No Small QuiPBIL: Truth, Recombination, and MAXSAT

Dustin Hines, Duncan Gans, David Anderson

1 October 2018

1 Abstract

Evolutionary Algorithms provide a valuable optimization and search tool by mimicking evolutionary processes in nature. Just as evolution works by natural selection to evolve populations better suited to succeed in their environment, evolutionary algorithms work by evolving a population of solutions to a problem to better solve this problem. The broad tree of computational evolutionary algorithms includes several algorithms, all of which have several parameters and possible implementation options. For this project we focus on optimizing Population Based Incremental Learning (PBIL), and a Genetic Algorithm (GA). We first implement these two algorithms, then test various parameter values for each algorithm, and ultimately compare their effectiveness to optimal solutions on MAXSAT problems. After optimizing parameter values, we determined that both evolutionary algorithms provide promising optimization tools, coming within a few percentage points of the optimal solutions. Between PBIL and GA, we determined that our Genetic Algorithm was slightly more successful and also more consistent.

2 Introduction

MAXSAT (maximum satisfiability) refers to a version of satisfiability (SAT) problems where there is not necessarily a solution that satisfies all clauses. Instead, the solution to a MAXSAT problem is the maximum number of clauses that can be satisfied. In MAXSAT, clauses are strings of literals in conjunctive normal form. Related to MAXSAT, a SAT problem is the set of literal assignments that make all Boolean clauses true. MAXSAT is a more general form of SAT in the sense that there does not necessarily need to be a solution that makes every clause true.

To solve MAXSAT problems, we implement both a genetic and population-based incremental learning algorithm. These algorithms are both inspired by the natural process of natural selection and are similar in many ways. In general, genetic algorithms and PBIL both work by iteratively developing a solution to a many dimensional problem. A standard genetic algorithm has three main components: selection, recombination, and mutation. First, a random population is generated where each individual represents a potential solution to the problem. Then individuals are selected by some measure for recombination (crossover). All selection methods will account for a fitness score that is an evaluation of how good the individual's solution is at solving the given problem. At the recombination step, there is a chance that the individuals are not recombined in which case they would progress to the next generation. After the crossover step, there is a round of mutation where individuals have some (small) chance of having their solution changed. This increases the randomness of the search, thereby increasing exploration. These steps are repeated with each new generation until a stopping point is reached.

PBIL is very similar to GA but instead of having individuals, PBIL uses a probability vector (PV) for representing the population. The probability vector is iteratively adjusted by generating individuals and evaluating the fitness of their solution. The probability vector is adjusted towards good solutions using supervised learning. For each component of a good individual, each component of the probability vector is adjusted in that direction by some learning rate.

In section 2, we describe the MAXSAT problem and our two approaches to solving it. In section 3, we cover the first of these approaches, which is a genetic algorithm. In section 4, we discuss our implementation of PBIL, which is our second approach to solving MAXSAT. In section 5, we cover the methodology we used to arrive at a recommendation for which algorithm we think is better suited for solving MAXSAT problems. Then, in section 6, we describe the results from our experiments. In section 7, we discuss some opportunities for future work. Finally, in section 8 we conclude with a summary of our results.

3 MAXSAT

For a given series of conjunctive normal form clauses, the maximum satisfiability solution is the assignment of true/false values to each literal that maximizes the number of clauses that are true. Each clause is made up of at least two literal values which can either be true or false. A clause evaluates as true if any of the literals in the clause have a value of true. For example, for literals *L1*, *L2*, *L3*, *L4*, consider the clauses:

1. $(L3)\ or\ (L1)$
2. $!(L1)\ or\ !(L1)$
3. $!(L3)\ or\ (L2)$
4. $!(L2)\ or\ !(L2)$
5. $(L4)\ or\ !(L1)$

The MAXSAT solution for this set of clauses is 4, since at most 4 clauses can be satisfied. If *L1* is FALSE, *L2* is TRUE, *L3* is TRUE, and *L4* is TRUE, clause 1, 2, 3, and 5 will be satisfied.

Brute forcing a maximum satisfiability problem would potentially involve checking 2^n possible solutions. To avoid needing to use an exponential approach to solve MAXSAT problems, we implement both standard genetic and population-based incremental learning algorithms to solve these problems.

4 Standard GA

Genetic Algorithms (GAs) are optimization and search algorithms inspired by processes of natural selection. The gradual evolution of solutions towards better solutions parallels the evolution of populations towards the traits of highly successful individuals. The algorithm begins by choosing a population of random solutions in a solution space. This is the starting population. It then iteratively selects the fittest individuals for breeding, breeds them by combining aspects of both solutions, and then randomly alters some of the solutions. At each iteration, the solution population improves by continually selecting for better solutions and combining good solutions together for potentially better solutions.

4.1 Generating The Population:

The biggest question in generating the population is how large to make the population. Larger populations will lead to more exploration of the solution space but will result in longer run-times of the algorithm. Although population sizes vary, they are often between 30 and 100 individuals. Once the population size is determined, each individual's initial solution is determined randomly in the multi-dimensional solution space. Sometimes some of the individuals will intentionally be solutions known to perform well, but more often they are truly random.

4.2 Selection

After the initial population is generated, and at the beginning of every generation, the solutions go through selection. This mimics natural selection and chooses the most fit solutions out of the subset. This requires an evaluative function that can score the fitness of each solution. For instance, if the goal is to maximize an equation, the evaluative function would just proscribe higher fitness to solutions that produce larger function values. There are several methods of selection, with each of them having various strengths and weaknesses. We will use three:

Selection by Ranking: This selection requires ranking the solutions from most fit to least fit, from 1 to n, with n being the population size. The worst solution receives a score of one, and the best receives a score of n. These are then chosen with the probability being an individual solutions rank divided by the sum of all the ranks. This method ignores the significance of differences between levels of fitness, which can help to ensure the breeding pool is not a small subset of a few highly fit individuals, while still favoring greater fit.

Selection by Tournament: This randomly selects some number M of individuals from the population, and from these individuals chooses a subset k with the highest fitness levels to go into the breeding pool. Like ranking, this ignores the weight of differences between fitness levels, instead opting to factor in the order.

Boltzmann Selection: This method factors in the actual fitness of individuals, rather than simply how they compare to other individuals. Each solution is given the probability below, where e is the mathematical constant of the base of the natural logarithm.

$$Probability_i = \frac{e^f}{\sum_i^n e^f}$$

4.3 Recombination

Recombination is the process of choosing members of the breeding pool randomly, and breeding them. When two solutions are breeding, you give them some probability of crossover (typically [.6, .9]). If they do not crossover, they will immediately go to the next stage (mutation) unchanged. Otherwise, for this project they will engage in one of two forms of crossover. The strategy of crossover is to mix up good solutions to allow for the possibility that mixing solutions will add the best parts of both solutions and create a solution that is better than either unmixed individual

1-Point Crossover: This chooses a random crossover point in the solution string and swaps what comes after the crossover point. For instance, with the two individuals,

$$AABBABAAB|BBABAA$$

$$and$$

$$ABBBAABBA|BABAAB.$$

If the crossover points are denoted by the vertical lines, then the parts on either side of the solution will swap, and the resulting individuals will be

$$AABBABAABBABAAB$$

$$and$$

$$ABBBAABBABBABAA$$

Uniform Crossover: This method is simpler. To mix the individuals, you will iterate through each element in the solution string, and flip a coin to determine which parent the solution element will be from. Unlike 1-point crossover, this means that there wont necessarily be continuous sections of solution elements that belong to at least one parent. Because this creates just one individual, you either double the breeding pool, or select two individuals from each pair of parents.

4.4 Mutation

The final step of the genetic algorithm is mutation. Mutation allows individuals to alter their solutions without needing those solutions to exist in the population. The rates of mutation per solution element are very low, typically between [.001, .02], but can be higher or lower depending on the problem. This percentage is typically either one divided by the population size, or one divided by the number of symbols. Each element of the solution will be changed at a rate equal to the mutation percentage, and it will change to a randomly selected alternative value if there are more than one.

4.5 Advantages and Disadvantages

Genetic Algorithms are a relatively common way to search/optimize a solution in a solution space, but they are not the only one. Genetic Algorithms have several benefits and drawbacks in comparison to other methods. On the positive side, they are straightforward you always have an answer, the environment can be messy, it can handle a large search space, and you can often hybridize it with other learning methods. However, determining representation can be difficult, you need a fitness function ahead of time, and there are lots of parameters that need to be optimized to get the best solutions. Finally, there is the inevitable problem of early convergence towards a local optima, which needs to be solved by changing parameters. On the whole, genetic algorithms provide a straightforward and powerful method for searching and optimizing.

5 Population Based Incremental Learning (PBIL)

Population based incremental learning is a commonly implemented alternative to more traditional genetic algorithms. Like genetic algorithms, PBIL is inspired by the processes of natural selection and is used to solve hard optimization problems. Unlike genetic algorithms, PBIL does not maintain a population of "individuals" that compete, recombine and reproduce over time. Rather, PBIL maintains a "probability vector" $\vec{p}$ that is used to generate candidate solutions. The fitness of these solutions is then assessed and the best solution is used to update the probability vector. Before iterating again, the population vector may be mutated slightly. This process is repeated until an optimum solution is found.

For example, $\vec{p} = (0.5, 0.5)$ might generate individuals $v_1 = (0, 0)$ and $v_2 = (1, 1)$. If we selected v_2 as the more fit individual, both values in $\vec{p}$ would be increased and the generation and selection process would begin again.

5.1 Generation of Initial Solutions

Since PBIL does not maintain a constant population, it is not necessary to generate a random field of individuals. Rather, one must generate a probability vector that will subsequently be used to generate individuals. Unless the designer of the implementation has some prior knowledge of the problem at hand, a common approach is simply to set the likelihood of every solution to be equal at initialization.

After setting the initial probability vector, a population of N candidate solutions is generated according to the probability vector. For example, assuming a large value of N , the vector $\vec{p} = (1, 1)$ would be expected to only generate candidate solutions of the form $(1, 1)$. By contrast, if $\vec{p} = (0.5, 0.5)$ approximately one quarter of the generated solutions are expected to be of the form $(1, 1)$, $(1, 0)$, $(0, 0)$ or $(0, 1)$ respectively. This generative step repeats every time the algorithm is run.

5.2 Selection and Learning

After generation of a pool of candidate solutions, the single best candidate is selected and the rest are discarded. Metrics of "best-ness" may vary, but for the purposes of optimization minima and maxima within the candidate pool are often used. In MAXSAT problems, the best solution is the one that makes the largest number of clauses true.

Once the best candidate has been selected it must be used to update the probability vector. This is usually accomplished via a competitive supervised learning algorithm where each index in the probability vector is updated according to the formula

$$\vec{p}_i = (\vec{p}_i)(1 - a) + \vec{s}_i a$$

where $\vec{s}$ is the vector representing the selected candidate and a is a constant representing learning rate (typically close to 0.1. When this process is iterated, p will eventually approach a binary vector (that is, a vector containing all 0's and 1's. When this occurs, candidate solutions will typically all be the same, indicating that some sort of optimum has been reached. Ultimately, the binary form of $\vec{p}$ should be the solution to the problem.

5.3 Mutation

PBIL is structured to forgo recombination, but mutation is still implemented. Rather than applying mutation to individuals, PBIL applies mutation to the probability vector. Essentially, before the generation of each new pool of candidate solutions, each index in $\vec{p}$ is shifted may be shifted up or down by ι_u or ι_d respectively at rate μ. Higher values for μ promote exploration, whereas lower values favor rapid convergence to a solution. Like almost all parameters in evolutionary algorithms, these values may be updated dynamically.

5.4 Advantages and Disadvantages

PBIL has many of the same advantages as traditional genetic algorithms. It also comes with its own set of extra disadvantages. First, there is the question of how many candidate solutions to generate. This is akin to setting population sizes in genetic algorithms. A second difficulty arises when choosing how many of those individuals to select for learning. The approach described above simply takes the single most fit individual, but it is possible to select multiple individuals and using each individual to update the probability vector, though this raises the further problem of how to weight each individuals contribution to $\vec{p}$. Furthermore, PBIL introduces several new parameters. The learning rate is an extremely important parameter that must be kept track of, and mutation alone requires three parameters.

Nevertheless, PBIL has its advantages. It may be simpler conceptually to implement than genetic algorithms, and in many problems has been shown to outperform them. PBIL does not require the maintenance of a large continuous population, so may be faster to run or make better use of memory. Lastly, PBIL does not implement recombination, so difficulties associated with recombining complex data structures are at least partially glossed over by the algorithm.

Despite these obvious advantages and drawbacks, it is difficult to say which algorithmic approach is more effective at solving a given problem until it has actually been implemented and tested on that problem. Experimental comparative implementations of the two approaches are necessary to determine which approach is best for a particular problem type.

6 Experimental Methodology

6.1 Genetic Algorithm

To test our genetic algorithm, we first looked at the performance of difference parameter ranges and then used this insight to run our GA against MAXSAT problems with known solutions and to look at our margin of error compared to the optimum solutions. Additionally, we looked how at run-time is effected by each parameter.

6.1.1 Parameter Optimization

A major question for testing a genetic algorithm is finding optimal parameter values. Our GA has 7 parameters: the population size, selection method, crossover method, crossover probability, mutation probability, and number of generations. There are some rules of thumb for what to set these parameters to, but we decided to check a wide combination in a brute force style to gain some initial insight. For testing our GA, we began by testing a wide array of possible combinations of parameters together to find the ideal mix of parameter values. This gives us an idea of how the parameters interact with each other to find ideal solutions. For every parameter we would check several values within a given range, and do this for all parameters simultaneously. This leads to a multi-dimensional array with the values being the success of the algorithm, and the array indices being the parameter values. By converting the data into an excel file, we could see which combinations of parameters preformed best.

We then looked at parameters in a more isolated way. This was done by setting the other parameter values at a constant level and iterating through a range of values for a single parameter. We generally used tournament selection, uniform crossover, a population size of 100, a crossover probability of .7, a mutation probability of .01, and 100 generations as constants while varying individual parameters. These values were based on rule-of-thumb knowledge and which values had been found to perform well during initial testing. When testing for the effect of a parameter, we would vary this parameter and leave the other parameters at these values. By isolating variables we were able to get a more specific idea of what optimal parameter values might be. To be fair, this depends on other parameters, so it is possible that some parameters that perform poorly with a subset of other parameter values would perform better with different parameter values. However, for meaningful analysis, we test for how one varying parameter value will work with an otherwise mostly optimal set of parameter values. We ran the isolated parameter tests against six crafted MAXSTAT test files from http://www.maxsat.udl.cat: maxcut-140-630-0.7-1.cnf, maxcut-140-630-0.7-2.cnf, maxcut-140-630-0.7-3.cnf, maxcut-140-630-0.7-4.cnf, maxcut-140-630-0.7-5.cnf, maxcut-140-630-0.7-6.cnf. Since these files are crafted, the optimal solutions are known. We considered the number of correct clauses in the solution, the run-time, and the generation the best solution was found to measure the success of the GA as we varied the parameters.

Finally, we used the our insights from our parameter testing to run additional tests to consider the accuracy of our solutions.

6.2 PBIL

Testing of PBIL followed in a similar vein to our tests of the GA. However, as the algorithms operate differently they rely on separate parameters and must be approached individually. The parameters relevant to PBIL are population size, the number of individuals selected from each generated population for adjustment of the population vector, the learning rate α, the mutation rate, the amount to shift by when mutating, and the number of generations to run.

First, we set parameters based on typical values in other studies and adjusted them until we had a good qualitative idea of the ranges of parameter values we wanted to test. Then, we took a similar approach to parameter optimization as in GA optimization, testing ranges of individual parameters against known solutions while holding the other parameters constant. The constants used during optimization were population size = 100, number of individuals selected = 1, learning rate = 0.1, mutation rate = 0.01, mutation shift = 0.01, and 100 generations.

6.2.1 Testing Efficacy of PBIL

To test the effectiveness of PBIL for solving MAXSAT problems, we used the same crafted MAXSAT files used for testing the GA. We ran the PBIL with optimum parameters three times

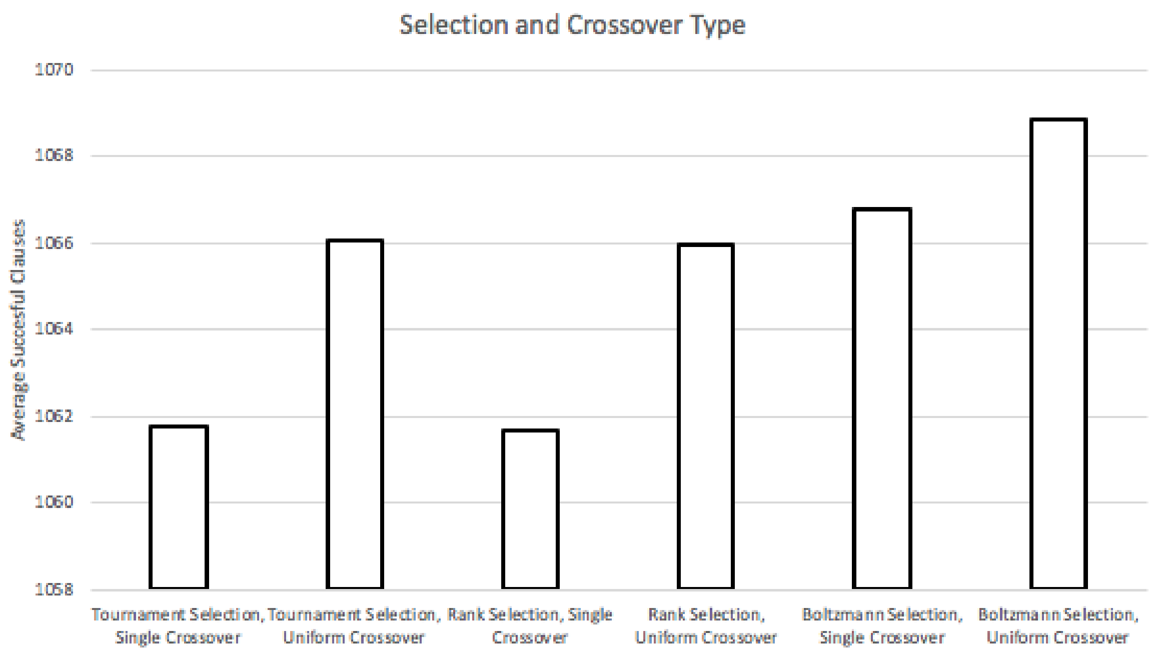

Figure 1: Selection Methods and Performance.

for each of the ten crafted problems and calculated average performance as the percentage of clauses fulfilled. We then compared these results to those generated by the GA to determine the relative merits of each approach.

7 Results

7.1 Genetic Algorithm Parameter Optimization

7.1.1 Selection and Crossover Type

For selection, we tested Boltzmann selection, tournament selection, and rank selection each with uniform and single-point crossover. Rather than use 100 generations and a population of 100, we decided to increase the number of generations so that our results wouldn't favor selection types that converged quickly. So, we used 400 generations, and a population of 60 to offset the additional run-time caused by increasing the number of generations. Rank select and tournament select performed similarly, with no discernible differences between the two, other than run time (rank selection was slightly slower). As shown in figure 1, Boltzmann consistently performed better, typically getting a few more correct literals than tournament and rank select. This improvement in performance was more notable when using single crossover. It is also notable that tournament and rank selection with single crossover performed the worst in our tests. In terms of the time to completion, Boltzmann was the slowest, followed by rank select, and then tournament select. This makes since given the computational simplicity of tournament selection. Boltzmann performed slower than rank select, likely because of the calculations based on Euler's number at every iteration.

Uniform crossover performed better than single crossover across the board. For each selection type, it worked best when paired with Boltzmann. The differences in performance between crossover types were of a slightly greater magnitude than differences in selection method. That said, single crossover was a faster method of crossover, performing consistently, if not significantly, faster than uniform crossover. On the whole, this suggests that Boltzmann combined with uniform crossover performs the best. These findings suggest a trade-off between optimality and performance where more computationally expensive parameter options like uniform crossover and Boltzmann selection yield better results than the computationally simpler options.

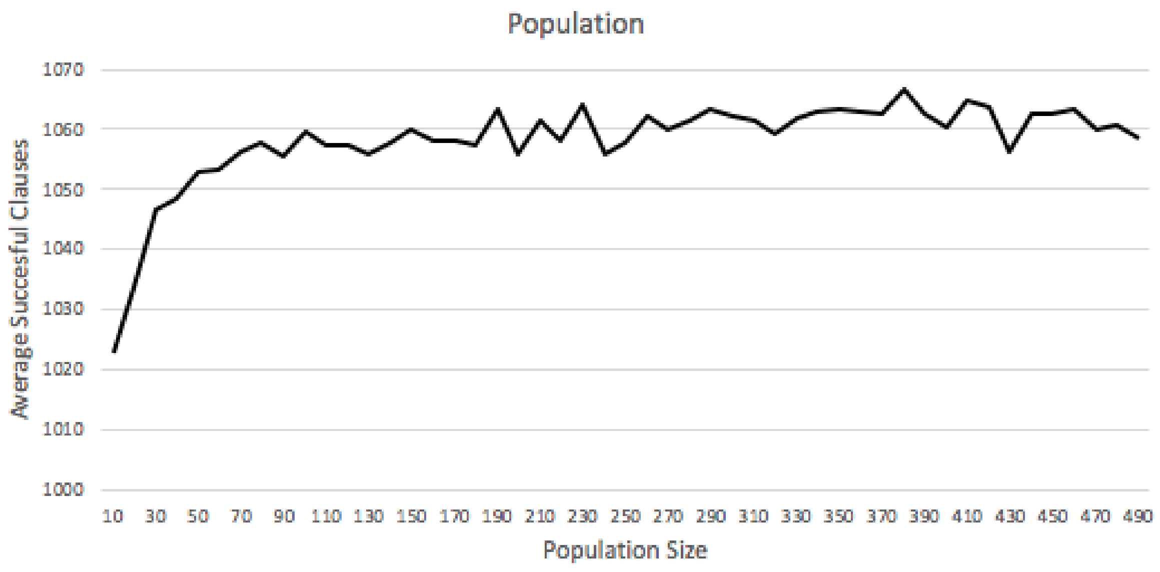

Figure 2: Population Size and GA Performance

7.1.2 Population Size

To evaluate the effect of the population size, we tested our GA on populations from 10 to 500. Increasing the population size up to 100 showed clear benefits in the quality of the solutions, but beyond that, there was little improvement. This falls in line with rule-of-thumb that a population size should be around 100. Furthermore, increasing the population size significantly increased the run time, so it's likely a population of 100 can combine the benefits of optimality and speed. This could be dependent on the problem, as ours we were able to get a large subset of the solution space with just 100 individuals. However, it's conceivable that with a larger solution space with more variables, you would want a larger population.

7.1.3 Mutation Probability

To evaluate the effect of changing mutation probabilities on the GA, we tested a range of mutation probabilities from 0 to .04 (increments of .002). Varying mutation between within this range had impacts on run-time and average solutions. As shown in figure 4, a mutation value of .08 performed the best, but this is consistent with the rule-of-thumb that a mutation value of .01 is effective. It is also clear that as mutation values get smaller than .004 and larger than .012, the GA will perform worse. It is not surprising that higher mutation rates led to worse solutions over 100 generations since increasing mutation rate increases exploration. Additionally, beyond high mutations values leading to low average solution values, higher mutation rates lead to longer run times. For example, for a mutation rate of 0, the average run-time was 4.07 seconds across out test files while the average run-time for a .04 chance was 5.29 seconds. This is intuitive since higher a higher mutation values results in more computation as the individuals are modified. Overall, we found that mutation values around .01 were best. It is likely that optimal mutation values would be a ratio of the population size. Testing how the population size and the mutation probability interact is a good opportunity for future work. [b]

7.1.4 Crossover Rate

We tested the effect of crossover rates on the effectiveness of our GA with the probability of crossover ranging from .1 to 1. In terms of effectiveness, increases in crossover probability led to increases in clause success. However, the crossover rate might not have improved the solution beyond around .9. Our function continued to have higher performance at .95 than at any other rate, but diminished when it went to 100 percent, at least within 100 generations. As far as run-time goes, changing the crossover rate had a significant impact on the run time of the genetic algorithm. At .1, the algorithm took a mere 1.15 seconds on av-

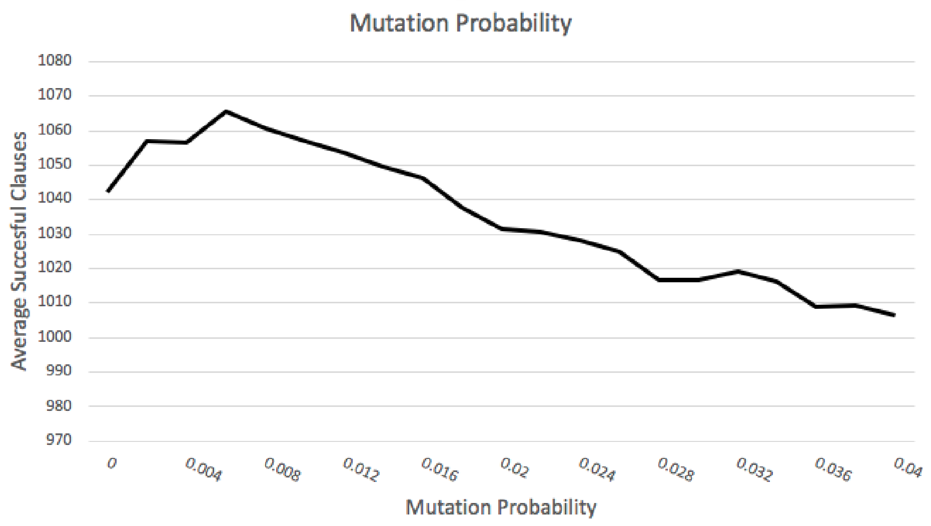

Figure 3: Mutation Probability and Performance.

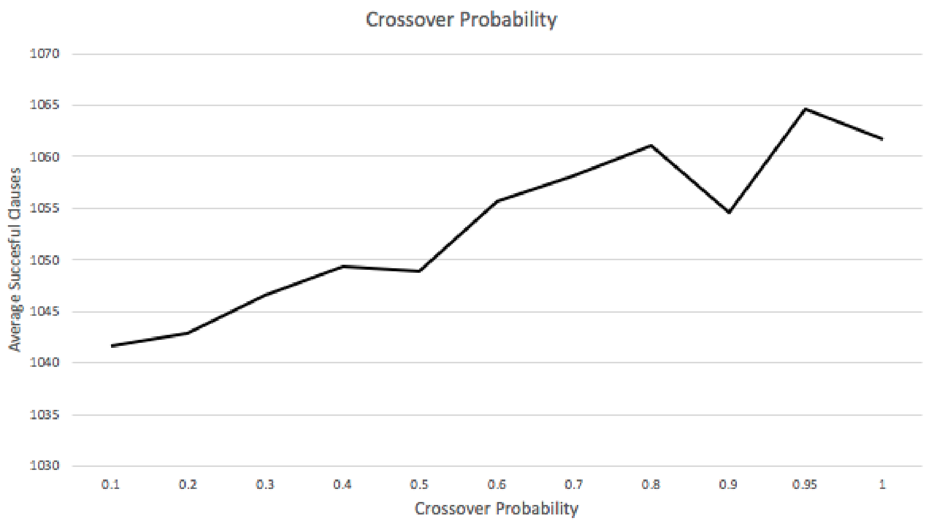

Figure 4: Crossover Probability and Performance.

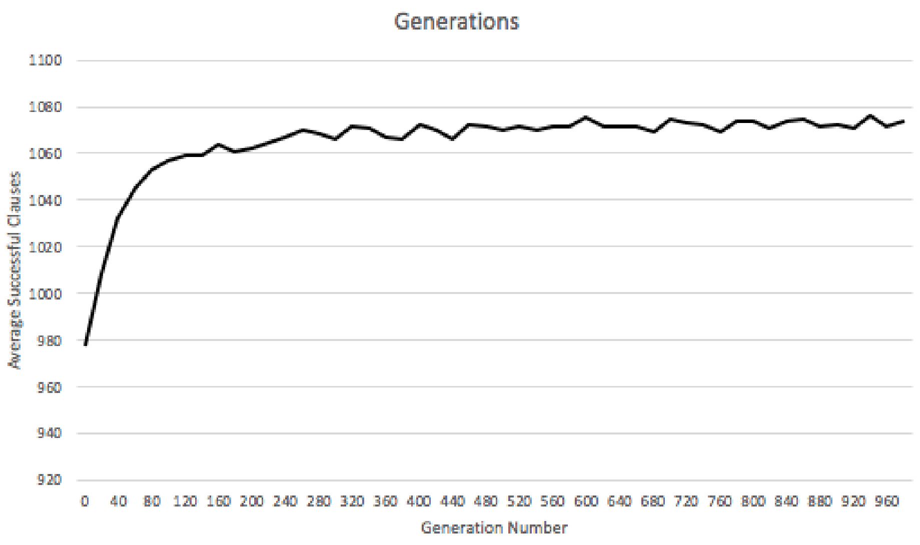

Figure 5: Number of Generations and Performance.

erage across the test files, but took close to six seconds when the crossover rate was 100 percent. This broadly falls in line with the idea that performance is best from .6 to .9. However, it is possible that increasing it beyond that could be helpful, at least with some solutions and a small amount of generations.

7.1.5 Number of Generations

The number of generations appeared to have an effect similar to population size on performance. Larger generations increased the success of the algorithm, but with diminishing returns. The returns of increasing the generation tapered off from 100 generations to 200 generations, meaning this was typically when the individuals would converge to a local maxima in the search space. Still, beyond this point, there was typically modest gains in continuing to run the algorithm. This shows that while increasing the generations always can help the algorithm, the amount that it improves it will continually diminish. Figure 4 shows that increasing the population to values in the 100s will increase performance with the increase in performance becoming more moderate through around 250 generations. Running for more than 300 generations did not appear to have a meaning impact on the average solutions.

7.2 Optimality of Solutions

To determine optimality of our Genetic Algorithm, we tested it on 10 different problems, three times each. All three of these had an optimality score of 167, meaning that the best solution would get 1093 clauses successfully. For parameters, we used a mutation score of .01, a crossover rate of .9, a 100 individual population, and uniform crossover combined with Boltzmann. We ran this for 800 generations to give it time to further optimize. We measured our algorithms against the optimal solution by comparing our solutions to the optimal as a percentage. Our best solution we garnered out of these 30 run times was 1082, or 99 percent of the optimal solution. A histogram of the results can be seen in figure 6. The average of these thirty runs was 1071, or 98 percent of an optimal solution, and the Standard Deviation was about four clauses, or about a third of a percent of the total optimality. In terms of time, each solution took about forty seconds, largely because we were running it for 800 generations, rather than the 100 generations we would run in the tests.

7.3 PBIL

7.3.1 Optimization of Parameters

After testing a wide range of parameters, we chose to continue testing with a population size of 130, 1 individual included in population vector adjustments, a learning rate of 0.17, a mutation rate

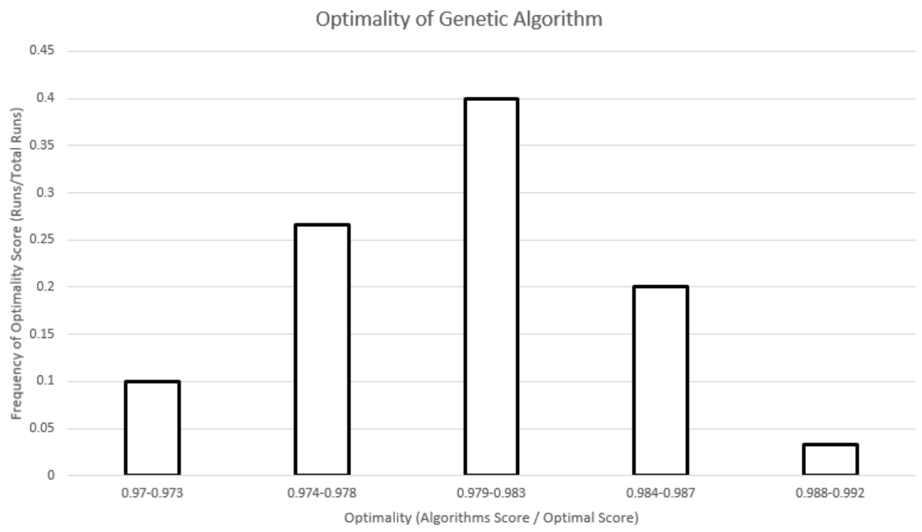

Figure 6: Optimality of GA Solutions.

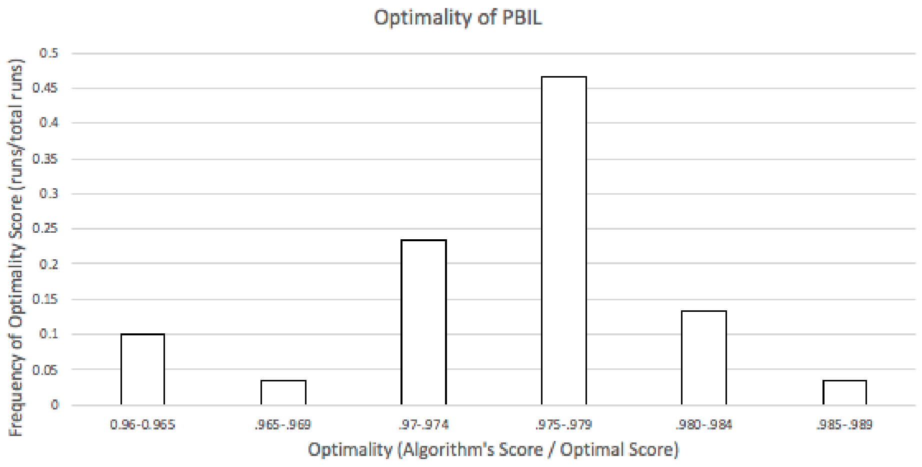

Figure 7: Optimality of PBIL Solutions.

of 0.15, a mutational shift of 0.11, and a run time of 1000 generations. These parameters were found to provide optimal answers without excessively inflating run times. However, it should be noted that further tweaking of the parameters could result in better solutions. For example, our run times for tests are short relative to many other studies. Longer run times could almost certainly find more optimal solutions than ours, but we found there to be a ”law of diminishing returns” on increases in optimality as run-time was increased, so in the interest of running more tests we kept run-times relatively short.

7.3.2 PBIL Solution Optimality

In our tests, PBIL performed near optimally, though it never found the optimum solution. On average PBIL was able to generate solutions that were 97 percent as optimal as true optima with a negligible variance. As for variance, PBIL also had a pretty narrow variance, with the standard deviation being about two thirds of a percentage of total optimality. This means there is about twice as much variation as our Genetic Algorithm, but still not very much. This indicated that while PBIL was not able to find perfect optima, it was able to find relatively decent optima consistently across problems.

7.4 GAs vs. PBIL Recommendation for MAXSAT

On test problems PBIL performed marginally more poorly than the GA, typically coming within three percentage points of true optima rather than one percentage point as observed in GAs. Nevertheless, both approaches provided decent approximations of true optima, particularly when the relatively short run-times used for experiments are taken into account. In solutions where peak optimality estimation is extremely important, GA approaches are better than PBIL.

However, GAs have numerous qualitative drawbacks relative to PBIL. First of all, implementing a GA requires a great deal more coding work than implementing PBIL. This was reflected in the length of our algorithms: while the code for GAs ran over 500 lines, code for PBIL ran to just over 200 lines, less than half the length. The PBIL approach is also somewhat simpler and easier to understand as it is not complicated by recombination and multiple selection types. In MAXSAT problems, PBIL does lose some optimality to GAs, but is more advantageous in terms of time invested in coding and testing. In terms of speed, our implementations of PBIL ran slower than GA by a couple seconds (about 40 seconds for GA, and around 50 seconds for PBIL on 800 generations).

Because of these tradeoffs, we tentatively provide two alternate recommendations for evolutionary approaches to MAXSAT problems. If the end goal is close approximation to the true optimum of a problem, we recommend that GAs be used as they are more likely to find the true solution. However, in cases where a close but looser approximation of MAXSAT optima is allowable or where implementation time and simplicity are important.

8 Further Work

Our project allows for many avenues of future research. We could improve both the implementations for the GA and PBIL algorithms and the experimental methodology. First, to improve our algorithms we could implement additional crossover methods such as 2-point or n-point crossover. It would be interesting to explore the effect of different n-point crossover methods on performance. Additionally, we could implement dynamically changing parameters. This idea is inspired by simulated annealing which is a technique in particle swarm optimization where the constants affecting exploration are decreased throughout the course of the algorithm. Both the GA and PBIL algorithms have constants that could be adjusted in this manner. For GA, the mutation and crossover rates could be adjusted over time. In PBIL, the learning rate and the number of best performing individuals used to adjust the probability vector could be adjusted throughout the run-time. Alternatively, exploration could periodically be increased by sporadic increases in mutation rate, helping the algorithm avoid convergence on local optima. Dynamically varying these constants in our GA and PBIL algorithms would open up additional testing in order to optimize how to change these values over time and where they should start.

Additional frontiers could be explored using inspiration from biology. One of these could be non random mating, where recombination will involve crossing over parents that are similar, dissimilar, or have higher fitness levels, depending on the goals. Another one could be population is-

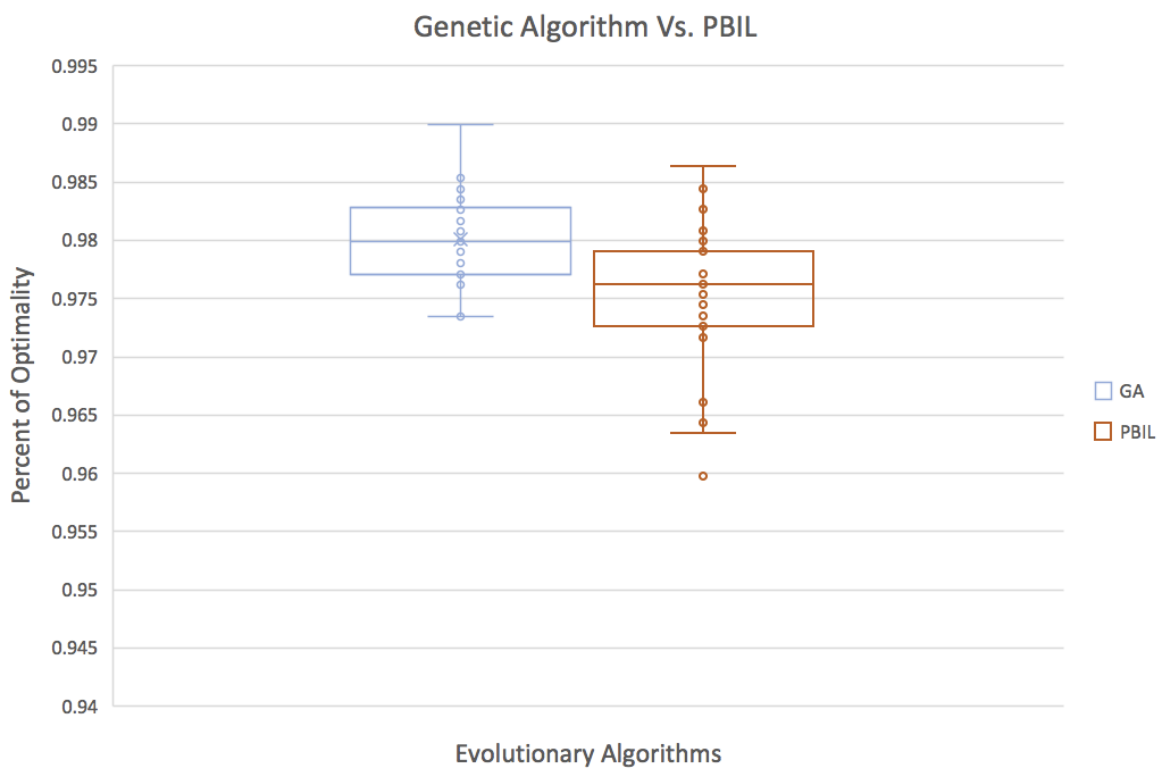

Figure 8: Genetic Algorithm and PBIL Performance

lands. This is where the different islands would have their own species and would evolve independently from other islands. This would help to explore multiple areas in the solution space at once in an attempt to avoid local minima. Another option would be to have each of the individuals hill climb around their current solution before the selection, recombination, and mutation process. Overall, there are many opportunities to improve on our algorithms and explore interesting variations.

9 Conclusion

On the whole, our evolutionary algorithms provide an efficient and near optimal method of finding solutions to MAXSAT problems. Using Genetic Algorithms and iterations of less than 100 generations, we could reliably get within a percent or two of the optimal solution on test problems, and at best were able to get within one half of one percent of the optimal solution. This demonstrates that Genetic Algorithms could provide close to optimal solutions very quickly. PBIL performed similarly, also coming within a few percentage points of the optimal solution. However, our implementation of PBIL was slightly less effective, and took slightly longer to run. This suggests, that at least for the purposes of optimization, Genetic Algorithms work better than PBIL. PBIL may be useful in scenarios where absolute optimality is less of a concern as it is easier and quicker to implement. However, given the relatively intuitive nature of both algorithms, this is largely a moot point.