Problems and Algorithms Homework 5 : MAX-SAT Solution Using an Advanced Iterative Method

Jimmy Leblanc

1 Problem Definition

Given a Boolean formula F having n variables, $X = (x_1, x_2, , x_n)$ in a Conjunctive Normal Form (CNF). Next, integer weights of the n variables $W = (w_1, w_2, , w_n)$ are given.

Your task is to find such an assignment $Y = (y_1, y_2, , y_n)$ of input variables $x_1, x_2, , x_n$, so that $F(Y) = 1$ and the sum S of weights of variables set to 1 is maximized.

2 Algorithm and implementation

MAX-SAT can be reduced to the weighted 3-SAT problem, where each clause consists of exactly 3 variables. Since the complexity of this problem is the same, but it's easier to be programmed, I chose to implement this version.

I chose to implement a genetic algorithm to solve this problem. I will assume that you are familiar with the concepts used in genetic algorithms and won't detail all of them here. I will sometimes use the abbreviation GA for Genetic Algorithm.

The code of my algorithm is in *maxSatSolver.py* and it is commented so for a technical insight (data structures ...) I advise you to check it, although I will explain the main principles in this report.

2.1 Data

$generateInstances(path, numberOfInstances, k, nbClauses, nbVariables, isSat, rangeMax, remove)$ generate SAT instances randomly accordingly to the parameters, k being the number of variables in each clause, all instances created and used in the experiments will use $k = 3$. All created instances will be stored in the given path. Instances respect the DIMACS SAT format but between the first line and the

line representing the first clause, I write a line representing weight of the variables.

You can load the instances to use them with $loadAllInstances(path, n, k, nbClauses, nbVariables)$, n being the number of instances with the given parameters you want to load. Nevertheless, the aim of the homework being to solve 3-SAT weighted problem, the rest of the algorithm has been implemented to work with 3-SAT instances only.

Then you case use $testGenetic(instances, k)$ to run the genetic Algorithm on the instances and get the best population.

2.2 The Basic Structure of a Genetic Algorithm

- Start : Generate random population of n chromosomes (suitable solutions for the problem)
- Fitness : Evaluate the fitness of each chromosome in the population
- Test : If the end condition is satisfied, stop and return the population. If not, generate a new population.
- New population : Create a new population by repeating following steps until the new population is complete:
  - Selection : Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
  - Crossover : With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
  - Mutation : With a mutation probability mutate new offspring at each locus (position in chromosome).
  - Accepting : Place new offspring in a new population
- Replace : Use new generated population for a further run of algorithm
- Loop : Go to the Fitness step

2.3 Generating a population

Regarding the genetic algorithm itself, we create a population with the $generate(nbVars, popSize)$ function, which generates a population of random assignment of variables coded by boolean values. So for example if we have 5 different variables, a gene of the population could be $[0, 1, 1, 0, 1]$ meaning $(x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1)$.

2.4 Fitness function

getFitness(pop,cnf,nbClauses,nbVars,k) function compute the fitness of a given population. In our case, fitness of an assignment is represented by the sum of the weights of selected variables in the population, divided by the number of clauses unsatisfied (relaxation).

2.5 Selection

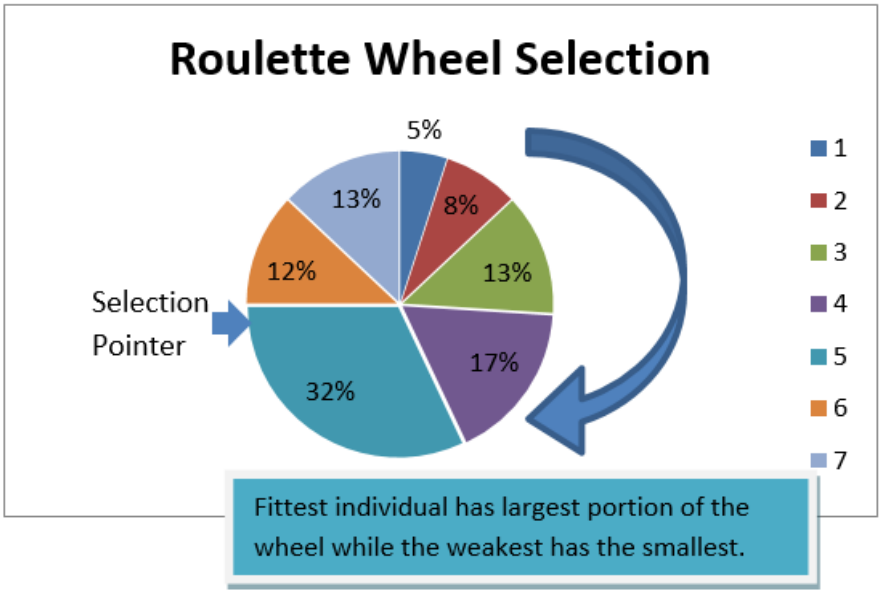

Figure 1: roulette wheel selection

To select populations, I chose to use Fitness proportionate selection, also known as roulette wheel selection. In fitness proportionate selection, as in all selection methods, fitness levels of populations are used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}, \tag{1}$$

where N is the number of individuals in the population. While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be.

With fitness proportionate selection there is a chance some weaker solutions may survive the selection process. This is an advantage, as though a solution may be weak, it may include some component which could prove useful following the recombination process.

2.6 Crossover and mutation

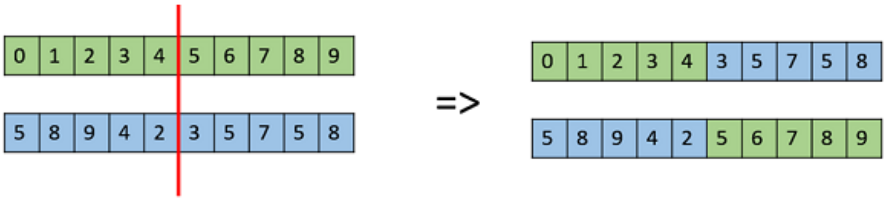

Figure 2: One-point crossover

Once we have selected two mate by using fitness proportionate selection, we carry out crossover on them. Crossover is the process of taking more than one parent solutions (chromosomes) and producing a child solution from them. By recombining portions of good solutions, the genetic algorithm is more likely to create a better solution. I chose to implement one-point crossover.

To do so, I chose a random point in the range of the length of the chromosome and then I create a child from both parents. A single crossover point on both parents' organism is selected. All data beyond that point in either string is swapped between the two parent organisms.

2.7 Termination

We run those steps until we have generated as many generation as the maximum given number of generations or until a given percent of the chromosomes fitness have converged.

3 Heuristics

3.1 optimum parameters

Find optimum parameters for a genetic algorithm is often quite difficult. Because of the no Free Lunch theorem for example. This basically states that there is no general search algorithm that works well for all problems. Thus it seems that the best we can do is tailor the search for a specific problem space and then manually tweak our parameters to fit your solution.Nevertheless, there have already been discussion that disputes this theorem ans even though the parameters does depend on the problem, there are methods which can help to find them.

There are two types of parameter setting:

- Parameter Tuning (offline parameter search - before the GA is run) :
  - Simple parameter sweep (try everything)
  - Meta-GA ontop of a GA
  - Racing strategy to find best parameters
  - Meta-GA and Racing together
- Parameter Control (online parameter tweaking - during the GA run)
  - Adaptive probabilities for mutation and crossover
  - Self adaptive population size
  - Deterministic Genetic Operators

Using a second GA to tune the parameters of the first GA is perilous, indeed how do we get the optimum parameters for the second GA ? However, I read several articles which stated that meta GA could be very efficient and find really good parameters, this is why I decided to implement a meta-GA. I initialized parameters of my meta-GA with the PyEvolve library default settings as they did in [1], i.e. population size 80, mutation rate 0.02, 1-point crossover, elitism enabled, maximum number of meta-GA generations was set to 20.

3.2 The meta-GA

The algorithm works like my first GA so I will just describe what is different. You can find its code in *metaGA.py*

3.2.1 Generating a population

The variables here will be the different parameters of the GA, so I create arrays containing a range of values for each parameter in $generateParams(popSize)$, and then I create a population by choosing randomly a value for each parameter.

The different possible values are as follows :

- population Size : between 20 and 100 with an interval of 10.
- mutation rate : between 0 and 50% with an interval of 1%.
- rate of convergence : between 50% and 100% with an interval of 10%.
- max number of generation : between 10 and 200 with an interval of 10.

3.2.2 Fitness function

How to estimate the fitness of a population in our case ? Well, first it must product the best solution possible. We do not have benchmarks to estimate our best solution, so to estimate a good solution, I execute this process :

1. run the meta GA
2. take the best parameters found
3. when running the meta GA again, before doing anything, we run the initial GA with the best parameters found before and then estimate the best obtainable score which will be used to sort the different population for further steps of the meta GA

But this is just a prerequisite because we want to get the best possible solutions. Nevertheless the fitness values won't be defined by this score but by the time its need to run on all the given instances, because we want our algorithm to process data as fast as possible. Thus, the problem becomes a minimization problem (usually, we want to maximize fitness, it is the case for the initial algorithm because we wanted to get the highest weight). A little modification is done to allow this.

3.2.3 Termination

We run those steps until we have generated as many generation as the maximum given number of generations. The only difference is that we do not wait for the populations to converge because there are too many possible fitness values.

Those are the only differences, otherwise, the implementation is similar from the initial GA.

The main drawback of this solution is that it is computationally expensive even for low population sizes, and if we want to approach the best solution, we need to increase the population size of the meta GA algorithm, which increases the running time. [2]

4 Experimental evaluation

4.1 Performance of the Genetic Algorithm

To test my algorithm, for each value of the number of clauses / number of variables ratio between 0.5 and 10, I created 100 instances of 3-weighted maxSAT, i.e. 100x19 = 1900 examples. The number of clauses is between 2 and 38. The number of variables is always 4. I then run my algorithm on every instance, and averaging results for each different value of ratio. For this purpose, I used the best parameters found by my meta GA (see next section), that is :

PopSize = 30
Mut = 4100
MaxGen = 10
Rate = 0.6
Elitism = True

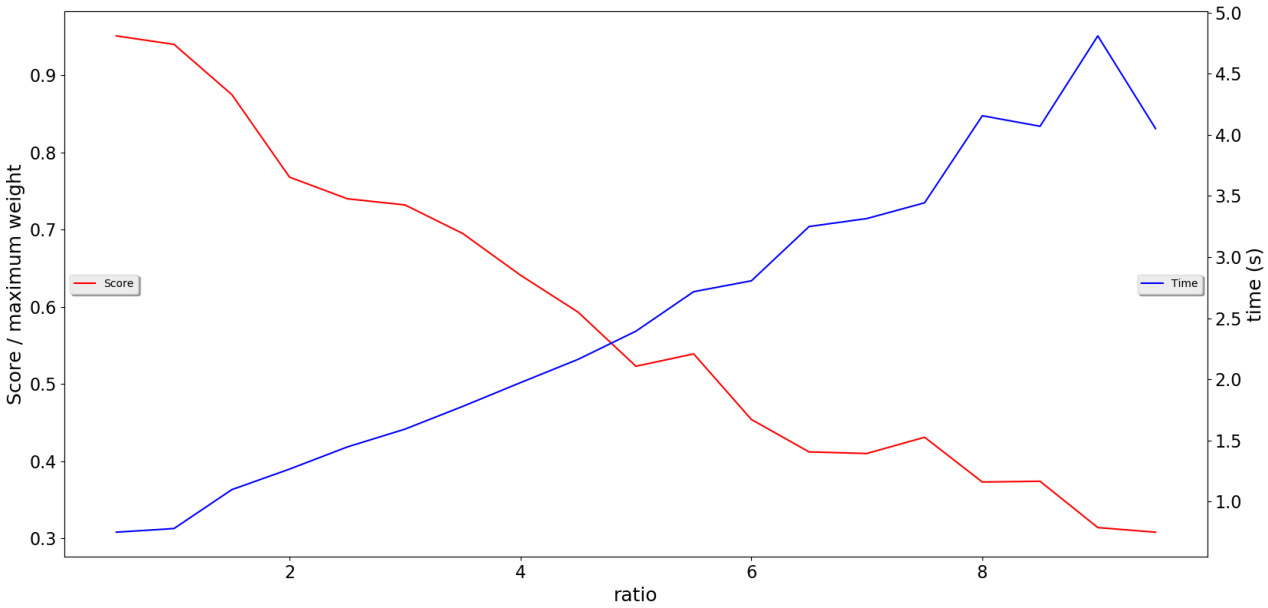

Figure 3: Number of clauses : 2-38, Number of variables : 4

As the ratio increases, the number of satisfied clauses deceases and so is the score / maximum weight. Of course the computing takes more time.

The number of clauses and variables are all defined by the *getFraction* function which takes a ratio as an argument and gives matching number of clauses and variables. Nevertheless when I increase the ratio, it keeps the same values for the number of variables and increases the number of clauses. I wanted to test to increases the number of variables too, in order to see if it has an effect on the performance. The following graphs shows the results with different values.

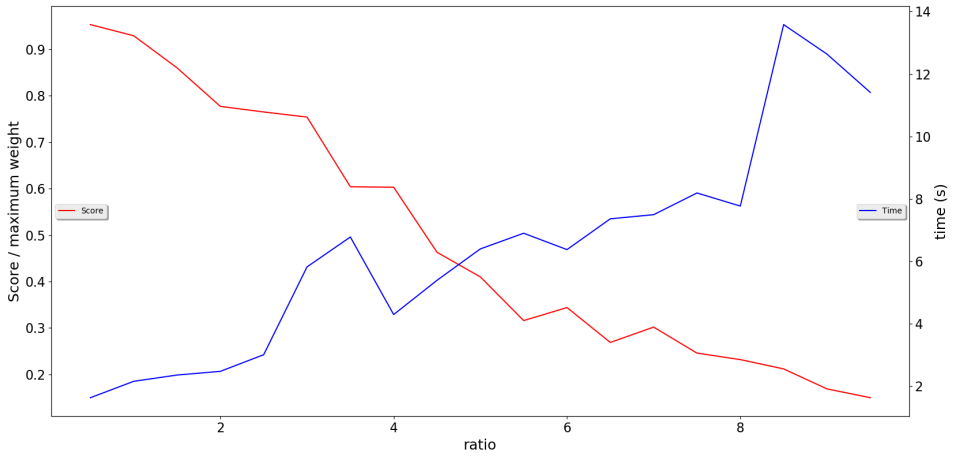

Figure 4: Number of clauses : 4-76, Number of variables : 8

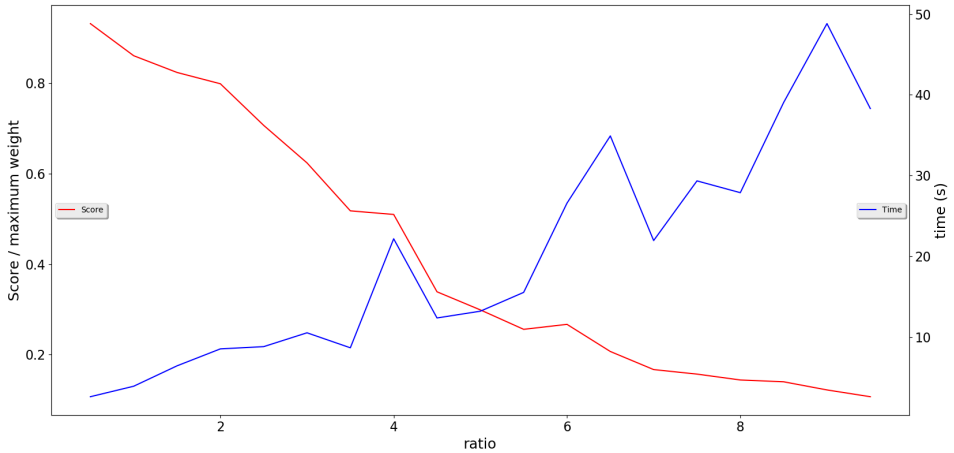

Figure 5: Number of clauses : 6-114, Number of variables : 12

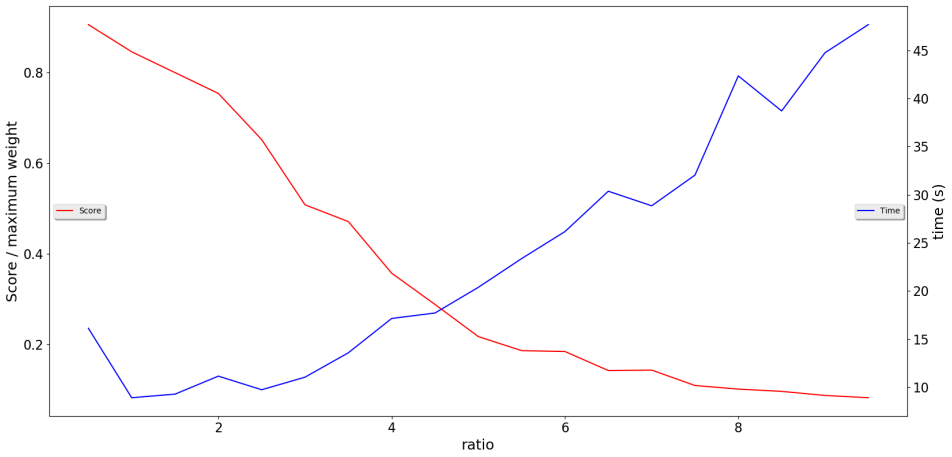

Figure 6: Number of clauses : 8-152, Number of variables : 16

Finally a graph plotting all the previous results. It is not really comprehensible but it shows that there is no really differences between them, they seem to follow the same trend and only the ratio seems determining on the results.

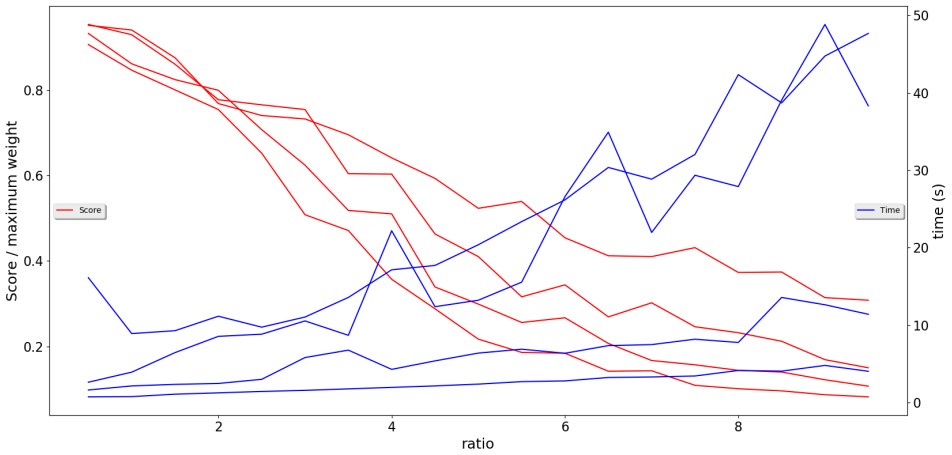

Figure 7: All the previous graphs

My algorithm seems fast and efficient, nonetheless it is hard to evaluate it correctly considering that we do not have any benchmarks. It would be

interesting to compare the two important results : the score / maximum Weight ratio and the computing time for the same instances with other algorithms.

4.2 Performance of the Meta Genetic Algorithm

Note : I realized recently that I made a mistake initializing my array of values for the mutation rate, indeed taking a value between 0 and 5000 with a step of 100 is actually take a rate between 1/100 and 1/5000 and not between 1/100 and 50/100, and I don't have the time to compute again my results. However, it seems that it is not a big deal because as you will see thereafter, the best values that we found are usually around 4000 which means that the mutation rate is around 0.025% and we rarely found higher rate than 0.033%.

At first I wanted to create a lot of instances of 3-weighted maxSAT with different values of the number of clauses / number of variables ratio to find some "universal" parameters, that is parameters which would work well, on average, on all the different instances. This is why I created 30 instances for each ratio, between 0.5 and 10, i.e. 570 examples, but since the meta GA is very computationally expensive, it would have taken me days of computing, especially if I set a high value for the size of the population, in both algorithms. I had to lower my ambitions. So I took 10 examples for each ratio, and runned the meta-GA with a population size of 20. Even with this, it is very long to get a result. It is too bad that I haven't been able to do as I wanted to because we may had discover real optimum parameters, but I was unfortunately limited by my hardware. Anyway I was able to found conclusive results even with this experiment. Indeed, here are the results I obtained for the different values of the ratio :

Ratio	PopSize	MutRate	Rate of convergence	MaxGen	time (s)
0.5	20.0	3000.0	0.8	10.0	0.182
1.5	20.0	4000.0	0.7	10.0	0.224
2.0	20.0	4100.0	0.6	10.0	0.267
3.0	50.0	1000.0	0.6	20.0	2.099
4.0	30.0	3200.0	0.7	10.0	0.689
5.0	20.0	3900.0	0.5	30.0	1.467
6.0	40.0	3700.0	0.5	20.0	2.549
7.0	20.0	3700.0	0.7	10.0	0.861
8.0	20.0	4000.0	0.6	20.0	1.504
9.0	50.0	4600.0	0.8	10.0	2.634

This table seems to indicate that no matter the ratio, some parameters stand out from the crowd :

- population Size : it is hard to distinguish a better value, but in general, more is better (but it increases computing time), nevertheless it seems that we can obtain very good values with a size of 20 in certain cases so I think even with 20 we would get good results in the other cases too.
- mutation rate : we have values between 1000 and 4600 but 1000 seems to be an outlier, around 4000 seems to be pretty good.
- rate of convergence : we have values between 0.5 and 0.8, 0.6 seems to be a happy medium.
- max number of generation : apparently we don't need a high number of generation, 10 is enough in a lot of cases, 20 should be good almost everytime.

Of course time increases with the ratio as we saw before, but there are some anomalies I think they would be eliminated with a higher number of examples and a higher population size for the meta GA (which was 20) . It seems that we succeed to detect some good parameters with our meta GA even though we would have got better results with more resources.

This idea of implementing a meta GA is a fruitful strategy, and it is not required to test some random values to compare the results to the GA's ones because this is exactly what the GA is doing, taking the best random parameters.

Further experiments :

- Try to run the meta GA with more instances with better hardware
- compare our results with some other algorithms on the same instances

References

[1] M. A. Addicoat and Z. Brain, "Using a meta-ga for parametric optimization of simple gas in the computational chemistry domain," 01 2010.

[2] B. Selman, "Stochastic search and phase transitions: Ai meets physics," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, (San Francisco, CA, USA), pp. 998–1002, Morgan Kaufmann Publishers Inc., 1995.