# CSCI 446 Artificial Intelligence Project 1 Design Report

Roy Smart Nevin Leh Brian Marsh

September 24, 2016

## 1 Introduction

The *Graph Coloring Problem* (GCP) is the problem of attempting to color a set of interconnected regions such that no region has the same color as its neighbors using three or four colors. For example, consider the problem of coloring a map of the USA (Figure 1), using only four colors and ensuring that no neighboring states share the same color. This is the motivation of the graph coloring problem.

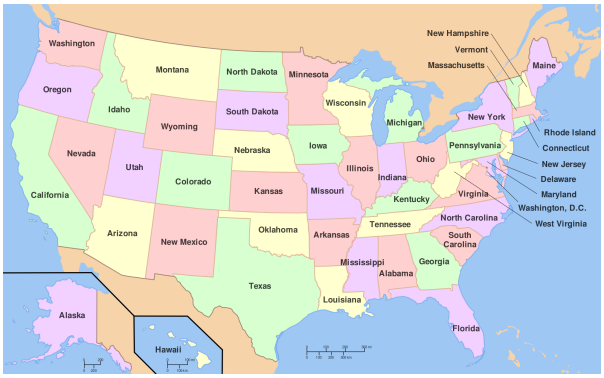


**Figure 1:** *Map of the United States of America satisfying the graph coloring problem.*

It can be shown that the map coloring problem reduces to the graph coloring problem if we represent the states as the vertices of the graph, and the borders between states as the edges of the graph[1]. This configuration produces a *planar* graph, a graph with no edge intersections.

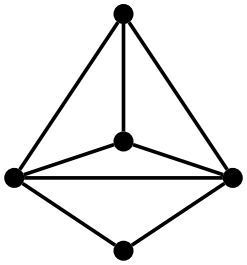

**Figure 2:** *Simple example of a maximally planar graph.*

For this project, we will concentrate on solving the graph coloring problem for *maximally planar* graphs (Figure 2), planar graphs for which adding any addition edge would make the graph non-planar.

We are tasked with solving the GCP five different ways: Minimum Conflicts, Simple Backtracking, Backtracking with Forward Checking, Backtracking with Constraint Propagation (MAC), and Local Search using a Genetic Algorithm. To test these algorithms, we will first need to build a problem generating program that can produce a random set of maximally planar graphs. Using the problem generator, we will calculate a set of graphs between the sizes {10, 20, 30,...,100} and then use the five graph coloring algorithms to solve the GCP. We will measure GCP algorithm performance by how many vertex read/write operations it requires to find a solution.

The solution to this project will be implemented in Python 3.4 using the graphics library to visualize the graphs.

## 2 Problem Generation

### 2.1 Computing a Maximally Planar Graph

For this project, we will need to create maximally planar graphs (MPGs) from a set of randomly scattered points. For an arbitrary set of points, there is no unique MPG that can be constructed. To solve this issue, the problem statement has provided a prescription for calculating an MPG.

> Select some point $X$ at random and connect $X$ by a straight line to the nearest point $Y$ such that $X$ is not already connected to $Y$ and line crosses no other line. Repeat the previous step until no more connections are possible.

To implement this algorithm we will first create a complete graph (where each vertex is connected to every other vertex). Each vertex will have a list of

associated edges, sorted by length. As instructed, we will then select a point at random and inspect the first unchecked edge $E$. If $E$ doesn't cross any of the accepted edges, it is added to the list of accepted edges and marked as checked. We then repeat this process until every edge has been checked.

An algorithm to determine if two edges (or line segments) cross has been outlined by LaMothe [2] and described in detail here. Let's start by defining two line segments

$$\begin{aligned} A &= \{\mathbf{a},\ \mathbf{a'}\} \\ B &= \{\mathbf{b},\ \mathbf{b'}\} \end{aligned} \tag{1}$$

where $\mathbf{a}$, $\mathbf{a'}$, $\mathbf{b}$, and $\mathbf{b'}$ are vectors from the origin to the ends of the line segments. Next, compute the direction vectors for each line segment

$$\begin{aligned} \boldsymbol{\alpha} &= \mathbf{a}' - \mathbf{a} \\ \boldsymbol{\beta} &= \mathbf{b}' - \mathbf{b} \end{aligned} \tag{2}$$

Now, the trick to solving this problem is to parameterize the line segment using the direction vector and a parameter $t_i$ in the domain $[0,\ 1]$.

$$\begin{aligned} \mathbf{p} &= \mathbf{a} + \boldsymbol{\alpha} t_a, \quad t_a \in [0,\ 1] \\ \mathbf{q} &= \mathbf{b} + \boldsymbol{\beta} t_b, \quad t_b \in [0,\ 1] \end{aligned} \tag{3}$$

From here, the solution is obvious. To find the point of intersection, we set $\mathbf{p} = \mathbf{q}$ and solve for the values of $t_a$ and $t_b$. Then, if the solution is outside the domain of $t_i$, the line segments do not intersect. Thus,

$$\Rightarrow \begin{cases} p_x = q_x \\ p_y = q_y \end{cases} \tag{4}$$

$$\Rightarrow \begin{cases} a_x + \alpha_x t_a = b_x + \beta_x t_b \\ a_y + \alpha_y t_a = b_y + \beta_y t_b. \end{cases} \tag{5}$$

Equation 5 is a system of two equations and two unknowns, solving for $t_a$ and $t_b$ gives

$$\Rightarrow \begin{aligned} t_a &= \frac{(-a_y + b_y)\beta_x + (a_x - b_x)\beta_y}{\alpha_y \beta_x - \alpha_x \beta_y} \\ t_b &= \frac{(a_y - b_y)\alpha_x + (-a_x + b_x)\alpha_y}{-\alpha_y \beta_x + \alpha_x \beta_y}. \end{aligned} \tag{6}$$

Finally, we can determine whether $A$ and $B$ intersect if $0 < t_a < 1$ and $0 < t_b < 1$ evaluates to true.

Utilizing the algorithms described above we have implemented problem generation code in Python to compute MPGs of arbitrary size. The output of a random execution of the problem generator is shown in Figure 3

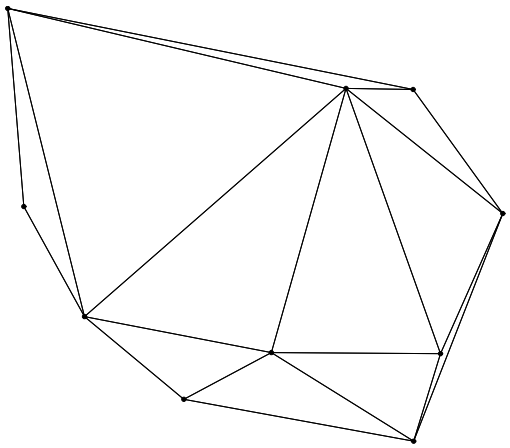

**Figure 3:** *Example of the MPG produced by our problem generator code with the number of vertices, $N = 10$.*

## 2.2 Computing Vertex Polygons

We have also implemented a way to produce a map of regions that is analogous to the graphs created by the problem generator, shown in Figure 4.

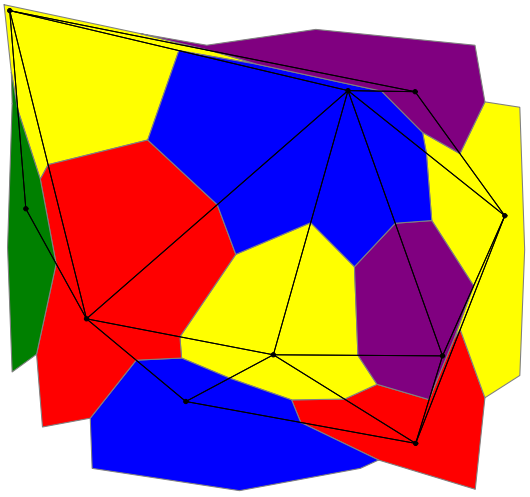

**Figure 4:** *The example of Figure 3, overlaid with the regions corresponding to each vertex. Each vertex has been randomly colored in this example.*

Since the graph coloring problem does not uniquely define the map coloring problem, we had to invent a metric that produces appropriate graphs.

To define our metric, consider that if a polygon $G_i$ corresponding to each point $P_i$ were constructed out of the midpoints $M_{i,m}$ of the associated edges $E_{i,m}$ of $P_i$, then an adjacent polygon $G_j$ would only touch at the corners, that is, our map would be full of gaps. Therefore, to fix the problem we also define the centroid $C_{i,n}$ of each triangle $T_{i,n}$ formed by the two adjacent edges $E_{i,m}$ and $E_{i,m+1}$. We then define the final polygon for $P_i$ using the points $M_{i,m}$ and $C_{i,n}$. This procedure is unnecessary in terms of the assignment, but is helpful for debugging and for aesthetic pleasure.

# 3 Experiment Design

## 3.1 Invariant Performance Measurement

Our goal with this project is to solve the graph coloring problem using five different algorithms while comparing the relative performance of each algorithm. Since CPU time and wall-clock time are not reliable measurements of computation, we will have to define a metric that allows us to compare different algorithms.

We propose to solve this problem using the concept of counting vertex read/write operations. The procedure will be to tally how many times each algorithm peeks into a vertex to determine its color and also how many times the algorithm must poke a vertex to update its color. This metric has the advantage of being algorithm independent, since all of our algorithms need the ability to set vertex colors and check the colors of adjacent vertices.

To acquire good performance statistics we will run ten experiments on each of the ten possible values for $N$ (10, 20, 30, 40, 50, 60, 70, 80, 90, 100) using each of the five GCP algorithms for a total of 500 experiments. We will then plot the average number of total vertex reads/writes vs $N$ for each algorithm. These graphs will allow us determine relative algorithm performance and also the time complexity, $\mathcal{O}(N)$, of each GCP algorithm.

## 3.2 Algorithm-Specific Performance Measurement

In addition to measuring the reads and writes to each vertex, we will also implement a metric for each algorithm. This will allow us to better compare an algorithm against itself for different problems. These metrics will be described in Section 5.

# 4 Software Design

We will use Python 3.4 for this project as it has simple graphics support for program testing, object-oriented paradigms, and convenient ways of parsing and storing data structures. The program entry point is the `Main()` function. This function will be responsible for generating the datasets, managing the experiments, collecting results, and displaying the output.

`Main()` will call the `ProblemGenerator(N)` function 100 times to create a list of `Map` objects upon which we will run our algorithms. We will refer to the list of `Map` objects as the dataset. `ProblemGenerator(N)` will return `Map` objects, which contain a list of `Point` objects. The class `Point` contains a list of `Edge` objects, which in turn reference another `Point`. This forms a doubly-connected graph and is the main data structure that will be used by our the GCP algorithm. Also the `Point` class will feature private variables `reads` and `writes` that will be incremented each time the functions `get_color()` and `set_color()` are called, respectively.

Once the dataset is constructed, `Main()` will run each of the five GCP algorithms on the entire dataset. Each algorithm will return a colored `Map`, a total of the variables `reads` and `writes` for each node, and additional algorithm-specific performance statistics described in Section 5. We will then plot the average and standard deviation of each measurement vs. the vertex number $N$ using Python's Math Plotting Library (`matplotlib`).

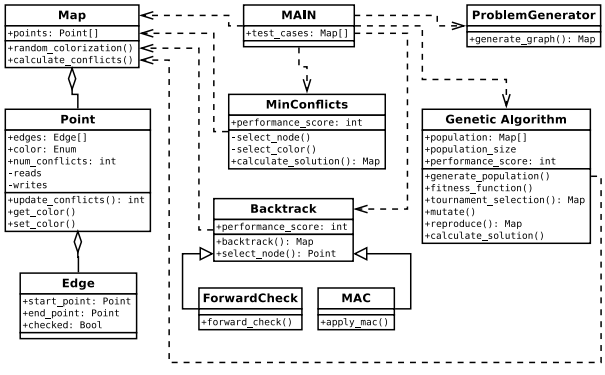


**Figure 5:** *A UML diagram describing the objects that each algorithm will use to solve the GCP and their relationships.*

# 5 GCP Algorithms

## 5.1 Minimum Conflicts

### 5.1.1 Design Implications

Minimum conflicts is a local search algorithm that can be used to solve the graph coloring problem. The approach is to randomly color the graph and assign each node a conflict score determined by how many of its neighbors have a color in common with that node. The algorithm then randomly chooses a conflicting node and changes it to a color that results in the least conflicts. The algorithm continues to choose conflicting nodes and changing them until there are no more conflicts or an until a predefined limit on how long the program runs is reached [1].

There are several design decisions that have been made to facilitate the use of this algorithm. One decision is to include a conflict score variable with each point. In addition, some functions will have to be implemented such as a node selector that ignores nodes with zero conflicts and a function that selects a color that causes the least amount of conflicts.

### 5.1.2 Experimental Design

The experimental design for Minimum conflicts is fairly simple. The intention is to increment a counter every time a node is selected to be minimized. This will give an accurate value that defines how many steps it took to find an answer.

## 5.2 Simple Backtracking

### 5.2.1 Design Implications

Backtracking is a variant of the depth-first search. For the graph coloring problem the approach is to choose a node and assign it a valid color. The algorithm continues to do this until a solution is found or a node is reached that cannot be assigned a valid color. If the node cannot be assigned a valid color the algorithm back tracks to the previous node and tries a different color.

This algorithm will have several design features that are required to make the algorithm work. The foremost feature is that the algorithm will be recursive as it is a convenient way of keeping track of old states. Each instance of the recursive function will have a list of previously used colors so it can know when to backtrack.

### 5.2.2 Experimental Design

The experimental design for backtracking is to increment a counter each time the `backtrack` method is called. This will give an accurate value that will define how many steps it took to find a solution.

## 5.3 Backtracking with Forward Checking

### 5.3.1 Design Implications

Backtracking with forward checking improves upon simple backtracking by looking forward for possible conflicting values in unassigned nodes when the current node is assigned a color. The algorithm deletes the conflicting colors from the domains of the unassigned nodes, and it backtracks if any of these domains becomes empty.

The design of the algorithm will be recursive in order to effectively backtrack. Forward checking can be achieved by a simple looping procedure that checks each of the neighboring nodes. Another design implication is that each node will have a way to save its domain which specifies what colors that node can still be assigned.

### 5.3.2 Experimental Design

The experimental design will be the same as simple backtracking. A counter could be added to see how many times the `forward_checking` method is called, but this method will just be called the same number of times as the `backtrack` method so the other counter can be used for both purposes.

## 5.4 Backtracking with Constraint Propagation

### 5.4.1 Design Implications

Backtracking with constraint propagation takes the idea of forward checking a bit further by deleting values from domains of nodes beyond just the immediately neighboring nodes. If a color is removed from a domain during the process of forward checking, constraint propagation looks further ahead to check if any colors of the neighboring nodes must be removed from their domains due to their dependency on the deleted value.

The design of this algorithm will remain fairly similar to backtracking with forward checking. However, the more extensive nature of constraint propagation requires an equally more extensive loop. The

algorithm will use a nested loop in order to look ahead beyond simply the neighboring nodes and into the neighbors of neighbors, and so on.

#### 5.4.2 Experimental Design

Again, the experimental design will be identical backtracking. Counting each time the `backtrack` function is called will give an accurate number of steps it took to solve the problem.

### 5.5 Local Search using a Genetic Algorithm

#### 5.5.1 Design Implications

Local search using a genetic algorithm involves a process similar to biological evolution to produce an individual that maximizes a fitness function. The fitness function for the GCP is the performance of the individual. The genetic algorithm begins with an individual with a unique string of genes that define its characteristics. A selection process chooses which individuals to use for reproduction, which are weighted by their score on the fitness function. The individuals are recombined by cross-over and the resulting individuals may be mutated from a small probability on each gene. For this project, tournament select will be used to select individuals for crossover. The design implications of this that we will need a `crossover` method to deal with reproduction.

#### 5.5.2 Experimental Design

The experimental design for the genetic algorithm is slightly more complicated. The first metric that will be used is the number of generations it takes to find a solution. This metric is not perfect because a very large population size may result in very few generations, but the overhead of processing such a large population would slow down the algorithm considerably. This is where the node `reads` and `writes` variables can be used to further assess the performance of the algorithm.

## References

[1] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 3rd edition, 2010.

[2] André LaMothe. *Tricks of the 3D Game Programming Gurus.* Sams Publishing, 201 West 103rd Street, Indianapolis, Indiana 46290, 2003.