CS 3445 – Fall, 2018
Project 1

Evolutionary Algorithms for MAXSAT:
A Comparison of Genetic Algorithms and Population Based Incremental Learning

Due Dates: September 21 and October 1

In this project, I want you to write a program that implements both a genetic algorithm (GA) and a population-based incremental learning (PBIL) algorithm to solving MAXSAT problems. I would prefer that you use Java, C, or C++ for your coding. Python is okay, too. I guess. If you have a strong preference for some other language, come talk to me. Whatever language you use, I need to be able to run it from the command line in the way described below for Java.

MAXSAT is a generalization of satisfiability (SAT). The solution to a SAT problem is an assignment of values to the Boolean variables that satisfies (makes true) *all* the clauses. The solution to a MAXSAT problem is an assignment of values to the Boolean variables that *maximizes* the number of clauses that are satisfied. Your program should implement both a basic genetic algorithm and the PBIL algorithm and the user should be able to specify which algorithm to run. Note that solving MAXSAT problems is more than an academic exercise. Like SAT, MAXSAT has been used to encode and solve very large practical problems. In fact, the repository I got the test problems from for this assignment (see the Problems section) has a test suite of "industrial-strength" circuit debugging problems with hundreds of thousands of variables and millions of clauses.

When you have written and debugged your program, design a set of experiments that will allow you to make a recommendation regarding which algorithm to use for MAXSAT problems. Of course, I'm not looking for a simple GA-yes/PBIL-no (or vice-versa) recommendation, although I suppose it's possible that your results indicate that one is always better than the other. Consider a number of factors:

1. degree of difficulty of implementation,
2. execution time,
3. size and or variety of problems that the algorithm is able to solve,
4. ease of use, e.g. how difficult is it to find good parameter settings and does it vary from problem to problem, and
5. anything else that seems appropriate to consider.

The Genetic Algorithm

Your GA code should allow the user to make the following choices through command line arguments in the order specified:

1. the name of the file containing the problem,
2. the number of individuals in the population,
3. the method for selecting the breeding pool; you should implement the following methods:
   (a) selection by ranking (`rs`), i.e. rank the individuals from 1 to N, where N is the population size and the most fit individual is given N, the highest rank, then select an individual with rank $1 \leq i \leq N$ with probability
   $$\frac{i}{\sum_{i=1}^{N} i}$$
   (b) selection by tournaments (`ts`), i.e. choose some number M of individuals randomly and take the best $k < M$ of those (where M and k are both constants for a given run), and
   (c) Boltzmann selection (`bs`), i.e. select an individual i with probability:
   $$\frac{e^{f_i}}{\sum_{j=1}^{N} e^{f_j}}$$
   where f_i is the fitness of individual i.
4. the crossover method to use; these should include:
   (a) 1-point crossover (`1c`), and
   (b) uniform crossover (`uc`).
5. the crossover probability,
6. the mutation probability,
7. the number of generations to run, and
8. which algorithm to run (`g` for the GA and `p` for PBIL).

For example, assuming you are using Java and the class with your main method is called EvolAlg, the following command:

```
java EvolAlg t3pm3-5555.spn.cnf 100 ts 1c 0.7 0.01 1000 g
```

would specify to run the GA on the problem in `t3pm3-5555.spn.cnf` for no more than 1000 iterations, with 100 individuals in the population, using tournament selection, 1-point crossover with a probability of 0.7, and a mutation probability of 0.01.

The PBIL Algorithm

As far as I know, there is no standard version of the PBIL algorithm, so I am providing pseudocode at the end of this handout.[1] This pseudocode differs slightly from the algorithm I described in class in that, out of the population of individuals generated during an iteration, only two are used to update the probability vector—the best and the worst. The probability vector is updated toward the best individual and "away from" the worst individual. The scare quotes are because the update based on the worst is not, strictly speaking, away from the worst. For each probability, if the bits in the best and the worst are the same, don't do anything. If you updated away from this, you would just be undoing what you did when you updated the probablity *toward* this in the update for the best. If the bits in the best and the worst are different, then update that probability toward the best one *again*. The two updates – best and worst – have potentially different learning rates. Thus, the parameters that must be specified on the command line are:

1. the name of the file containing the problem,
2. the number of individuals to generate in an iteration,
3. the positive learning rate (for the best-individual update),
4. the negative learning rate (for the worst-individual update),
5. the mutation probability (the probability that an element of the probability vector is mutated),
6. the mutation amount (the amount that an element of the probability vector is changed if it is mutated), and
7. the number of iterations to run, and
8. which algorithm to run (`g` for the GA and `p` for PBIL).

For example, assuming you are using Java and the class with your main method is called EvolAlg, the following command:

```
java EvolAlg t3pm3-5555.spn.cnf 100 0.1 0.075 0.02 0.05 1000 p
```

would specify to run PBIL on the problem in `t3pm3-5555.spn.cnf` for no more than 1000 iterations, generate 100 individuals for each iteration, update the probability vector toward the best individual using a learning rate of 0.1, update the probability vector away from the worst individual using a learning rate of 0.075, and mutate elements of the probability vector with probability 0.02 by an amount of 0.05.

[1] Taken from Baluja, S., "An Empirical Comparison of Seven Iterative and Evolutionary Function Optimization Heuristics," Technical Report CMU-CS-95-193, Carnegie Mellon University, 1995.

Output

Your program should stop immediately if it finds an assignment that satisfies all the clauses before hitting the maximum number of iterations. After the last iteration, it should print out the following information.

1. the name of the file containing the problem,
2. the number of variables and clauses in the problem,
3. the number *and* percentage of clauses that the best assignment you found satisfies,
4. the assignment that achieves those results, and
5. the iteration during which the best assignment was found.

The Problems

There is a repository of MAXSAT benchmark problems at:

http://www.maxsat.udl.cat/09/index.php?disp=submitted-benchmarks

I have provided files with some of the unweighted MAXSAT problems from this repository. You might want to start with a couple of tiny problems you create that you know the answers to, and use these for debugging purposes. The smallest problem I found in the repository was `maxsat-crafted/MAXCUT/SPINGLASS/t3pm3-5555.spn.cnf`, which has 27 variables and 162 clauses, so this might be a good next step. The `maxsat-crafted/MAXCUT/DIMACS_MOD` directory contains problems that are somewhat larger. Many of them have only about 40 variables and about 1,000 clauses. The `maxsat-random/max3sat` directory contains directories of 60-variable, 70-variable, and 80-variable problems. If you get really ambitious, try the industrial problems from the repository (about 5.5 GB expanded). As I mentioned above, many of these problems have hundreds of thousands of variables and millions of clauses.

These problem files all have the suffix `.cnf` (conjunctive normal form) and have the same format:

1. Lines that begin with a `c` are comments.
2. There will always be a line before the clauses that looks like:

   ```
   p cnf 40 600
   ```

   where `p` means problem type, `cnf` indicates conjunctive normal form, the first number is the number of variables, and the second number is the number of clauses.
3. Variables are numbers.
4. Positive numbers are positive literals; negative numbers are negative literals.
5. A clause is a list of numbers (literals) ending with a zero.

Solutions to many of the problems are available at:

http://www.maxsat.udl.cat/09/index.php?disp=results

Search for the problem name, look for the best result, which will be the one with the lowest O value, and download the "out" file. On my machine, I had to append `.txt` to the file to open it (in TextEdit).

For example, the solution to the `maxsat-crafted/MAXCUT/SPINGLASS/t3pm3-5555.spn.cnf` problem reads, in part:

```
Job file: /home/jplanes/evaluation/benchmarks/
    maxsat_crafted/MAXCUT/SPINGLASS/t3pm3-5555.spn.cnf
Job solver: /home/jplanes/evaluation/solvers/HanLin/inc-maxsatz
o 25
o 21
o 19
o 17
c Best Solution=17
c Program terminated in 0.000 seconds.
o 17
s OPTIMUM FOUND
v -1 -2 -3 -4 -5 -6 -7 -8 9 10 11 -12 13 14 -15 -16
17 -18 -19 -20 -21 -22 -23 -24 25 26 27
```

Note that the lines starting with a lower case "o" are indicating the number of unsatisfied clauses for the current best assignment and "Best Solution=17" means that the best solution satisfies all but 17 of the clauses. The assignment that produces that result is given as a list of variable numbers, where a positive number indicates that that variable is assigned `true` and a negative number indicates that that variable is assigned `false`.

Experiments

Remember, the main point of this assignment is to be able to make a recommendation regarding which algorithm to use for MAXSAT problems. The performance of each algorithm is highly dependent on its parameter settings. Ideally, one would explore the entire parameter space for each algorithm, but it's a *huge* space and this is clearly not possible. You should, however, explore some of the parameter space in order to discover whether there are differences between the GA and PBIL. More on this below. It's up to you to decide what experiments to run, but here are some general suggestions for determining the capabilities of each approach. Feel free to ignore any or all of them in favor of your own approach, as long as it allows you to make a well-supported recommendation.

For PBIL, start with the parameter values described under the pseudocode on the last page—they provide values for all the command line arguments you can play with, except for number of iterations. The number of iterations used in the experiments in the report was 2,000, so you might want to start with that value for that parameter. For the GA,

you might as well start with the same number of individuals and the same number of iterations. Use any selection method, 1-point crossover, and use values for crossover and mutation probabilities from the rule-of-thumb parameter ranges that I mentioned in class: maybe 0.7 probability of crossover and 0.01 probability of mutation. I would expect that these parameters will allow you to do reasonably well on the smaller problems. If not, try varying population size, number of iterations, type of selection method, and type of crossover method, one at a time. For the numerical ones–population size and number of iterations–vary them over a small set of values widely spaced values to get a sense of the effect. For example, you could try, for number of iterations, 6,000 and 10,000. You should be able to find a combination of parameters that deals well with samll problems.

Of course, you shouldn't test only on small problems with negligible solution times, since varying the parameters is unlike to have a measurable impact on a solution time of, for example, one second. So, using the parameters that gave you good results on the small problems, test your program on increasingly large problems, until you reach a point at which you can't get "good" results in a "reasonable" amount of time. I would suggest that "good" means satisfying at least 90% of the number of clauses satisfied by the optimal solution, and "reasonable" amount of time means a few minutes. Of course, if you want to take the time to let your program run longer on some of the larger problems, that's great. At this point, you would want to try varying the parameter settings again. One possibility would be to leave the population size, number of iterations, type of selection method, and type of crossover method fixed, and investigate changes in the probabilities of crossover and mutation.

There are a number of ways to do this. One possibility would be to leave the probability of crossover at 0.7 and vary the probability of mutation, say testing $\{0.0, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0\}$. Then, based on those results, fix the probability of mutation and vary the probability of crossover using the values $\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. This would require 13 tests. Another approach, that requires more tests, but explores the interaction between these two parameters more thoroughly, would be to test all possible combinations of crossover and mutation probabilities from the sets $\{0.1, 0.5, 1.0\}$ and $\{0.005, 0.01, 0.1\}$, respectively. This requires 9 tests. Then, based on those results, explore more deeply. For example, suppose the best combination is 0.5 probability of crossover and 0.1 probability of mutation. Try all possible combinations of crossover and mutation probabilities from the sets $\{0.4, 0.5, 0.6)$ and $\{0.05, 0.1, 0.5\}$, respectively. That's 9 more tests. But those 18 tests have given you a much better sense of the interplay between these two parameters.

Again, you don't need to follow these suggestions. You may decide to focus on the effect of population size and number of iterations. It's difficult to specify exactly what your experiments should be and where you should stop. You should push your program to its limits (within reason) and test it on enough problems to get a good sense of the capabilities of the GA compared to those of the PBIL algorithm.

Report

Being able to write a clear, well-organized scientific paper is a skill that is just as important as being able to write a program that uses a GA or PBIL to solve MAXSAT problems. Arguably, more important. I have seen many articles in which employers cite *communication*

skills as one of the most important qualities they are looking for. The other most cited skill is *the ability to work on a team*, which you are also getting experience with here. To make it worth your while to spend a good deal of time on this part of the assignment, I am making this 50% of your grade. There is some guidance in this section on how to write such a report. I will also post on BlackBoard a recent conference paper of mine that will give you a concrete example (more on this below).

Your report should have the following:

- **Title**: The title of this handout would be suitable.
- **Abstract**: (optional) The abstract is basically a condensed version of the Introduction.
- **1 Introduction**: An overview of what you did, briefly addressing the following points (each in *about* a paragraph):
  - Explain the problem you are attacking, i.e. MAXSAT, at a fairly high, conceptual level at this point.
  - Explain the algorithms you are using, i.e. GA and PBIL, again at a conceptual level at this point.
  - A summary of what you did.
  - A summary of what you found.
  - A roadmap of the paper (e.g. "In Section 2, we describe.... In Section 3, we ... In Section... Finally, we conclude with a section...")
- **2 MAXSAT**: A more detailed description of the problem and your approach.
- **3 Genetic Algorithms**: A more detailed description of how a GA works.
- **4 Population Based Incremental Learning**: A more detailed description of how PBIL works.
- **5 Experimental Methodology**: A more detailed description of your experimental methodology. What experiments did you run? What did you measure? Why? How many tests did you run? On what problems? These questions are not intended as a checklist. You don't need to make sure you answer each one and I'm sure they aren't exhaustive.
- **6 Results**: A description and analysis of your results (see below for more details).
- **7 Further Work**: What would you do next if you were going to work on this for a few more weeks, or a few more months?
- **8 Conclusions**: Summarize your results.

Of course, the section describing and analyzing your results is the most important section. Keep in mind that your data is not self-explanatory. You should use graphs, tables, charts, and/or figures to present your data and support your conclusions, but this section should not be a collection of graphs and tables presenting your data whose interpretation is

left as an exercise for the reader, nor should it be a series of one or two sentence descriptions of your graphs, tables, etc. This may seem obvious, put previous experience suggests otherwise. The text should refer to the graphs/table/charts/figures in support of your claims. For example "As the graph in Figure 8 shows, the effect of..." Any graphs should be clearly drawn (a title, axis labels with units, a legend if more than one series of values is being shown) and captioned.

Clarity is *very* important. Your report should tell a connected, coherent, and self-contained story. Although this is a scientific paper, you are still telling a story and your reader needs to be able to follow the plot. You may want to use subsections within your sections to enhance readability, e.g. the results section might benefit from further subdivision, especially if your project addressed multiple issues.

Don't make the mistake of writing this report for an audience of me. Someone who knows nothing about what you worked on should be able to read your report and understand, at least at a high level, what you did. Assume you're writing it for a CS major who knows nothing about MAXSAT, GAs, or PBIL. Even in conference papers and presentations, which are being prepared for an audience of CS researchers, the author(s) will provide a basic explanation of everything, even though some readers or people in the audience will already know that.

Example Paper

I have posted a recent conference paper of mine on BlackBoard in a folder called `majercik-example-paper`. This is a good example of the type of paper I want you to be patterning your paper on. I don't expect you to write a paper of this length, and you will notice that I don't follow the above guidelines on sections. Like you, I had two algorithm descriptions (mine were Standard PSO and GR-PSO), but I made them subsections of a single section, rather than two separate sections. I combined **Experimental Methodology** and **Results**. And I combined **Further Work** and **Conclusions.** Thing like this are fine, as long as you cover all the bases. Also, I don't expect you to have a **Related Work** section.

You don't need to follow the style of my paper exactly. This just happened to be the style required by the particular conference that this paper was in. For example, although the 2-column format is widely used in conferences, some conferences (and all journals) use the standard one "column" format. In case you know Latex and you would like to use this style, I have included the style file in the folder. I have also included the `bib` file. (If you don't know what that is, you probably won't need to.)

Deadlines

Starting this project a few nights before it's due is not a good idea. To help prevent that, there is an intermediate deadline for a draft of part of your paper:

Friday, September 21:

Submit a hardcopy of your report, *except for*:

- the few sentences in the Abstract that will summarize your results,
- the few paragraphs in the Introduction that will briefly explain your results, and
- Sections 5 through 8.

I will review your paper and provide feedback in time for you to improve it, if necessary.

Monday, October 1:

- Submit a folder containing a copy of your code and your finished report on BlackBoard. All the files I need to run your program should be in a directory that includes a `README` file containing clear instructions for running your program. Your code should be documented well enough that I can easily see what you are doing.
- Submit a hard copy of your report.
- Submit, **individually**, a confidential report assessing each group member's contribution to the project.

Between October 2 and October 5 (last day before Fall Break):

- A project review session with me. These sessions should take between half an hour and an hour. I will ask you questions about your code, e.g. where the code is that does a particular thing, why you did things in a particular way, etc. and your report. I will expect anyone on the team to be able to answer any questions, so even if, for example, someone did not work on the code, they should still understand it well enough to be able to answer questions about it. And everyone should be able to talk about what's in the paper. I will also ask you to run your program for me on some test problems.

Do not count on my usual cavalier attitude toward deadlines. Missing any deadline will reduce your grade. The Particle Swarm Optimization project, which is of a similar size, will be coming out when this project is due (October 1) and will have a similar schedule.

Grading

Your assignment will be graded as follows (roughly):

40%: the correctness and quality of your code,

20%: the quantity and quality of your experiments, and

40%: the content, organization, and clarity of your report.

```
****** Initialize Probability Vector *****
for i :=1 to LENGTH do P[i] = 0.5;

while (NOT termination condition)
        ***** Generate Samples *****
        for i :=1 to SAMPLES do
                sample_vectors[i] := generate_sample_vector_according_to_probabilities (P);
                evaluations[i] :=Evaluate_Solution (sample[i]);

        best_vector := find_vector_with_best_evaluation (sample_vectors, evaluations);
        worst_vector := find_vector_with_worst_evaluation (sample_vectors, evaluations);

        ***** Update Probability Vector towards best solution *****
        for i :=1 to LENGTH do
                P[i] := P[i] * (1.0 - LR) + best_vector[i] * (LR);


        ***** Update Probability Away from Worst Solution *****
        for i :=1 to LENGTH do
                if (best_vector[i] ≠ worst_vector[i]) then
                        P[i] := P[i] * (1.0 - NEGATIVE_LR) + best_vector[i] * (NEGATIVE_LR);


        ***** Mutate Probability Vector *****
        for i :=1 to LENGTH do
                if (random (0,1) < MUT_PROBABILITY) then
                        if (random (0,1) > 0.5) then mutate_direction := 1
                        else mutate_direction := 0;
                        P[i] := P[i] * (1.0 - MUT_SHIFT) + mutate_direction * (MUT_SHIFT);
```

USER DEFINED CONSTANTS (Values Used in this Study):

SAMPLES: the number of vectors generated before update of the probability vector (100).

LR: the learning rate, how fast to exploit the search performed (0.1).

NEGATIVE_LR: the negative learning rate, how much to learn from negative examples (PBIL = 0.075, EGA = 0.0).

LENGTH: the number of bits in a generated vector (problem specific).

MUT_PROBABILITY: the probability for a mutation occurring in each position (0.02).

MUT_SHIFT: the amount a mutation alters the value in the bit position (0.05).