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

Ethan Zhou, Jake Adicoff, and Mac Groves

**Abstract:** In this paper we aim to explore differences in two algorithms implemented to solve MAXSAT problems. The algorithms we have implemented are a Genetic Algorithm and a Population Based Incremental Learning algorithm. We designed tests to measure performance differences in the two algorithms, noting runtime and quality of the solution produced by the two implementations. These tests were constructed to answer the question: Which algorithm is better suited for solving very large MAXSAT problems? From our testing, we have determined that Population Based Incremental Learning outperforms the Genetic Algorithm in both time and solution quality.

## 1. Introduction

MAXSAT is a type of satisfiability problem that consists of *variables* and *clauses*. Clauses are a conjunction of instances of variables, or *literals*, and are *satisfied* when at least one of the literals evaluate to true. Given a set of variables and a set of clauses, the MAXSAT problem asks what an optimal variable assignment is such that the most clauses are satisfied. This is a NP-Hard problem, and therefore (probably) does not have a polynomial time solution.

Genetic Algorithms are a nature-inspired optimization method that operate on symbol strings to find the highest scoring string based on a fitness function. The symbol strings are possible solutions to a problem, such as variable assignments for a MAXSAT problem. Typically, a population of symbol strings is randomly generated. Then the population is *evolved* by breeding individuals together. In this breeding phase, a new symbol string is generating via parent symbol string crossover, and random mutation. By strategically weighting high-fitness individuals from the population for breeding, the Genetic Algorithm seeks to improve the population every generation and, eventually, converge on the optimal symbol string.

Population-Based Incremental Learning (PBIL) algorithms share some similarities with Genetic Algorithms. PBIL also optimizes bit strings based on a fitness function, except a *probability vector* is used in place of a population. The probability vector is a vector where each element is the probability that the bit at that index is 1. During each PBIL generation, a representative population of individual bit strings are generated bit-by-bit, using the probability vector to randomly set each bit. All of these bit-strings are then scored with a fitness function. The probability string is weighted towards the best bit-string(s) in the population, and may also be weighted away from the worst bit-string. Individual probabilities can be mutated by randomly adjusting the probabilities by a small value. Eventually, the probability vector should converge on the values of the optimal bit-string.

In section 2, we describe the MAXSAT problem and our approach to solving it with nature-inspired optimization algorithms. In sections 3 and 4, we give a more detailed description of Genetic Algorithms and PBIL respectively. In section 5, we explain the experimental methods used in this study. In section 6, we describe and analyze the results of our experiments and we discuss the two algorithms used in this study. In section 7, we suggest further research in the application of these nature-inspired algorithms on the MAXSAT problem. In section 8, we conclude the study with a summary of our findings.

## 2. MAXSAT

As described in section 1, the MAXSAT problem consists of a conjunction of clauses that are each composed of a disjunction of literals. A literal is a boolean variable or its negation. For example, if the variable $\mathbf{x_3}$ set to FALSE, and the literal that appears in a clause is $\neg\mathbf{x_3}$, then the

literal evaluates to TRUE. Because a clause is a conjunction of literals, if any literal in a clause is true, the entire clause is satisfied. The solution to a MAXSAT problem is the variable assignment that satisfies the maximum number of clauses.

The MAXSAT problem is a good subject for nature-inspired optimization techniques because it is NP-hard, and the naive solution is expensive. In order to test every possible combination of boolean variables, $2^n$ iterations would be required, where n is the number of variables in the MAXSAT problem. Using a nature-inspired algorithm to find a solution to the MAXSAT problem may converge on a relatively good solution in a shorter amount of time than the naive approach.

Both PBIL and Genetic Algorithms require a fitness function to evaluate candidate solutions. For the MAXSAT problem, the fitness of a candidate variable assignment is simply the number of clauses satisfied. Instead of evaluating every single combination of variables, these algorithms search the solution space with an *exploration* and *exploitation* phase. Exploration transforms the population in a random fashion, to avoid fixating on local minima. Exploitation moves the population towards higher fitness, or local maxima. PBIL and Genetic Algorithms have different exploitation methods, but both achieve exploration through random mutation and Genetic Algorithms achieves additional exploration through crossover. By adjusting the parameters of the algorithm, the balance of exploration versus exploitation can be tuned.

## 3. Genetic Algorithms

Genetic Algorithms (GA) are inspired by the effects of natural selection and mutation, found in genetics. A *population* is made up of *individuals*, which are strings that represent a candidate solution in the search space. The individuals are capable of *breeding* with each other and mutating. Each individual has a different probability of entering the breeding pool, based on how fit the candidate solution is. Fitness is evaluated with a *fitness function*, which is hugely important in the operation of a Genetic Algorithm. The better the fitness function, the higher the quality of the final solution. In the breeding pool, two individuals are used to produce children that have traits (characters from the strings) from both parents. These children are also subject to small random mutations that induce exploration of the solution space. The children thus produced constitute create the next generation of the population, and this process is repeated to ideally converge on the optimal individual in the solution space. The details of the GA are best described in phases.

Initial population generation: A population of a user-specified size (usually 50-100 individuals) is generated randomly. The strings should be designed to be constrained in such a way that each string is a valid representation of a possible solution to the problem. For MAXSAT, an individual is simply an array of boolean variables which are (initially) assigned a random true/false value. The randomness allows for breadth in the exploration of a solution space.

Evaluation: Here, individuals are tested with the fitness function to determine the strength of each individual, or solution, in the population. The fitness of an individual allows the selection phase (described next) to preferentially pick more fit individuals to be bred. Therefore, the fitness function defines the topography of the search space, and it is important that the fitness

function properly reflects the problem. For MAXSAT, the fitness function can simply be the number of clauses that are satisfied by the variable assignment specified by an individual.

Selection: Individuals are chosen to enter the breeding pool based on some heuristic. We have implemented three heuristics in our code: Ranked Selection (RS), Tournament Selection (TS), and Boltzmann Selection (BS).

In RS, for a population of $n$ individuals, the individuals are sorted in an array by fitness from least fit to most fit. Then, each individual is selected to enter the breeding pool with probability

$$\frac{i}{\sum_{j=1}^{n} j}$$

Where $i$ is the rank of the individual's fitness ($i$ can be thought of the index of the individual in a array sorted low to high by fitness). The denominator simply normalizes the probabilities. Individuals can be selected more than once to enter the breeding pool, so this type of selection allows more fit individuals to preferentially enter the breeding pool.

In TS, two individuals are chosen at random from the population and the more fit of the two is selected to enter the breeding pool. This type of selection has a significant drawbacks - low fitness individuals may be selected if two individuals of very low fitness enter a tournament

In the final selection heuristic, BS, individuals are chosen with probability

$$\frac{e^{f_i}}{\sum_{i=1}^{n} e^{f_i}}$$

Where $f_i$ is the fitness of individual $i$ in the sorted array. The denominator again serves to normalize the probabilities. This type of selection, like RS, allows fitter individuals to have a higher probability of entering the breeding pool. However, if there is a large variance in the fitnesses among the population, BS will favor individuals with disproportionately high fitness considerably more than RS.

Breeding: The selected individuals in the breeding pool are used to generate the next generation of individuals. All individuals in the pool are paired up with others in the pool, and each pair generates children to create the next generation. The children are generated via *crossover* and then undergo *mutation*. The algorithm may also automatically promote the most fit individual from the pool to the next generation without any change. This is called *elitism*.

Crossover: The child string's characters come directly from the parents, i.e. the character in position $i$ in the child string is the same as character in position $i$ of one of the parent strings. A *crossover probability* parameter is provided by the user to set the chance of crossover occurring. If crossover is not performed, then one or both of the parents enter the next generation unaltered. There are many crossover methods; the methods covered in this study are 1-point crossover and uniform crossover.

In 1-point crossover, an index $i$ in the symbol string is chosen randomly. The child is generated by combining the characters from indices 0 to $i$ from one parent with the characters from indices $i + 1$ to the end of the other parent. Uniform crossover also combines two parent individuals to form one child, but it traverses each character in both parent strings and selects one to pass to the child with equal probability.

Depending on how the breeding pool is constructed, crossover of two parents may generate two offspring, so that the next generation is the same size as the breeding pool. Otherwise, the breeding pool needs to be twice as large as the original population size to maintain equal population size per generation.

Mutation: Each symbol in the child individual has a random chance to be changed to a different value. The character mutation probability is usually low (~0.01) so that good symbol sequences aren't often destroyed by mutation. Mutation enables the Genetic Algorithm to explore new symbol assignments that may not exist in the original population, since crossover can only generate children with characters that already exist in the population. A low mutation probability will preference exploitation while a high mutation probability will preference exploration.

After the breeding phase is completed, the Genetic Algorithm has a new population of individuals. The algorithm can then either stop and return the most fit individual it has found across all generations, or it can continue the process of evaluation, selection, and breeding to continue to search for better solutions.

## 4. Population Based Incremental Learning

PBIL is similar to the GA method, as it has conceptually similar phases. As such, certain parts of the PBIL method will be covered in less detail. There is no explicit population in PBIL as there is in GA, but information about the population is encoded in the *probability vector*. The vector is the same length as a bit string, and the bit string encodes a solution to the problem at hand. Each element of the probability vector represents the probability that that element of the bit

string will be 1. Using this vector, different individuals can be generated to create a population whose bit strings will reflect the probability vector. For example, the vector:

```
< 1.0, 0.7, 0.2, 0.5, 0.0 >
```

Could generate the population:

```
< 1, 1, 1, 1, 0 >
< 1, 1, 0, 1, 0 >
< 1, 0, 0, 1, 0 > ...
```

Probability Vector Generation: The length of the vector is decided based on the parameters of the problem, and each element is initialized to 0.5. A population generated from this initial vector would consist of all random bit strings. The randomness allows for breadth in the exploration of a solution space. For the MAXSAT problem, each bit in an individual's bit string is interpreted as a boolean value for a corresponding variable in the problem.

Population Generation: A fixed number of individuals are generated using the vector, as specified above. This procedurally generated population serves a similar purpose to that of the GA. Unlike GA, a new population is generated in each iteration of PBIL, and is discarded at the end of the PBIL iteration.

Evaluation: Each member of the generated population is evaluated using a fitness function, exactly in the same way as GA. The fitness function for the MAXSAT problem in PBIL is the same as that used for GA.

Probability Vector Update: The most fit and least fit individuals of the population are selected. A user specified positive learning rate (*PLR*) and a negative learning rate (*NLR*) are used in concurrence with the most fit and least fit individuals to make changes to the probability vector, to exploit high fitness areas of the solution space. The probability vector is updated to be

more likely to generate the most fit individual, and less likely to generate the least fit individual, based on the specified rates, in the following manner:

Let the most fit individual's bit string be *mostFit* and the least fit individual *leastFit*. For each index in probability vector *P*:

$$P[i] = P[i] \cdot (1 - PLR) + mostFit[i] \cdot PLR$$

$$P[i] = P[i] \cdot (1 - NLR) + mostFit[i] \cdot NLR$$

Conceptually, these equations take each value of the probability vector and change it very slightly in the direction of the the most fit solution. For example, if P[i] = 0.75 and PLR = .01, and if mostFit[i] is **0**, P[i] will be updated to:

$$0.75 \cdot (0.99) + 0 \cdot (0.01) = \mathbf{0.7425}$$

Otherwise, if mostFit[i] is **1**, P[i] will be updated to:

$$0.75 \cdot (0.99) + 1 \cdot (0.01) = \mathbf{0.7525}$$

If the best and worst individuals are different at a certain index, the algorithm will use the second equation to further bump the probability vector towards the most fit solution at that position. After the update is completed, the representative population is discarded.

Mutation: Finally, the probability vector is randomly changed based on the *mutation probability* and *mutation shift* parameters. Every probability in the vector has a chance to be mutated, which entails randomly increasing or decreasing by the shift value. It is best to keep both of these values low (on the order of magnitude of 0.01 probability of mutation chance and magnitude of shift) so that not too much randomness is introduced. If these values are too large, the random noise introduced could negate the effect of the Probability Vector Update in the previous step.

When this process is repeated continually, all the values in the vector will converge towards 1.0, 0.5, or 0.0. The 1 and 0 values signify that the optimal bit string has a 1 or 0 in that position, and a 0.5 signifies that that bit does not have an effect on the optimal solution. Once enough iterations are run and/or the values have converged to within a certain distance from these final values, the algorithm can terminate. The algorithm returns the probability vector and/or an individual created from rounding all the elements of the probability vector to the nearest integer, 0 or 1.

## 5. Experimental Methodology

In order to asses the performance of the two algorithms, we chose to do our primary testing on large, industrial scale MAXSAT problems. We wanted to measure performance differences on problems that are difficult to solve with a deterministic algorithm. SAT-solvers can easily produce the optimal satisfying assignment for small problems, so it is less useful to implement GA or PBIL to solve small scale problems. On larger problems, a deterministic algorithm would still produce an optimal assignment, but the time cost would be large. Probabilistic algorithms become more useful here, as they can find a solution that is close to optimal in shorter time.

We designed our experiments in such that each algorithm had the best chance at producing a good satisfying assignment. To do this, we used a number of small files to tune the parameters of each algorithm in order to find optimal parameter settings. The small files had less than 200 variables and 2000 clauses. In both algorithms, increasing the number of generations and the population size produce better solutions. Therefore, we chose to keep these two

parameter settings constant, using 500 generations and 100 individuals throughout our testing. We did not do this calibration on larger problems because runtime for solving those problems is on the order of minutes, and optimizing just a single parameter may take a dozen executions. We determined the input parameters that allowed highest average satisfaction over many tests on a small selection of MAXSAT problems. For GA, these parameters were:

| | |
|---|---|
| Selection type: | Ranked Selection |
| Crossover type: | Uniform crossover |
| Crossover probability: | 0.93 |
| Mutation probability: | 0.02 |

For PBIL, the optimal parameters were:

| | |
|---|---|
| Positive learning rate: | 0.1 |
| Negative learning rate: | 0.075 |
| Mutation probability: | 0.05 |
| Mutation amount: | 0.05 |

It is worth noting that we did not consider Boltzmann Selection for our GA, because the fitness values could get arbitrarily and excessively large for the MAXSAT problem. Given that our fitness scores increased proportionally with the number of clauses in the problem, and that the Boltzmann Selection requires calculating *e*^(fitness), this selection algorithm overflows floating point values. Also, changing the parameters appeared to have a larger effect on the outcome of the GA algorithm, while making changes of similar magnitude did not affect the PBIL algorithm as much.

We made the assumption that the optimal parameters for these smaller problems would be appropriate for the larger problems. We downloaded large MAXSAT problems from the "industrial" problems found on:

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

Each problem we selected had between 100,000 and 400,000 clauses. For each problem, we ran each algorithm three times and recorded the average runtime, percent of clauses satisfied, and the generation at which the algorithm found the fittest individual. The performances of the Genetic Algorithm and PBIL were then compared for each problem in order to draw conclusions regarding the efficacy of each algorithm.

## 6. Results

The results of our industrial MAXSAT tests are recorded in Table 1. For every problem, PBIL consistently had a faster runtime and a larger percent of satisfied clauses. The difference in the runtimes was significant. On average, PBIL was 33.75% faster than the Genetic Algorithm to perform 500 generations. PBIL also consistently satisfied an average of 1.2% more clauses than the Genetic Algorithm. This 1.2% translates to about 3000-4000 more satisfied clauses. The Genetic Algorithm was never faster nor better at finding a good solution than PBIL.

Both time and percent of clauses satisfied measurements displayed very little variance among the three measurements, with Relative Standard Deviation (standard deviation/mean) around 0.33% variance in time, and around 0.026% variance in percent satisfied. These very low variance values suggest that the values we measured are well representative of each algorithm's performance, and are unlikely to be outliers.

Furthermore, PBIL tended to find its fittest individual in a later generation than the Genetic Algorithm; PBIL consistently found a best individual past the 400th generation, while the Genetic Algorithm usually took fewer than 350 generations to find its best individual. This

suggests that GA converges upon a local maxima sooner than PBIL does, or takes more time to escape local maxima, both of which are undesired behaviors.

The differences in fittest individual generation displayed significantly larger relative variance in three trials. PBIL was not universally better in this measure as well, as the Genetic Algorithm would sometimes discover its best individual after PBIL does. However, the average Genetic Algorithm best individual discovery generation was 290 while the average PBIL discovery generation was 440. This indicates that PBIL is still improving at later generations, while the Genetic Algorithm gets stuck on a single individual with medium fitness. If the algorithms were given more time to run, PBIL would probably continue to incrementally improve, while the Genetic Algorithm may be more likely to get caught in a local maximum.

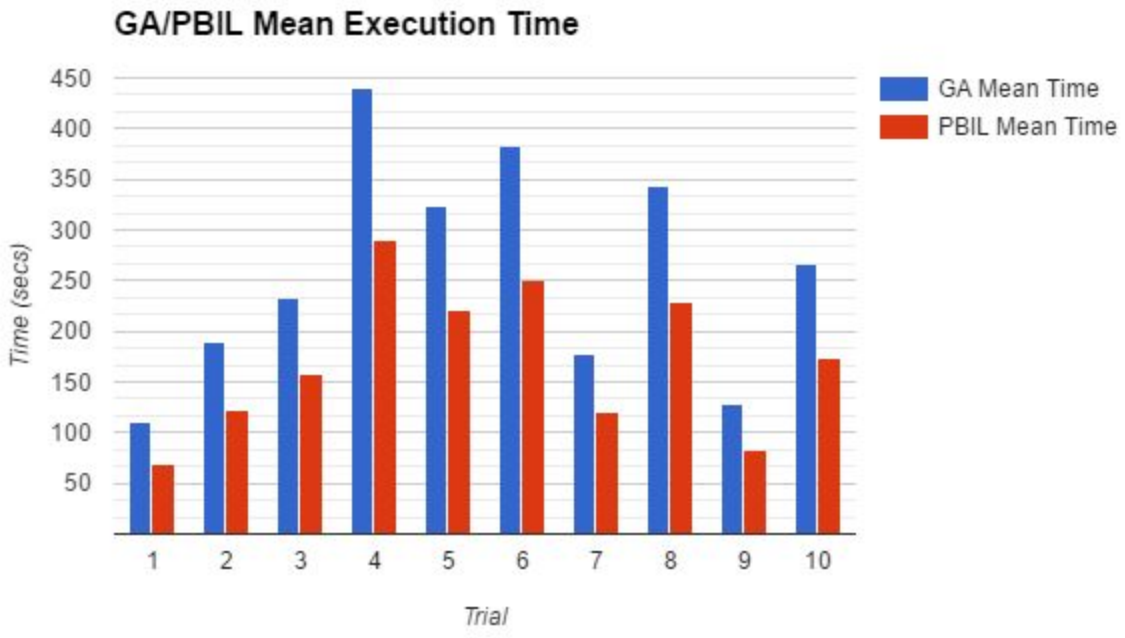


Figure 1: Mean execution time for 500 generations between our Genetic Algorithm and PBIL Algorithm, taken over three measurements per algorithm, per trial. The average relative standard deviation (Standard Deviation / Mean) between the three measurements was about .0035.

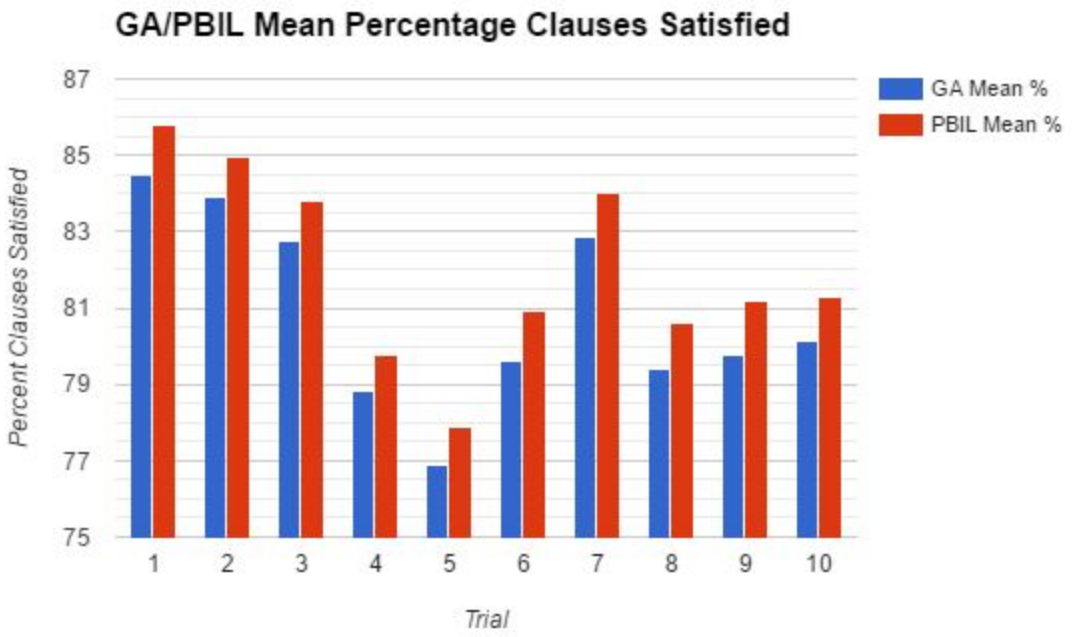


Figure 2: Mean percent clauses satisfied over 500 generations between our Genetic Algorithm and PBIL Algorithm, taken over three measurements per algorithm, per trial. The average relative standard deviation (Standard Deviation / Mean) between the measurements is .00026.

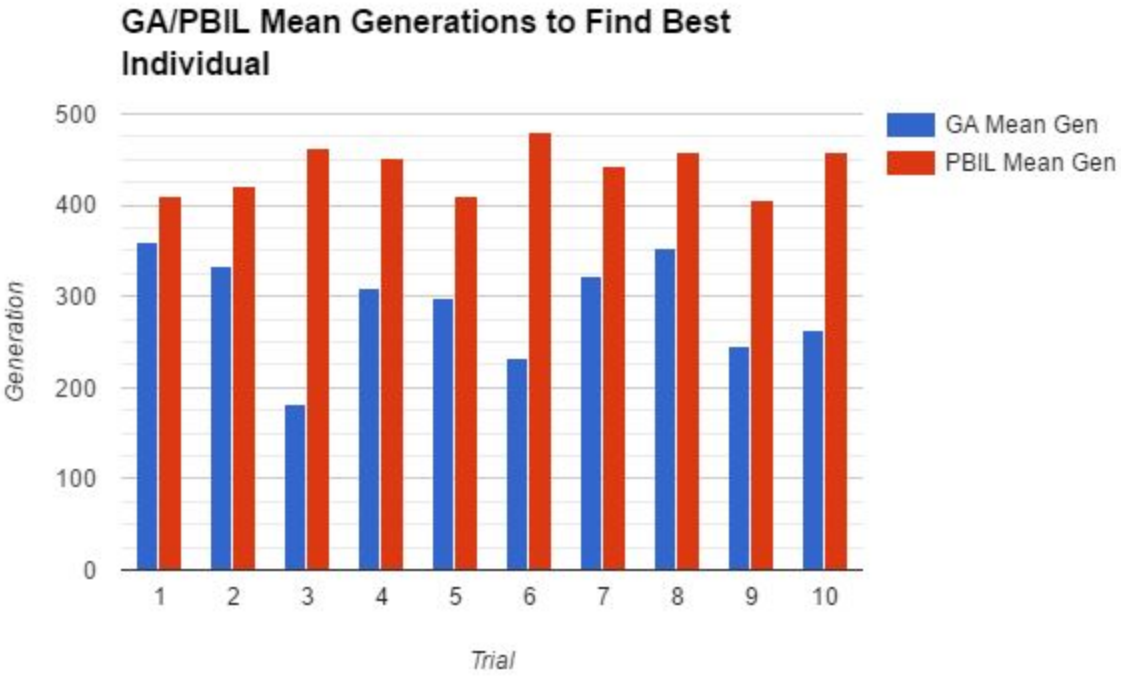


Figure 3: Mean generations taken to find the best individual over 500 generations between our Genetic Algorithm and PBIL Algorithm, taken over three measurements per algorithm, per trial. The average relative standard deviation (Standard Deviation / Mean) between the GA measurements is .40723, while the average RSD for PBIL is .08562.

## 7. Discussion

PBIL terminates sooner (34% faster on average) than the Genetic Algorithm because it is much less computationally intensive. Both algorithms generate populations of individuals and score their fitness. However, PBIL only has to update the probability vector with the best and worst individual and add some random mutation [O(n) operation]. On the other hand, the Genetic Algorithm must select a breeding pool using a rank sorting [O(n lg n)], and perform crossover and mutation in order to generate the next generation of individuals [each O(n)]. This explains why PBIL is so much faster than the Genetic Algorithm.

Furthermore, PBIL always satisfied a larger percent of the clauses than the Genetic Algorithm, which indicates that it is better at solving MAXSAT problems. It is worth noting that we only tested one set of parameters that were optimized on small problems, so it is possible that optimizing the parameters on these large inputs could draw the two algorithms' performances closer. However, as discussed in Section 5, optimizing parameters for large problems is not a trivial task, and PBIL seems to generate good results for a larger range of parameters than the Genetic Algorithm could. While we cannot determine for sure whether or not PBIL is always better than Genetic Algorithms, it is certainly easier to set up to produce good results.L

Looking first at the best individual discovery generation, it appears that PBIL continues to effectively search the solution space right up until the final generation. On the other hand, the Genetic Algorithm tends to converge earlier. This suggests that the Genetic Algorithms are more likely to get stuck at local maxima and converge prematurely. The premature convergence could be a result of the selection method and the dependence on a population of individuals. The Genetic Algorithm selection methods give preference to the fitter individuals, so the fittest

individuals in the initial population could dominate the population. This would inhibit diversity in the population and make it harder to explore by crossover, meaning mutation is the only significant means of exploration left. PBIL does not have quite the same issue because its population is regenerated every generation. This enables PBIL have a higher degree of exploration, especially in earlier generations. PBIL is not restricted to the genetic material from the first generation, and it will be able to generate different individuals as long as the probability vector does not converge too soon.

The Genetic Algorithm's dependence on the initial population of individuals also gives it some limitations. The algorithm will have trouble exploring if it gets initialized with poor fitness individuals, or individuals with similar genetic material. The ability of PBIL to continuously regenerate its population and a low learning rate enables the algorithm to explore more of the solution space early in the iterations. It is likely that the Genetic Algorithm's population is quickly filled with individuals with similar genetic material because the fittest individuals are most likely to breed. On the other hand, PBIL's representative population may take longer to become homogenous, and it appears to be able to escape these local maxima more readily.

## 8. Further Work

Given more time, we would consider 2 questions. First, why does GA underperform compared to PBIL, and can the margin of difference in runtime be reduced? There are ways in which we could have better implemented our GA code to run faster. Since sorting the population for Raked Selection is an O(n lg n) operation, we have identified selection as a potential area to improve the Genetic Algorithm. It may be possible develop a new selection type that is neither

dependent on sorting nor reduces the quality of a selection in the same manner as Tournament Selection does.

Second, we would like to know why the Genetic Algorithm tends to find its best individual in earlier generations than PBIL. As noted above, we believe that the Genetic Algorithm finds local maxima more quickly than PBIL and eventually converges on these areas and does not find a global maximum. This may have been a result of our experimental design. We optimized the algorithm's parameters on a few small problems, and never any of the industrial-sized problems, as this would have been time consuming. We are now curious to know optimizing the Genetic Algorithm parameters, mutation and crossover probabilities in particular, would have produced more favorable results. It is possible that increasing these parameters that increase exploration in the Genetic Algorithm would produce better results in these larger problems.

## 9. Conclusion

Our results showed that PBIL is consistently more effective at solving industrial MAXSAT problems than the Genetic Algorithm. PBIL runs faster and satisfies more clauses, making it superior on both fronts. We believe PBIL outperforms the Genetic Algorithm because it is better at exploration, since it regenerates its representative population per generation. The Genetic Algorithm appears to get stuck at local maxima and converge prematurely. Therefore, on large scale MAXSAT problems that are too big to reasonably solve with deterministic algorithms, we recommend using PBIL with because it achieves can satisfied clauses in less time than the Genetic Algorithm.

## 10. Appendix

A) Files Tested

These files were downloaded from the following website: http://www.maxsat.udl.cat/09/index.php?disp=submitted-benchmarks, under the "industrial" section of the Unweighted MAXSAT problems. The number corresponds to the "File" number in the figures included in the Results section.

1 dividers6_hack.dimacs.filtered.cnf
2 dividers7.dimacs.filtered.cnf
3 i2c_master1.dimacs.filtered.cnf
4 c6_DD_s3_f1_e2_v1-bug-fourvec-gate-0.dimacs.seq.filtered.cnf
5 c4_DD_s3_f1_e2_v1-bug-onevec-gate-0.dimacs.seq.filtered.cnf
6 b14_opt_bug2_vec1-gate-0.dimacs.seq.filtered.cnf
7 i2c_master2.dimacs.filtered.cnf
8 b15-bug-onevec-gate-0.dimacs.seq.filtered.cnf
9 wb1.dimacs.filtered.cnf
10 c5_DD_s3_f1_e1_v1-bug-fourvec-gate-0.dimacs.seq.filtered.cnf