Homework 5

Introduction

The MAX-SAT problem is a well-known combinatorial optimization. Given a Boolean formula FFF in Conjunctive Normal Form (CNF) with weighted variables, the goal is to determine an assignment of variables that satisfies FFF while maximizing the total weight of satisfied variables. This problem, a generalization of the satisfiability problem (SAT), is NP-hard, making heuristic approaches essential for finding approximate solutions in a reasonable time.

This report explores the use of a **Genetic Algorithm (GA)**, an advanced iterative metaheuristic inspired by natural evolution, to solve the MAX-SAT problem. The GA operates on a population of candidate solutions (binary strings representing variable assignments), evolving them over generations using selection, crossover, and mutation. By iteratively refining solutions based on their fitness—the weighted sum of satisfied clauses—the algorithm aims to find near-optimal solutions efficiently.

Key elements of the GA implementation include:

- **Representation:** Solutions are encoded as binary strings, where each bit corresponds to the truth value of a variable.
- **Selection Strategy:** A tournament selection mechanism chooses parent solutions based on their fitness.
- **Variation Operators:** Solutions undergo one-point crossover and random bit-flip mutation to explore the search space.
- **Elitism:** The best solutions are preserved across generations to ensure monotonic improvement.

The evaluation process includes:

1. **White-Box Testing:** Parameter tuning using a subset of problem instances to identify optimal values for population size, mutation rate, crossover probability, tournament size, and other hyperparameters.
2. **Black-Box Testing:** Applying the optimized parameters to a broader set of instances, including varying problem sizes and difficulties, to assess the GA's robustness, runtime, and solution quality.

Evaluation

White box

This section documents the process of parameter tuning and early experimental results to optimize the GA for solving the MAX-SAT problem. The following parameters were tested:

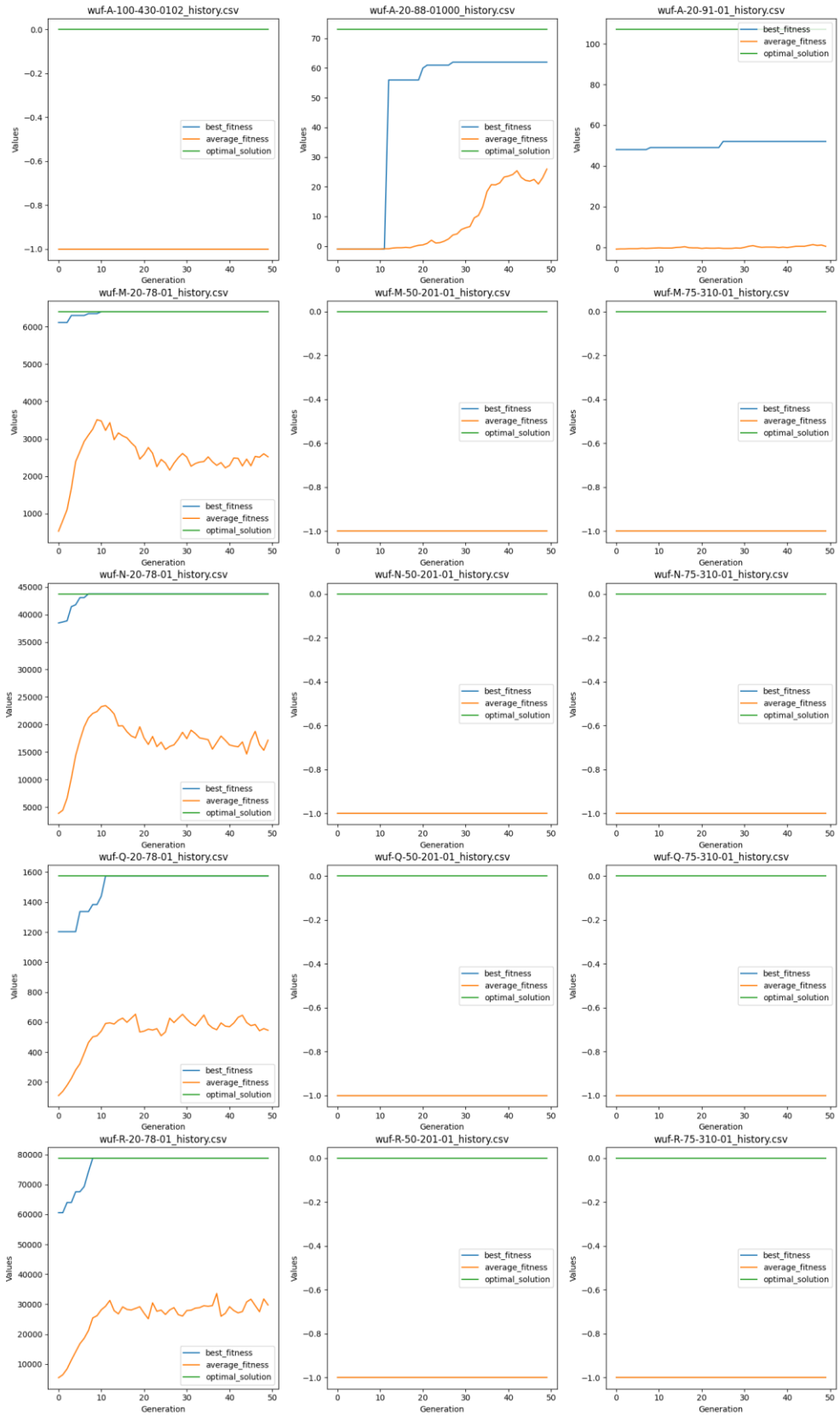
- **Population Size (ρ):** The number of candidate solutions in each generation.
- **Tournament Size (τ):** The number of candidates considered during selection.
- **Number of Elites (ε):** The number of top-performing solutions preserved each generation.
- **Number of Generations (γ):** The total number of evolutionary steps.
- **Mutation Rate (μ):** The probability of altering a bit in the chromosome.
- **Crossover Rate (χ):** The probability of recombining parent chromosomes.
- **Max Repair Attempts (α):** The maximum attempts to repair infeasible chromosomes.

Initial Parameter Settings

The following values were initially chosen:

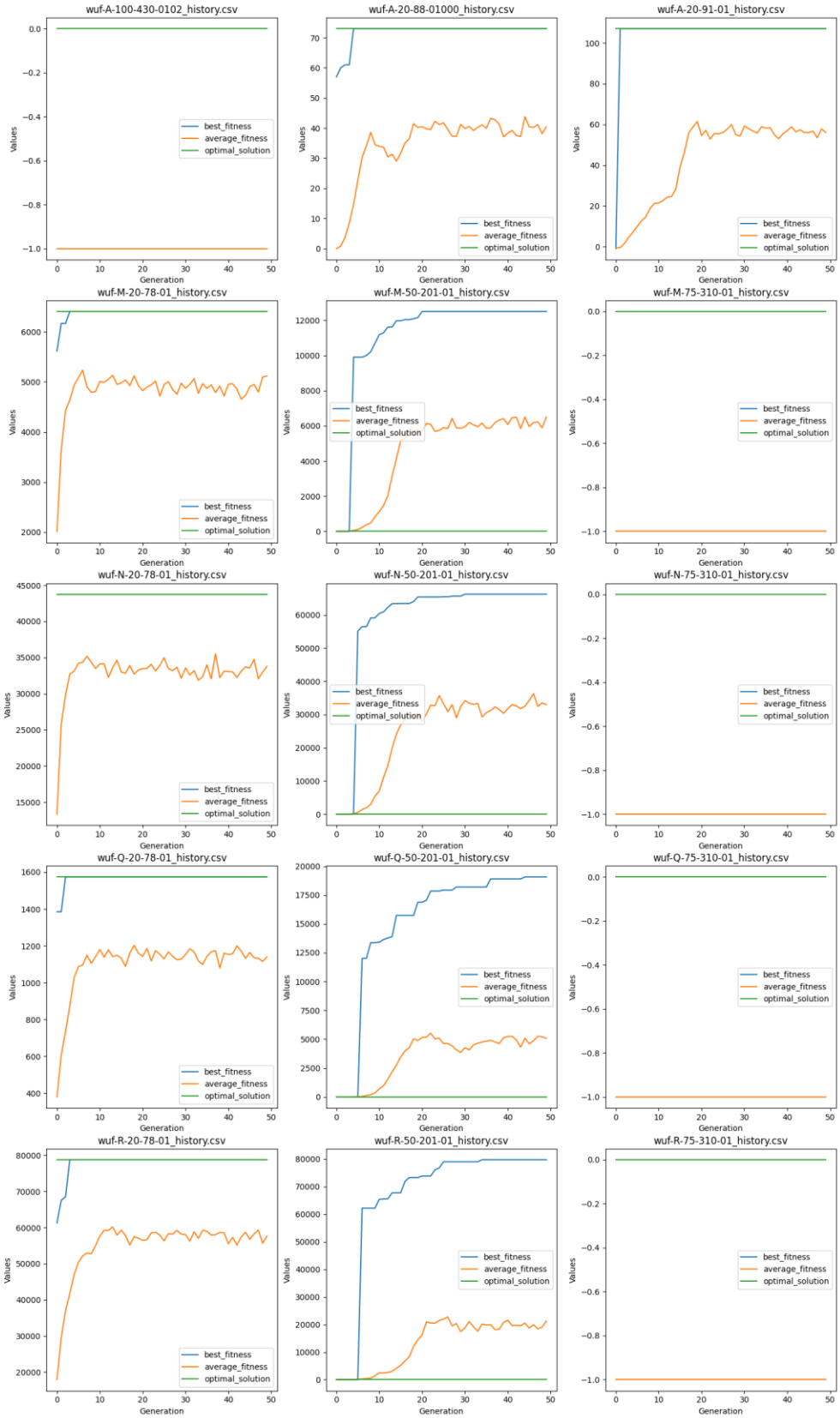
- Population size: 500
- Tournament size: 4
- Number of elites: 3
- Number of generations: 50
- Max repair attempts: 5
- Mutation rate: 0.05
- Crossover rate: 0.9

Initial experiments revealed issues with generating feasible solutions. Initially, the repair mechanism was applied only during population initialization.



The high proportion of infeasible chromosomes indicates that the initial repair mechanism, applied only during population initialization, was insufficient for maintaining solution feasibility throughout the evolutionary process.

When I extended the repair mechanism to include newly generated chromosomes, significant improvements were observed in the feasibility rate.



We can see that the runtime is greatly increased with every increase in maximum number of repair requests. This is expected behaviour as each repair attempt requires multiple iterations through the chromosome structure, attempting to modify variables while maintaining clause satisfiability. More repair attempts mean more computational overhead per chromosome.

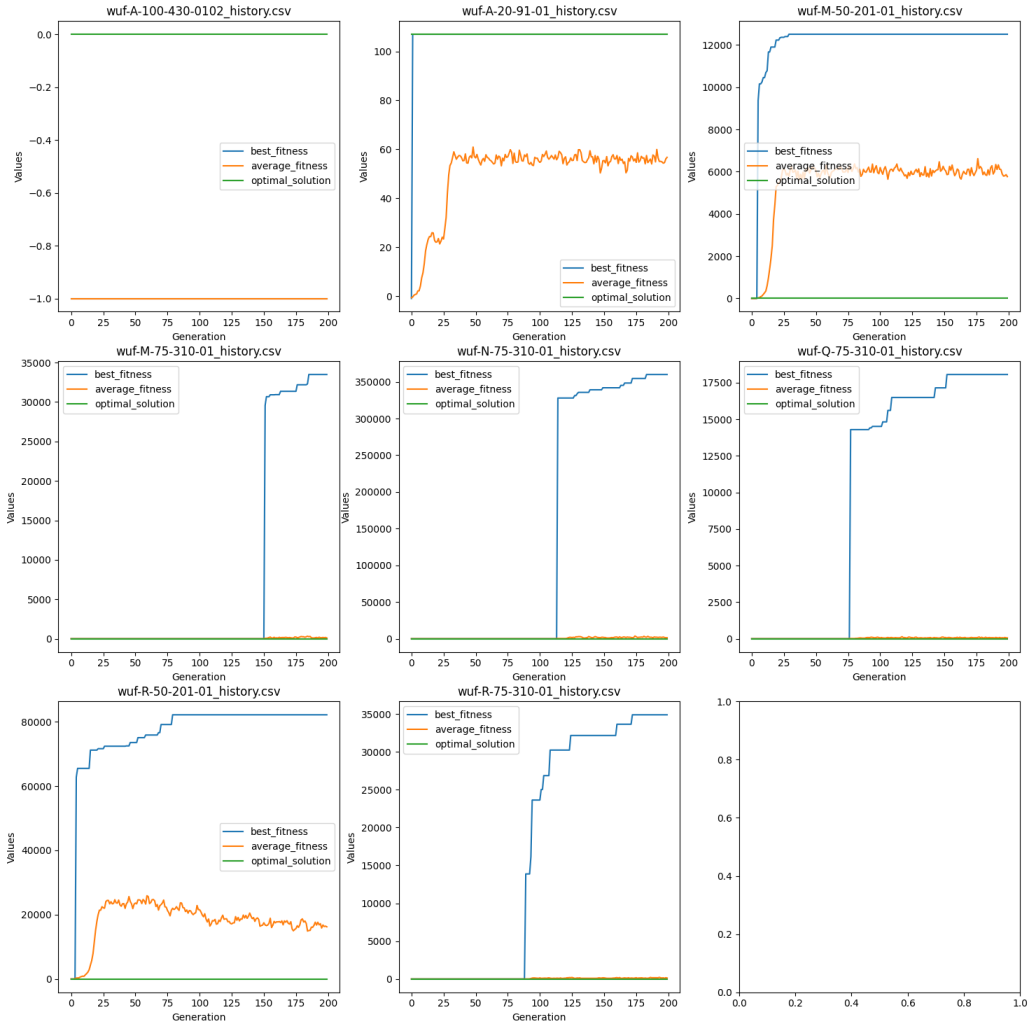
Looking at the convergence results (as shown in the graph), we observe that there's a significant improvement in solution quality when moving from 10 to 50 repair attempts, but the benefits diminish considerably for higher values (100 and 150). For that reason I decided to leave it at 50 to make a difference but still not to make runtime much bigger.

We can see that this modification helped a lot to problem instances to generate feasible solutions. This modification proved essential for handling problem instances effectively. However, further experiments were conducted to refine parameters.

I will now remove some examples that seem to have found some good results, and later at the end I will check on them again.

I will also increase generation number to 200 and population number to 1000 to see if I can find at least some feasible solutions for some harder examples. These adjustments aimed to improve solution quality for larger or more complex instances.

Results:



We can see that the best fitness was found for most of the problems and that population size mattered because it helped algorithm explore deeper space in search for optimal solution and by increasing generation number it gave more time to find those solutions.

I have now decided to keep only one that seem the most important to figure optimal parameters: these would seem to be:

wuf-N-75-310-01

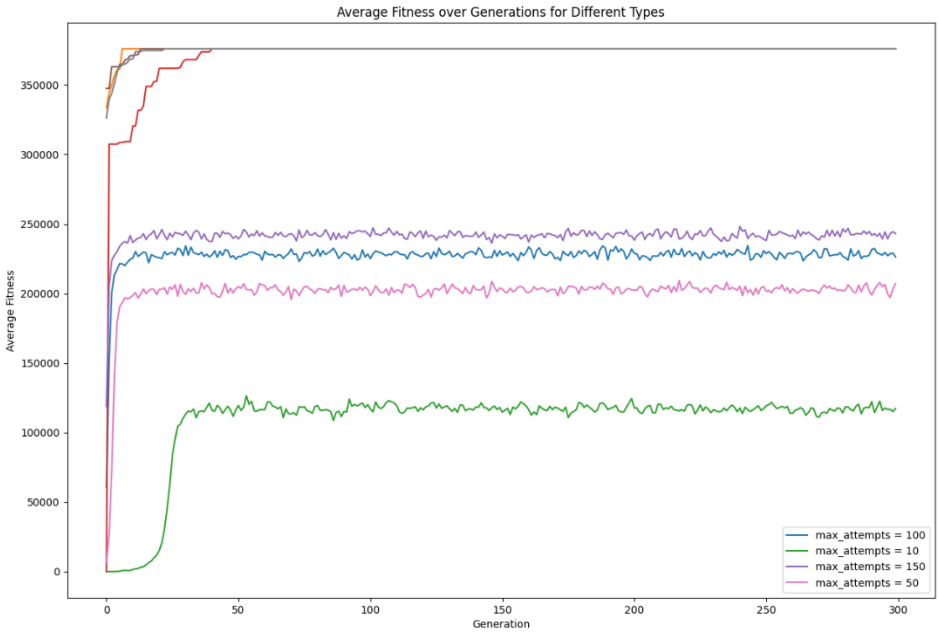
Also, I have decided to increase population and generation numbers to 5000 and 300 respectively just to try and find some solutions and at the same time try with parameter valuations.

First, I decided to test max number of attempts to repair infeasible chromosome with values: 10, 50, 100, 150

type	variable_nu	clauses_nu	measured_	average_e	max_attempts	
N	75	310	32106	0	10	
N	75	310	92639	0	50	
N	75	310	147192	0	100	
N	75	310	192232	0	150	

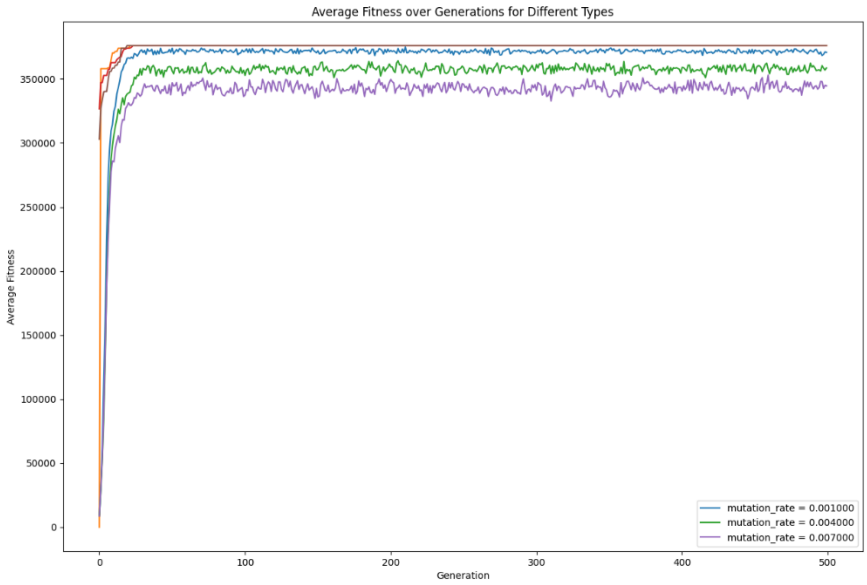
We can see that the runtime is greatly increased with every increase in maximum number of repair requests.

Let's look if we are any closer to convergence for our example.



We can see that there is a big difference between 10 and 50 while the difference decreases with increase in number of attempts. That is why I have decided to leave the number 50 for now and try to test some other parameter.

Now I remembered when playing with experimental web application that one of the values that greatly affected the algorithm was mutation rate so let us try testing those values. I tried values: 0.001, 0.004, 0.007



The experimental results demonstrate that mutation rate significantly impacts solution quality. This effect occurs because even after finding an optimal solution, the algorithm continues applying mutations to the entire population, potentially disrupting good solutions. This observation suggests the need for an adaptive mutation mechanism.

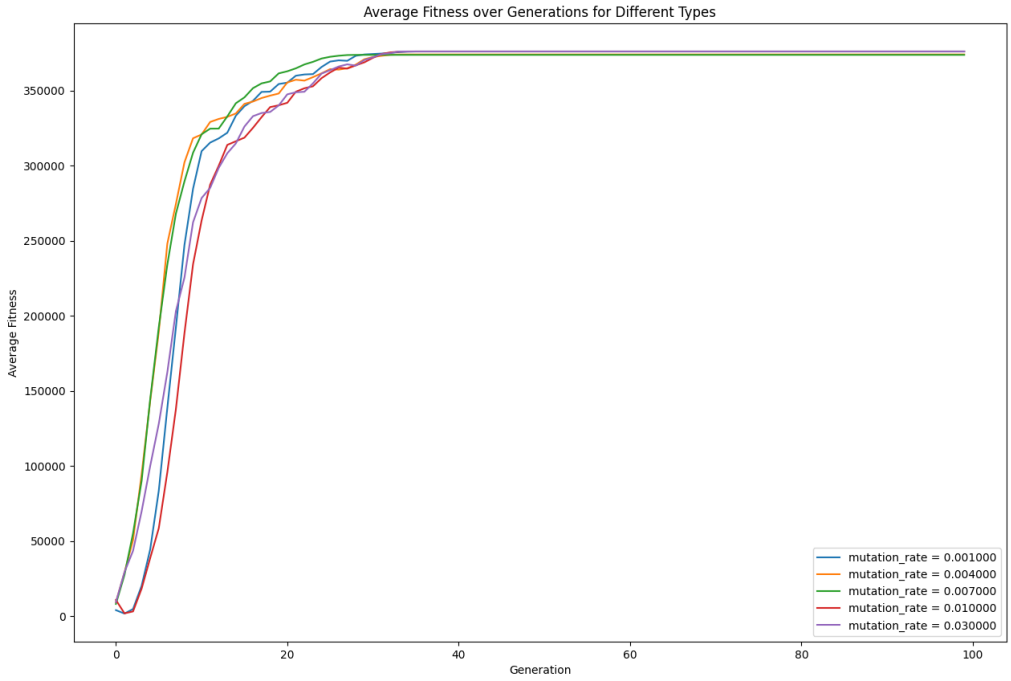
My idea is to introduce new formula to calculate mutation ratio on every iteration. I want it to decrease the closer average fitness is to the maximum fitness. In theory, that way would introduce bigger mutation ratio on the beginning but as the algorithm is about to converge to the optimal, our mutation ratio would decrease to 0. The formula would look something like:

$$mutation = initial * (1 - \frac{average}{maximum})$$

At the beginning of the search mutation rate would be bigger as the maximum is greater than the average, but as the algorithm converges so would mutation rate decrease to 0 and getting our chromosomes closer to the optimal

Let's see the results:

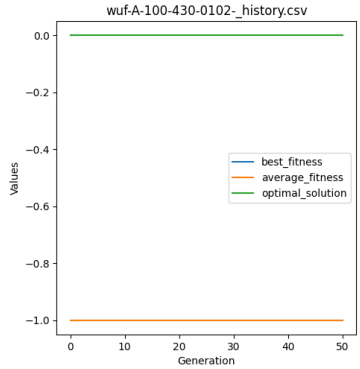
```
mutation_rate = 0.001000: 375961.0
mutation_rate = 0.004000: 373967.0
mutation_rate = 0.007000: 373686.0
mutation_rate = 0.010000: 375961.0
mutation_rate = 0.030000: 375961.0
```

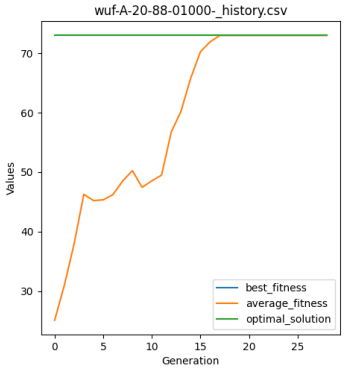


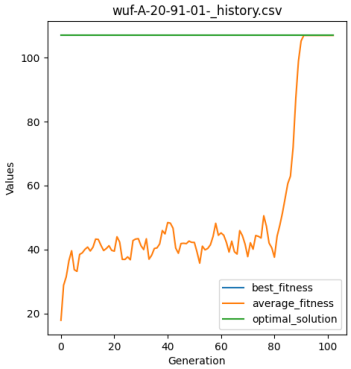
When printing last average value, we can see that for some mutation rates, the algorithm does not quite converge, that is why I choose to take 0.03 rate which gives me quite big ratio at the beginning and decreasing as it converges too optimal.

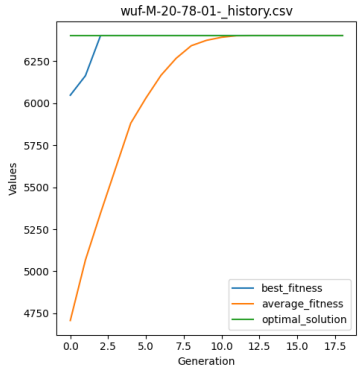
Also, to speed up calculations for later black box testing I introduced stopping criterion which is based on clauses number so that easier problems stop after few iterations without change and the harder problems have more iterations to find feasible states. To check how much has the algorithm changed I have a parameter “change_threshold” which check did the new average population improve from the previous ones by that percentage.

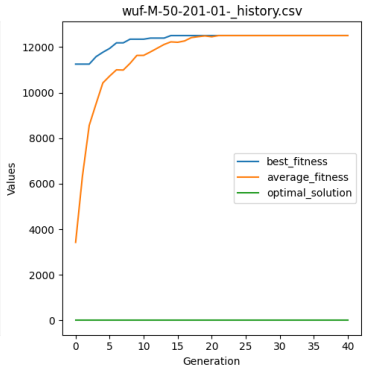
Let’s now check our all examples and see how we are progressing:

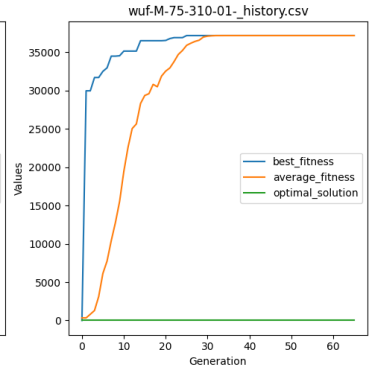

wuf-A-100-430-0102-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

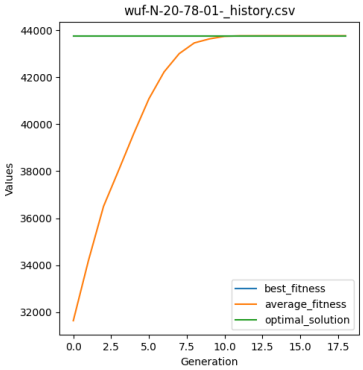

wuf-A-20-88-01000-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

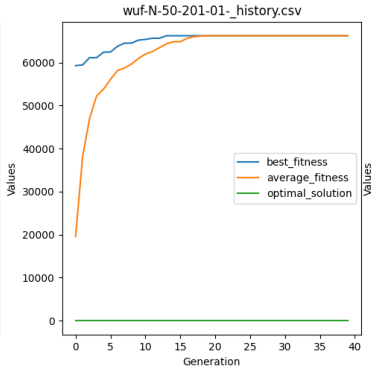

wuf-A-20-91-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

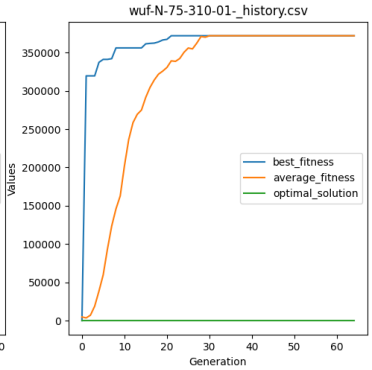

wuf-M-20-78-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

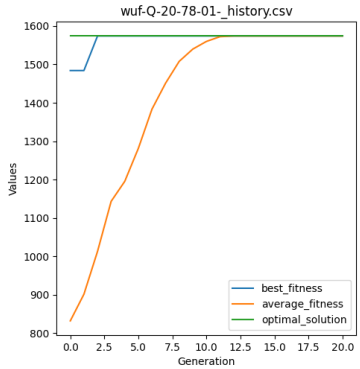

wuf-M-50-201-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

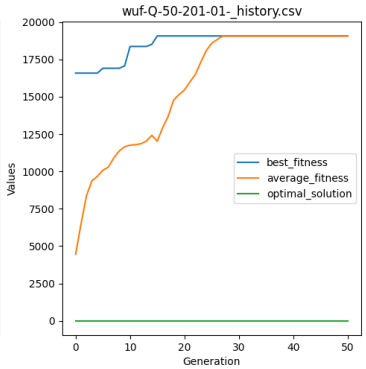

wuf-M-75-310-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

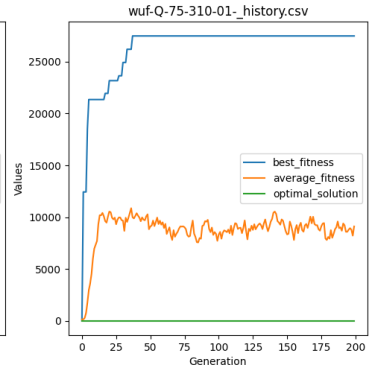

wuf-N-20-78-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

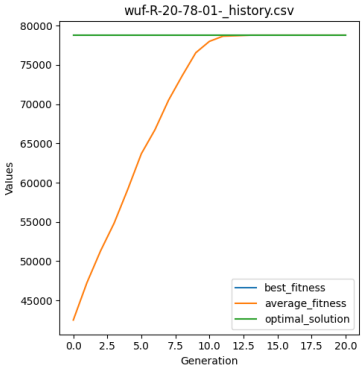

wuf-N-50-201-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

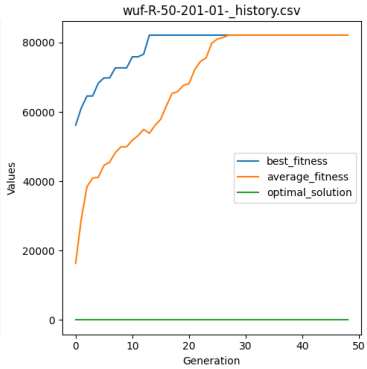

wuf-N-75-310-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

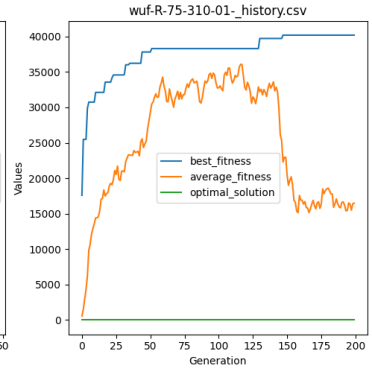

wuf-Q-20-78-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution


wuf-Q-50-201-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution


wuf-Q-75-310-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution


wuf-R-20-78-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

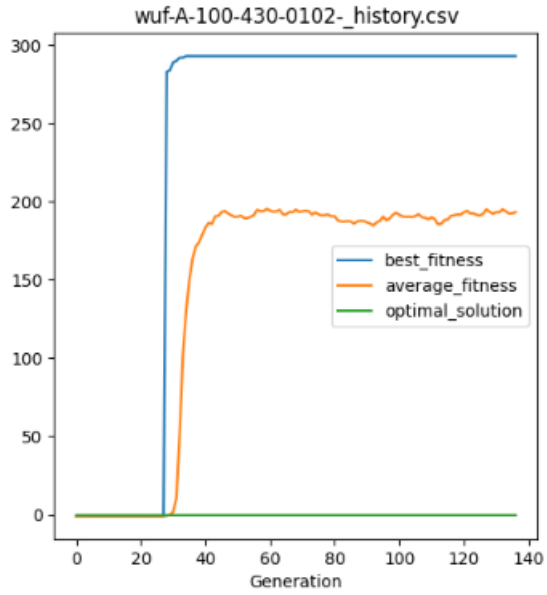

wuf-R-50-201-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution


wuf-R-75-310-01-_history.csv
Values
Generation
best_fitness
average_fitness
optimal_solution

We can see that we made great progress on most of the algorithms, they are converging successfully with small number of generations needed. Still few specific harder examples are still not converging so we will focus on them. First, I will focus on hardest one ("wuf-A-100-430-0102") and hopefully the others will also converge with it.

For the wuf-A instance I experimented with many different parameters and the one that seem to help the most is population size. To find suitable solution I needed to increase population size to a value of 5000. Because of that I decided to implement an adaptive population size as well.

After testing my algorithm, I also figured out that even though it finds good solution, after some time the newer populations do not converge their averages to optima.



That is why I decided to modify my method for adapting my algorithm after every few iterations. Now it works in a way that at the beginning population size is smaller (500) and tournament size and number of elites is only 2. I leave few numbers of generations for my algorithm to find best fitness before checking the ration of best and average fitness. For the hardest (A-instance) examples the algorithm needs some time to find feasible solutions so if the population did not find enough feasible solutions, I increase the population size by big number and max number of attempts to enable the algorithm to find feasible solutions.

If it already got out of infeasible chromosomes, I increase it by little at beginning and little more every time that it is getting closer to the optimal solution by increasing tournament and elites' numbers. This way my algorithm is running a bit slower but at least I am pretty sure that for all examples, my algorithm will have enough time to find the best solution and converge towards it successfully. Let's first see how this new modification affects all the examples except the hardest one and then also check if it is good for the hardest one.

```
Number of unoptimal instances: 0
Number of unoptimal instances: 1
Number of unoptimal instances: 0
Number of unoptimal instances: 0
Number of unoptimal instances: 1
Number of unoptimal instances: 0
Number of unoptimal instances: 1
Number of unoptimal instances: 0
Number of unoptimal instances: 0
Number of unoptimal instances: 2
Number of unoptimal instances: 0
Number of unoptimal instances: 1
Number of unoptimal instances: 1
Number of unoptimal instances: 0
```

This represents the number suboptimal average fitnesses which were tested on some smaller part of every type of instance except A-100-430. We can see that it successfully converges except few instances which could be because it does not converge fast enough and gets terminated before. We will check in the evaluation when we run all examples, why is that.

To help with the convergence I modified my adaptive mutation ratio equation:

$$mutation = initial * (1 - \frac{average}{maximum}) * (1 - \frac{\text{current generation}}{\text{\# of generations}})$$

I did this because as the algorithm is getting closer to the end of the generations, I want it to reduce the mutation rate to help it converge even faster with the assumption that it has at least found one optimal solution.

BLACK BOX TESTING

Let's once again run our algorithm with default parameters:

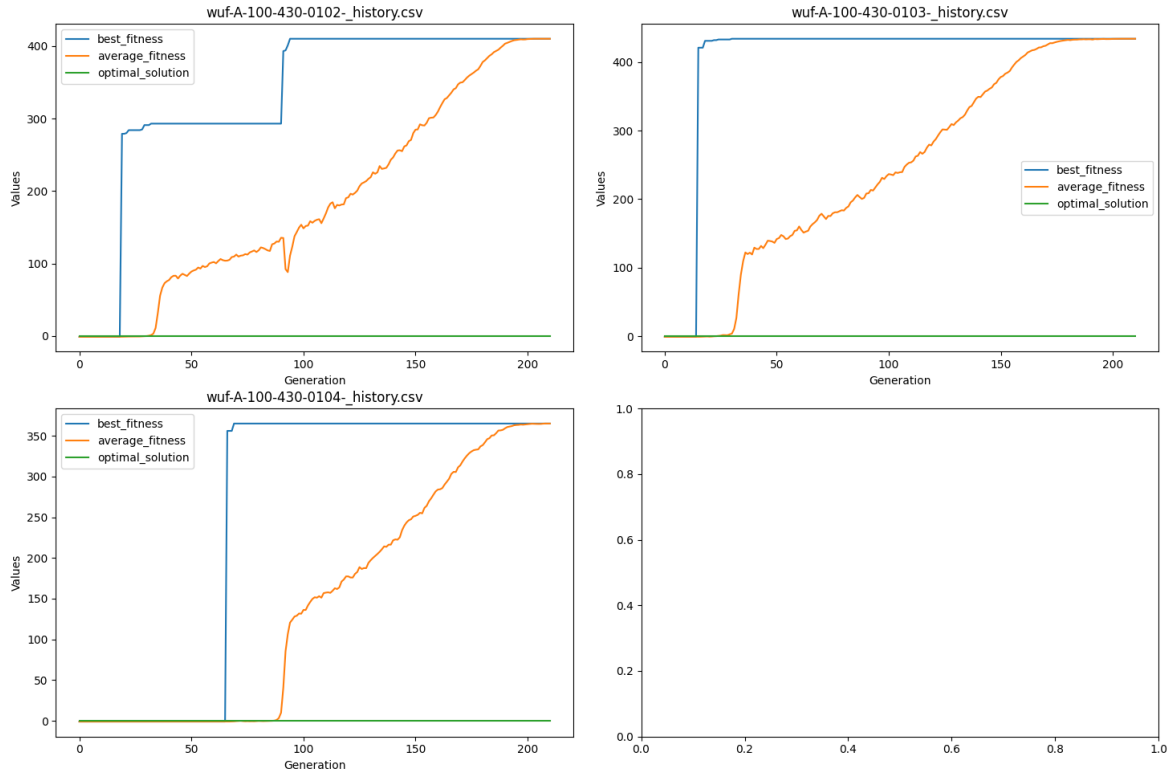
```
population_size = 500;
tournament_size = 2;
num_elites = 2;
num_generations = 300;
max_attempts = 50;
mutation_rate = 0.05;
crossover_rate = 0.9;
change_threshold = 0.05;
```

type	variable	clauses	average_error	best_fitness_error	measured_time	average_runtime
N	20	78	1.75E-05	1.65E-05	27925	27.925
N	50	201	0	0	159644	159.644
N	75	310	0	0	64579	645.79
M	20	78	1.70E-06	0	28531	28.531
M	50	201	0	0	160808	160.808
M	75	310	0	0	63361	633.61
A	20	88	0.000229737	0	15543	15.543
A	20	91	3.43E-05	0	53872	53.872
Q	20	78	7.83E-05	7.83E-05	42397	42.397
Q	50	201	0	0	334533	334.533
Q	75	310	0	0	272483	2724.83
R	20	78	3.51E-05	3.51E-05	33599	33.599
R	50	201	0	0	339801	339.801
R	75	310	0	0	263355	2633.55

We can see that almost always our algorithm successfully finds optimal solutions and most of the time successfully converges the whole population towards optimal. The error is almost always zero if not, close to the minimal, no matter the instance size.

We can see that for the bigger instances the average runtime is getting higher, but it is still negligible. The smallest average runtime is 27 milliseconds for easiest problems and the highest one is for 2.5 seconds for “R” instances with 75 variables and 310 clauses.

The only problem is “nasty” instances when the weight is made to fool the search and therefore take much longer to find the optimal solutions. But we will show now that eventually it is found, and the algorithm starts to converge towards the optimal.



We can see that sometimes (1st example) it takes algorithm long time to discover the new optimal, about 100 generations.

All in all, we can see that with few exceptions the algorithm converges correctly to optimal and is almost always available to find at least the best optima solution without error but few times it ends a bit too early due to the stopping number criteria. If I had set this number a bit higher than it would increase my runtime by a lot because all the other thousands of examples that would end early will take more time to stop.