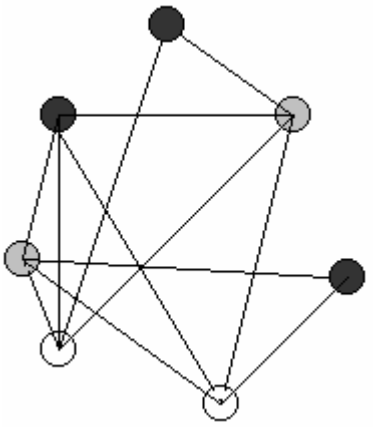
Table of Contents

Implementation of a Hybrid Evolutionary Algorithm for Graph Coloring

1. Introduction

Graph coloring is a problem that came about from people wanting to color adjacent countries on a map. Given a map, we would like to color countries or regions such that no adjacent country or region share the same color. It was conjectured in the 19th century, that at most four colors would be necessary for any given map. The problem has been studied extensively over the years, and from the coloring problem many results in Graph Theory and related studies, such as scheduling and resource allocation have been obtained.

Formally, in *vertex coloring*, we are given a graph G=(V,E) and a coloring function c:V->C; we are to find an assignment of colors from C such that for all adjacent vertex v,u in V, we have c(v) != c(u). The most interesting property of the set C of colors is its size. Wherein, we are typically concerned with the smallest number of colors needed such that no adjacent vertex has the same color. Such a number is known as the *chromatic number.*



A 3-coloring of a graph

The result of coloring a graph is that we find k disjoint sets of vertices/edges associated with each color from {1,2,...,k}.

Several immediate questions can be asked given the above problem statements. Firstly, given a graph G=(V,E) and an integer k, is G k colorable? That is, can we color G using k colors but not less? Second, given a graph

G=(V,E) and an integer k, can we find an assignment of colors to vertices such that we have a proper coloring ? Determining if a given graph is k-colorable is different than finding a coloring using k colors, generally speaking. It is not necessarily the case that if we can find a coloring of k+1 colors, that the graph cannot be colored using k colors.

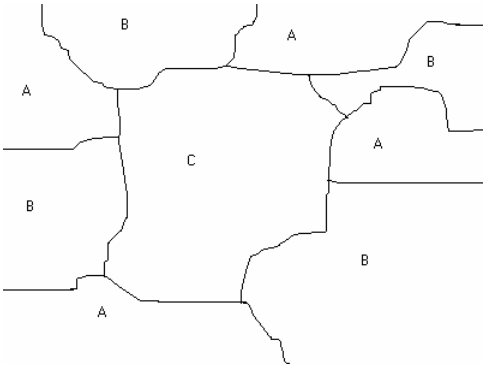
2. Complexity of Graph k-Colorability

Given an instance of a graph G=(V,E), is G k-colorable ? That is, does there exist a function c:V->{1,2,...,k} such that c(u) != c(v) whenever {u,v} ∈ E ?

The above problem is NP-Complete for all fixed k >= 3 and for K=3 and remains NP-Complete for planar graphs with no vertex degree exceeding 4. This was proved by Stockmeyer, in 1973.

3. Applications of Graph Coloring

Consider the problem of scheduling finals at a local university. The problem can be stated as: *Find a scheduling of final examinations such that no two students have a conflicting schedule date*. This can be modeled using a graph by letting each vertex represent a final examination and letting each edge e=(v,u) represent a student if the student has to take exam v and u. The colors represent times at which an exam may be given. Thus, finding an appropriate coloring for the graph implies we have an appropriate schedule of exams (because no two exam times conflict by our definition of coloring).



The map coloring problem

Consider the problem of assigning channels to radio stations. The problem can be stated as: *Find a channel assignment to R radio stations such that no station has a conflict*. This can be represented as a graph coloring problem similar to the previous problem, by letting vertices correspond to radio stations, and letting each edge correspond to a channel if the channel would conflict with the two given radio stations. For example, two radio stations maybe in close vicinity to one another such that assignment of channel 107.5 FM to both corresponds to a conflict between the two. Letting colors correspond to available channels, and it should be clear that finding an appropriate coloring implies we have an appropriate assignment of channels to radio stations.

Consider the problem of register allocation in a compiler. The problem can be stated as: *Given a set of available registers, find an assignment of variables to registers within a given scope*. In a loop or body of a function, it is often desirable for the compiler to load variables from memory onto registers so that while in execution, the number of references to memory is reduced. Here the problem maybe modeled with vertices representing variables and colors representing registers. An edge exists between any two vertices if there would be a conflict in assigning the two variables to the same register at the same time. Given a coloring of this graph, we would have an appropriate assignment of variables to registers.

The allocation of hardware buses in a VLSI design is another practical example. It should be clear from the above examples as to why we are interested in the smallest number of colors needed. For example, with respect to register allocation, the colors represent registers and it is usually not the case that their are enough registers to accommodate all variables.

4. Need for an efficient algorithm for graph coloring

Given the wide applicability of graph coloring and the inherent inefficiency of the classical Turing computational model in solving large instances of the problem, an efficient algorithm for graph coloring is required. For instance it is not uncommon in VLSI design to have hundreds of components competing to use a few buses.

Hence we employ other models of computation to solve the problem. In this project we successfully apply a Hybrid Evolutionary Algorithm to solve many large benchmark instances of the graph-coloring problem.

5. Genetic Algorithms

Genetic algorithm (GA) is an algorithm used to find approximate solutions to difficult-to-solve problems through application of the principles of evolutionary biology to computer science. Genetic algorithms use biologically derived techniques such as inheritance, mutation, natural selection, and recombination (or crossover). Genetic algorithms are a particular class of evolutionary algorithms.

Genetic algorithms are typically implemented as a computer simulation in which a population of abstract representations (called chromosomes) of candidate solutions (called individuals) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but different encodings are also possible. The evolution starts from a population of completely random individuals and happens in generations. In each generation, multiple individuals are stochastically selected from the current population, modified (mutated or recombined) to form a new population, which becomes current in the next iteration of the algorithm.

6. Tabu Search

Tabu search is a mathematical optimization method, belonging to the class of local search techniques. Tabu search enhances the performance of a local search method by using memory structures. Tabu search is generally attributed to Fred Glover.

Tabu search uses a local or neighbourhood search procedure to iteratively move from a solution x to a solution x' in the neighbourhood of x, until some stopping criterion has been satisfied. To explore regions of the search space that would be left unexplored by the local search procedure and---by doing this---escape local optimality, tabu search modifies the neighbourhood structure of each solution as the search progresses. The solutions admitted to N* (x), the new neighbourhood, are determined through the use of special memory structures. The search now progresses by iteratively moving from a solution x to a solution x' in N* (x).

7. The hybrid coloring algorithm

To solve the GCP, i.e. to color a graph with a minimum possible number of colors, a particular approach consists in applying a coloring algorithm to look for a k-coloring for a (sufficiently high) number of colors $k = k_0$. Then whenever a k-coloring is found, one re-applies the same algorithm to look for k-colorings with decreasing numbers of colors $k = k_0 - 1$, $k_0 - 2$ …Therefore, the graph coloring problem is reduced to solving increasingly difficult k-coloring problems. In this section, we present the components of our Hybrid Coloring Algorithm (HCA) for finding k-colorings.

7.1 Search space and cost function

Recall that the k-coloring problem consists in coloring a graph $G=(V, E)$ with a fixed number of k colors. To solve a k-coloring problem, we consider the set of all possible partitions of V in k classes including those which are not proper

k-colorings. Each partition is attributed a penalty equal to the number of edges having both endpoints in a same class. Therefore, a partition having a 0 penalty corresponds to a proper k-coloring (a solution). The purpose of the search algorithm is then to find a partition having a penalty as small as possible. So solving the minimization problem can solve the k-coloring problem. In the following we call configuration any element of the search space *S* and reserve the word solution for a proper k-coloring.

7.2 The general procedure

In HCA, the population P is a set of configurations having a fixed constant size |*P*|. We present below the general algorithm:

The hybrid coloring algorithm
Data : *graph G* D *.V; E/*, *integer k*
Result : *the best configuration found*
Begin
    P=*InitPopulation(|P|)*
    While *not Stop-Condition ()* **do**
        (s1,s2)=*ChooseParents*(P)
        s=*Crossover*(s1,s2)
        s=*LocalSearch*(s, L)
        P=*UpdatePopulation*(P,s)
End

The algorithm first builds an initial population of configurations (*InitPopulation*) and then performs a series of cycles called *generations*. At each generation, two configurations *s*1 and *s*2 are chosen in the population (*ChooseParents*). A crossover is then used to produce an offspring *s* from *s*1 and *s*2 (*Crossover*). The LS operator is applied to improve *s* for a fixed number *L* of iterations (*LocalSearch*). Finally, the improved configuration *s* is inserted in the population by replacing another configuration (*UpdatePopulation*). This process repeats until

a stop condition is verified, usually when a pre-fixed number *MaxIter* of iterations is reached. Note however that the algorithm may stop before reaching *MaxIter* , if the population diversity becomes too small (see Section 6.2).

Let us notice that this hybrid algorithm differs from a genetic algorithm by some features. The fundamental difference is naturally that the random mutation operator of a GA is replaced with a LS operator. Another difference concerns the selection operator of a GA, which encourages the survival and reproduction of the best individuals in the population. In the hybrid algorithm, the selection is ensured jointly by *ChooseParents* and *UpdatePopulation*.

7.3 The initialization operator

The operator *InitPopulation(|P|)* initiates the population *P* with |*P*| configurations. To create a configuration, we use the greedy saturation algorithm of Br´elaz (1979) slightly adapted in order to produce a partition of *k* classes. The algorithm works as follows. We start with *k* empty color classes $V_1 \ldots V_k$. At each step, we chose a vertex $v \in V$ such that *v* has the minimal number of allowed classes (i.e. a class that does not contain any vertex adjacent to *v*). To put *v* in a color class, we chose among all the allowed classes of *v* the one V_i that has the minimal index *i*. In general, this process cannot assign all the vertices. Each unassigned vertex is then put into one color class randomly chosen. Once a configuration is created, it is immediately improved by the LS operator for *L* iterations. Due to the randomness of the greedy algorithm and the LS improvement, the configurations in the initial population are quite different. This point is important for population based algorithms because a homogeneous population cannot evolve efficiently.

7.4 The crossover operator

The crossover used here is the GPX crossover presented. Let us show now how this crossover works. Given two parent configurations $s1 = \{ V_1^1, \ldots, V_k^1\}$ *and* $s2 = \{ V_1^2, \ldots, V_k^2\}$ chosen randomly by the *ChooseParent* operator from the

population, the algorithm *Crossover(s1,s2)* builds an offspring $s = \{V_1,\ldots,V_k\}$ as follows:

The GPX crossover algorithm

Data : *configurations* $s1 = \{ V_1^1, \ldots, V_k^1\}$ *and* $s2 = \{ V_1^2, \ldots, V_k^2\}$

Result : *configuration* $s = \{V_1,\ldots,V_k\}$

begin

  for *l*(1 <= *l* <= *k*) **do**

    if *l* is odd, then *A* := 1, else *A* := 2

    choose *i* such that V_i^A has a maximum cardinality

    $V_l := V_i^A$

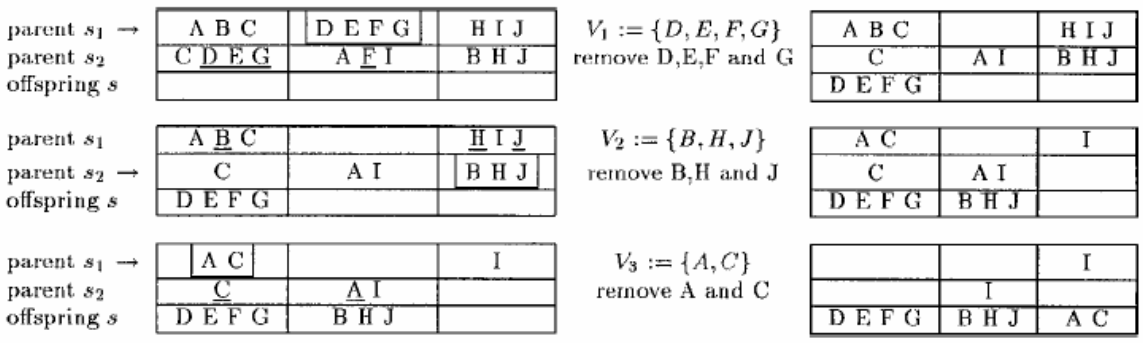
    remove the vertices of V_l from *s*1 and *s*2

    Assign randomly the vertices of $V - (V_1 \cap V_2 \cap V_3 \ldots \cap V_K)$

end

The algorithm builds step by step the *k* classes $V_1,\ldots,V_k$ of the offspring at step *l*(1 <= *l* <= *k*), the algorithm builds the class V_l in the following way. Consider parent *s*1(*A* = 1) or parent *s*2(*A* = 2) according to whether *l* is odd or even. In the considered parent, choose the class having the maximum number of vertices to become class V_l and remove these vertices from parents *s*1 and *s*2. At the end of these *k* steps, some vertices may remain unassigned. These vertices are then assigned to a class randomly chosen.

The crossover algorithm: an example.



Let us notice that the edges of the graph, which are used in the cost function f, are not involved in the crossover operator.

7.5 The LS operator

The purpose of the LS operator *LocalSearch(s, L)* is to improve a configuration *s* produced by the crossover for a maximum of *L* iterations before inserting *s* into the population. In general, any local search method may be used. In our case, we use tabu search (TS), an advanced local search meta-heuristic (Glover and Laguna, 1997). TS performs guided search with the help of short and eventually long term memories. Like any LS method, TS needs a neighborhood function defined over the search space $N : S \rightarrow 2^S$. Starting with a configuration, a typical TS procedure proceeds iteratively to visit a series of locally best configurations following the neighborhood. At each iteration, a *best* neighbor is chosen to replace the current configuration, even if the former does not improve the current one. This iterative process may suffer from cycling and get trapped in local optima. To avoid the problem, TS introduces the notion of *Tabu lists*. The basic idea is to record each visited configuration, or generally its attributes and to forbid to re-visit this configuration during next *tl* iterations (*tl* is called the tabu tenure).

The TS algorithm used in this project is an improved version of the TS algorithm proposed by Hertz and de Werra (1987). Here a *neighbor* of a given configuration *s* is obtained by moving a single vertex *v* from a color class to another color class V_i. To make the search more efficient, the algorithm uses a simple heuristic: the vertex *v* to be moved must be conflicting with at least another vertex in its original class. Thus a neighboring configuration of *s* is characterized by a *move* defined by the couple $(v, i) \in V \times \{1 \ldots k\}$. When such a move *(v, i)* is performed, the couple *(v; s(v))* is classified tabu for the next *tl* iterations, where *s(v)* represents the color class of vertex *v* in *s*. Therefore, *v* cannot be reassigned to the class *s(v)* during this period. Nevertheless, a tabu move leading to a configuration better than the best configuration found so far is always accepted (*aspiration criterion*). The tabu tenure *tl* for a move is variable and depends on the number nb_{CFL} of conflicting vertices in the current configuration: $tl = Random(A) \, C \times \alpha \times nb_{CFL}$ where *A* and α are two parameters

and the function *Random(A)* returns randomly a number in [0,A]. To implement the tabu list, it is sufficient to use a *V x { 1 … k }* table.

The algorithm memorizes and returns the *most recent* configuration *s*§ among the best configurations found: after each iteration, the current configuration *s* replaces *s** if *f(s) <= f(s*)* (and not only if *f(s) <= f(s*))*. The rational to return the last best configuration is that we want to produce a solution which is as far away as possible from the initial solution in order to better preserve the diversity in the population. The skeleton of the TS algorithm is given below.

The TS operator
Data : *graph G = (V, E)*, configuration *s*0
Result : the best configuration found
begin
    *s := s*0
    while *not Stop-Condition()* **do**
        choose a best authorized move *(v,i)*
        introduce the couple *(v,s(v))* in the Tabu list for *tl* iterations
        perform the move *(v,i)* in *s*
end

The configuration created by a crossover and improved by TS is now to be inserted in the population. To do this, the worst of the two parents is replaced.

8 Experimental results

8.1 Experimental settings

Graphs from the well-known second DIMACS challenge benchmarks are used (Johnson and Trick, 1996). We are interested in these graphs because they were largely studied in the literature and constitute thus a good reference for comparisons. Moreover these graphs are difficult and represent a real challenge for graph coloring algorithms. Note that according to the optimization approach

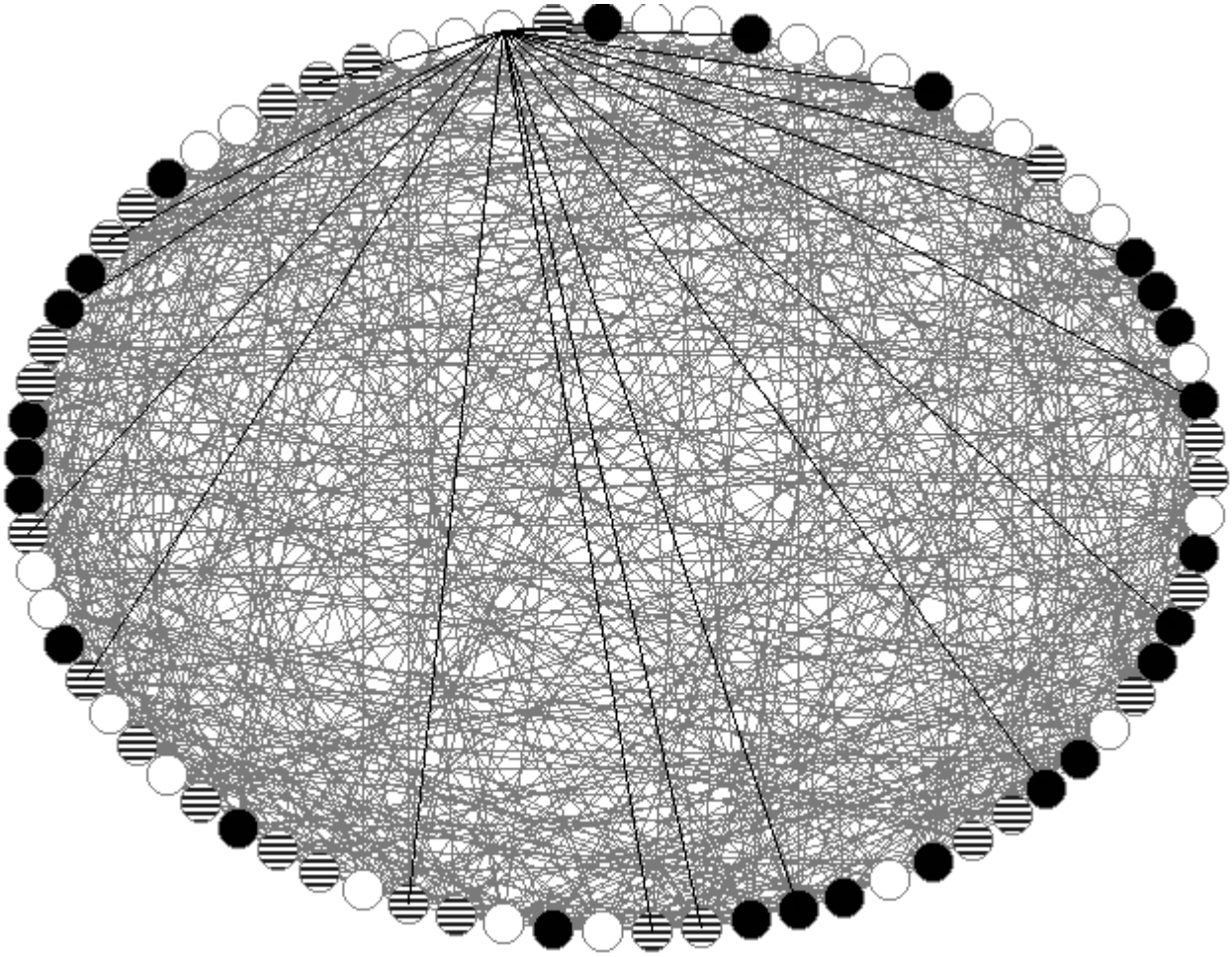
used (Section 4), each graph defines in reality a set of k-coloring instances for different values of *k*.

It was observed that the Hybrid algorithm performed better than plain a tabu search or genetic algorithm approach, without any noticeable delay in speed. Infact the genetic algorithm approach failed to evolve better solutions from existing solutions in the population.

9. References

1. This algorithm is due to the paper: *Hybrid Evolutionary Algorithms for Graph Coloring by Galiner and Hao (Journal of Combinatorial Optimization 1999).*
2. *Goldberg, David E.:* Genetic Algorithms in Search, Optimization and Machine Learning
3. Graph Theory with Application to Engineering and Computer Science, by Narsingh Deo.
4. Graph Theory by *Reinhard Diestel* (Electronic Edition 2000).
5. The C++ Programming Language, *Bjarne Stroustrup.*

10. A Sample Coloring



A 3-coloring of a random graph with 75 nodes. The fine mesh within the elliptical arrangement of the nodes are the edges. The edges of one particular node are highlighted.