|  |
| --- |
| **Research on the topic Of Rectangle Free Grid Coloring** |
| A SAT/Genetic Algorithm Based Approach |
|  |
| Given an N x M grid, is it possible to color the grid using only C colors in such a way that no rectangle in the grid has four corners of the same color? For instance the colorablity of a 17x17 grid using 4 colors is unknown. It is conjectured that this grid is colorable with 4 colors. However, a brute force attempt to find a valid coloring is computationally infeasible, as there are 10^173 different possible colorings of this grid. Hence, it is necessary to devise a clever algorithm in order to solve this problem. We investigate methods using Genetic Algorithms, Satisfiability Solutions, and Hybrid approaches to determine the colorablity of a 17x17 grid. |
|  |
| **Paul Varoutsos and Michael Groh** |
| **Spring 2010** |
|  |

Table of Contents

[Introduction 3](#_Toc259971942)

[Grid Coloring Defined 4](#_Toc259971943)

[SAT-Based Approach 7](#_Toc259971944)

[The Mapping 7](#_Toc259971945)

[The Un-Mapping 14](#_Toc259971946)

[DPLL-Based Solver 19](#_Toc259971947)

[Walksat 20](#_Toc259971948)

[Validation 24](#_Toc259971949)

[Evolutionary Computation 26](#_Toc259971950)

[The Mapping 26](#_Toc259971951)

[Approach 28](#_Toc259971952)

[Hybrid approaches 30](#_Toc259971953)

[WalkSat to MiniSat 30](#_Toc259971954)

[WalkSat to GA 32](#_Toc259971955)

[Future Research 33](#_Toc259971956)

[Summary 35](#_Toc259971957)

[Appendix 36](#_Toc259971958)

[The DPLL Algorithm 36](#_Toc259971959)

[The GSAT Algorithm 37](#_Toc259971960)

[The ECJ Open Source Library 38](#_Toc259971961)

[Works Cited 41](#_Toc259971962)

# Introduction

On November 30, 2009, William Gasarch proposed a problem to the world via a post on the popular blog, Computational Complexity. Gasarch has been doing research on a problem dealing with the concept of rectangle free grid coloring. However, he has reached some difficulty when it comes to the solution of a specific grid. Gasarch has even attached a small prize if a solution is found. He states in his blog post:

“The *n* x *m* grid is *c-colorable* if there is a way to c-color the vertices of the *n* x *m* grid so that there is no rectangle with all four corners the same color…The first person to email me a 4-coloring of the 17x17 grid will win $289.00.” (Gasarch)

In addition to the problem definition, Gasarch laid out various resources for anyone who was willing to take a stab at the problem. These resources included various papers written by Gasarch and his fellow researchers, recent papers and research done by others, and the types of methods that have been used in previous attempts to solve the problem. Gasarch made one particular statement about previous attempts that caught our attention:

“SAT-solvers and IP-programs have been used but have not worked--- however, I don't think they were tried that seriously.” (Gasarch)

We decided to take this as an opportunity to further exhaust the attempt at using SAT-solvers in order to determine whether or not the 17x17 grid was colorable and, hopefully, perhaps the other three remaining unknown 4-colorable grid sizes, 17x18, 18x18, and 12x21.

# Grid Coloring Defined

In Stephen Fenner, William Gasarch, and Charles Glover’s paper titled, *Rectangle Free Coloring of Grids*, the problem is formally defined as:

(I) “A two-dimensional grid is a set *Gn*,*m*= [*n*] × [*m*]. A grid *Gn*,*m*is *c-colorable* if

there is a function *Xn*,*m* : *Gn*,*m* [*c*] such that there are no rectangles with all

four corners the same color.” (Fenner, Gasarch and Glover)

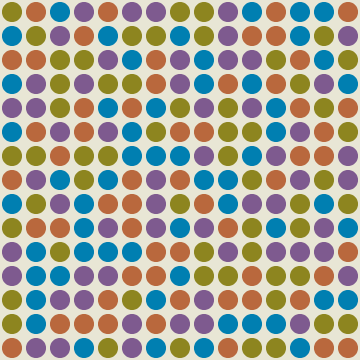


Fig 1. A 4-coloring of 15x15 (Hayes)

This means, that for any grid of size *n* x *m* and c-colors, there is a total of total possible color combinations. So, for the 17x17 grid, there are possible 4-color combinations, more than the number of picoseconds since the Big Bang (This number is 1.3 \*, still only a very small fraction of the number of grid color combinations). As a result, it becomes obvious that any type of brute force algorithm is out of the question and one needs to find a clever algorithm in order to solve this problem.

The 17x17 is of specific interest because it is a border case. All grid sizes smaller than 17x17 have known colorings. Similarly, all grid sizes larger than 18x18 have been shown mathematically to not be colorable. The goal, for Gasarch, at this point in time, is to determine what is called the obstruction set, also notated OBSc.

(II) “If you fix *c,* then OBSc is the set of all grids *Gn*,*m*such that *Gn*,*m*is not *c-colorable*

but all grids properly contained in *Gn*,*m*are c-colorable. We also call such grids *c-minimal”* (Fenner, Gasarch and Glover)

Informally, the obstruction set grids are the smallest grids known not to be colorable for a given C. The sets of OBS2 and OBS3 are all conclusive, but OBS4 is still unknown. If this set is determined, then all 4-colorings will be known as noted by the theorem above.

Another important idea to introduce at this time, as it deals with our research, is the concept of *rectangle free subsets*.

(III) A *rectangle-free subset* *A*  *Gn*,*m*is a subset that does not contain a rectangle as defined by (I). A problem that is closely related to grid-colorability is that of finding a *rectangle-free subset* of maximum cardinality. (Fenner, Gasarch and Glover)

(IV) If *Gn*,*m*is *c-colorable*, then it contains a rectangle-free subset of size .

When 4-coloring an *n* x *m* grid, a rectangle-freesubsetof note is the set containing squares using only one color without any rectangles. If we can color more than ¼ of a 4-coloring grid using the same color we can significantly narrow down the size of the problem. In fact the following conjecture has been made:

(V) “Let *n*, *m*, *c * 2. If there exists a *rectangle*-*free subset* of *Gn*,*m*of size , then *Gn*,*m*is *c-colorable*.” (Fenner, Gasarch and Glover)

This concept, known as the rectangle free conjecture, has been true for all colorable grids. Luckily for Gasarch, a rectangle free subset of size has been found for the 17x17 grid. As a result, he believes it to be colorable.

# SAT-Based Approach

The Mapping

Despite being NP-Complete, SAT-solvers, in many cases, can solve SAT problem instances very quickly. We decided to take the rectangle free grid coloring problem and map it to an instance of SAT. After this mapping is performed, we can run the instance on different types of SAT-solvers using different techniques.

The mapping technique we used was very similar to the technique used by Andrea Lobo in her paper titled, *Teaching Problem Reduction: NP-Completeness via Sudoku* (Lobo). The mapping involves three types of constraints in order to represent the grid coloring problem as an instance of SAT:

1. Each square must contain at least one color.
2. Each square must contain at most one color.
3. For each sub-rectangle in the n by m grid, no four corners should have the same color.

The general mapping algorithm for any *n* \* *m* square in a *c-coloring* problem is as follows:

We represent columns by *n*, rows by *m* and the number of colors by *c.* We also must note that *n *, *m *, *c* ** 1

We will generally derive the number of variables and number of clauses mapped from an *n x m* grid.

*The number of variables*

There are *n* \* *m* \* *c* actual variables, but in order to encode to a conjunctive normal form, or CNF, standard formula we must use modular encoding techniques. The encoding is done by considering as our base. So for this reason we have

Variables, with and . Any variable in any clause will be represented by a number in the span of [

*The number of clauses*

We must have clauses for the following cases.

*Constraint 1: Each square must contain at least one color.*

We will need 1 clause for every square in the grid so this is represented by *n \* m* clauses*.* The clauses are simply disjunctions of all assignments associated with a particular space. There are *C* of these disjunctions in each clause.

The clauses are

:

:

,

*where*

,

*and is as noted above.*

*Constraint 2: Each square must contain at most one color.*

We will need an additionalclauses for each square on the grid. If we consider the disjunction of 2 falsed variables we know that one of the two needs to be false. So we will consider the conjunction of every possible disjunction of 2 falsed variables for each square assignment. It is guaranteed that only 1 such variable can be true and then the constraint is satisfied. Therefore we will need to add *\* n \* m* clauses to our list of clauses. These clauses are

:

:

:

:

:

:

*where,*

,

*and is as noted above.*

*Constraint 3: For each sub-rectangle in the n by m grid, no four corners should have the same color.*

(VI) There are many combinations of rectangles in a given *n* x *n* grid.

Therefore for an *n* by *n* grid there are rectangles cases to consider. This is because we are considering the set of rectangles for every color in the grid.

(VI) Given an *n x m* grid the number of rectangles can be represented by the following formula.

Let and , then we have

,

rectangles for each color. So the total number of rectangle cases is

Given the aforementioned formula we must have 1 clause to check each rectangle and this gives us *R \* c* clauses the size of each clause is a constant 4 variables. This is because no matter the size of our grid a rectangle always has 4 vertexes and as long as these 4 points aren’t all true at the same time, then we are rectangle free at those coordinates. The clauses are in the general format

.

This concept of checking for every rectangle is complex in theory but we will explain this in detail below.

*Example*

For simplicity, we will describe an example mapping for a 4-coloring of a 10x10.

We will map the variables in the following manner:

The colors will be represented with the numbers 1-4, and the rows/columns will be represented with the numbers 1-10. The variables will be mapped using the digits for the color, row and column in the following manner. For example, the square at row 5, column 6, and containing color 4 will be mapped to the variable: . For a 10 by 10 grid, this will give us a total of 1104 variables, however, some variables will not be used (e.g. mapping row/column/color 0, these variables do not affect the satisfiability).

We then map the constraints in the following manner:

In order to satisfy constraint 1, we need to make sure that each square contains a color. This will produce clauses, one for each square. The structure of the clause has the following format:

The above constraint can be interpreted to mean that the grid location at row 1, column 1 must be either color 1, 2, 3, or 4.

In order to satisfy constraint 2, we need to make sure that no square is given more than one color. This will produce six clauses for each square for a total of 600 clauses. For example, this will produce,

-111 OR -112

-111 OR -113

-111 OR -114

-112 OR -113

-112 OR -114

-113 OR -114

The above set of constraints can be interpreted that the grid location at row 1, column 1, cannot be any two colors.

In order to satisfy constraint 3, we need to make sure that no rectangle contains four corners of the same color. This is where a majority of the clauses are formed. We need to check for a total ofrectangles, or 2025 rectangles. For each of these rectangles we need to check all four colors. This means we have a total of 2025 \* 4 rectangles. The clauses would look like the following:

-111 -221 -211 -121

The clause above checks for the rectangle with top left corner at (1,1) and bottom right corner at (2,2).

For constraint 3, the way we map each individual rectangle’s constraint is using the property that you only need two points to create a rectangle. For example, we iterated through each location of the grid starting at (1,1). For location (i,j), we choose (i+1,j+1) to be the second point that completes our rectangle. We create this clause then move to the next grid location. We do this by increasing the j value until we reach the end of the grid. At this point, we increase the i value and, again, iterate through the j values. A pictorial example for choosing rectangles is found below.

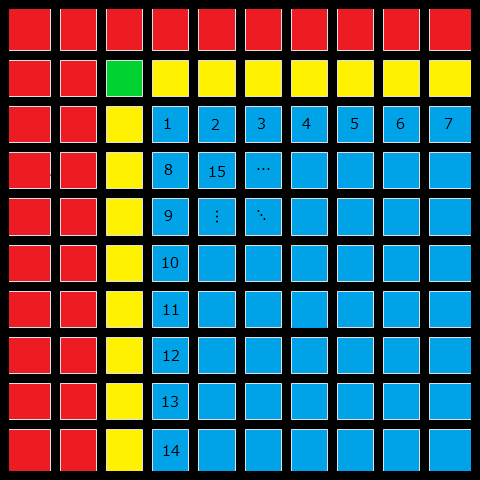


Fig 2. Choosing Rectangles

The green location represents a starting grid location, the top-left part of the rectangle we will be making. The blue area represents all of the possible second points that we can choose from to create a rectangle. The yellow area represents, depending on what second point we choose, the valid bottom left and top right sub rectangle points. The red area contains grid locations that we do not choose from as they do not create any new rectangles that have not already been created using this approach. The squares are numbered to represent the order in which the second point of our rectangles were chosen.

The Un-Mapping

The Un-mapping software will take the solution or partial solution to our problem that has been mapped to SAT, and displays the indecies in a printable grid. Knowing that an instance has been satisfied as a set of variables and clauses in a SAT-solver, we need to un-map this data back into a grid so we can determine if it is a valid rectangle free grid coloring. After this is done, we can then validate the possible solution using a validator which will be described in a later section. A SAT-solver will print out whether or not the formula can be satisfied, and if so, it will supply an assignment. These variables are going to be an encoded number using the strategy that was mentioned in the mapping section.

*The Decoding*

Remembering from the Mapping section, we know that the scope of the variables mapped are in the span [ , where both are defined in that section. This is not, unfortunately, the official assignment statement that will be given to us by a standard SAT-solver. It will return to us every variable assignment in the span [. This means we will have to exclude all the values in the span. All of these values are row/column/color combinations that do not exist in our gird.

We also need to consider the fact that our row, column, and color values are often times not equal and we need to consider this in our un-mapping. For cases where this is true, we need to discard every value that will map to a color we are not considering or a row and column combination that isn’t contained in the grid. This can be determined once the final computations are done, and we will see this at the end of our theoretical analysis.

Once we have established our notable values we can start the decoding process. The type of encoding that was done was modular, and we need to consider this thought process to decode the variables.

Consider the encoded value. First we must derive whether or not this value is true or false. Only the true variables are the ones that represent the grid color and location that we want. In a standard solution assignment a false value is denoted by a “-” sign. Therefore the span we are looking for is where all For each 𝛾 we must determine the quotient and remainder as it pertains to our base. We will name these values respectively. These 2 values will help determine where we are in the grid and what color we have at that location. We will also need a value called the correction factor, denoted 𝛿. This value will be needed in order to ensure that values that lie on boundaries will be represented in the right row and column.

*Correction Factor:*

, where we know that and are the base and the number of colors respectively from the mapping section.

*Quotient:*

*.* It is important to note that computers will be using integer division and this calculated value will be rounded down to the nearest whole number. This is important to do because we cannot have fractional indexes. This is why the Correction Factor 𝛿 is needed. It is purposely a big enough number to keep the last column of a row in the right row, but it is also purposely just small enough so that we are not throwing the first column of a row into the previous row. The value we obtain for q is our row.

*Remainder:*

This value will be used to obtain the column that we are in. We must note that this is not directly the column number. Consider the actual numerical

index for column to be,

Again this is computer integer division and will be rounded down to the nearest value. This time no correction factor is needed. Lastly we can obtain the numerical value of our color denoted 𝜍, where c is the number of colors. This will be

which is simply what we have left and will most certainly be the amount of colors under the assumption that our mapping was done correctly (which it was).

After we have obtained our row, column and color values, we must validate that they are all within the bounds that were originally set for this grid. If we are mapping an grid with colors, we need to validate that our,, and our . If they are all valid we may add this to our grid at the given indices. Otherwise these values were just generated by the Sat-solver and can be thrown out.

We perform this operation for each positive value. From these variables, we obtain a complete C-coloring of our grid. From here, we can pass the grid on to validation which again will be described in a later section.

*Example*

Let’s again consider the 10 by 10 grid in our example case. In the previous section we talked about the assignments that are produced from a mapping to SAT. Also earlier in this section we mentioned the format of a standard assignment statement. Now let us look at an example of how our Un-mapping software will handle the SAT-solver solution.

Here is an excerpt from a 10 by 10 solution file:

*-632 -633 -634 635 636 637 638 639 640 -641 642 -643 -644*

Any variable that has a “-” sign will automatically be discarded so we can now just consider

*635 636 637 638 639 640 642*

As we mentioned in our general theory above the first thing we need to do is obtain the row that each of these values are contained in. It’s easy to see that all of these are in row 6. This is because the row is obtained by dividing the number by *base2* then subtracting our correction factor. For example the first number 635/100 = 6.35. Then our correction factor is = .08. Then we have 6.35 - .08 = 6.27. This number can be rounded down to 6 to show that it is in the 6th row. This will be found true for all of these numbers.

Next we need to determine the column that we are in. This is done by taking the remainder of our first row division and dividing it by the base. Again let us demonstrate this by looking at 635. When we divide 635 by 100 we get 6 with a remainder of 35. We can see that =3.5 and the column that this is representing is 3.

The final variable we need to consider is the color of our spot we just un-mapped. This is done with another remainder function from our column computation. For example we can once again consider 635. After our row and column un-mapping we need to determine the remaining value. When we divide 35 by 10 we are going to have 5 left over. We can discard this value because 5 is not contained in the span of our colors, which only go up to 4. The values *636, 637, 638, 639,* and *640* can also be thrown out for this very reason*.*

The remaining number *642* can be decoded to the 6th row and the 4th column and the number 2 is leftover. This is a valid value and therefore can be represented in our grid as part of our solution. It can be noted that in the case of 10x10 grids with 4 colors, only 1 out of every 10 variables are actually mapped into our solution grid.

DPLL-Based Solver

After the Mapper and Un-Mapper software were completed we were able to test our SAT instance on SAT-solvers. There are two main categories of SAT-solvers: DPLL based SAT-solvers and local search based SAT-solvers. The former is discussed here, and the latter can be found in the next section.

DPLL SAT-solvers are based off of the Davis-Putnam-Logemann-Loveland algorithm. One of the main benefits of the DPLL algorithm is the fact that it is a complete algorithm. This means, given a formula, the algorithm will be able to determine whether or not the formula is or is not satisfiable in a finite amount of time. For the case of local-search based SAT solvers, if a solution does not exist, the solver will not be able to make such a conclusion.

Our initial run of problem was with a Sat-solver called zChaff on a Sun Solaris 9 machine with 32 GB RAM and 4x 1.2GHz processors run and maintained by Rowan University’s Computer Science department. zChaff is a very powerful SAT-solver, developed at Princeton Univerrsity, that implements the previously mentioned algorithm. zChaff has won awards at SAT competitions as one of the fastest available SAT-solvers. Since then, zChaff has become the result of much research and the basis of new SAT-solvers. zChaff was able to give solutions for grid sizes up to and including 14x14 almost instantly. However, for 15x15 and greater computation was stopped after 48 hours.

As a result, we decided to try a second mapping technique. Gasarch had noted that, for all known colorable grids, there existed a rectangle free subset. A rectangle free subset of size 74 exists for the 17x17 grid. We took this as a starting point and programmed these values into our SAT formula as a partially colored grid. These values will allow zChaff to perform unit propagation on 74 variables in the formula, hence significantly reducing the number of clauses that need to be satisfied. We tried the formula with the newly added constraints, and again, the computation was stopped after 48 hours.

We decided that perhaps the problem just needed more computation time on a more powerful machine. We ran the above 17x17 SAT instance with the rectangle free subset on a Windows XP machine with 4GB RAM and quad-core 3.0GHz processor. The solver ran for over a week, with no result.

As a result of our previous runs we decided to look into other DPLL based SAT-solvers. After much experimentation and comparisons of runtimes on smaller grid sizes, we determined that the most efficient SAT-solver for this problem instance was Minisat. Minisat is another award winning, competition-grade DPLL based SAT-solver.

Minisat was able to outperform zChaff in all grid sizes. Minisat was also able to solve 15x15, which zChaff was unable to do. The 17x17 grid was reattempted using this solver on the above Windows XP machine, but failed to determine a solution after running for over one month.

Walksat

The second type of SAT-solver we used is a local search based SAT-solver. These types of SAT-solvers are not deterministic as with DPLL, so they will never be able to determine that a solution cannot be found. Local search SAT-solvers use a more probabilistic approach than DPLL solvers. GSAT, a popular greedy local search algorithm works in the following manner: Start with a randomly-generated assignment and then repeatedly flip the assignment of variables that lead the greatest decrease in the number of unsatisfied clauses (Selman, Kautz and Cohen). The GSAT algorithm has been shown to significantly outperform backtracking-DPLL based SAT-solvers on problem instances including graph coloring (Selman, Kautz and Cohen).

This approach can be very successful at finding solutions, but can get stuck at local optima when the variables that lead to the greatest number of unsatisfied clauses lead to a dead end. There are a few ways to escape this local optima, one of which is implemented in the local search based SAT-solver, Walksat. Walksat implements a random walks strategy as described in the paper *Local Search Strategies for Satisfiability Testing* (Selman, Kautz and Cohen):

With probability p, pick a variable occurring in some

Unsatisfied clause and flip its truth assignment.

With probability 1-p, follow the standard GSAT scheme,

i.e. make the best possible local move.

Where DPLL-based solvers could only solve square grid colorings of size 15x15 and below, sometimes taking as long as 45 minutes in the case of 15x15, Walksat could solve these instances almost instantly.

Walksat began to have trouble solving 16x16 but was able to in under a minute when given specific parameters. Walksat allows several parameters to be input to customize a run. Depending on the SAT problem instance, different parameters can greatly affect the outcome of a solution. Given certain parameters, some instances may never come to a solution. However, given the correct parameters, some solutions can be found very quickly. There is an entire science behind choosing the correct parameters for a given problem.

For grid coloring problems, through scripts and much trial and error, we have narrowed down the best parameters for Walksat.

One of the most important parameters to change is the *cutoff*. Cutoff, which defaults at 100000, determines the amount of variable flips before the assignment is randomized again and the algorithm starts over. Given the number of variables for a 16x16 grid, the default cutoff is not enough to come to a solution. The cutoff should be increased to 3-400000 to allow for more variable flips.

The most important parameter however, is the heuristic for choosing the variable within the selected clause chosen from the GSAT algorithm. In the case of 16x16, the best heuristics are novelty, rnovelty, novelty+, and rnovelty+. The heuristics will be briefly described below.

Novelty considers the variables in the selected clause sorted by the number of clauses they would satisfy when flipped. If the best variable in this list is not the most recently flipped variable, it is flipped. Otherwise, it is flipped with a probability of 1-p. Otherwise, the second-best variable is flipped (Hoos).

The R-Novelty heuristic is similar to Novelty, but in the case that the best variable is the most recently flipped, the decision between the best and second best variable depend on their score difference. Also, every 100 steps, a random variable is flipped instead of the above described heuristic (Hoos).

The +heuristics are similar, however, with a probability 1-wp, the heuristic will flip a variable according to the standard mechanism, but otherwise a random walk step is performed.

In association with each of these heuristics, you can specify a noise parameter. This noise is the p variable for the GSAT algorithm described above. For grid coloring problem instances of size 15x15 and greater, a noise parameter between 10-15 percent have been found to give the best results. If using a “+” heuristic, you may also specify the noise for the wp probability.

When 17x17 is tried, the Walksat algorithm consistently satisfies all but 5~7 clauses, but fails to come any closer to a solution despite any changes in parameters and runtime.

An example run of a good Walksat command for a grid coloring instance can be found below:

./walksat.exe –novelty+ -cutoff 400000 –noise 12

# Validation

For all of our software and applications it was important to create a function that can validate the colorablity of a certain grid. The validation can be done in 2 different ways. In the first method we check the grid until we have obtained something to contradict a valid solution. In the 2nd method we can keep checking the entire grid to calculate the number of rectangles that were formed. The second way can be used to determine an overall fitness to a certain assignment when using the genetic algorithm approach. First we will focus on proving a given assignment false.

Given an n x m grid we are going to have a number of rectangles as defined above in the mapping section. In order to check for rectangles we need to check the vertices of every possible rectangle that can be created for every color used in the grid. We devised a simple brute force checking method for this. Refer back to Fig. 2 in the Mapping section for a picture of the amount of rectangles in a certain grid. Our algorithm for validation is based off this idea.

We need to devise a scheme so that no rectangle will be considered more than once. Starting with the index (1,1) and ending with the index (n-1,m-1), we will search for every possible rectangle that can be considered below and to the right of the index noted. We don’t need to include the last row and the last column in our checking because they cannot create any new rectangles.

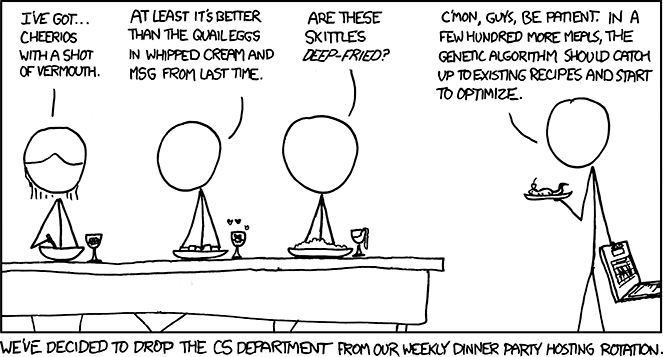
More generally, at every index , where , we start considering sub-rectangles at each index in the span of (i+1,j+1) and (n,m). Each combination of vertices will create a potential rectangle. Next we must check to see if both these vertices are both the same color. If this is the case, then we will look at the image of these 2 indices to check the other 2 points that would make this rectangle. If both the indices and are the same color, we must look at the indexes and . If both these points also yield the same color, then we have a rectangle being formed of all the same color and this coloring is invalid.

Now that we have considered the idea of a sub-rectangle and how to check for it, we can systematically check all the possible rectangles. We will use a nested algorithm that will perform the above computation using loops and conditionals.

As mentioned before there are a few ways we can use the power of this type of checking. The first is a simple check to see if an unmapped Sat-solver assignment talked about in the Un-mapping section is valid. Our unmapper software will display the rectangle obtained from SAT in a 2-dimensional array. This is the grid that will be looked at in our validator. Because we have the possibility of the transfer between the solver and our programs going wrong, we must have a means to validate the solution. If our validator goes through the entire grid and doesn’t record a single rectangle we know the solution is valid. If we find a single rectangle then we can stop and consider the validation invalid.

The 2nd method we used our validation software for has to do with the “fitness” of a given solution or partial solution when used in evolutionary computation. In this version of the validator, we will check for every possible rectangle that can be created. The given assignment’s fitness is the amount of rectangles that the grid contains. If we can compute the number of monochromatic rectangles in a grid, then we can determine how close it is to a solution. This idea is important for Evolutionary Computation which will be described in the next section.

# Evolutionary Computation



(Munroe)

The Mapping

Another approach we used in our attempts to color the 17x17 grid was to use genetic algorithms. Genetic Algorithms, or GAs, are a search technique used in computing to find exact or approximate solutions for search problems. Genetic algorithms are a subset of evolutionary algorithms which take into account concepts from evolutionary biology such as selection, mutation, and crossover.

Genetic Algorithms work in the following manner:

1. Create a population of chromosomes, or individuals (usually randomly).
2. Determine, based on a user-defined function, the fitness of each individual.
3. Select the next generation using some user-defined or preset selection method.
4. Perform reproduction using crossover.
5. With a probability, p, perform mutation on individuals in the population.
6. Replace the population of individuals with the new selection and crossover individuals.
7. Repeat steps 2 through 6, n times.
8. Print the solution(s) with the highest fitness.

For our project the genetic algorithm library we used was called ECJ. In order to use any genetic algorithm for a given problem, one must create a way to un-map a given individual to a possible solution to the problem at hand. In most genetic algorithm libraries, such as ECJ, the individuals are represented as vectors of the basic primitive types: bit, byte, double, float, integer, and long. For these vectors, one may specify the length of the vector and valid ranges for each value in the vector.

One must also specify a fitness function that calculates how close an individual is to the “optimal” solution. This is done by un-mapping the individual to the given problem instance. We can then perform any necessary calculations to determine its fitness based on how close it is to the “optimal” solution. The individual is then assigned this fitness and passed into the evolutionary computation framework.

As described above, the Genetic Algorithm library will randomly generate vectors of a specified size and within the valid ranges for each value. This will be the starting population for the first generation of the run. Using the defined fitness function, each individual is assigned a fitness which is then used in the selection and breeding processes for future generation populations.

In the case of rectangle free grid coloring, individuals are represented by a vector of integers. The size of each individual is equal to the product of the row and column sizes. For example, the 17x17 grid, the individual size would be The range of values for an individual is determined by, c, the number of colors you wish to use to color a grid. In the domain of 4-coloring, we used the integers [1,4] as the possible vector values.

Using the validation process as described above in our validation section, the fitness of each individual is calculated based on the number of monochromatic rectangles in the grid that the individual represents. Many selection strategies dictate the fitness must be a positive number so we used the following calculation:

For example, the 17x17 grid could containrectangles. An individual containing 22 monochromatic rectangles would be given a fitness of:

As a result, the fewer rectangles an individual contains, the more “fit” the individual is.

Approach

For genetic algorithms, one of the most important factors is the mutation rate. This is integral to bringing the evolution out of local optima. Through much experimentation, we found that the most effective range of mutation probabilities was somewhere between [.005, .02]. Any mutation probability greater than 2% prevented the evolution from converging to a maximum fitness level. Any mutation lower than .005% causes the evolution to get stuck at a local optima.

The graph below shows several different sized tournament selection runs over various mutation probabilities for a 17x17 coloring. Each line represents a different tournament size. The x-axis represents the mutation probabilities and the y-axis represents the highest fitness for that run. As the mutation probability increases, the fitness decreases. It should also be pointed out that larger tournament size runs lead to higher finesses as well.

Another important factor in genetic algorithms is population size. Our research found that a population size of ~500 individuals yielded the best results. When the population size is too small, the search space is reduced as there are not as many combinations being compared. When the population size is too large, there are so many individuals that the population is unable to converge to an optimal fitness level.

We also tried various selection methods all with similar results. The main selection method is called tournament selection. Tournament selection works by randomly choosing n individuals from the population, then choosing the one with the highest fitness. Another popular selection method is GreedyOverselection. In this selection method, with a probability p, the selection algorithm will choose a random individual from the top n percent of individuals. Otherwise, an individual from the bottom 1-n percent is selected.

We were able to color all square grid sizes up to and including 16x16 using this approach. However, coloring the 16x16 grid took longer with this genetic algorithm approach than the SAT-solver approach using WalkSat. However, when it comes to coloring the 17x17 grid, this approach can come within just 10 monochromatic rectangles compared to the ~19 monochromatic rectangles with WalkSat.

# Hybrid approaches

Neither of the two previously described approaches have been able to solve the rectangle free grid coloring problem. As a result, we developed a couple hybrid approaches that combine the strengths of each approach into one.

WalkSat to MiniSat

As stated above WalkSat is a local search based SAT-solver that is not complete. WalkSat by nature will not exhaust a given search space, it will only make local search decisions that look at the short term satisfiability of a formula. However, WalkSat was able to very quickly generate partial solutions. WalkSat can generate grids that are within ~19 rectangles of a fully colored 17x17grid. It can also generate many fully colored solutions to smaller grid sizes less than 17x17 very quickly. We would like to combine this strength of WalkSat with the property of completeness of a DPLL-based algorithm.

This combination of SAT-solvers was done in two ways. The first combination involved taking a close rectangle free coloring of a 17x17 grid and punching approximately 150 “holes” into it. This was done by removing colors from the grid at random. We would then hard code the remaining grid colors into our boolean formula and run the formula on MiniSat. This would allow us to exhaust all possible colorings that remained from the partial solution output by WalkSat. However, the more colors that are removed from the partial solution, the longer MiniSat takes to execute. Leaving only 125 squares colored in a 17x17 grid seemed to be the best balance between the number of holes and MiniSat execution time. We want to allow MiniSat to go through as many color combinations as possible, but still execute quickly enough to try out many different partial coloring combinations. We wrote scripts that would generate an approximate solution using WalkSat, removed random colors from the grid, then ran MiniSat on the remaining partially colored grid. This would allow us to try several hundred approximate solutions per day for several days.

A similar approach takes advantage of the fact that all colorable grids contain within themselves, smaller sized colorable grids. For example, a 4-colored 10x10 grid contains within itself a 4-colored 5x5 grid. Using a similar approach as just described, we generated solutions for small grid sizes, placed this small grid inside a 17x17 grid, and attempted to fill in the remaining rows and columns with MiniSat. The smaller the sub-grid that we place inside the 17x17 grid, the more combinations we were able to check. Again, the smaller the grid, the more execution time required for MiniSat. In this approach, we ran scripts by generating 12x12 solutions, punching approximately 25 holes into it, placing the remaining squares into a 17x17 grid, and running it in MiniSat.

This strategy was probably the most likely to find a solution, however, we were still unable solve the 17 by 17 grid. With these hybrid approaches we are not sure whether or not we came any closer than the 19 monochromatic rectangles using the WalkSat approach or the 10 monochromatic rectangles using the GA approach. In order to know how close we come with this approach, we would need to investigate Max-Sat SAT-solvers, which could be investigated in future research. These types of SAT-solvers will tell us the maximum number of clauses that can be satisfied from a given formula. So if a grid is not colorable using this approach, these SAT-solvers will at least tell us how close we can come.

WalkSat to GA

Another approach that we investigated was a combination of WalkSat and GA. We attempted to take approximate colorings containing a small amount of rectangles as described above in our previous approach and to code them into the GA approach as population individuals.

The idea here is to saturate the population with as many highly fit individuals as possible from the start and have the evolutionary computation converge to a more optimal solution than with a randomly generated initial population. Using ECJ, the individuals were given 1 at a time with varied time intervals. We allowed the library to determine the best possible coloring assignment for each hardcoded assignment. This way we could let the greedy selection strategy keep the very best mutated individuals around. Eventually we were able to obtain a grid coloring that was within 9 rectangles, beating our best by1.

This approach seemed to be promising at first. The downfall was that we were probably getting stuck in local optima for each given branch hardcoded to GA. Our hope was that, by combining multiple WalkSat partial solutions, we would be able to break out of local optima and come closer to a fully solved grid. This is not currently the case, but this Hybrid approach showed promise in that it came up with our research’s best colored grid to date. We will discuss other future research strategies in the next section.

# Future Research

During our research project, we exhausted many approaches, but in the process opened doors for some great ideas. For each approach we attempted, we found more ideas to try. Given a finite amount of time, we tried as many as we could, but there are still some ideas we believe could be promising.

Given that the WalkSat-GA approach worked the best, yielding a grid colored to within 9 rectangles, we would like to investigate in more detail how we can use this approach to come up with better solutions. There are many different ways of feeding individuals into the evolutionary computation system, which we would like to investigate further.

We would like to look into the concept of MaxSat. Max-Sat SAT-solvers determine the maximum number of clauses that can be satisfied for a given formula. We would like to use these types of solvers in order to investigate how close our WalkSat-Minisat approach gets to a fully colored grid. This solution seems like it could generate very close approximations, but we are not sure how close we can come without this type of solver. Most modern SAT-solvers will only output the result of SAT/UNSAT, not how close it came to a solution.

Another similar concept is that of MaxWeight-SAT. This type of SAT applies a weight to each clause in the formula. Perhaps using this method, we could weigh clauses using a heuristic which will yield the highest weight. For example, high weights can be applied to the clauses that dictate that each square must contain a color. After these are satisfied, the SAT-solver can work on creating assignments that reduce the number of rectangles in the grid. The problem with the WalkSat approach is that it would just try to satisfy all clauses. This sounds like a good thing, but these few clauses that remain are usually the ones that dictate that each square must contain at least one color. The problem with this is that once colors are applied to these squares, many rectangles form. So a partial WalkSat solution that came within just 3 clauses could actually contain 20+ rectangles. Using a weighted formula, we could put more emphasis that each square must contain a color. We can also look into experimentation of the heuristic that chooses the weights of each clause.

Another attempt that we did not try but was considered was a mathematical proof. Many of the other grids that were determined unsatisfiable were done so mathematically. It may be the case that there is no 17 by 17 solution and if this is true we have provided a great deal of evidence that Sat-solvers would not be able to search the entire search space in a reasonable amount of time. Therefore it would strongly be the case that general mathematics would be needed to determine this was unsatisfiable.

# Summary

As described in the specification document, we set out to attempt two different approaches to determining a 4-coloring of the 17x17 grid.

The first attempt was a mapping of the grid coloring problem to SAT. Using this approach, we were able to use DPLL based SAT-solvers to solve all grids up to and including 15x15. However, when using local search based SAT-solvers such as WalkSat, we were able to color all grids up to and including 16x16, at a much quicker rate than DPLL based solvers. When attempting to solve the 17x17 grid, WalkSat was able to come within ~19 monochromatic rectangles.

The second attempt was a mapping of the grid coloring problem to be used in a genetic algorithm approach. Using this approach, we were able to color grids up to 16x16, but at a much slower rate than WalkSat. However, when it came to 17x17, this approach was able to come within 10 monochromatic rectangles. This is a better approximation of the 17x17 solution than the WalkSat approach.

After exhausting much of the SAT and GA based approaches, we mixed together some of the main strengths of each approach together. Using a combination of WalkSat generated approximate solutions as an input to our GA approach; we were able to come within 9 monochromatic rectangles of a solution.

We tried many different approaches; however we were still unable to completely solve the 17x17 grid. We have determined that, if a solution really does exist for the 17x17 grid, using a computational approach to find it is very difficult. One must get lucky or parse out a significant part of the search space using a very clever algorithm.

# Appendix

This appendix contains background information on any concepts that deal with our research that we feel, if explained, can help a reader better understand our research.

The DPLL Algorithm

The DPLL algorithm, or Davis-Putnam-Logemann-Loveland algorithm, is an algorithm introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann, and Donald Loveland. The DPLL algorithm is an algorithm to determine the Satisfiability of a propositional logic formula (DPLL Algoritm). The formula must be in conjunctive normal form (CNF), which any propositional logic formula can be transformed into. This algorithm is the basis for many modern SAT-solvers used in the CNF-SAT problem today, including the SAT-solvers that we used in our research. Pseudocode for the algorithm is found below (DPLL Algoritm):

function DPLL(Φ)

if Φ is not a consistent set of literals

then return false;

if Φ contains an empty clause

then return false;

for every unit clause l in Φ

Φ=unit-propagate(l, Φ);

for every literal l that occurs pure in Φ

Φ=pure-literal-assign(l, Φ);

l := choose-literal(Φ);

return DPLL(ΦΛl) OR DPLL(ΦΛnot(l));

Where,

Φ, is the formula which we would like to determine the satisfiability of.

l, is a pure literal, or variable in Φ

In the formula, each clause is considered one of the disjunctions of variables. An empty clause is a clause in which all variables are assigned a value, but the clause is not satisfied. This prevents the overall formula form being satisfied.

Each literal is a variable in the formula. A unit clause is a clause that contains one unassigned variable, this is called a unit variable. This means that the clause can only be satisfied when this unit variable is assigned true. When a unit clause is found, the algorithm calls for unit propagation. This means that we automatically assign the unit variable true. If this assignment creates more unit variables, we continue the process until no unit variables exist, or the formula is satisfied.

If no unit clauses exist and the formula is not satisfied, we choose a literal which is not assigned and choose a value for it based on a heuristic.

The algorithm finishes when the assignment of variables satisfies the formula, or not.

The GSAT Algorithm

The GSAT uses a completely different approach to solving Boolean formulae than algorithms such as DPLL. For one, GSAT is not a complete algorithm; this means that GSAT will never halt execution if no solution to a formula exists. For determining unsatisfiability, GSAT will never be able to determine such a conclusion. GSAT randomly assigns values for all of the variables in a formula, and then continually flips variable values that reduce the amount of clauses in the formula. Pseudocode for this algorithm is found below (GSAT):

**procedure GSAT**

**for** i:=1 to MAX-TRIES

T:= a randomly generated truth assignment

**for** j := to MAX-FLIPS

**if** T satisfies **then return** T

Flip any variable in T that results in greatest

Decrease (can be 0 or negative)

in the number of unsatisfied clauses

**end for**

**end for**

**return** “No satisfying assignment found”

This algorithm has been shown to be very successful at determining Satisfiability compared to complete solvers such as DPLL.

One pitfall for this algorithm is that it can get stuck at local optima. GSAT is essentially a greedy algorithm. A variable change that decreases the overall number of clauses may be a good move in the short term, but may not be optimal for the overall satisfiability. This pitfall has been addressed in the WalkSat algorithm described above.

The ECJ Open Source Library

ECJ is an Evolutionary Computation platform in written in Java. ECJ is used in both Genetic Programming and Genetic Algorithms. Our research deals strictly with the GA portion of the library so that is what will be addressed here.

ECJ, as with many GA platforms, are very easy to use as you only need to define a fitness function. For more information on this fitness function, see the Evolutionary Computation section.

The fitness function is implemented by creating a class in Java that extends the Problem class and implements the evaluate() method. Once this is done, the evolutionary computation can be run when specified with a parameter file.

Parameter files in ECJ allow you to customize a GA run. The parameter file allows you to specify parameters such as the following:

1. A random seed.
2. Genome size.
3. The problem (as described above, the class containing the fitness function).
4. Subpopulation size.
5. Number of evolutionary generations
6. Vector species type.
7. Mutation probability
8. Valid gene values
9. Selection method.

An example of a parameter file is found below:

parent.0 = ../../simple/simple.params

seed.0 = time

pop.subpop.0.species.genome-size = 289

eval.problem = ec.app.grid.Grid

pop.subpop.0.size =500

generations = 5000

pop.subpop.0.species = ec.vector.IntegerVectorSpecies

pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline

pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline

pop.subpop.0.species.pipe.source.0.source.0 = ec.select.GreedyOverselection

pop.subpop.0.species.mutation-prob = .01

pop.subpop.0.species.pipe.source.0.source.0.top = .1

pop.subpop.0.species.pipe.source.0.source.0.gets = .99

select.tournament.size = 7

pop.subpop.0.species.pipe.source.0.source.1 = same

pop.subpop.0.species.min-gene = 0

pop.subpop.0.species.max-gene = 1

pop.subpop.0.species.ind = ec.vector.IntegerVectorIndividual

pop.subpop.0.species.fitness = ec.simple.SimpleFitness

pop.subpop.0.species.crossover-type =two

# Works Cited

DPLL Algoritm. 2010. 20 April 2010 <http://en.wikipedia.org/wiki/DPLL\_algorithm>.

Fenner, Stephen, et al. Rectangle Free Coloring of Grids. 2009. 30 November 2009 <http://www.cs.umd.edu/~gasarch/papers/grid.pdf>.

Gasarch, William. The 17x17 challenge. Worth $289.00. This is not a joke. 2009 November 2009. 30 November 209 <http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html>.

GSAT. 2010. 20 April 2010 <http://en.wikipedia.org/wiki/GSAT>.

Hayes, Brian. The 17x17 Challenge. 5 Devember 2009. 20 January 2010 <http://bit-player.org/2009/the-17x17challenge>.

Hoos, Holger H. On the Run-time Behaviour of Stochiastic Local Search ALgorithms for SAT. 30 February 2010 <http://www.satlib.org/ubcsat/algorithms/Hoos99b.pdf>.

Lobo, Andrea. Teaching Problem Reduction: NP-Completeness via Sudoku. 30 January 2010 <http://elvis.rowan.edu/~lobo/npc/Papers/CIESC2007SudokuPaper.pdf>.

Munroe, Randall. Recipies. 2010. 20 April 2010 <httip:/xkcd.com/720>.

Selman, Bart, Henry Kautz and Bram Cohen. Local Search Strategies for Satisfiability Testing. 30 January 2010 <http://www.cs.rochester.edu/u/kautz>.

Software Used:

ECJ. <http://cs.gmu.edu/~eclab/projects/ecj/>

MiniSat. Niklas Eén, Niklas Sörensson. <http://minisat.se/>

WalkSat. Bart Selman, Henry Kautz. <http://www.cs.rochester.edu/~kautz/walksat/>

zChaff. Princeton University. < http://www.princeton.edu/~chaff/zchaff.html>