# Artificial Intelligence Project 1: Design Document

**Group 11: Lisa Peters, Janette Rounds & Monica Thornton**

## 1. Description of the Problem

The graph coloring problem is a well-established constraint satisfaction problem (CSP) with a number of practical applications. In its simplest form, vertex coloring, the goal is to assign colors to the vertices of a graph $G$ such that no two vertices that share an edge have the same color. If a proper coloring of $G$ uses (at most) $k$ colors, then it serves as a $k$-coloring of the graph $G$. The software we are developing for Project 1 has two major functions, and is largely focused on the issues of search and constraint satisfaction as they apply to the graph coloring problem. The first task our software will perform focuses on the generation of several planar graphs varying in size from 10 to 100 vertices. The second task involves using five variations of a constraint solver to obtain $k$-colorings for all of the generated graphs. More specifically, the software will try to find consistent, complete assignments where $k = 3$ and $k = 4$ for each graph using min-conflicts, three different versions of backtracking, and a local search technique using a genetic algorithm as constraint solvers. Additional details regarding the implementation of these constraint solvers can be found in Section 3 of this document, and our plan to evaluate the performance of the various algorithms will be outlined in Section 4.

## 2. Software Architecture

In developing our software, we chose to utilize the Template Method design pattern as a guide. The Template Method defines the framework for an algorithm in an abstract class, leaving it to the subclasses to redefine parts of the algorithm as needed (Freeman, Robson, Bates, & Sierra, 2004). This approach is useful with respect to the problem specifications, as there is a fair amount of overlap between the five constraint solvers we need to implement. For example, each of the constraint solvers will need to have methods to assign colors to the vertices, evaluate the proposed solution, and check for constraint violations. Specifics regarding how these methods are implemented will vary according to the algorithm, so while many of these methods are specified in the abstract class, the details of the implementation are stored in the concrete classes. Furthermore, many of the algorithms will have methods that do not apply to the other constraint solvers, for example, only the `LocalSearchGA` will have a method for tournament selection.

A model of our software is provided as Figure 1. This diagram illustrates the major classes implemented in our software, as well a selection of some of the methods in each class, and the connections between the classes. The `RunModels` class provides the user with a method to run the `GraphGenerator`, as well as each of the five constraint solvers. The graph generator takes the number of vertices $n$ as input, and outputs a planar graph according to the process detailed in the project specifications. These planar graphs are output as a flat file that specifies the vertices and edges of the graph, which get read in and passed as input to the specified constraint solver. The five constraint solvers implemented in this work each inherit from `AbstractAlgorithm` which provides a general template of the relevant methods, which are fleshed out in the classes that inherit from it. In addition to the classes described above, our software also includes a class for scoring the solutions produced by the algorithms, a class that holds the penalty function for the local search, and an abstract and concrete class for the vertices of the graph.

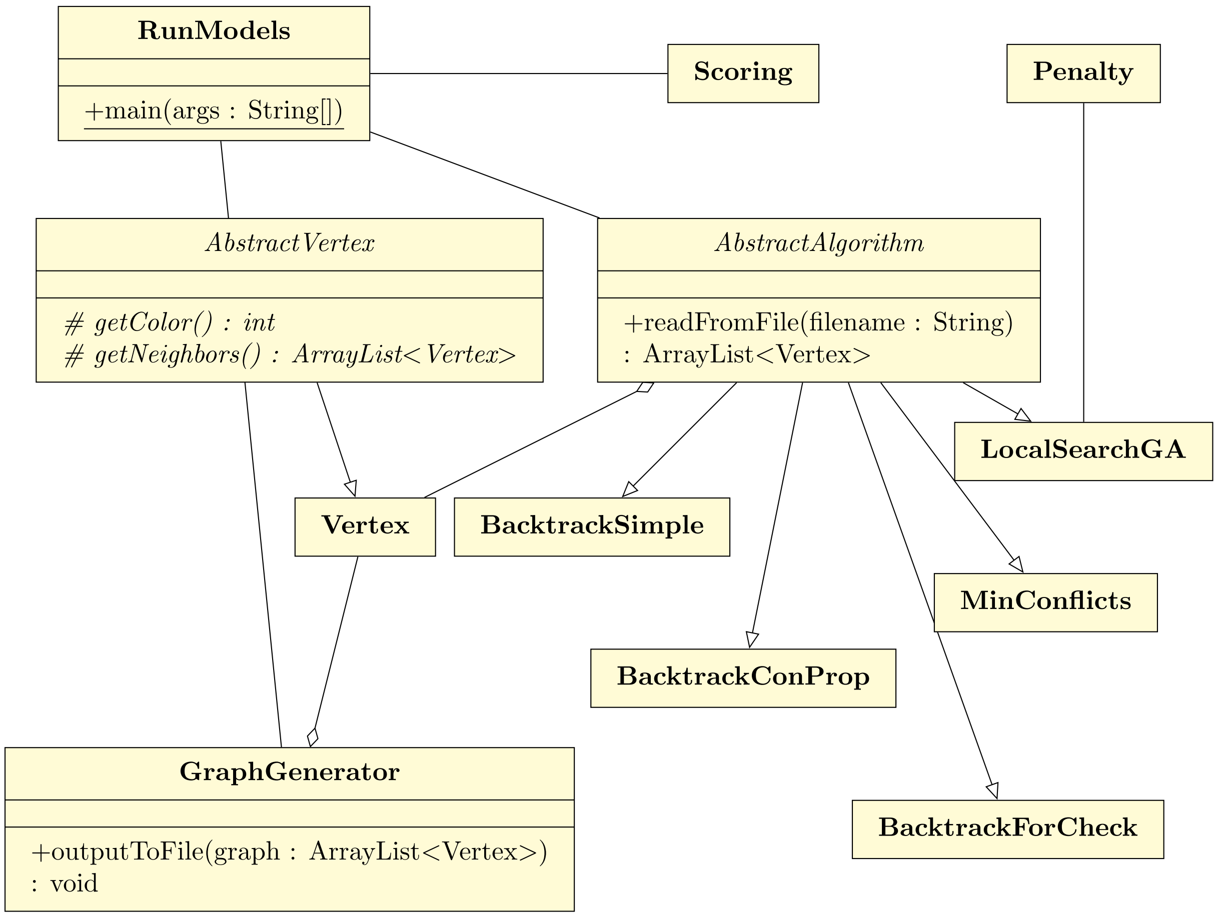

Figure 1: UML diagram that outlines the architecture of the described graph coloring software.

## 3. Design Decisions

In the following subsections, we outline a number of the key design decisions we have made when implementing our solution. The decisions more directly related to our experimental design are presented in Section 4 of this document. Additionally, although not depicted in Figure 1, our software will also include classes for running tuning experiments. Time permitting we will also include unit tests for all of the major functions, to ensure that the software continues to perform as expected.

### 3.1 Graph Generator

As described previously in this document, the problem generator constructs planar graphs to test the five specified variations of the constraint solver. The problem specifications provide a list of instructions for randomly generating the $n$ vertices for each graph, and the circumstances under which to connect two vertices with an edge. An edge can be drawn between two points $X$ and $Y$ if $Y$ is the nearest point that is not already a neighbor of $X$, and the edge between $X$ and $Y$ does not cross any existing edges.

There are a number of different methods to determine if two line segments intersect, but the approach we employ is inspired by one of the computational geometry algorithms presented in (Cormen et al., 2001). This algorithm employs the cross product to determine the orientation of the line segments, and also checks for the boundary case where the endpoint of one segment lies on another. The orientation information is valuable because we can say line segment $\overline{p_1p_2}$ intersects line segment $\overline{p_3p_4}$ if $p_1$ lies on one side of $\overline{p_3p_4}$, and $p_2$ lies on the other. We chose this method because it has no restrictions on the given points, is not prone to problems with respect to round-off error, and all of the necessary operations (addition, subtraction, multiplication and comparisons) can be done relatively quickly - which may be a concern for graphs with larger $n$ values.

The other major design decision with respect to the graph generator involves how we are going to pass the graphs to the constraint solvers. We chose to have the graph generator write out the graphs to a flat file, which will be read in by a parser that passes this information to the constraint solver.

### 3.2 Min Conflicts

Our implementation of the min-conflicts variation of the constraint solver is based on the algorithm provided in (Russell & Norvig, 2003). Like other local search algorithms, this approach uses a complete-state formulation where every variable is assigned a value, and the value of a variable is changed one at a time through the search process. The assignment of values to variables can be chosen randomly, or through a greedy assignment process that chooses the value that results in the least conflicts at every iteration. For our implementation, we will use the version of the min-conflict algorithm which uses the min-conflict heuristic to select the value that causes the minimum number of conflicts for each variable (node) at each iteration. The number of maximum iterations will be a tunable parameter, and we will select the maximum iteration value after running our tuning experiment.

### 3.3 Backtracking Approaches

Three types of backtracking will be implemented: simple backtracking, backtracking with forward checking, and backtracking with constraint propagation (MAC). Simple backtracking, as its name implies, simply backtracks to the parent node when a conflict is discovered during an assignment. Forward checking will add another layer of efficiency, in removing conflicting values for child nodes when an assignment is done. Finally, backtracking with the constraint propagation of MAC, or maintaining arc consistency will iterate through all nodes, removing conflicts at each assignment.

There are many heuristics we could use to improve the backtracking algorithm. We will consider three. The first of these is to assign colors to nodes with higher degree first. The second heuristic would prioritize nodes with fewer legal values. Our final heuristic would select values for each node that rules out the fewest choices for neighboring values. The procedures for backtracking with forward checking and backtracking with constraint propagation would obviate many of the benefits of these heuristics. Therefore, we will only compare these heuristics on the simple backtracking algorithm. During tuning, we will compare simple backtracking and simple backtracking with each one of these heuristics. For the actual algorithm comparisons, we will choose the heuristic (or no heuristic) that provided the best performance.

### 3.4 Local Search Using a Genetic Algorithm

For our local search algorithm, we will be implementing a genetic algorithm with a penalty function. Our penalty function (which will act as the fitness function)is as follows:

$$penalty(i) = -c * conflict \tag{1}$$

where $conflict$ refers to the number of conflicts in the current coloring of the graph. Additionally, $c$ is a constant that will affect how quickly the search process is directed away from solutions with more conflicts. We could have used additional terms in the penalty function representing blank nodes, or additional colors in the graph. However, as we discuss in Section 4, we will be comparing the number of conflicts at each iteration between algorithms. Given that, and given that we have a target number of colors, we could randomly initialize individuals in the population with a full coloring at the target number of colors and then minimize conflicts. Anything else would be an added complication and unlikely to be beneficial.

The encoding for our genetic algorithm will consist of a list of colors. Each position in the list will represent the color of a particular node. Our implementation will also incorporate a tournament selection operator so as to prevent the premature convergence problems found with fitness proportionate selection operators (Miller & Goldberg, 1996). Additionally, we will use a uniform crossover operator because uniform crossover has been shown that it can preserve subsequences without sacrificing diversity (Hu & Di Paolo, 2009). Our tuning approach is covered in Section 4.

## 4. Experimental Design

We will be running each algorithm on randomly generated graphs. In order to prevent extremely easy or hard graphs from skewing the results, we will be running five different randomly generated graphs for each graph size. Furthermore, in order to prevent one algorithm from getting especially hard or easy graphs, we will run all the algorithms on the same set of graphs. Finally, since the constraint solvers have stochastic elements, we will run each algorithm five times for each graph. Parameter tuning will be performed on each algorithm by running each algorithm multiple times with different parameter values (or parameter value combinations). Our tuning graphs will include a graph with 10 vertices, a graph with 50 vertices, and a graph with 100 vertices.

In order to ensure that algorithms finish, we will keep track of the number of iterations for each algorithm. Once a certain maximum number of iterations is reached, we will report the number of conflicts in the best-so-far coloring. If the algorithm finds a valid coloring before the maximum number of iterations is reached, we will report how many iterations were required for the algorithm to reach a valid coloring. At each iteration, we will track the number of conflicts in the current coloring of the graph. An uncolored graph would be the same as a graph that was colored all one color. In the final report, we will show how the number of conflicts changed with increasing number of iterations for each algorithm. We will compare algorithms on number of iterations reached and the number of conflicts in the final graph, if there are any.

## References

Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2001). *Introduction to Algorithms* (3rd edition). McGraw-Hill Higher Education.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.

Hu, X.-B., & Di Paolo, E. (2009). An efficient genetic algorithm with uniform crossover for air traffic control. *Computers & Operations Research*, *36*(1), 245–259.

Miller, B. L., & Goldberg, D. E. (1996). Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computation*, *4*(2), 113–131.

Russell, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education.