**Evolutionary Algorithms for MAXSTAT:**

**A Comparison of Genetic Algorithms and**

**Population Based Incremental Learning**

*By Jimmy Lemkemeier, Jake Rourke, Alex Weinberger*

**Abstract**

The MAXSAT problem is an NP-hard computation that seeks to find the maximum number of clauses that can be made true (i.e. “satisfied”) in a boolean proposition. MAXSAT is not an academic exercise -- any solution to it can also solve other NP-hard problems, meaning a dramatic increase in our ability to use computers to solve incredibly difficult problems. Genetic Algorithms (GA) and Population-based Incremental Learning (PBIL) are two nature inspired algorithms that seek to approximate solutions to optimization problems subject to constraints. In this paper, we implement GA and PBIL and assess the relative benefits of each in approaching the MAXSAT solution. Our results indicate that GAs are slower but more accurate than PBIL.

**1. Introduction**

The problem we are addressing is MAXSAT, a variation of satisfiability problems. Satisfiability problems consist of boolean variables that make up clauses. In the MAXSAT the goal is to satisfy as many full clauses as possible. To do this we are using two evolutionary algorithms. The first is a genetic algorithm (GA). The genetic algorithm comes from the generational evolution observed in nature. The idea is to allow the best individuals in a population of solutions to survive, mutate, and combine with each other, thereby improving the solutions in the long run. Our other algorithm that we will be using is Population Based Incremental Learning (PBIL). PBIL takes a lot of the same ideas from the genetic algorithm and also uses learning techniques. Rather than having a population of solutions, PBIL uses a probability vector to generate candidate solutions. At each iteration, the probability vector is updated toward the best candidate and away from the worst. In the next section we will describe the MAXSAT problem in greater depth. In sections 3 and 4 we will go into greater detail about the genetic algorithm and PBIL respectively. Section 5 will be a detailed account of the tests we did with these algorithms, and why we chose to do these tests. Section 6 will detail and analyze the results we collected from our testing. Section 7 will describe the future work left to be done with these algorithms and this problem. We will then give a summary of our work and our conclusions in section 8.

**2. MAXSAT**

The MAXSAT problem is a NP-hard problem derived from satisfiability (SAT). In a SAT problem, there are n number of boolean variables and m number of clauses. A boolean variable is a variable whose value is either *true* or *false*. A clause is a list of boolean variables with specifications as to which variables need to be true and false in order for the clause to be true. These n variables are combined in various ways to make up those m number of clauses. The solution to a SAT problem is finding the values of those n boolean variables which makes all of the clauses true.

A MAXSAT problem starts with the same premise; there are n number of boolean variables and m number of clauses. However, instead of trying to find the values of the variables which makes all the clauses true, you are simply trying to find values of the variables which makes as many clauses as possible true. This allows there to always be a solution to the MAXSAT problem.

**3. Genetic Algorithms**

Genetic Algorithms are based on the evolutionary process of natural selection. The idea is that only the fittest individuals in a population will be chosen to reproduce and carry offspring into the next generation. Similarly, the algorithm attempts to improve its population of solutions with each iteration.

Before we get into the algorithm, there are two important components that must be initially defined. First we must decide how to represent the solutions. Some generic ways of doing this could be using bits in a string or using a list of ID’s. This will change depending on the problem. Still, there will be various options to choose from for any problem. Next, a proper fitness function is needed. This too will be different for any given problem. The fitness function should be able to evaluate how close a solution is to the optimal solution. It’s also important to consider the efficiency of the fitness function as it will be performed on each individual during each of the iterations.

The first step of the algorithm is to *generate the population*. The size of the population, N, will be specified in a command line argument for our algorithm. The population could be seeded with solutions that are already known to be good, but for our algorithm we will generate N individuals randomly. This step will be done only once, rather than with each iteration.

The next step is to *evaluate the fitness* of each individual. At this point, we run the predetermined fitness function on each individual to allow us to gauge how close each individual is to the optimal solution.

Once the fitness for each individual is determined, a *selection* process is used. At a high level, selection is the process of choosing generally more fit individuals to use for the breeding pool. There are several ways to do this, but for our algorithm we will have 3 options to choose from. The selection method will be specified as a command line argument. The first method we will use is selection by ranking. In this method, each individual is given a rank based on fitness. Individuals are selected with replacement based on a probability determined by their rank: . In this equation, *i* is the individuals rank, and N is the population size. The benefits of this method include the ability to handle outlying fitness values and negative fitness values. The next method we will use is selection by tournaments. In this method we randomly choose n individuals, without replacement, and put the best m in the breeding pool. Typically in this method n=2 and m=1. The last method we will use is Boltzmann selection. In this method we choose individual i with fitness with probability where N is the population size.

Now that we have our breeding pool, we perform the process of *crossover*. In general, crossover is the process of choosing individuals from the breeding pool and mixing them up with some probability (usually [0.6, 0.9]). In our algorithm, we use two variations of crossover. The first is 1-point crossover. In this method we first choose two individuals from the breeding pool. Then a probability, usually [0.6, 0.9], is used to decide if the process is going to continue. If the process is to continue, both chosen parents are split at the same point. The location of this split point is usually chosen with equal probability. The parents then exchange one side of the split point, in the end creating 2 children to go on to the next step. The other crossover method we will be using is uniform crossover. In this method, we start off similarly by selecting two parents from the breeding pool and deciding whether or not to continue. If we are to continue with the process, we form one new child by choosing each symbol for it from the parents. For each symbol for the child, we choose from the parents with equal probability. It's important to note that in this method, we will only create one child each time. To handle this we can either double the breeding pool size or do this method twice. Also, for either method we will have the probability that crossover happens as a command line argument.

The next step is *mutation.* In mutation, we go through each child that we just made, and give each symbol a small probability of changing. Probability values for this are usually [0.001, 0.02]. This mutation probability will be specified as a command line argument. In some applications it makes more sense to do swaps within the child rather than a random change.

At this point we have a new population and have completed one iteration of the algorithm. The next iteration will begin with this new population having its fitness evaluated and continue through the rest of the steps. The number of iterations that will be performed is specified as a command line argument.

**4. PBIL**

PBIL combines many of the evolutionary mechanisms of GA with a learning process. As discussed, two important concepts drawn from nature undergird GA -- survival of the fittest, as replicated in the selection process, and random information exchange, which we manifest in the recombination and mutation phases. PBIL borrows these same concepts but applies them to a very different idea of what constitutes a population and individual.[[1]](#footnote-0)

The PBIL algorithm uses a probability vector as a “prototype” for vectors that do well in exploring the solution space. The goal of PBIL is to create a probability vector that represents a population of high evaluation solution vectors.[[2]](#footnote-1) That means that each operation in the iterations of PBIL occur not directly on sample solutions, as in GA. Beginning with an even probability vector, where the likelihood that every bit be flipped “on” is .5, we generate new vectors, evaluate the fitness of these new vectors, then update the existing probability vector based on a combination of the relative changes in fitness and the speed at which we want to incorporate improved vectors.

Before we outline the algorithm’s strategy, we need to define certain terms. The *probability vector* represents the likelihood of every bit position containing a “1”.[[3]](#footnote-2) The *learning rate*, alluded to above, decides how fast we move the probability vector toward the best generated individual. The *negative learning rate* does the converse -- it decides how quickly we move the probability vector away from the worst generated individual. (Note that some PBIL algorithms use more than just the best and worst two individuals to update the probability vector; this implementation uses only the two.) The *mutation probability* represents the likelihood that an element in the probability vector is mutated; the *mutation amount* represents the magnitude of that change.

Set a given number of iterations to run and a given number of samples to create at each iteration. We then operate four steps: sample generation/eval, updating toward best, updating away from worst, and mutating.

*Generate and evaluate samples:* For each iteration in the number we are to run, create that number of sample using the existing probability vector.Evaluate the samples: say we find that a sample with a lot of 1s in the 4th position and 0s in the 6th position outperforms all other samples; we want to adjust upward the probability of getting 1s in the 4th and downward the probability of 1s in the 6th. Additionally, say that a sample with a lot of 1s in the 5th position and 0s in the 7th position. We would want to move away from the likelihood that we get lots of 1s in the 5th and 0s in the 7th.

*Update toward best solution:* So we update each bit probability by replacing part of the existing probability of each index. The “update” equation is

P[i] = P[i] \* (1.0 - LR) + best\_vector[i] \* LR

where LR is the “learning rate.” The greater the learning rate, the more the probability vector shifts toward the best solution sample. In this case, we would replace part of the 4th probability with a more positive segment, and part of the 6th probability with a less positive segment.

*Update away from worst solution:* Just as we updated up the 4th and 6th, we want to update the 5th and 7th to include the information about the worst probabilities. Say that at this iteration probability\_vector[5 (ignoring 0-base indexing)] = .3; we want to replace some of that with the adjusted vector, but not all of it (in case we are approaching a local maximum, let’s keep our options open). The actual equation is

P[i] = P[i] \* (1.0 - NLR) + best\_vector[i] \* NLR

where NLR is the negative learning rate, or the amount that each iteration moves away from the worst solution. (The equation uses best\_vector[i] rather than worst because if the individual bit is different in best\_ and worst\_vector, then moving toward best is functionally the same as moving away from the worst.)

*Mutate:* lastly, we want to mutate our probability vector some. This step is fairly straightforward: randomly check if a number is less than the mutation probability, and if so, randomly pick a direction and shift every vector in the probability vector by that amount.

The twin concepts of GA -- movement toward promising solutions and randomization -- are found in PBIL; however, the actual mechanisms by which the two algorithms achieve these goals are starkly different. PBIL continually updates a probability vector based on the evaluation of generated samples and returns the final probability vector.

**5. Experiments**

In order to test GAs vs PBIL, we split our experiments into three tests: a small problem, a medium problem, and a large problem. The small problem consisted of 27 variables and 162 clauses. The optimal number of clauses which could be satisfied was 145 or 90% of the clauses. This problem was the smallest one available and therefore the easiest. The medium problem consisted of 64 variables and 384 clauses. The optimal number of clauses which could be satisfied is 346 or 90%. The largest problem consisted of 125 variables and 750 clauses, of which 672 or 90% could be satisfied at the same time. By comparing GAs vs PBIL on these problems, it can determined if there is a better algorithm on both easier and harder problems.

In order to compare these algorithms at optimal performance, a parameter tuning was done for both algorithms on the easy and medium level problem. For the sake of time, a parameter sweep was not done for the hardest problem. The number of iterations was kept constant at 2000 in order to keep results consistent and relatively time efficient. The other parameters were originally set the way they were suggested in the assignment and adjusted from there. We did not tune parameter combinations for the sake of time. After we found the optimal values for each parameter, we reran the program with those values to see what the most accurate score the algorithm could achieve on that problem.

For the largest problem, we simply used the optimal parameter values found in the previous problems and ran each algorithm once to compare the results.

Results from all of the problems were compared to determine the benefits of each algorithms.

**6. Results**

Looking at the results from the smallest problem, both algorithms were able to find the optimal solution. However, PBIL was able to do it in less than half of the time.

As shown in Table 1, the parameter sweep for PBIL on the smallest problem showed that the original parameters hit the optimal solution very efficiently. The original parameters found that 90% of the clauses could be satisfied and it found that solution in 21 seconds. No parameter changes made the algorithm arrive at a better solution or arrive at the same solution in a faster amount of time.

Looking at Tables 3-5, the parameter sweep for the Genetic Algorithm on the smallest problem found parameter variations which strengthened the algorithm’s performance. The original parameters found that 88% of the clauses could be satisfied. A couple of parameter changes found that 90% of the clauses could be satisfied. Increasing the population size helped the algorithm achieve that optimal result but at a severe time increase. Increasing the crossover probability as well as the mutation probability also helped the algorithm hit that optimal solution and these variations did not noticeably increase the time. Switching the selection method to ranking selection instead of tournament selection also allowed the algorithm to hit the optimal solution with a slight increase in time. Therefore, we determined that the optimal parameter setting would be a population size of 100, using tournament selection and 1-point crossover, with a crossover probability of 0.8 and a mutation probability of 0.1. This resulted in finding the optimal solution (90% of the clauses could be satisfied) most efficiently (46 seconds).

Therefore, both of these algorithms could be used on this easy problem, however PBIL is better given it found the same optimal solution in less than half of the time.

For the medium level problem, the genetic algorithm found the optimal solution more often than PBIL at the expense of added time.

As shown in Table 2, the original parameters for the PBIL did not find the optimal solution, (optimally 90% of the clauses can be satisfied). In 53 seconds, PBIL found that 89% of the clauses could be satisfied. Surprisingly, that accuracy could not be increased with a larger population size. The only parameter change which allowed the optimal solution to be found was increasing the mutation amount from 0.05 to 0.5. However, this increase in accuracy was most likely a lucky run since a mutation amount that high can cause anything to happen. Upon rerunning that same test multiple times the original score of 90% could not be replicated. Therefore, PBIL could not consistently find the optimal solution to this harder problem.

The GA was able to consistently find the optimal result at the expense of using some extra time. Tables 6-8 show that the GA using the original parameters found, in 108 seconds, that 90% of the clauses could be satisfied. This is the optimal solution however it took much longer than the 53 seconds that PBIL took on average for this same problem. Varying the GA parameters did not help in finding a more efficient way to find the optimal solution. Most surprisingly, increasing the population size for the Genetic Algorithm actually resulted in a decrease in accuracy along with an increase in time. One possible explanation for this would be that there is a local minimum which the population can easily become stuck at for this problem. Therefore, the two trials with a larger population size got stuck at this local minimum. The accuracy of the trials with a larger population size was the same as the accuracy for the PBIL which could further imply that there is a tricky local minimum for this problem. However, overall the GA was able to find the optimal solution more frequently for this problem at the expense of added time.

At this point in the experiment, the prevailing hypothesis emerged that GA’s were slower but more accurate. We continued to test this hypothesis by running both algorithms on the largest test. Running PBIL with the original parameters found, in 106 seconds, that 664 clauses could be satisfied in this problem. The GA found, in 222 seconds, that 666 clauses could be satisfied. Neither approach found the optimal solution to this problem, however these results confirmed our conclusion that GAs were slower but more accurate.

This increase in accuracy most likely stems from the GAs ability to keep the best solutions in the population over time. PBIL’s technique of summarizing and regenerating all of the individuals does not allow the outlying best individuals to survive throughout multiple generations. The time increase for GAs is probably due to the combination and mutation of individuals. This is a time costly process which needs to be done many times for many individuals. PBIL merely adjusts the probability vector, a more time efficient but less accurate operation. Another concern with PBIL is the small number of sample solutions responsible for updating the probability vector; only the best and worst samples are used. Consider the possibility that the best sample exhibits a particular trend A, while the 2nd best, 3rd best, …, 10th best all exhibit a separate trend B. PBIL will not recognize the likelihood that B may be more likely to lead to optimization across a number of iterations, and will instead update toward A.

Regardless of the reasoning behind the conclusion, the results imply that GAs are slower but more accurate. As for giving a recommendation on which algorithm to use, it would depend on the situation. For the easiest problems, PBIL would be better since it can find the solution more efficiently. Once you get into more complex problems, it would depend if time or accuracy is valued more. If accuracy is valued more, GAs would be recommended. If time was a higher priority, PBIL would be able to achieve a slightly less accurate result significantly faster.

**7. Further Work**

There are a number of different ways to research this comparison further. First, running each experiment multiple times would help in minimizing the random variation of the accuracies which could be affecting our results. Secondly, optimizing parameter combinations could optimize each algorithm more effectively, leading to a more clear analysis of each algorithms potential. Third, running these algorithms on more than three tests and parameter tuning all of the parameters on each test would allow the results to be more enforced. Lastly, other research could be done on comparing the two approaches on different problems rather than MAXSAT in order to determine if the same conclusions hold across different problems.

**8. Conclusions**

By using the MAXSAT problem as judge, this paper compared the accuracy and speed of GAs vs PBIL. By testing on three different sized MAXSAT problems, the algorithms were compared on both easy and difficult tasks. Both algorithms found the optimal solution to the easiest problem, however the GA became more accurate as the problems became more difficult. In order to achieve this added accuracy, the GA took almost twice as much time on each problem so it was found to be much less time efficient. Therefore, it was concluded that GAs are slower but more accurate.

**Tables:**

Table 1 (PBIL Small Problem Parameter Sweep):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Large Decrease | Small Decrease | No Chance | Small Increase | Large Increase |
| Num Individuals | .88 (10)  2 seconds | .88 (50)  11 seconds | .90 (100)  21 seconds | .88 (200)  46 seconds | .90 (500)  109 seconds |
| Positive LR | .82 (0.001)  21 seconds | .90 (0.01)  22 seconds | .90 (0.1)  21 seconds | .88 (0.25)  22 seconds | .90 (0.5)  23 seconds |
| Negative LR | .87 (0.001)  22 seconds | .90 (0.01)  23 seconds | .90 (.075)  21 seconds | .88 (0.2)  22 seconds | .90 (0.5)  23 seconds |
| Mutation % | .90 (0.001)  22 seconds | .87 (0.01)  22 seconds | .90 (0.02)  21 seconds | .90 (0.1)  22 seconds | .88 (0.5)  22 seconds |
| Mutation Amount | .87 (0.001)  21 seconds | .87 (0.01)  22 seconds | .90 (0.05)  21 seconds | .90 (0.1)  23 seconds | .90 (0.5)  22 seconds |

Table 2 (PBIL Medium Problem Parameter Sweep):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Large Decrease | Small Decrease | No Chance | Small Increase | Large Increase |
| Num Individuals | .88 (10)  5 seconds | .88 (50)  25 seconds | .89 (100)  53 seconds | .89 (200)  98 seconds | .89(500)  254 seconds |
| Positive LR | .79 (0.001)  52 seconds | .89 (0.01)  52 seconds | .89 (0.1)  53 seconds | .87 (0.25)  49 seconds | .88 (0.5)  54 seconds |
| Negative LR | .89 (0.001)  49 seconds | .87 (0.01)  50 seconds | .89 (.075)  53 seconds | .89 (0.2)  53 seconds | .88 (0.5)  54 seconds |
| Mutation % | .89 (0.001)  50 seconds | .88 (0.01)  50 seconds | .89 (0.02)  53 seconds | .89 (0.1)  51 seconds | .87 (0.5)  55 seconds |
| Mutation Amount | .86 (0.001)  54 seconds | .88 (0.01)  52 seconds | .89 (0.05)  53 seconds | .89 (0.1)  54 seconds | .90 (0.5)  52 seconds |

Table 3 (GA Small Problem Parameter Sweep Part 1):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Large Decrease | Small Decrease | No Chance | Small Increase | Large Increase |
| Num Individuals | .88 (10)  5 seconds | .88 (50)  23 seconds | .88 (100)  46 seconds | .88 (200)  95 seconds | .90 (500)  235 seconds |
| Crossover Prob. | .88 (0.1)  45 seconds | .90 (0.5)  45 seconds | .88 (0.7)  46 seconds | .90 (0.8)  47 seconds | .90 (0.9)  45 seconds |
| Mutiation Prob. | .88 (0.0001)  46 seconds | .90 (0.001)  45 seconds | .88 (0.01)  46 seconds | .90 (0.1)  46 seconds | .87 (0.5)  46 seconds |

Table 4 (GA Small Problem Parameter Sweep Part 2):

|  |  |  |
| --- | --- | --- |
|  | 1c (No Change) | uc |
| Crossover | .88  46 seconds | .88  48 seconds |

Table 5 (GA Small Problem Parameter Sweep Part 3):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ts (no Change) | rs | bs |
| Selection | .88  46 seconds | .90  49 seconds | .87  47 seconds |

Table 6 (GA Medium Problem Parameter Sweep Part 1):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Large Decrease | Small Decrease | No Chance | Small Increase | Large Increase |
| Num Individuals | .88 (10)  11 seconds | .89 (50)  55 seconds | .90 (100)  108 seconds | .89 (200)  213 seconds | .89 (500)  538 seconds |
| Crossover Prob. | .83 (.1)  117 seconds | .88 (0.5)  121 seconds | .90 (0.7)  108 seconds | .90 (0.8)  117 seconds | .88 (0.9)  116 seconds |
| Mutiation Prob. | .89 (0.0001)  110 seconds | .90 (0.001)  110 seconds | .90 (0.01)  108 seconds | .84 (0.1)  113 seconds | .82 (0.5)  111 seconds |

Table 7 (GA Medium Problem Parameter Sweep Part 2):

|  |  |  |
| --- | --- | --- |
|  | 1c (No Change) | uc |
| Crossover | .90  108 seconds | .89  117 seconds |

Table 8 (GA Medium Problem Parameter Sweep Part 3):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ts (no Change) | rs | bs |
| Selection | .90  108 seconds | .90  112 seconds | .90  114 seconds |

**Citations:**

Baluja, Shumeet. “Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning.” Technical Report, CMU-CS-94-163, Computer Science Department, Carnegie Mellon University, *January 1994*.

1. Shumeet Bajula, “Population-Based Incremental Learning: A Method for Integrating Genetic Search Based on Function Optimization and Competitive Learning,” Technical Report, CMU-CS-94-163, Computer Science Department, Carnegie Mellon University, January 1994, 1. [↑](#footnote-ref-0)
2. Shumeet, “Population-Based Incremental Learning,” 11. [↑](#footnote-ref-1)
3. Shumeet, “Population-Based Incremental Learning,” 11. [↑](#footnote-ref-2)