MI-PAA: Assignment #5

Petr Hanzl
hanzlpe2@fit.cvut.cz

Czech Technical University - Faculty of Information Technology — February 16, 2019

Weighted Maximum Satisfiability Problem - 3SAT (WMSP-3SAT)

Given a conjuctive normal form formula F with positive weights $w_i \in \{w_1, \dots, w_n\}$ assigned to each variable $v_i \in \{v_1, \dots, v_n\}$ and each clause has exactly three variables, find values $x_1, \dots, x_n$ for its variables that satisfy the formula F and maximize weight of all truth variables.

Formula F in CNF with 3 variables in each clause:

$$(v_a \vee v_b \vee v_c) \wedge \dots \wedge (v_d \vee v_e \vee v_f)$$

$$\max \sum_{0}^{n} x_i \cdot w_i$$

1 Generator of problem instances

Algorithm for generating problems creates random clauses with random variables. If number of unique variables in a generated formula is not equal to gives number of variables provided by user, does not count, the algorithm replaces one random clause. It also checks if one variable in not already in the same clause. If true, the variable is changed with another one. It is possible to control number of variables, number of clauses, maximum and minimum in weight distribution and distribution of negation in the formula. Weights have normal distribution and negations have gaussian distribution with a posibility to control its skewness.

Parametr	**Description**
variables	Number of variables in a formula
clauses	Number of clauses in a formula
min	Minimal weight in a normal distribution
max	Maximal weight in a normal distribution
skew	Skewness of gaussian distribution of negations in a formula

2 Genetic Algorithm

Many natural processes have found an application into problem solving and optimization problems. The evolution and the natural selection is not an exception. There are more algorithms which can simulate the evolution, but they usually use similar mechanisms to reproduce an optimal or at least a suboptimal solution. These mechanism are: reproduction, mutation a selection. For every GA, a definition of a fitness function, which scores a quality of an individual, is very important.

2.1 Fitness

Fitness is defined as a quality of a feasible solution. I used just a simple sum of weights of all truth variables.

Algorithm 1: `computeFitness`

Input: $(chromosome[N])$
$chromosome[N]$ - a chromosome of one solution
Result: $(fitness)$
$fitness$ - a quality of a given solution

```
fitness ← 0
for i ← 0 to chromosome.length do
    if chromosome[i] == true then
        fitness+ = weights[i]
    end
end
satisfiable ← true
for clause ∈ clauses do
    clauseSatisfiable& = false for variable ∈ clause do
        if variable > 0 then
            clauseSatisfiable| = chromosome[variable]
        end
        if variable < 0 then
            clauseSatisfiable| =!chromosome[variable]
        end
    end
    if clauseSatisfiable! = true then
        randomVariable ← pickRandomVariable()
        if chromosome[randomVariable] < 0 then
            chromosome[randomVariable] ← false
        end
        if chromosome[randomVariable] > 0 then
            chromosome[randomVariable] ← true
        end
    end
    satisfiable&clauseSatisfiable
end
if satisfiable! = true then
    fitness ← 0
end
return (fitness)
```

2.2 Variance coefficient

The variance coefficent is calculated from fitness distribution in population every generation to control mutation and crossover. Its formula is:

$$varienceCoefficient = \frac{standardDeviantion}{average}$$

2.3 Reproduction

And algorithm for reproduction randomly picks individuals from the population, creates new feasible solutions and removes the original ones from population to not pick them again for crossover. If a variance in population is too low, reproduction process tends to return completely new randomly set individuals more often instead of crossovering already found solutions. This is controlled by a global variable **crossoverProbability** which is computed from variance. Another global variable **populationSize** is used as a stopping criterion of the reproduction. The probability that a sibling would be added to new population instead of putting there a random individual is calculated as a subtraction of 1 by variance coefficient divided by length of chromosome.

Algorithm 2: `crossover`

Input: $(population[M])$
$population[size]$ - a population with M individuals
Result: $(population[M])$
$population[N]$ - the same population but with different individuals

---

$originalSurvivorsCount \leftarrow population.length$
while $population.length \lg populationSize$ **do**
  $randomIndividual1 \leftarrow floor(random() \cdot originalSurvivorsCount)$
  $randomIndividual2 \leftarrow floor(random() \cdot originalSurvivorsCount)$
  if $randomIndividual1 == randomIndividual2$ **then**
    continue
  end
  $\{sibling1, sibling2\} \leftarrow singleCrossover(randomIndividual1, randomIndividual2)$
  if $crossoverProbability > random() \&\& population.length \leq populationSize$ **then**
    $population.push(sibling1)$
  end
  if $crossoverProbability > random() \&\& population.length \leq populationSize$ **then**
    $population.push(sibling2)$
  end
end
return $(population)$

2.3.1 Single-point Crossover (SPC)

Genomes of two parents are split at randomly picked point and the parts are swapped to make two siblings. After that, chromosomes of these siblings are repaired to satisfy given formula F.

Algorithm 3: `singlepointCrossover`

Input: $(chromosome1[N], chromosome2[N])$
$chromosome1[N]$ - a chromosome of one randomly chosen solution
$chromosome2[N]$ - a chromosome of second randomly chosen solution different from the first
Result: $(sibling1[N], sibling2[N])$
$sibling1[N]$ - a new feasible solution
$sibling2[N]$ - a new feasible solution

$randomPoint \leftarrow floor(random() \cdot chromosome1.length)$
$ch1p1 \leftarrow chromosome1[0..randomPoint]$
$ch1p2 \leftarrow chromosome1[randomPoint..chromosome1.length]$
$ch2p1 \leftarrow chromosome2[0..randomPoint]$
$ch2p2 \leftarrow chromosome2[randomPoint..chromosome2.length]$
$sibling1 = [...ch1p1, ...ch2p2]$
$sibling2 = [...ch2p1, ...ch1p2]$
$repair(sibling1)$
$repair(sibling2)$
return $(sibling1, sibling2)$

2.3.2 Uniform Crossover (UC)

In uniform crossover, each bit of a child is randomly picked from one of its parents.

Algorithm 4: `uniformCrossover`

Input: $(chromosome1[N], chromosome2[N])$
$chromosome1[N]$ - a chromosome of one randomly chosen solution
$chromosome2[N]$ - a chromosome of second randomly chosen solution different from the first
Result: $(sibling1[N], sibling2[N])$
$sibling1[N]$ - a new feasible solution
$sibling2[N]$ - a new feasible solution

$sibling1 = []$
$sibling2 = []$
for $i \leftarrow 0$ ***to*** $chromosome1.length$ **do**
  if $random() > 0.5$ **then**
    $newChromosome2[i] \leftarrow chromosome1[i]$
    $newChromosome1[i] \leftarrow chromosome2[i]$
  end
  else
    $newChromosome2[i] \leftarrow chromosome2[i]$
    $newChromosome1[i] \leftarrow chromosome1[i]$
  end
end
$repair(sibling1)$
$repair(sibling2)$
return $(sibling1, sibling2)$

2.4 Mutation

For the GA, there is a simple way how to mutate bits and it is flipping their values. Probability of flipping a bit is also controlled by variance of the population, concretely in a global variable **mutationProbability**. It is calculated from variance coefficient divided by length of chromosome.

Algorithm 5: `mutation`

Input: $(chromosome[N])$
$chromosome[N]$ - a chromosome of one solution
Result: $(chromosome[N])$
$chromosome[N]$ - a mutated feasible solution

---

for $i \leftarrow 0$ ***to*** $chromosome.length$ **do**
  $randomNode = floor(random() * chromosome.length)$
  if $random() < mutationProbability$ **then**
    $chromosome[i] \leftarrow !chromosome[i]$
  end
end
$repair(chromosome)$
return $(chromosome)$

2.5 Selection

2.5.1 Best-First (BF)

One heuristic used in this assignment is to pick the best solution from current generation a put it unchanged into next generation. This prevents losing the best-yet found solution. Some of the solutions are more difficult to find, because they can stay in very tight range of steep gradient with very few solutions.

2.5.2 Tournament

Five individuals are repeatedly picked from the population and the best two solutions with the highest fitness survives to the next generation.

Algorithm 6: `tournament`

Input: $(population[M])$
$population[size]$ - a population with M individuals
Result: $(population[M])$
$population[N]$ - a new population with surviving solutions

$newPopulation \leftarrow []$
$tournament \leftarrow []$
$i \leftarrow M$
while $i > 0$ **do**
  $randomIndividual \leftarrow floor(random() \cdot originalSurvivorsCount)$
  $tournament.push(randomIndividual)$
  $tmp \leftarrow population[i]$
  $population[i] = population[randomIndividual]$
  $population[randomIndividual] = tmp$
  $i - -$
  if $tournament.length \geq 5$ **then**
    $tournament.sortByFitness()$
    $newPopulation.push(tournament[0])$
    $newPopulation.push(tournament[1])$
    $tournament = []$
  end
end
return $(population)$

2.5.3 Catastrophe (C)

Most individuals in the population are killed, when the variance coefficient gets below 0.00001. The catastrophe leads to higher standart deviation in population and finds new ways of solutions. For my experiments, I set up, that only 10% of individuals, which are sorted in descending order by their fitness, survives the catastrophe.

2.6 Repair

To evaluate formula in CNF as a true formula, all clauses has to be true, which means, at least one variable or its negation has to be true. The repair algorithm tries to fulfill all clauses by random selection of variable and setting it in this way. After repairing all clauses, the algorithm checks if the formula is true. If it is not true, the algorithm continues to find other random configuration.

2.6.1 Full Random Repair (FRR)

Algorithm 7: `FullRandomRepair`

Input: $(chromosome[N], clauses[M])$
$chromosome[N]$ - chromosome of an individual with length of N variables, not necessarily a feasible solution
$clauses[M]$ - a list of M clauses
Result: $(chromosome[N])$
$chromosome[N]$ - a feasible solution

```
satisfiable ← false
while satisfiable! = true do
    satisfiable ← true
    for clause ∈ clauses do
        clauseSatisfiable& = false
        for variable ∈ clause do
            if variable > 0 then
                clauseSatisfiable| = chromosome[variable]
            end
            if variable < 0 then
                clauseSatisfiable| =!chromosome[variable]
            end
        end
        if clauseSatisfiable! = true then
            randomVariable ← pickRandomVariable(clause)
            if chromosome[randomVariable] < 0 then
                chromosome[randomVariable] ← false
            end
            if chromosome[randomVariable] > 0 then
                chromosome[randomVariable] ← true
            end
        end
        satisfiable&clauseSatisfiable
    end
end
return (chromosome[N]) ;
```

2.6.2 Heuristic Repair (HR)

I also implemented more sofisticated heuristic repairing algorithm. The heuristic is based on maximazing sum of clauses. As it was said before, every clause need to be true to fulfill requirement of true formula. If some clause is not true, the algorithm sets true its variable with highest weight. If there is negation of variable with highest weight, the algorithm sets correctly a different variable. Its value depends if it is negation of variable or not.

Algorithm 8: `HeuristicRepair`

```
Input: (chromosome[N])
chromosome[N] - chromosome of an individual with length of N variables, not necessarily a
feasible solution
clauses[M] - a list of M clauses
Result: (chromosome[N])
chromosome[N] - a feasible solution
--------------------------------------------------------------------------------
for i ← 1 to clauses.length do
    clauseSatisfiable& = false
    for variable ∈ clauses[i] do
        if variable > 0 then
            clauseSatisfiable| = chromosome[variable]
        end
        if variable < 0 then
            clauseSatisfiable| =!chromosome[variable]
        end
    end
    if clauseSatisfiable! = true then
        candidate ← candidatesOfClause[i]
        variable = abs(clause[candidate]) − 1
        if clause[candidate] < 0 then
            otherCandidate ← floor(random() · 3)
            while otherCandidate! = candidate do
                otherCandidate =← floor(random() · 3)
            end
            otherVariable = abs(clause[candidate]) − 1
            if clause[otherCandidate] < 0 then
                chromosome[variable] ← false
            end
            if clause[otherCandidate] > 0 then
                chromosome[variable] ← true
            end
        end
        if clause[candidate] > 0 then
            chromosome[variable] ← true
        end
    end
end
return (chromosome[N]) ;
```

3 Experimental part

3.1 Generated data

I have chosen one representative problem with 100 variables and 200 clauses. Negative variables were distributed equally as positive variables.

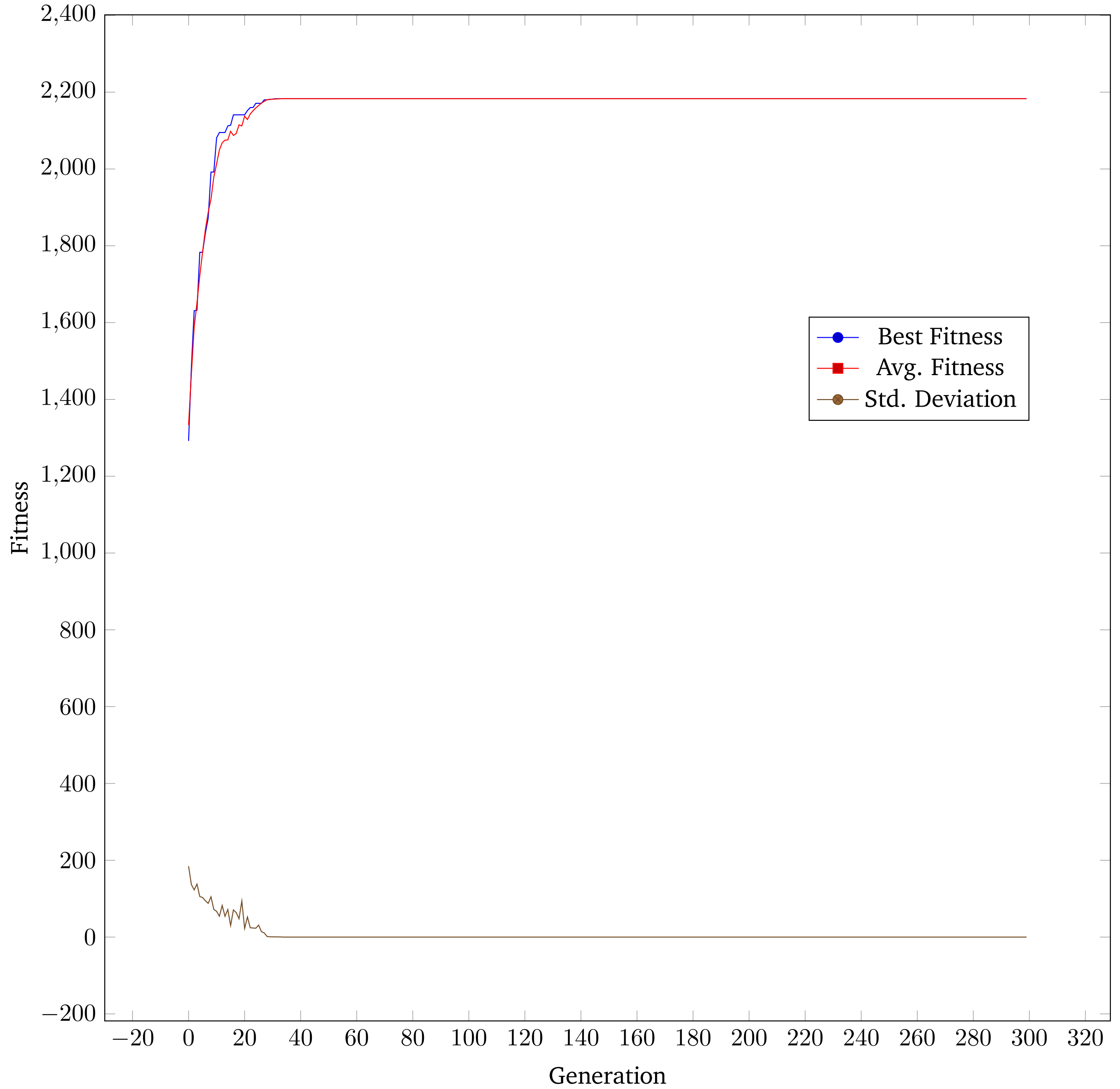
generator variables=100 clauses=200 skew=1 min=0 max=100

All the data were measured over 300 generations with 100 individuals in population.

3.1.1 Single-point Crossover & With Full Random Repair

It can be seen, that since the 30th generation, there is no deviation and since that, algorithm cannot find newer solution, because it stuck at local optimum. Maybe higher mutation rate would help to leave this local optimum. Even though GA has mutation, Full Random Repair probably repair mutated solutions. **Result: 2183, Average: 2219**

Graph 1 - Single-point Crossover & With Full Random Repair

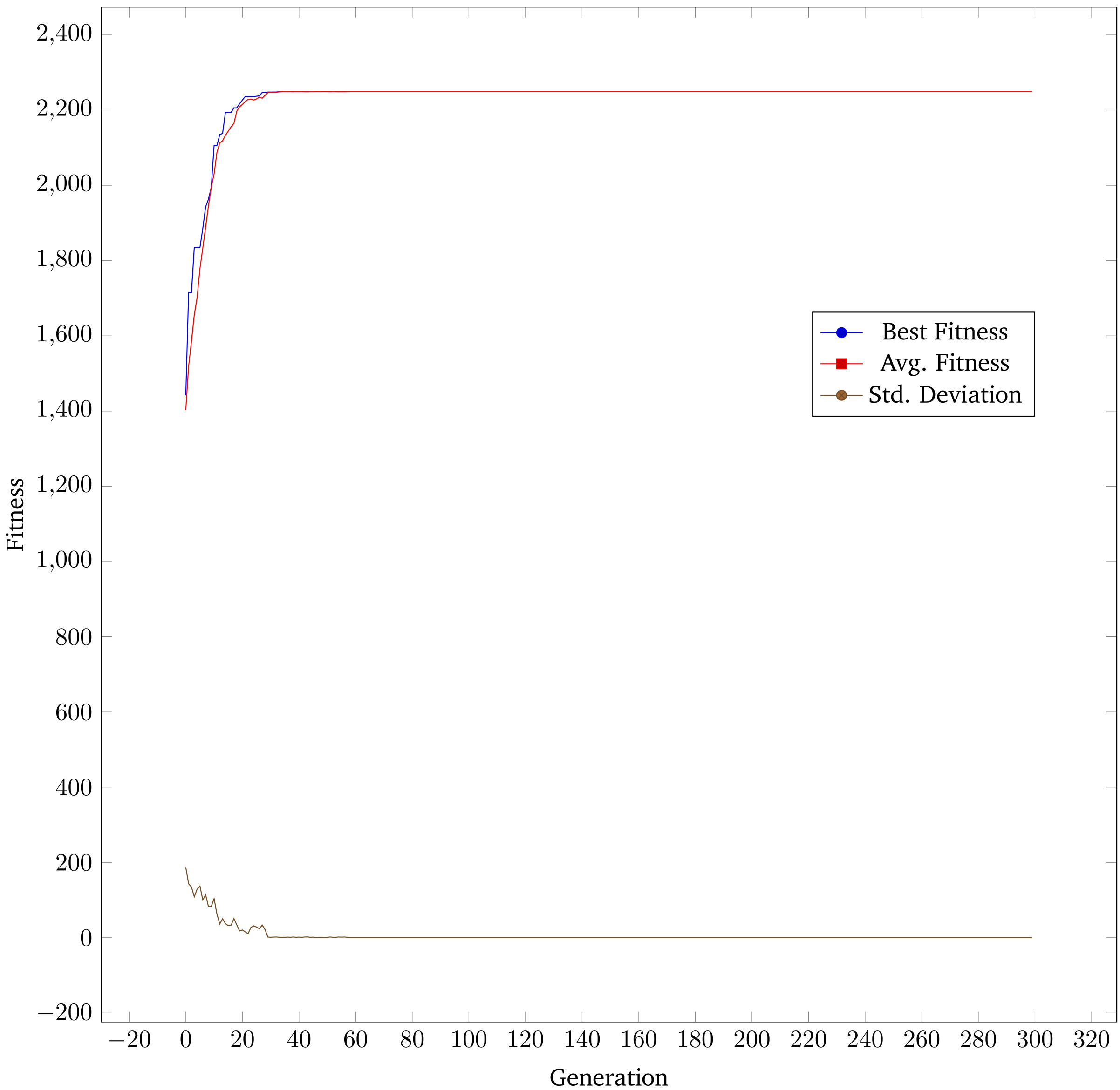


3.1.2 Adding Best-First method

Result: 2249, Average: 2227

Apparently, Best-First method is a good approach for our task, but algorithm stuck at local optimum as well.

Graph 2 - Adding Best-First method

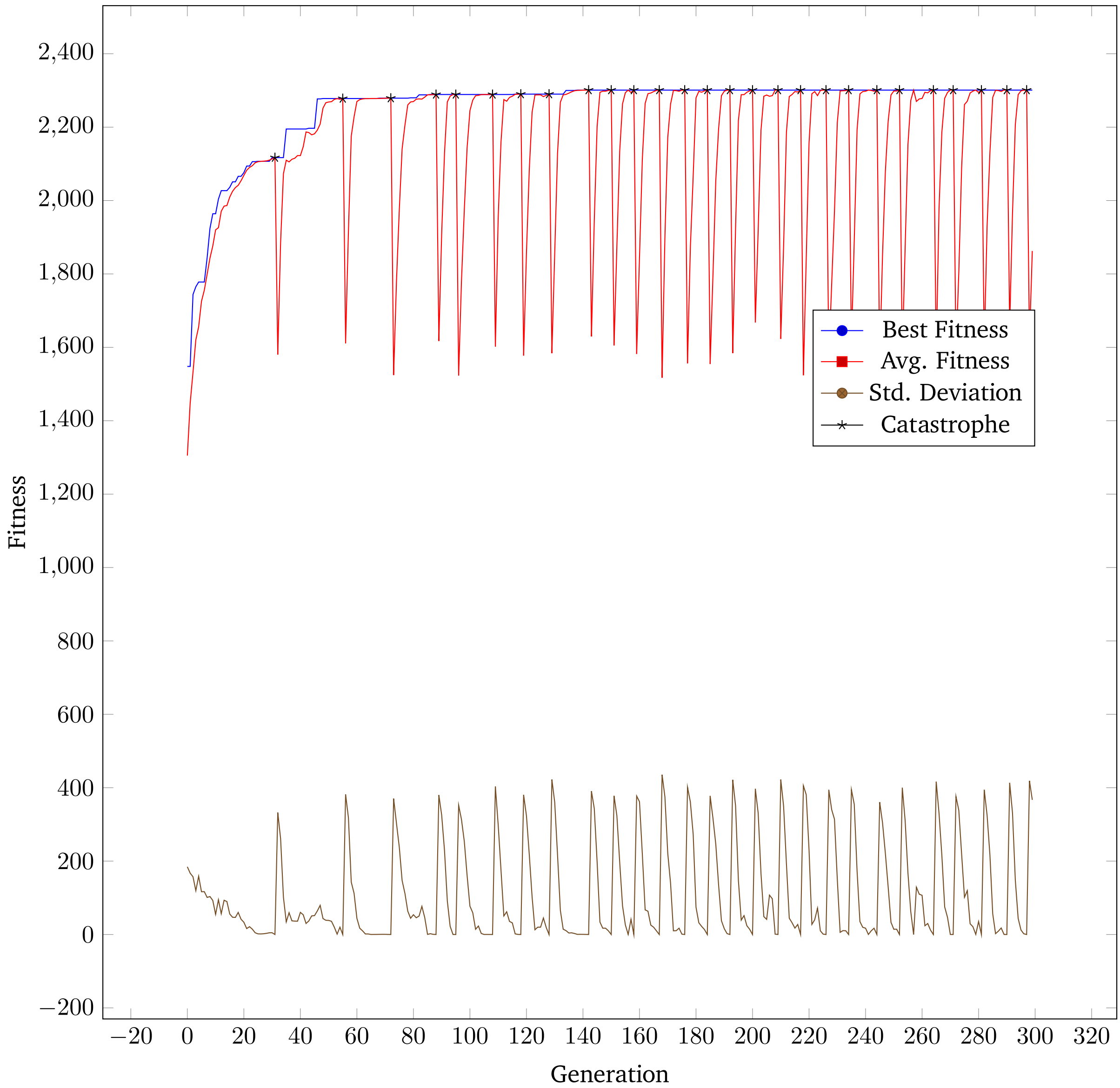


3.1.3 Adding Catastrophe method

Result: 2301, Average: 2282

Catastrophes adds new undiscoved solutions into population and because of that, GA is able to leave local optima. As It can be seen on graph, when std. deviation is near zero, catastrophe resets some of population and std. devitation is much higher.

Graph 3 - Adding Catastrophe method

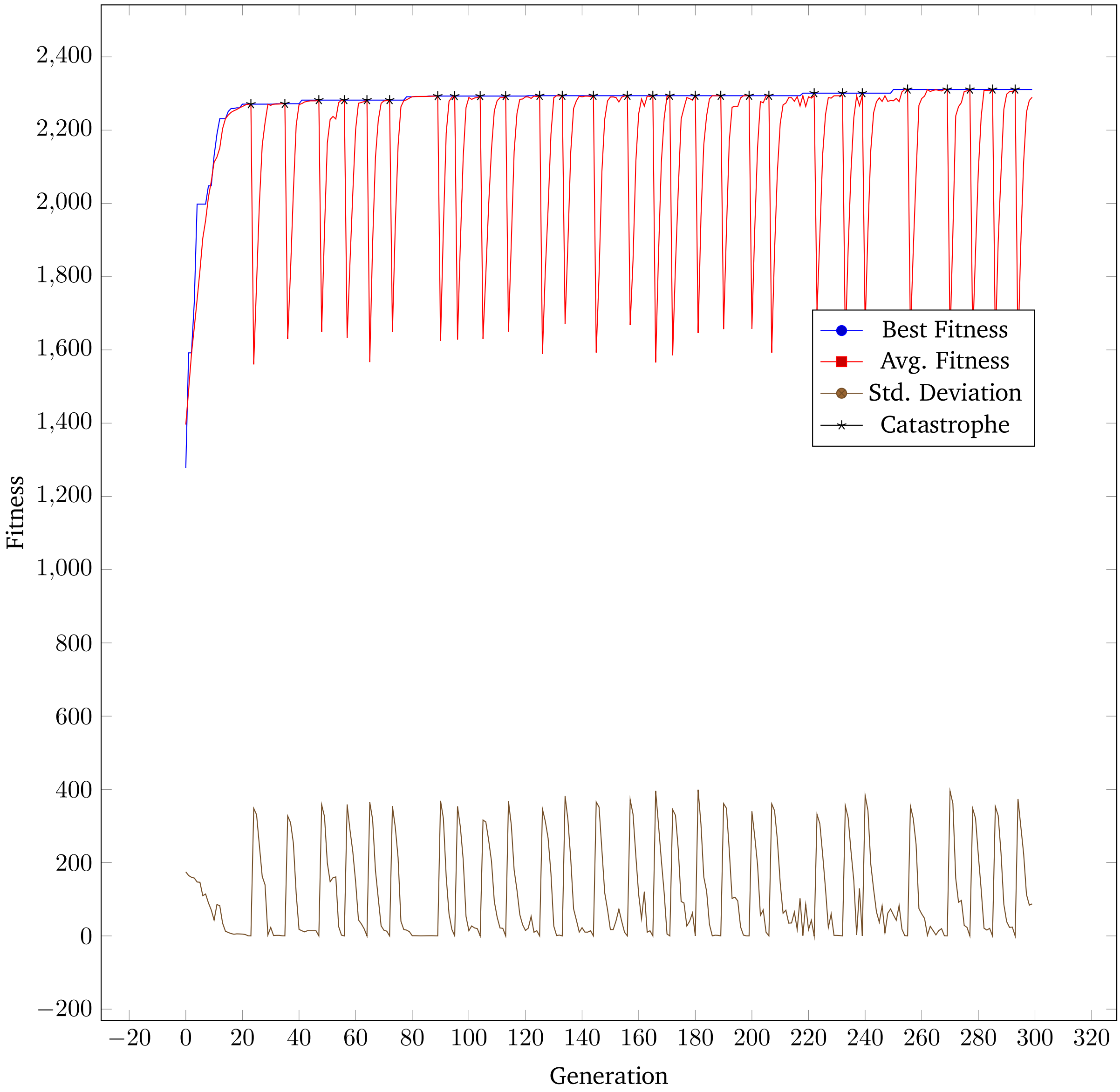


3.1.4 Adding Heuristic Repair method

Result: 2311, Average: 2287

What I am personally very pleased for is, that my self-developed heuristic repair actually worked. The heuristic approach helps to reach higher fitness much faster than without it.

Graph 4 - Adding Heuristic Repair method

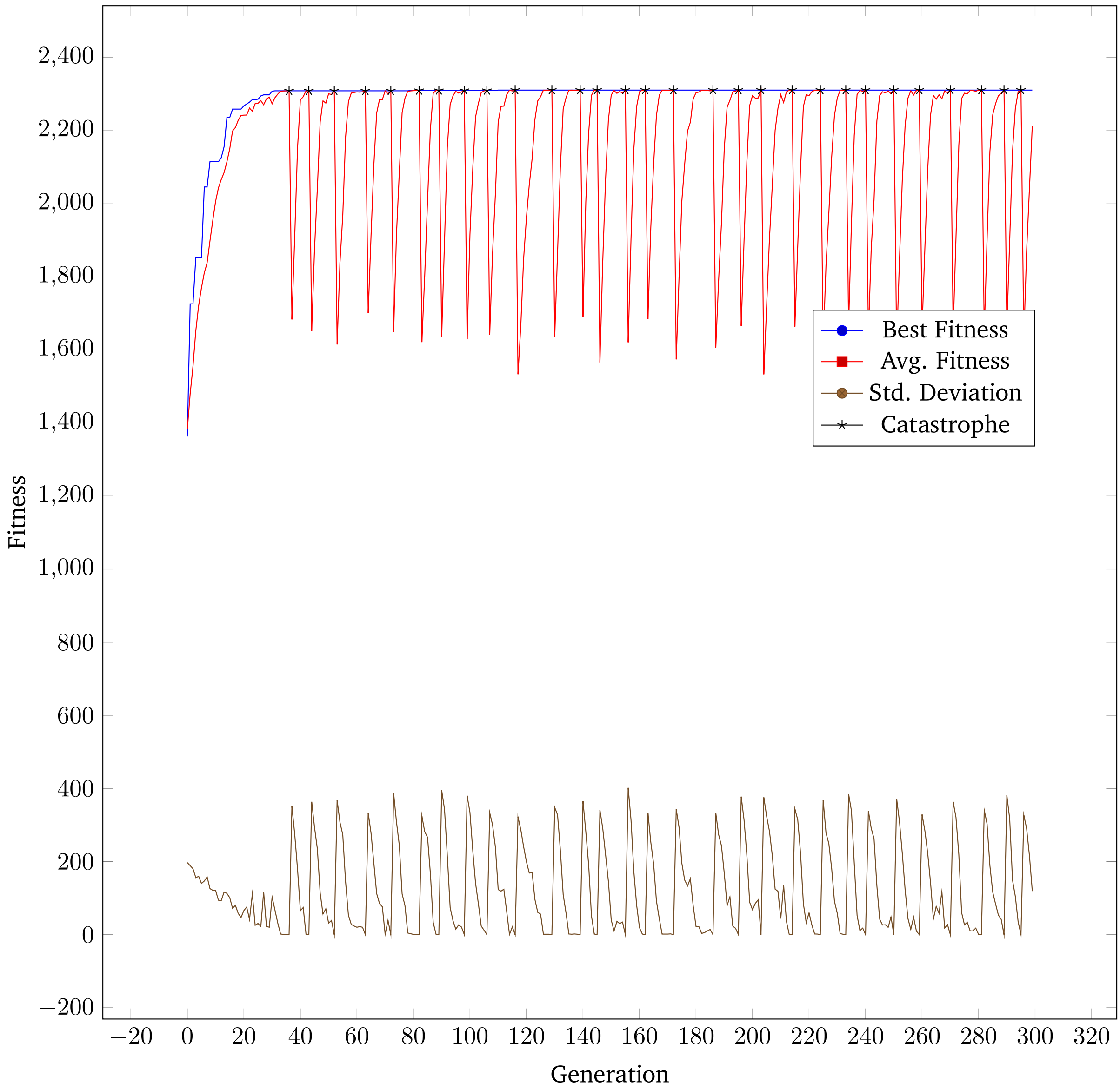


3.1.5 Using uniform crossover instead of single-point crossover

Result: 2311, Average: 2296

A uniform crossover is able to reach even higher average best fitness. I assume it is because of kind of 3 SAT, where an order of genes in chromosome is not important and uniform crossover can create much different solutions than single-point crossover.

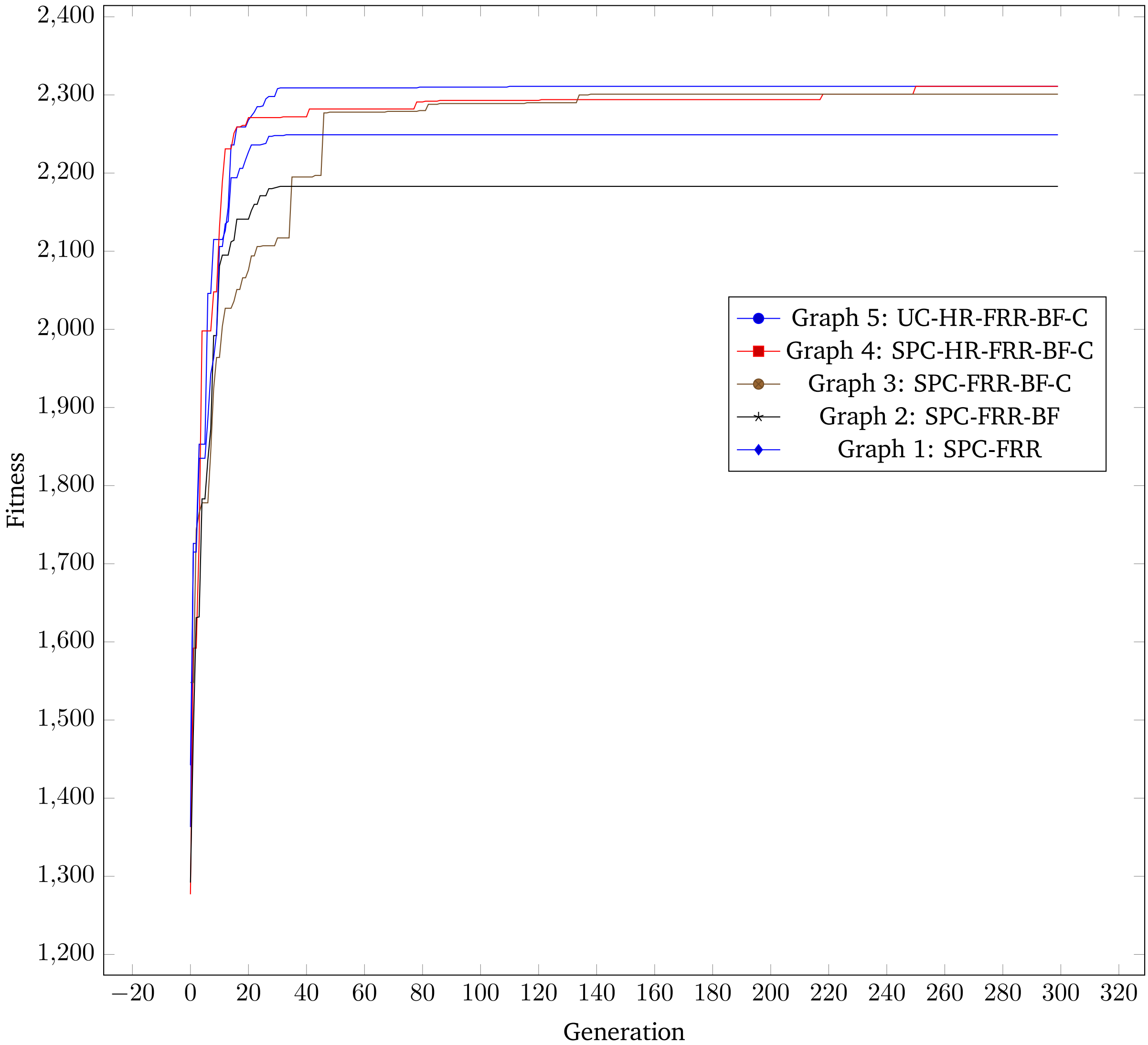
Graph 5 - Using uniform crossover instead of single-point crossover



3.1.6 Comparison of highest reached fitness over generations

It can be seen that my heuristic approch helped GA to find better solution in earlier generations. The uniform crossover also reached higher fitness in earlier generation than single-point crossover.

Graph 6 - Comparison of highest reached fitness over generations



Method	Best Fitness
Graph 1: SPC-FRR	2183
Graph 2: SPC-FRR-BF	2249
Graph 3: SPC-FRR-BF-C	2301
Graph 4: SPC-HR-FRR-BF-C	2311
Graph 5: UC-HR-FRR-BF-C	2311

3.1.7 Comparison of average best fitness for all used methods

SPC-FRR	SPC-FRR-BF	SPC-FRR-BF-C	SPC-HR-FRR-BF-C	UC-HR-FRR-BF-C
2167	2176	2214	2301	2292
2183	2249	2213	2299	2310
2196	2195	2310	2232	2222
2207	2195	2292	2292	2301
2277	2267	2232	2293	2301
2208	2295	2311	2309	2311
2147	2185	2301	2301	2301
2178	2276	2301	2302	2299
2198	2284	2301	2301	2301
2254	2217	2231	2291	2302
2274	2265	2222	2299	2293
2283	2185	2212	2292	2310
2186	2176	2232	2231	2293
2273	2168	2302	2222	2299
2173	2095	2299	2291	2311
2187	2254	2301	2299	2299
2209	2184	2302	2297	2302
2166	2266	2311	2288	2301
2277	2286	2301	2310	2311
2185	2197	2301	2309	2299
2166	2258	2310	2299	2299
2264	2290	2308	2306	2293
2286	2297	2301	2293	2230
2264	2287	2301	2275	2301
2178	2264	2301	2292	2301
2173	2237	2300	2308	2293
2295	2286	2301	2301	2301
2269	2173	2310	2293	2311
2265	2158	2231	2232	2308
2196	2165	2309	2260	2293
		Average		
2219	**2227**	**2282**	**2287**	**2296**

4 Conclusion

Apparently, the last method with uniform crossover gives the best results, which was experimentally tested and proved. I was very pleased that my own heuristic works very well and improved the finding of better solutions.