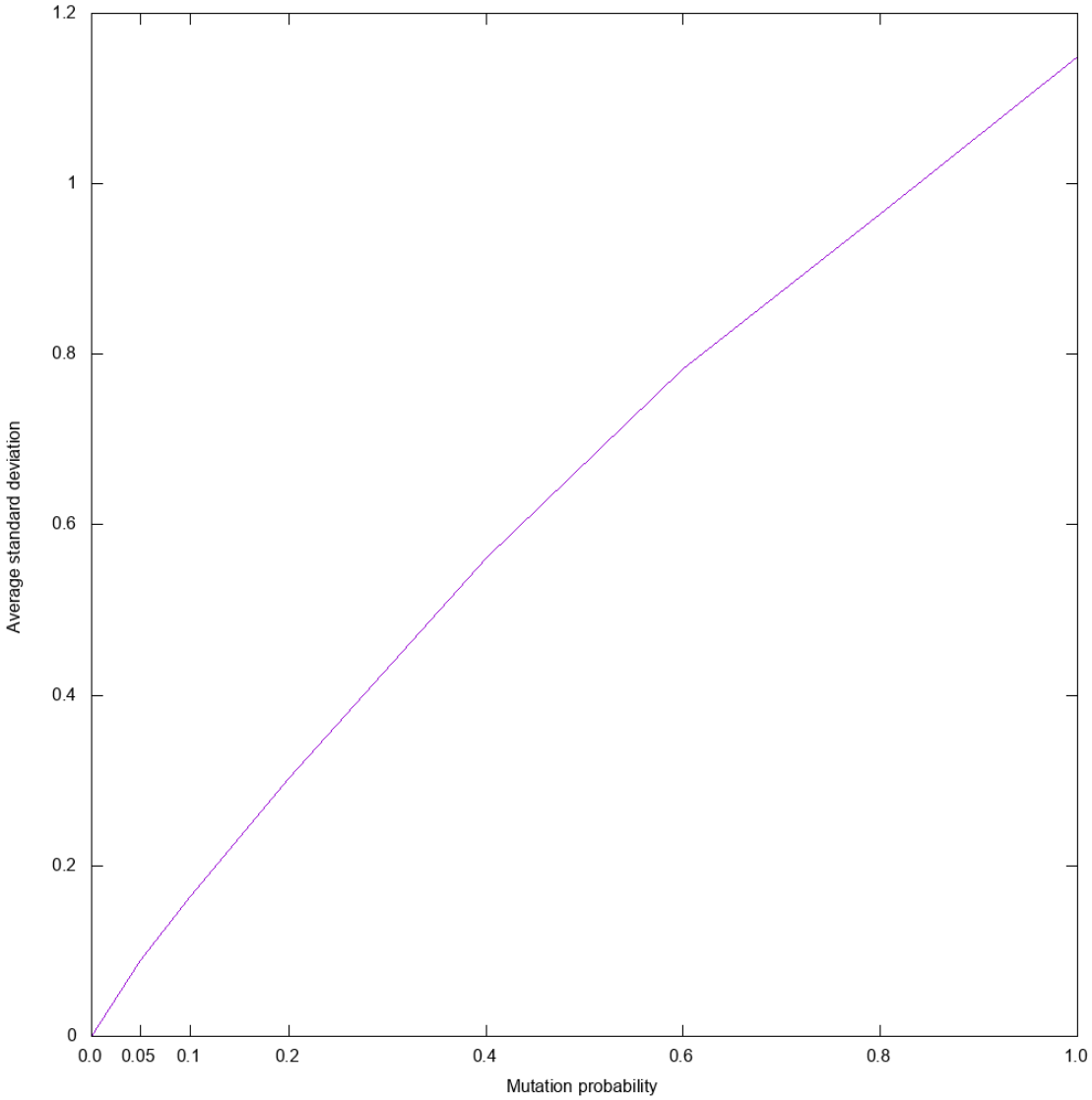


Slight increase of elitism does not seem to have big impact on general algorithm performance, as it turns out for this problem one elite individual should give best results.

3.8. MUTATION PROBABILITY

The graph shows average standard deviation of individuals fitness reduced by a baseline value computed discarding mutation operation. This allows to see how much individuals differ based on the mutation rate.

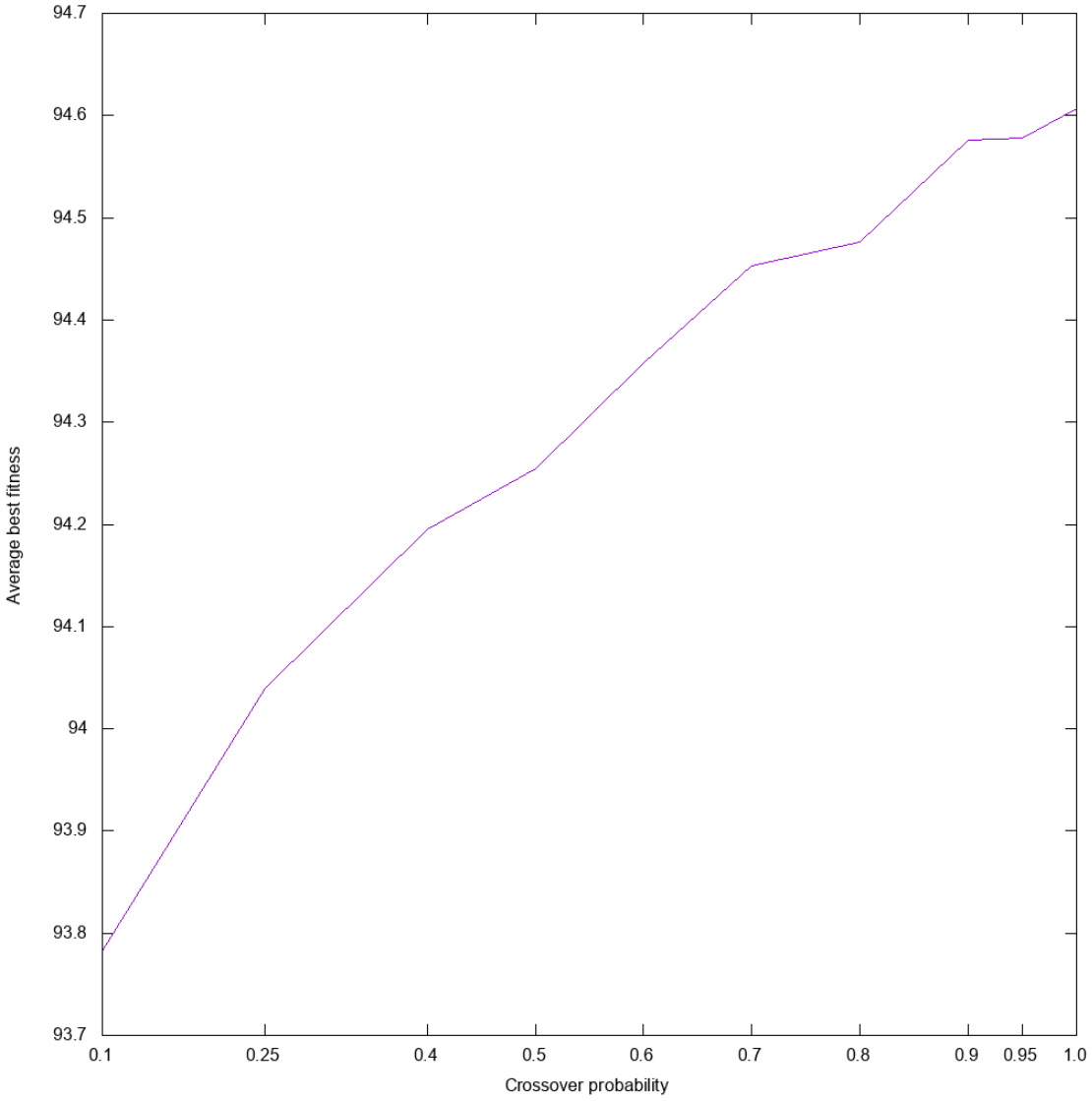
In contrary to previous tests, results are collected after `each` generation rather than after each `5` of them. Standard deviation is computed per results generated after each generation and then average value of it is presented in the graph.



Optimal solution will be to use lower mutation probability rate, as in case of high diversity of individuals there's a high probability of not preserving good genes, yet the value cannot be too small either, otherwise there would be not enough evolution of the individuals.

3.9. CROSSOVER PROBABILITY

To measure crossover impact on final result presented fitness value is based only on fitness of best individuals after all generations. Iterative step after `5` generations is not taken into account.



4. Algorithm evaluation

4.1. PARAMETERS

`Mutation probability: 0.1`

`Crossover probability: 0.9`

`Crossover method: uniform 3`

`Selection method: tournament 5`

`Elitism: 1`

`Size of population: 2000`

`Generations: 150`

4.2. ADAPTIVE MECHANISM

4.2.1. Repetition If after first iteration of 150 generations no solution satisfying the formula is found then size of population will be increased by `1000` and another attempt will be made. At maximum there will be 3 attempts trying to find solution. If then still no solution is found the algorithm will exit and return individual with the highest fitness value.

4.2.2. Weights If only one solution satisfying the formula is found its' weight will be computed and it will be given as final solution. If at least two solutions are found another instance of genetic algorithm will be created using the solutions as initial population, if amount of found solutions is lower than 10, then during execution there will be expansion of population, otherwise size of population will be the amount of found satisfying solutions. There will be `1000` iterations of the weights genetic algorithm, it's *fitness function* is weighted value of the solution and the tournament selection size is decreased to `2`. Best individual found by the second genetic algorithm instance will be the final solution.

4.3. RESULTS

4.3.1. Measurements

Variables	Clauses	Average standard deviation	Average failures	Average time(s)	Average adaptive help	Average error	Average unique results
20	91	17.23	0	2.07	0	0.00014	1
50	218	25.79	3	7.31	3	0.00229	2
75	325	17.3	8	13.77	0	0.00923	0
100	430	0	8	19.46	0	0.0066	0
150	645	0	8	25.5	0	0.01258	0

Measurements were done on 10 instances, repeating each of them `8` times. Weights were randomly selected once for all tested instances, possible values were in range `[1-100]`.

4.3.2. Description

- *Average standard deviation* - standard deviation of found weight results
- *Average failures* - average amount of repetitions that did not manage to find such assignment of variables that would satisfy the formula
- *Average time* - average computational time in seconds
- *Average adaptive help* - average amount of times within repetitions that adaptive part of the algorithm - mentioned second genetic algorithm - helped to improve final results
- *Average error* - satisfiability error, range `[0-1]` of satisfied clauses ratio, where 0 stands for satisfied formula and 1 means no clause was satisfied
- *Average unique results* - average amount of unique found solutions

4.4. BINARY VECTOR AND ENCODED INTEGER VECTOR PERFORMANCE

4.4.1. Measurements

Encoded Integer vector

Structure	Average standard deviation	Average failures	Average time(s)	Adaptive helped	Average error	Average unique results
Encoded integer vector	29.81	4	9.64	3	0.00287	2
Builtin bool array	39.89	5	115.33	3	0.0047	2

Measurements were done on `5` instances with `50` variables and `218` clauses, with `8` repetitions. The idea is to show execution time difference, while I present gathered results as well those are caused by the implementations itself. More experiments would be required to tell if encoded vector definitely provides better solutions.

4.4.2. Reasoning The main performance issue when using a binary vector is that each of the contained variable assignment has to be chosen separately, thus having an instance of `n` variables there will be `n` random number generations to choose whether to set the variable to true or false. In case of encoded `Integer` vector only `floor(n / 64) + 1` such operations will be required (in case of 64 bit machine). Also typical vector operations are quite faster i.e. accessing, storing values. Additional advantage of the encoded solution representation is lower memory usage, to store `n` variables in a binary vector `n` bytes will be required, in case of encoded vector only `8` bytes will be used per each 64 variables, which also has an indirect impact on performance - `Swift` the language I'm using to solve the problem has value type array storage, meaning such an expression

```
var a = [1, 2, 3, 4]
var b = a
```

Will result in copying contents of variable `a` to variable `b`, which if happens often will have significant impact on the performance. That's just the general idea though, actually there's a technique called copy-on-write implemented meaning `b` will be just using reference to `a` as long as it's not mutated, when mutation occurs (e.g. adding, removing, changing content of an array) then the copy is made.

4.5. CONCLUSIONS

The algorithm was implemented and tested with many different methods and parameters setting, but it performed well enough only in case of instances with 20, 50 variables, for bigger ones on average it does not manage to find an assignment of variables that would make the formula fully satisfied.

On the other hand the error is very small and does not significantly grow with increased instance size, in most of those cases removing few clauses would be enough to make the formulas satisfied.

As for weights themselves, the standard deviation may not be that small, but on average the algorithm returns 1-2 unique solutions provided that it was able to find such assignment of variables that would satisfy the formula.

Case of parameters is initially unclear and depends on the problem, with all the conducted experiments represented on the graphs it was quite clear for which of them the algorithm performed best.

The algorithm performs quite well when it comes to speed even with big populations, the problem for bigger tested instances is not being able to omit local minimas and to find solution that would fully satisfy the formula and it is now programmed to try to find solution regardless of spent time, thus longer average execution time. It would require further experiments with other instances to find good threshold function.