Homework Three

Iacobescu Tudor, A6

07 jan 2020

1 Introduction

This document means to examine the potential use of a genetic algorithm[1] in solving a real-world problem: the NP-complete problem of Satsfiability (or SAT). We will find, unfortunately, that our attempt yields unsatisfying results.

We will compare our algorithm with a random search algorithm to see how it performs, and why it's likely that the SAT problem requires a more specialized algorithm to be solved effectively.

2 Satisfiability problem

2.1 Decision problem

SAT is the first known NP-complete problem. It is defined as such:

$$\textit{input} : \text{a propositional logic formula } S$$

$$\textit{output} : \text{is } S \text{ satisfiable?}$$

A propositional logic formula is built from literals (representing boolean variables) and operators (AND, OR, NOT, and parantheses). A formula is said to be satisfiable if there exists a configuration of TRUE / FALSE values that can be assigned to its variables so that the value of the entire formula is TRUE.

2.2 Conjunctive normal form

Using the laws of boolean algebra, any propositional logic formula can be transformed into an equivalent conjunctive normal form - a conjunction (AND) of a number of clauses (disjunctions (OR) of a number of variables). This form can, however, be exponentially longer, but easier to properly analyze.

2.3 Optimization problem

A genetic algorithm cannot solve a decision problem. Genetic algorithms require a "gradient" - a gradual increase in the fitness landscape that allows the algorithm to climb towards better values. A decision problem doesn't have a gradient - a formula can either be satisfiable or unsatisfiable, and a configuration of values for the variables can either satisfy it or not.

One way to allow a genetic algorithm to attempt to solve the problem is by turning it into an optimization or maximization problem. The algorithm tries to find a configuration that satisfies the maximum number of clauses in the CNF of the formula - if it finds a configuration that satisfies all of the clauses, then the formula is satisfiable. In this way, a configuration satisfying more clauses is considered to have a larger fitness value than one that satisfies fewer.

3 Method

The test will run a genetic algorithm and a random search algorithm, both implemented in C++, against a set of 10 formulas in CNF. These formulas are inputted as text files in the DIMACS CNF format, obtained from the website of the Department of Scientific Computing of the Florida State University[2].

These files give the number of literals and clauses in the formula, as well as each clause formulated as a list of positive or negative integers. The absolute value of the integer represents the index of a literal, and the sign represents its value in the clause (i.e. if it is negated).

For example, the formula $(a \vee b) \wedge (b \vee \neg c \vee \neg d)$ can be represented as the following list of integer lists (clauses), where each integer corresponds to a literal:

$$[[1, 2], [2, -3, -4]]$$

A configuration for a certain formula is a list of boolean values, with the index in the list corresponding to the index of the literal as presented above. One configuration that can satisfy the above formula is this:

$$[true, true, false, false]$$

3.1 Algorithm

The test runner reads the files in the expected DIMANCS CNF format, ignoring comments, getting the number of literals and clauses from the problem line, and then reading the zero-terminated lists of characters. This data, encapsulated in a single object, is sent to the appropriate algorithm.

3.1.1 Genetic algorithm

The genetic algorithm is similar to the one used previously on Homework Two, with some changes specific to the problem made to try and enhance its performance.

Mutation While the crossover operator is the same, the mutation operator is changed in a way that should be, in theory, more helpful to the problem. After doing the standard set of random changes to an individual, it will then loop through the clauses of the formula and change the value of a random variable in clauses that are not satisfied by the configuration - a "positive mutation".

```
function mutate_SAT(population, problem):
    for individual in population:
        for boolean in individual:
            if randomFloat(0, 1) < 0.2 / individual.size():
                boolean = !boolean
        for clause in problem.clauses:
            if !satisfiedBy(individual, clause):
                positiveMutation(individual, clause)

function positiveMutation(individual, clause):
    index = randomInt(clause.size())
    value = clause[index]
    individual[abs(value) - 1] = (value > 0)
```

Selection Selection has a few changes as well. A couple of new optimizations were the addition of elitism (the best 10 individuals from every generation are guaranteed to be kept), and that of completely random individuals (also 10).

3.1.2 Random search

The random search algorithm is simple, designed mostly for comparison with the genetic one. It randomly generates configurations for ten minutes, and stores the best one.

```
function randomGeneration(problem):
    startTime = time()
    bestEval = 0

    i = 0
    while (true):
        individual = generateIndividual()
        currentEval = eval(individual)
        if currentEval > bestEval:
            bestEval = currentEval
            bestIndividual = individual
```

```
        i++
        if (i > 200):
            if time() - startTime() > tenMinutes():
                break

    return { bestEval, bestIndividual }
```

4 Experiment

Both algorithms were ran 30 times in parallel, their execution limited to 600000 ms (10 minutes). The genetic algorithm had 100 individuals per generation, and a limit of 3000 generations. The results of both algorithms are summed up in the table below.

5 Results

	#lit	#cl	S		rMean	rSDev	rMax	tMean	tSDev
1	50	80	Y	G	76.20	0.61	77	134649.43	2695.90
				R	78.10	0.31	79	600002.77	4.42
2	100	160	N	G	152.47	1.01	155	233597.50	3804.20
				R	153.77	0.57	155	600004.43	6.32
3	1040	3668	?	G	3335.20	32.44	3394	600568.33	248.08
				R	3125.70	8.67	3154	600007.67	9.89
4	60	160	N	G	154.00	1.55	157	219379.43	3937.70
				R	154.60	0.81	155	600007.07	6.91
5	63	168	N	G	161.67	1.09	165	228201.57	5068.99
				R	161.60	0.93	163	600005.13	6.08
6	66	176	N	G	168.93	1.34	171	221102.30	3281.13
				R	169.27	0.69	171	600002.43	3.89
7	42	133	N	G	129.67	0.48	130	144092.10	2267.90
				R	130.63	0.49	131	600003.80	5.60
8	64	254	?	G	238.47	2.30	243	299323.43	4745.90
				R	238.43	1.07	241	600003.17	4.60
9	16	18	Y	G	17.20	0.41	18	25006.57	12730.59
				R	18.00	0.00	18	113.37	117.69
10	155	1135	Y	G	1113.40	2.57	1118.00	600090.03	59.75
				R	1058.07	4.28	1064	600003.47	5.12

Table 1: Results table.

#lit = number of literals in a formula
#cl = number of clauses in a formula
S = is the given formula satisfiable? (yes, no, or unknown)
rMean, rSDev, rMax = mean, standard deviation and maximum value for the

number of clauses satisfied by the algorithm
tMean, tSDev = mean and standard deviation of time taken

6 Result analysis

At larger numbers of literals and clauses, the genetic algorithm outperforms the random search. At smaller scales, however, the random search algorithm gives similar results. As such, the genetic algorithm is more useful on large problems.

Neither algorithm managed to satisfy all clauses for any of the test cases, except for the simplest one, where the number of literals was small enough for the random search algorithm to find a solution very fast.

7 Conclusion

While a genetic algorithm can have applications in problems of satisfiability, it is more useful on large numbers of literals and problems where satisfying large numbers of clauses is more important than answering the question of satisfiability. In the end, it's likely that genetic algorithms are not the ideal solution for satisfiability problems[3], but as before, they prove to be possible solutions to a wide range of problems with only small specific optimizations.

References

[1] M. Breaban, "Algoritmi genetici." `https://profs.info.uaic.ro/~pmihaela/GA/laborator3.html`. Accessed on 6 nov 2020.

[2] J. Burkardt, "CNF files." `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`, May 2008. Accessed on 6 nov 2020.

[3] S. Harmeling, "Solving satisfability problems with genetic algorithms." `https://is.tuebingen.mpg.de/fileadmin/user_upload/files/publications/genetic_sat_[0].pdf`, March 2000. Accessed on 6 nov 2020.