3 SAT problem

3 SAT is a subtype of SAT problem. SAT is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In 3 SAT there is a restriction: each clause has to contain exactly 3 variables. Both SAT and 3 SAT are NP-complete problems. In this project I will develop a genetic algorithm for solving 3 SAT problem.

Instance Generator

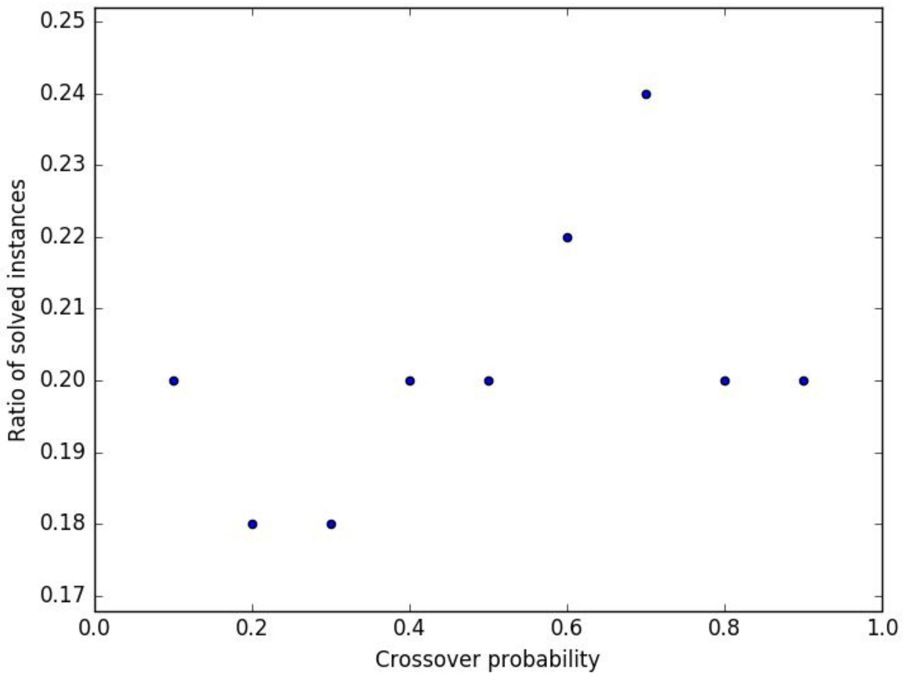
As the first step I implemented an instance generator for 3 SAT problem. This generator takes 3 parameters: **clauses/variables ratio**, **number of clauses** and **number of problem instances** user wants to generate. All of instances I used in this project have clauses/variables ratio 4.5, as this is close to ratio of the hardest instances and if my algorithm can cope with these complex instances, I can assume that it can solve easier instances with bigger or smaller clauses/variables ratio. I have mostly used number of clauses equal to 50, but in some experiments I tried different instances sizes. The generator calculates number of variables itself from number of clauses and clauses/variables ratio. As for the last parameter, I usually generated 30 or 50 instances for testing and measuring purposes.
The generator works in such a way, that for each clause it randomly chooses variables that will be in this clause, and for each variable it chooses if it will be negated with probability 50%. The generator does not verify satisfiability of these instances - that would be a SAT problem itself. That is why it is necessary to take into account that some of generated instances are not satisfiable by themselves.
The output of generator is saved to files, they can be found in project in directory “instances”. The format of output was borrowed from http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Basic Genetic Algorithm

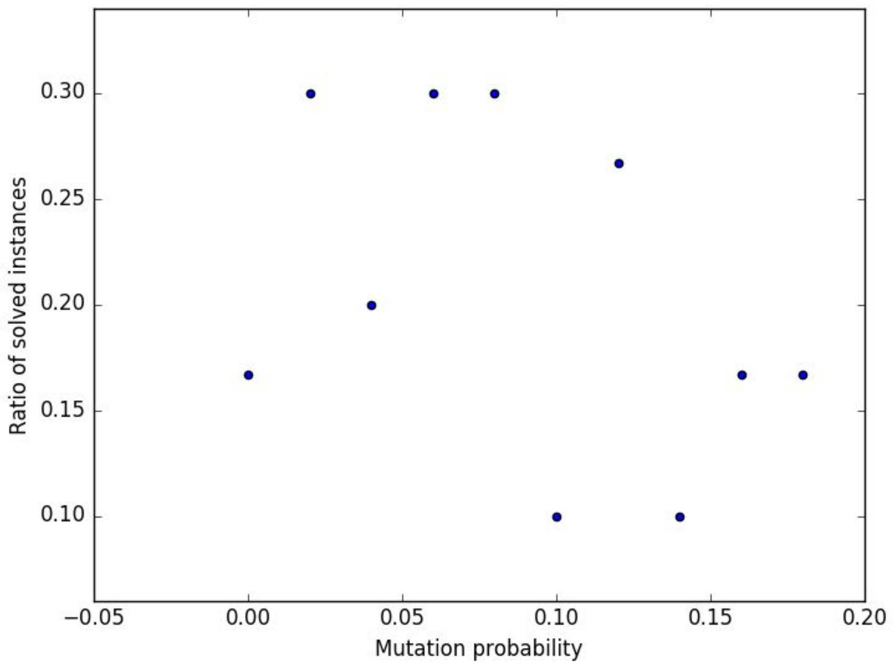
After implementing instance generator I implemented a basic version of genetic algorithm for solving 3 SAT problem. Individuals are implemented as vectors of 1’s and 0’s of size corresponding to number of variables. Each value in vector corresponds to value of certain variable: 0 for false and 1 for true. Fitness of individual is calculated as sum of weights of variables set to ‘1’ plus bonus for each satisfied clause plus bonus if the whole formula is satisfied. Selection is performed by roulette algorithm. One point crossover is used. Mutation is performed in such a way that with set probability a random bit is flipped.

Setting Algorithm Parameters

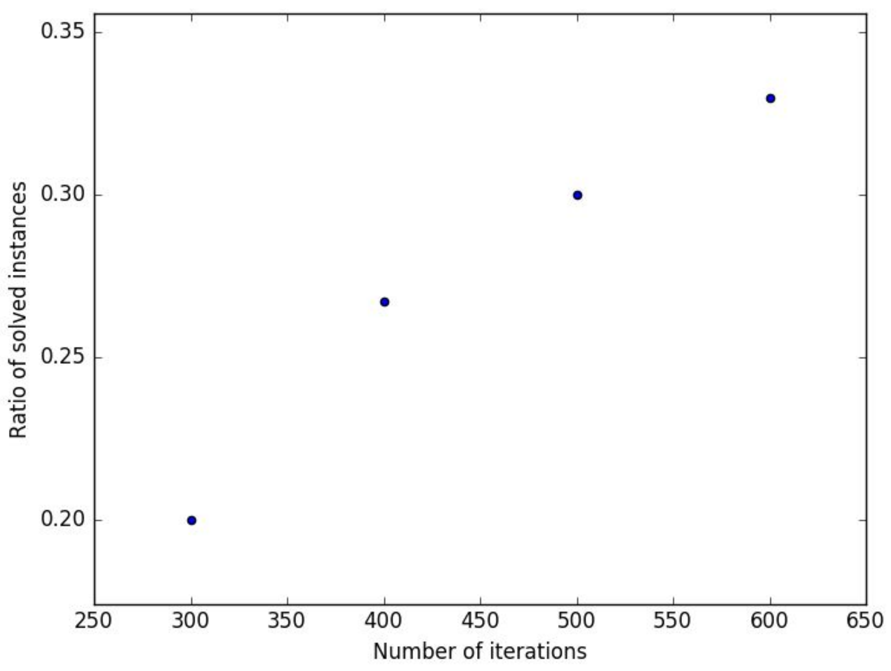
Algorithm has 4 inner parameters: size of population, number of iterations, crossover probability and mutation probability. Initial values of these parameters were 400, 50, 0.5 and 0.02 accordingly. In order to set these parameters to optimal values I needed some evaluation of performance of algorithm. I decided to use ratio of instances, for which algorithm found some solution, to all instances. So if I run algorithm on 50 instances and it finds solution for 20 of them, resulting performance is 20/50=0.4. It is important to remember that not all of instances, on which I tested algorithm, are solvable - some of them have no solution, as formulas coded in them are unsatisfiable. So if all the used instances were solvable, this ratio would be higher.
So I measured this performance for different values of each parameter, while leaving other parameters fixed. The graphs below show the results.



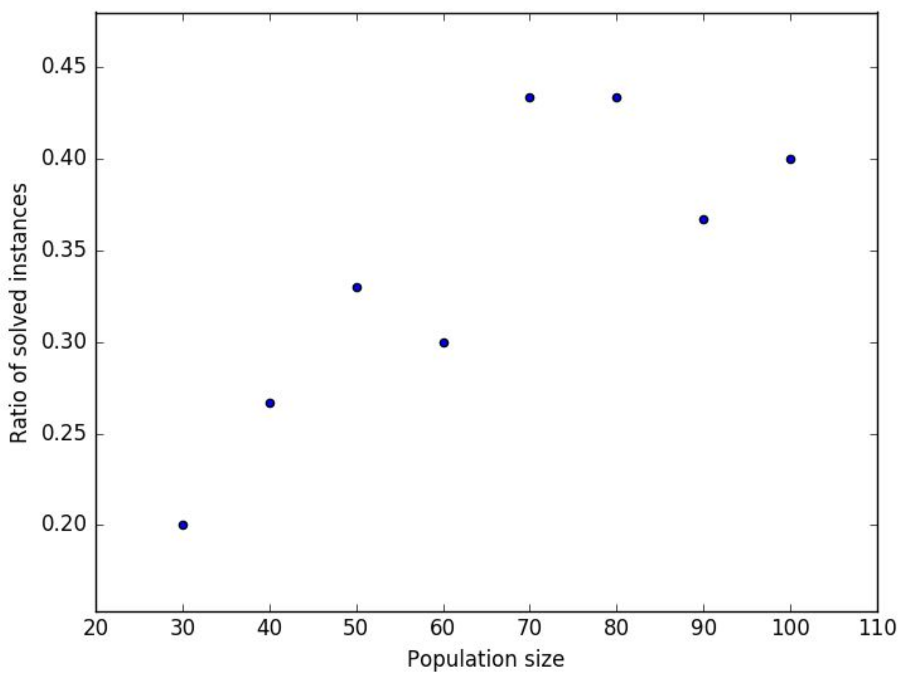
On the graph above it is clearly visible that the most optimal value for crossover probability is 0.7.



The graph looks a bit scattered, the reason for this is the fact that genetic algorithm is randomized. The best way to go around this is to repeat computation for as many times as possible and calculate average. However in this case the computational power and time available to me are limited and genetic algorithm is very computationally demanding. From the graph above I have chosen value 0.06.



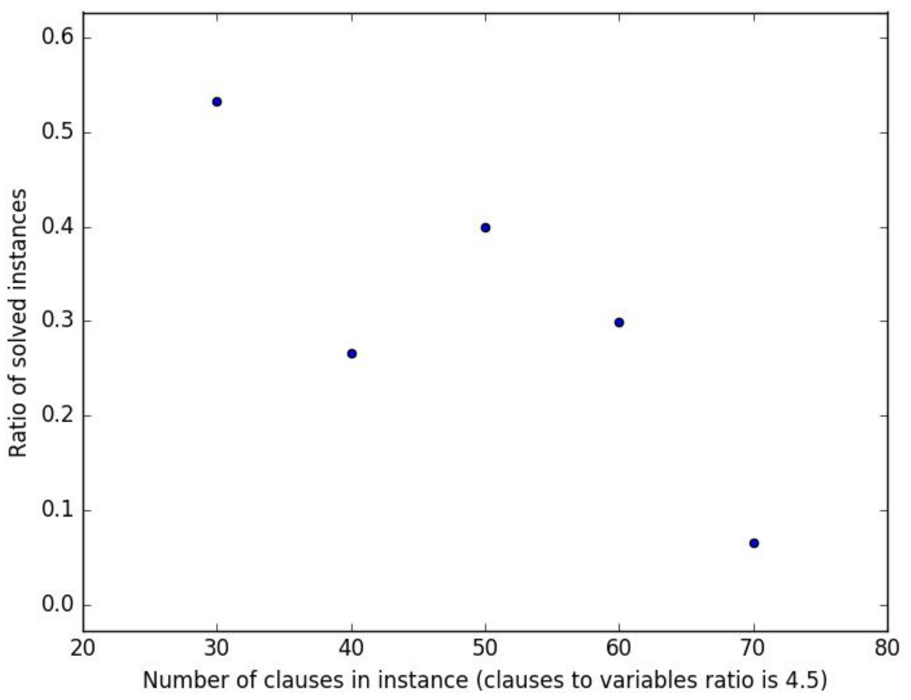
This graph clearly shows that the greater is number of iterations, the greater is ratio of solved instances. The question is to settle on optimal number according to computational time available. I have chosen 500 iterations.



A suboptimal value of 50 individuals was chosen.

In all further measurements this chosen configuration of parameters will be used.

Measuring Performance on Different Sizes of Instances



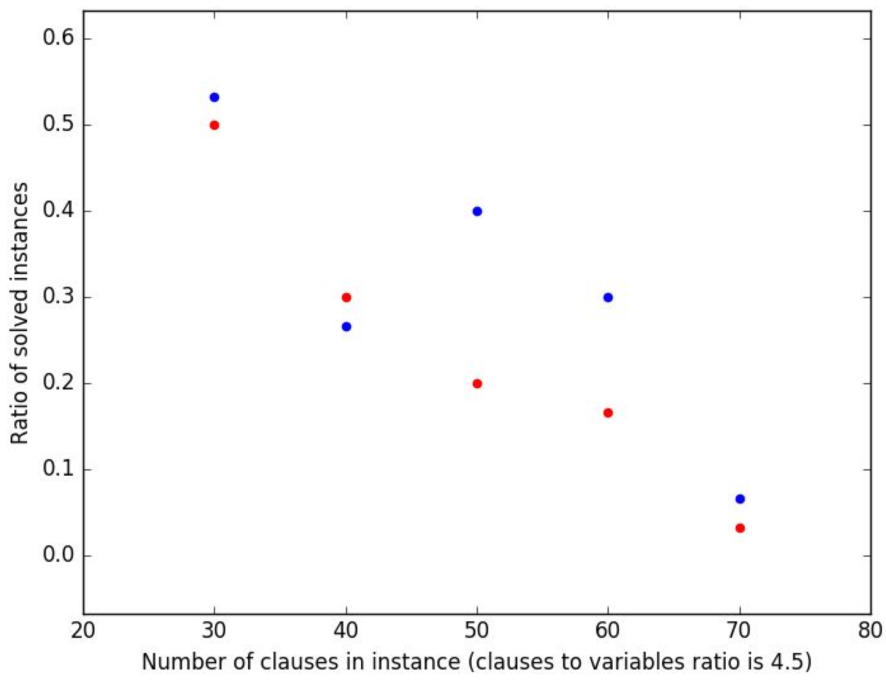
Ratio of solved instances declines with growth of sizes of instances.

Trying More Sophisticated Techniques

In this chapter I will try a few different techniques for genetic algorithm and see if they improve algorithm's performance.

Uniform Crossover

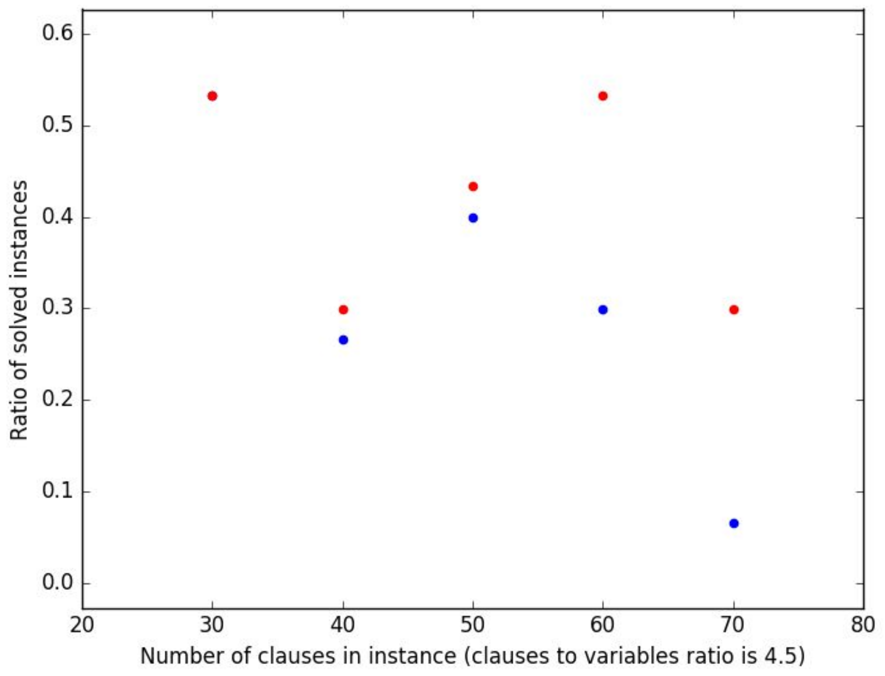
The uniform crossover evaluates each bit in the parent strings for exchange with a probability of 50%. I have measured algorithm's performance with uniform crossover for different sizes of instances and compared it to already collected data for algorithm with classical one point crossover. In all graphs below blue color shows performance of **basic** algorithm and red color shows performance of **changed** algorithm with new technique.



As you can see, original algorithm performs better, so one point crossover is more efficient than uniform crossover in this case and there is no need for change.

Linear Scaling

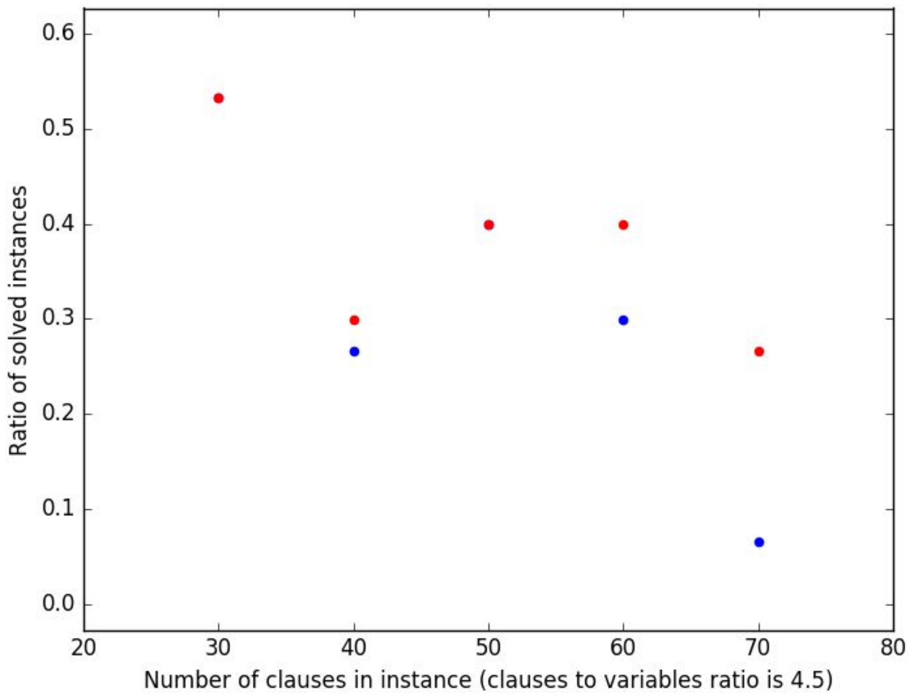
Linear scaling is a procedure that scales values of fitness function to certain range. This is performed for selection pressure to remain the same during the whole run of algorithm.



It is evident that linear scaling improves performance of algorithm for this problem.

Tournament Selection

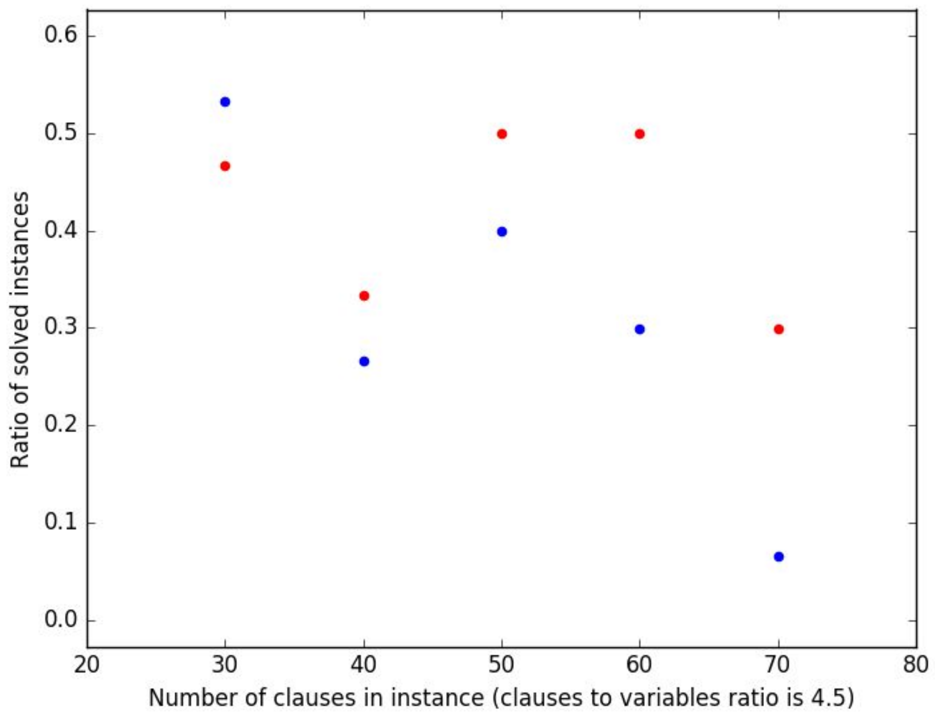
Tournament selection is a method of selection, when a certain group of individuals is selected and “tournament” is performed. The winner (an individual with biggest fitness) is added to new population. This is performed until population is full.



Tournament selection also improves performance of algorithm.

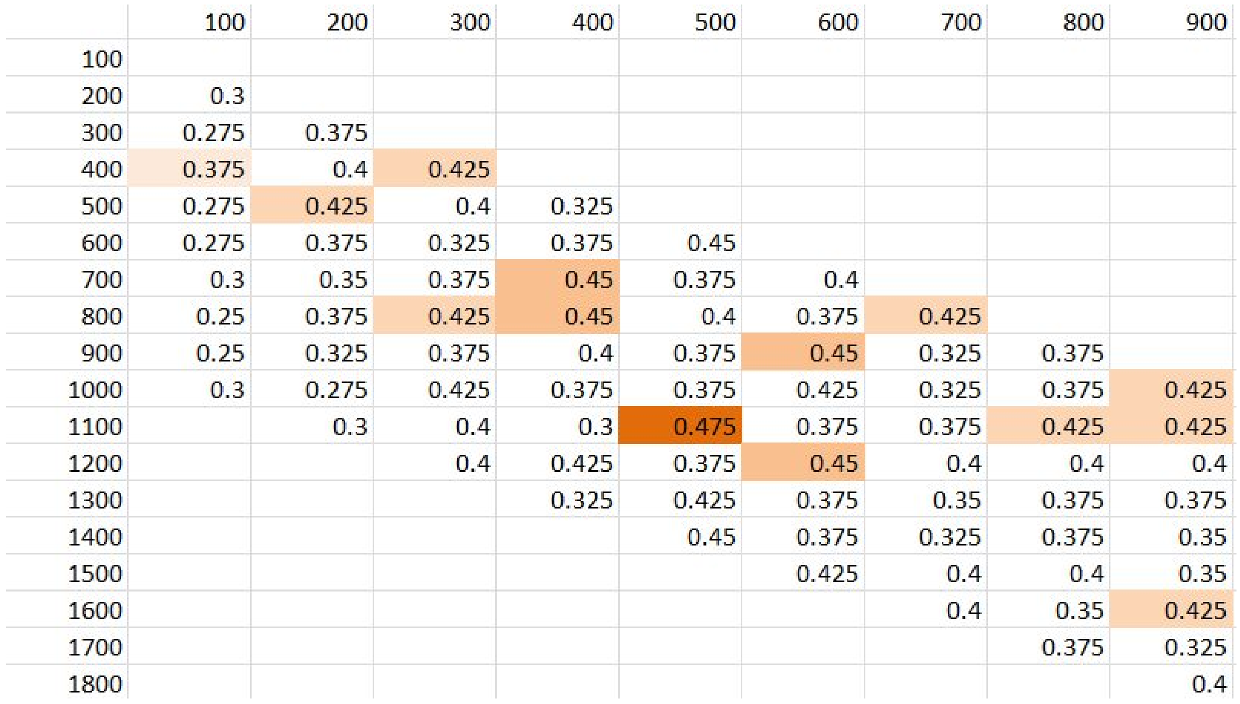
Combining Linear Scaling and Tournament Selection

Here is performance of algorithm with both linear scaling and tournament selection:



Setting Values of Satisfiability Bonuses

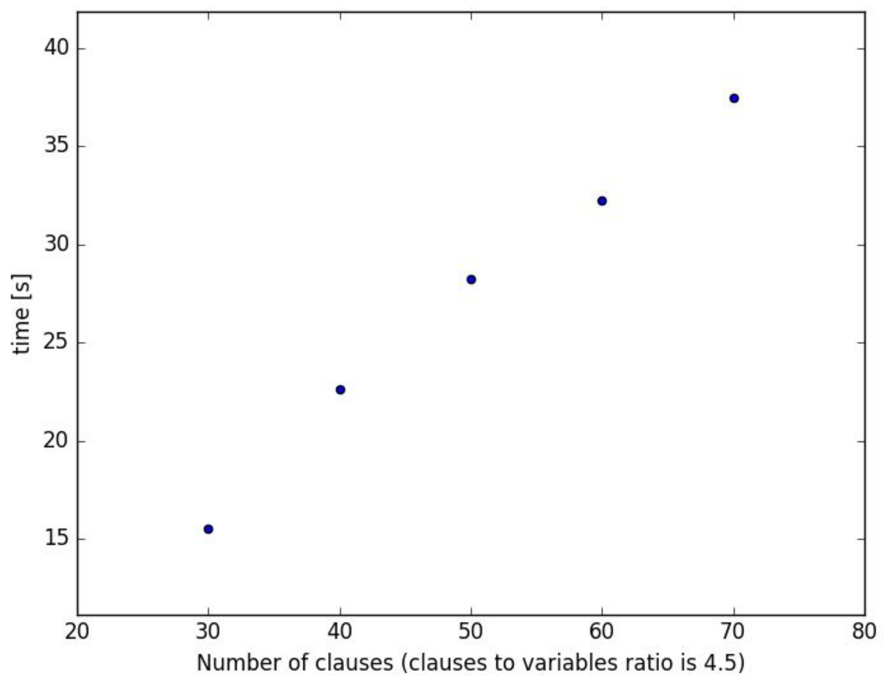
The algorithm computes fitness of individuals in such a way, that it adds a certain bonus for each satisfied clause and another bonus if the whole formula is satisfied. First I measured performance of algorithm with both bonuses equal to zero, and performance turned out to be zero as well. It is clear that awarding satisfied clauses and formula is necessary. I performed analysis of different combinations of values of these parameters and measured the result.



	100	200	300	400	500	600	700	800	900
100									
200	0.3								
300	0.275	0.375							
400	0.375	0.4	0.425						
500	0.275	0.425	0.4	0.325					
600	0.275	0.375	0.325	0.375	0.45				
700	0.3	0.35	0.375	0.45	0.375	0.4			
800	0.25	0.375	0.425	0.45	0.4	0.375	0.425		
900	0.25	0.325	0.375	0.4	0.375	0.45	0.325	0.375	
1000	0.3	0.275	0.425	0.375	0.375	0.425	0.325	0.375	0.425
1100		0.3	0.4	0.3	0.475	0.375	0.375	0.425	0.425
1200			0.4	0.425	0.375	0.45	0.4	0.4	0.4
1300				0.325	0.425	0.375	0.35	0.375	0.375
1400					0.45	0.375	0.325	0.375	0.35
1500						0.425	0.4	0.4	0.35
1600							0.4	0.35	0.425
1700								0.375	0.325
1800									0.4

In the table above columns correspond to bonuses for satisfied clause and rows correspond to bonuses for satisfied formula. In each column the greatest value is highlighted. The most optimal value of 0.475 corresponds to parameters values 500 and 1100.

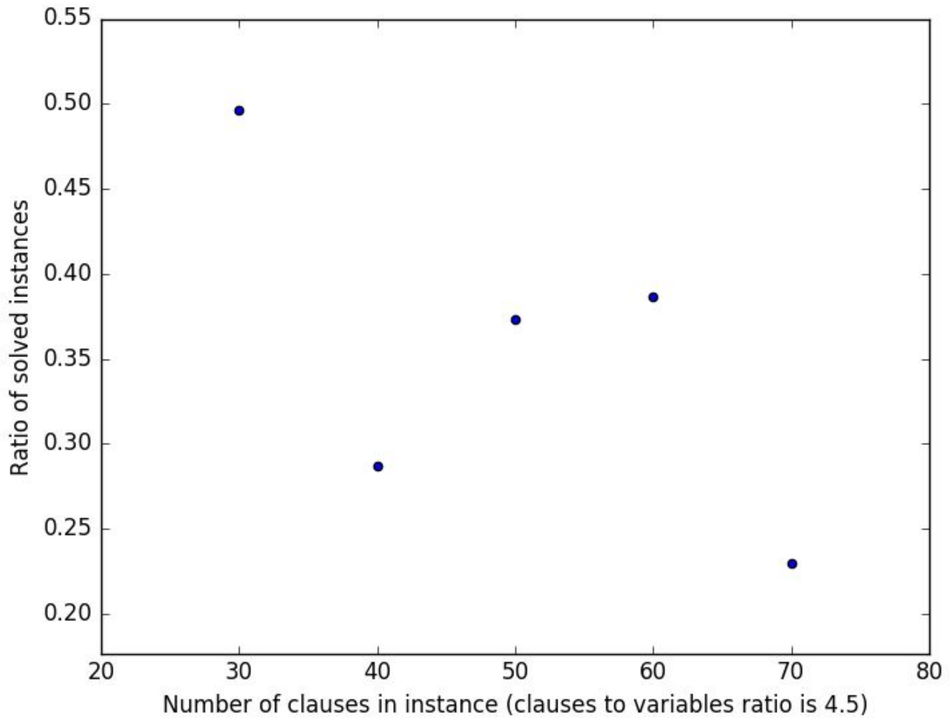
Measuring Runtime



The graph shows linear dependence between size of instance and computation time.

Final Performance Measuring

In previous sections I performed analysis and found the most optimal parameters values and techniques for genetic algorithm for solving 3 SAT problem. The final version of this algorithm uses tournament selection, one point crossover and linear scaling. Size of population, number of iterations, crossover probability and mutation probability were set to values 50, 500, 0.7 and 0.06 accordingly. Values of bonuses for satisfied clause and satisfied formula were set to 500 and 1100 accordingly. Below are results of measuring final algorithm's performance (ratio of solved instances) for different instance sizes.



The general trend is that with increasing instance size performance decreases. For 70 clauses and ratio 4.5 it reaches value 23%. However it is important to remember that not all of testing instances are solvable, that is why the real performance of this algorithm can be expected to be significantly higher. Also, instances of different sizes can have different ratios of solvable problems. This is most likely the reason for a drop for size 40.

Conclusion

In this project I have implemented genetic algorithm for solving 3 SAT problem and an instance generator, which I used for testing genetic algorithm. I have tuned algorithm's parameters such as size of population, number of iterations, crossover probability and mutation probability. For this I used measured ratio of solved problems to all problems. I have also tuned values of bonuses for satisfied clause and satisfied formula. I have tried implementing other techniques for subprocesses of genetic algorithm, such as uniform crossover, linear scaling and tournament selection. The last two improved algorithm's performance and were added to it. Then I measured final performance and runtime of my algorithm and it was significantly higher for bigger instances of problem.