Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Mária Vámošová

Barvení grafů pomocí mravenčí kolonie
Graph Coloring Using an Ant Colony

Department of Applied Mathematics

Supervisor: RNDr. Eva Jelínková

Study programme: Computer Science, General Computer Science

Prague, 2010

I would like to thank Eva Jelínková for her supervising, lot of advice and encouragement.

Contents

Název práce: Barvení grafů pomocí mravenčí kolonie
Autor: Mária Vámošová
Katedra: Katedra Aplikované Matematiky
Vedoucí bakalářské práce: RNDr. Eva Jelínková
e-mail vedoucího: eva@kam.mff.cuni.cz

Abstrakt: V této práci se zabýváme problémem barvení grafů. Podáváme přehled existujících algoritmů a zaměřujeme se na algoritmy mravenčí kolonie. Pro pozorování jejich principů a porovnání jsme vybrali tři existující algoritmy, otestovali je na známých grafech a podáváme přehled výsledků. Tyto výsledky jsou dále porovnány s dosud nejlepšími známými. Součástí práce je i návrh a otestování vlastního algoritmu.

Klíčová slova: Mravenčí kolonie; Problém barvení grafů; Barvící algoritmy

Title: Graph Coloring Using an Ant Colony
Author: Mária Vámošová
Department: Department of Applied Mathematics
Supervisor: RNDr. Eva Jelínková
Supervisor's e-mail address: eva@kam.mff.cuni.cz

Abstract: In the present thesis we deal with the Graph Coloring Problem. We provide an overview of the existing approaches and focus on the Ant Colony Optimization heuristics. We have selected three algorithms, implemented them and performed a set of tests on several commonly used graph instances to study their advantages and drawbacks. We also compared their results to the best known at time to find out how much they are suitable for the problem. In this thesis we also present an original algorithm.

Keywords: Ant Colony Optimization; Graph Coloring Problem; Coloring Algorithms

Chapter 1

Introduction

The Graph Coloring Problem (GCP) is a well-known combinatorial problem which consists of coloring the vertices of a given graph so that no adjacent vertices get the same color. At the same time, we try to minimize the number of colors used. The minimum number of colors we can color the graph with is called the chromatic number of the graph.

Although there are many practical reasons to solve the GCP, unfortunately, this problem is very hard to solve. Exact solution methods can solve instances of relatively small size (approximately 100 vertices), but upper bounds on the chromatic number can be obtained for larger instances by using heuristic algorithms.

In the present thesis we study a special class of such heuristics, Ant Colony Optimization (ACO) heuristics. ACO simulates the behavior of a colony of ants, more specifically, the way they collectively solve problems.

We implemented three different ACO algorithms and performed a set of tests on them. We also present our original ant algorithm.

1.1 Structure of work

In Chapter 2 we define the Graph Coloring Problem and provide an overview of the existing graph coloring algorithms. We then provide an introduction to the Ant Colony Optimization heuristics, and focus on the graph coloring heuristics.

In Chapter 3 we describe the three algorithms we selected for our implementation. At first, we show the similarities between these algorithms and then describe each algorithm in detail.

In Chapter 4 we present our original ACO algorithm.

In Chapter 5 we describe our program, Coloring Ants, which implements the three algorithms mentioned in Chapter 3 as well as our algorithm mentioned in Chapter 4. The program does not only compute the GCP, but also provides a visualisation during the computation for better observation of the run of the algorithm.

In Chapter 6 we provide the results of the experiments we performed on the algorithms mentioned above. We compare the algorithms with each other, and also with the best results known at time.

The enclosed CD contains the electronic version of this document. It contains the source code of the Coloring Ants program and the user and programmer documentation as well. We also added the graph instances used for the experiments.

Chapter 2

An overview of graph coloring methods

2.1 The Graph Coloring Problem

At the beginning, we provide some definitions. A graph G is defined as an ordered pair $G = (V, E)$ comprising a set V of vertices, together with a set E of edges, which are 2-element subsets of V. The Graph Coloring Problem (GCP) can be described as follows. Given a graph $G = (V, E)$, and given an integer k, a k-coloring of G is a function $col : V \to \{1, ..., k\}$. The value $col(x)$ of a vertex x is called the *color of x*. Vertices with the same color define a *color class*. If two adjacent vertices x and y have the same color, then vertices x and y are called *conflicting vertices* and the edge linking x with y is called a *conflicting edge*. The *conflict of a vertex* is the number of conflicting edges leading from that vertex. The *total conflict* of the graph is the sum of the conflicts of all its vertices.

A color class without conflicting edges is called a *legal color class*. A k-coloring without conflicting edges is said to be *legal* and corresponds to the partition of the vertices into k legal color classes.

The GCP is to determine the smallest integer k (called the chromatic number of G, and denoted by $\chi(G)$) such that there exists a legal k-coloring of G.

The Graph Coloring Problem has many practical applications. Vertex coloring can be used for example to solve the scheduling problem. A scheduling problem is a situation where a set of activities needs to be executed in a number of time slots. As the activities may be processed in parallel, we want to minimize the total time elapsed. Ideally, all activities are done in the same time slot. However, some activities may be dependent on each other or share the same resource and therefore cannot be performed at the same time. This problem can be solved by creating a graph where the activities are represented by vertices and an edge links two activities if they cannot be executed in the same time slot. The chromatic number of the graph is then the minimum number of time slots we need to execute all the activities without conflicts.

Another graph coloring application is the register allocation. A processor register is a small memory unit that can be accessed in a very short time. As the program executes the instructions, it is very advantageous to have all the variables it uses stored in the register. The register allocation is an optimization technique, in which the variables are assigned to the limited number of registers so that they can all be accessed as fast as possible. As not all variables are used in the same time, we can assign more than one variable to one register. However, variables which are used at the same time cannot be stored in the same register, as they would corrupt. This problem can be solved by creating a graph where the vertices represent the variables and an edge connects two variables if they are used in the same time. We then try to color this graph with k colors, where k represents the number of registers.

There are many other applications, for example the well-known logical game *sudoku*, which can be seen as a 9-coloring of given specific graph with 81 vertices.

2.2 Algorithms for the GCP

Through history, there have been many different approaches to the problem. They can be divided into two main categories: *exact solution methods* and *heuristics*. While exact solution methods solve the problem exactly and produce an optimal coloring, the heuristics may produce non-optimal solutions.

Since the GCP is NP-hard [24], exact solution methods can solve graphs of very limited size. Existing algorithms can cope with graphs that have up to approximately 100 vertices, more vertices would make the time-complexity unbearable. On the other hand, heuristics are often very time-efficient, therefore more suitable for larger problems. Although they may produce non-optimal solutions, good heuristics can find solutions close to the optimal one.

In the following we provide a brief overview of the graph coloring methods. We rely on surveys [35], [23], [9] and on original reseach papers.

2.2.1 Exact solution methods

Despite the relevance of the problem, the number of exact approaches is small with respect to the variety of heuristic algorithms proposed.

The simplest exact coloring method is the greedy approach. We take the vertices one at a time and assign to each vertex the lowest number of color possible without creating a conflict with already colored vertices. It is known that there exists an optimal ordering in which the greedy approach gives an optimal solution. However, to be sure we found the best coloring, we must consider all possible orderings of vertices. Therefore, the time complexity of this approach is $\Omega(|V|!)$, which makes the algorithm practically useless.

More sophisticated graph-theoretical methods were studied and there were proposed several algorithms such as the one of Brown [5], Brélaz [4] or Sewell [42].

The Graph Coloring Problem can be also solved with the Integer Linear Programming (ILP) approach. Multiple ILP formulations have been proposed, among others those by Mehrotra and Trick [36], Hansen et al. [27] or Méndez-Díaz and Zabala [37] [38].

2.2.2 Greedy constructive heuristics

The fastest methods to generate a (non-optimal) legal coloring are the greedy constructive heuristics. The algorithms construct the solution by sequentially coloring the vertices, each with appropriate color. Each iteration of the heuristics consists of two steps: first, the next vertex to be colored is chosen, and then this vertex is assigned a color.

The order in which the vertices are colored can be determined either *statically*, before the start of the algorithm, or *dynamically*, depending on the vertices already colored.

The color to assign is mostly chosen as the lowest color possible without creating any conflicts with already colored vertices.

The simplest greedy heuristics is the static *greedy sequential algorithm* (SEQ). SEQ applies the greedy approach mentioned in Section 2.2.1 on the initial ordering of the vertices. More specifically, it takes the vertices in the initial ordering, assigning each vertex the lowest color possible without creating any conflict. Although this approach is very time-efficient, it gives very poor results, as it completely depends on the initial ordering of the vertices.

The algorithm can be enhanced by ordering the vertices according to their degrees, so that we first color the vertices with larger degrees. This algorithm is called Largest First (LF). Although the algorithm performs better than SEQ, it still cannot produce satisfying results (according to [32]).

In contrast to static sequential algoritms, the dynamic sequential heuristics have been proposed.

One of the first ones, the DSATUR heuristics, was proposed by Brélaz in [4] (this DSATUR heuristics is not the same as the DSATUR exact algorithm proposed in the same paper). Suppose we have a partial coloring of the graph. We then define the *saturation degree* of the vertex as the number of different colors assigned to its neighbours in this partial coloring. At each iteration, DSATUR colors vertex with largest saturation degree. Ties are broken by taking the vertex with largest degree; futher ties are broken randomly.

Other dynamic sequential algorithm is the Recursive Largest First algorithm (RLF) proposed by Leighton in [33]. The algorithm sequentially creates color classes as maximal independent sets of vertices. Let c be current color class we are creating. Let P be the set of uncolored vertices that can be included in c (not creating any conflicts) and R the set of vertices that are adjacent to a vertex already colored with c. The first vertex to add to the color class c is chosen as a vertex from P having the maximum number of neighbours in P. The sets P and R are then updated, i.e., the adjacent vertices of the colored vertex are moved from set P to set R. Then, at each step, the algorithm adds to the color class c the vertex from P having the maximum number of adjacent vertices in R. Ties are broken by adding the vertex with minimum number of adjacent vertices in P, futher ties randomly. The process is repeated until P is empty. If there are any uncolored vertices, we start to create another color class, otherwise the algorithm ends.

The RLF algorithm was further enhanced by Johnson et al. In [31], they propose the XRLF algorithm. A modification of XRLF, the MXRLF algorithm was proposed by Bui et al. [6] (we decribe MXRLF in detail in Section 3.2.2).

The DSATUR and RLF heuristics have been tested and compared by Chiarandini et al. [9]. They also tested the Random Order Sequential Heuristic (ROH), which is similar to SEQ with the modification that the vertices are taken in random order. The conclusion was that RLF performs statistically significantly better than DSATUR for most instance classes and both heuristics are by a large margin better than ROH. RLF is not significantly better than DSATUR only on certain special graph classes, namely the insertions, full insertions, and course scheduling graphs. The time-complexity is inverse though, the RLF being the slowest algorithm and ROH the fastest.

Greedy construction heuristics can also be used if only a fixed number k of color classes is available. In that case, the usual steps of the construction algorithms are followed as long as the number of used color classes is lower than k. Then, if a vertex cannot be colored legally, it either remains uncolored, or is added to a color class randomly or according to some other heuristic that tries to minimize the number of arising conflicts. The result of such modified construction heuristics is typically a non-legal k-coloring or a legal partial k-coloring, which can serve as an initial solution for improvement methods, such as local search heuristics, which we address in the following section.

2.2.3 Local search heuristics

The local search heuristics are one of the most popular and most efficient approaches to the problem. A local search algorithm starts from an initial solution and then iteratively moves from the current solution to an improved one.

The algorithm must define the *set of candidate solutions*, the *neighbourhood relation*, and the *solution evaluation function*.

The set of candidate solutions represents the search space of the algorithm, i.e., the set of all possible solutions we consider. The candidate solutions do not need to be the solutions to the original problem. They can be legal colorings, non-legal colorings or legal partial colorings. The number of colors used in the colorings can be either fixed (if we are searching for a k-coloring) or variable.

The neighbourhood relation defines the neighourhood of a solution, i.e., the set of solutions we can move to from the current solution.

The solution evaluation function represents the advantage of a considered solution. Therefore, a move from current solution to a neighbour depends on this evaluation function.

One of the oldest local search heuristics is the *simulated annealing* (SA). This probabilistic method is inspired by the annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. At each step, the SA heuristic considers a neighbour s' of the current solution s, and probabilistically decides whether to move to solution s' or stay in solution s. The probability depends on the evaluation function and a global parameter called the *temperature*. The value of the temperature decreases every iteration; therefore, the solution is more likely to change at the beginning of the algorithm, while in the end, when the temperature is low, the solution is more likely to stay in current state.

The first SA algorithm for the GCP was proposed by Chams et al. in [8]. The SA algorithms were also studied and compared by Johnson et al. in [31]. In their work, the authors experimented with three different variants of SA heuristics for the GCP. In addition to the three SA algorithms, Johnson et al. have designed a procedure, called XRLF, for extracting legal color classes (as mentioned in Section 2.2.2).

A new approach for the local search heuristics was proposed by Morgenstern [40]. The author defines the *impasse class neighbourhood*. The idea is that we consider k legal color classes and the set of uncolored vertices. A move consists of picking an uncolored vertex, adding it to a color class and moving all conflicting vertices from this color class to the set of uncolored vertices (so that the partial coloring remains legal). Morgenstern used this idea, together with a procedure for the recombination of the solutions, to define a simulated annealing algorithm. This quite complicated technique made it possible to improve the best known results on some large graphs. Futhermore, these results are one of the best known at this time (accoring to the results published on website of Porumbel [46]).

Another well-known local search approach is the *tabu search*, originally proposed by Glover [25] and Hansen [26]. The idea of tabu search is that after moving from a solution s to its neighbour s', we store the solution s, or generally its attributes, in the tabu-list. We then forbid to re-visit this configuration (or similar ones, according to the attributes chosen) during next t iterations. The number t is called the *tabu tenure*. However, some algorithms propose that taking a tabu move is possible if the new solution s' is better than the best solution found so far (according to the solution evaluation function).

One of the most efficient tabu-search adaptations (according to [3]) is the *TABUCOL* algorithm proposed by Hertz and de Werra in [28]. The algorithm works with non-legal colorings and uses only a fixed number of colors. A move consists of recoloring a conflicting vertex while trying to minimize the total conflict of the graph.

An interesting tabu-seach algorithm was proposed by Blöchliger and Zufferey [3]. They use the *reactive tabu tenure* idea which was first introduced by Battiti and Tecchiolli [2]. The idea of this approach is that the value of the tabu tenure is changing during the computation. The algorithm combines the reactive tabu tenure with impasse class neighbourhood approach and provides very competitive results on a set of benchmark graphs which are known to be difficult. Moreover, the results are one of the best known at time (according to an overview website of Porumbel [46]).

The tabu search approach is also widely used in combination with other heuristics, such as genetic algorithms (more details in Section 2.2.4), but also some ACO algorithms or many others. The idea of tabu list can be easily embedded into various approaches.

There exist a number of other local search methods, for example the *variable neighbourhood search* proposed by Mladenovic and Hansen [39], Zufferey [43] or Avanthay et al. [1], or the ACO algorithms which are the focus of our thesis and will be described in Section 2.3. Futhermore, many algorithms are a combination of different approaches, for example the hybrid evolutionary algoritms described in the following section.

2.2.4 Genetic and Hybrid evolutionary algorithms

The genetic heuristics are algorithms inspired by evolutionary biology. The algorithms use techniques like inheritance, mutation, selection and crossover.

First of all, an initial population of solutions is constructed. A new solution is then built from the existing ones by selecting a pair of *parent* solutions, recombining them by *crossover* operator and then letting the offspring *mutate*. After this, the offspring is included into the population, where it replaces the worst solution according to an evaluation function.

Therefore, a genetic algorithm must define the way to select the parents, the crossover and the mutation procedure, and the evaluation function that represents the value of a considered solution.

A pure genetic algorithm for the GCP was proposed by Davis [14]. The algorithm encoded a solution as a permutation of the vertices, which are then colored through the SEQ algorithm. However, from the experiments followed that pure genetic methods do not produce satisfactory results for the GCP, since they are not able to capture the specificity of the problem.

Genetic algorithms (GA) can be extended by using the local search heuristics in the mutation phase, leading to *hybrid evolutionary algorithms* (HEA). Such an algorithm is essentially based on two key elements: an efficient local search (LS) operator and a highly specialized crossover operator. The idea is that the crossover operator produces new and potentially interesting configurations, which are then futher improved by the local search heuristic.

One of the most popular local search heuristic in the HEA algorithms is the tabu search (described in Section 2.2.3). Tabu search heuristic has been used in the algorithms proposed by Fleurent and Ferland [21] or Galinier and Hao in [22]. While Fleurent and Ferland use a simple crossover operator, Galinier and Hao introduced more specialized operator and therefore obtained better results. Their crossover operator works with whole color classes instead of color assignments of individual vertices. The offspring is created by iteratively taking the largest color class from the considered parent (the parents alternate) and assigning it to the new offspring. The vertices from this color class are then removed from both parents.

One of the most recent HEA algorithms was proposed by Malaguti et al. in [34]. Their local search algorithm combines the impasse class neighbourhood with the tabu search. The crossover operator is a variation of the one proposed by Galinier and Hao in [22]. The algorithm itself is very complex and a part of it solves the Set Covering Problem, but it gives excellent results on difficult graph instances, many of which are the best known at time (according to the overview website of Porumbel [46]).

Another heuristics is the *adaptive memory algorithm* proposed by Rochat and Taillard in [41], Eiben et al. [20] or Zufferey et al. [29]. Unlike the GA (and HEA) algorithms that use a population of solutions and perform crossovers, the adaptive memory algorithm stores portions of solutions in a central memory, and recombines them in order to build new solutions. In addition, the algorithm uses an LS operator, as it is the case of the HEA algorithm.

There are also many other heuristics for the Graph Coloring Problem, more information can be found at [48].

In the following section, we will focus on a special class of heuristics—the Ant Colony Optimization heuristics.

2.3 Ant Colony Optimization

The Ant Colony Optimization (ACO) is a family of probabilistic heuristics for solving computationally hard problems that is inspired by ants and their ability to collectively solve problems.

In nature, when ants are searching for food, at first they are moving randomly. However, if an ant finds a food source, it moves back to the nest laying down a pheromone trail marking the path to food. If another ant encounters such a path, it more likely follows the trail instead of continuing in random movement. The stronger the trail, the more the ant wants to follow this path. As more ants follow the same path, more trail is produced, so the path becomes even stronger.

On the other hand, the trail evaporates during the process. Therefore, if the path is too long, it takes longer to reinforce the trail. Shorter path gets marched over faster, so the trail is reinforced more often and the path becomes more attractive. This process helps the ants to find the shortest path to the food source.

The idea of ACO algorithms is the simulation of a colony of ants, walking on the graph, solving the given problem, using the trail system for communication with each other. The ants do not have to search for the shortest path anymore, the trail only represents how good the solution is.

Therefore—although the ACO algorithms can be very various—most of them follow similar principles: The algorithms create solutions step by step, using the colony of ants for the construction. Each ant produces a pheromone *trail*, which works as a “central memory” of the colony. The better the solution is, the more trail will be produced. The trail is evaporating during the process, so a newer trail is stronger than an old one. The ants then make decisions according to this trail, and the *greedy force*, which is a short term profit for the considered ant.

First ACO algorithm was proposed by Dorigo et al. [15] [16], as a heuristic for the Travelling Salesman Problem. Since then, the ACO algorithms have been widely used in many combinatorial problems. More information is provided for example by Dorigo and Stützle [17].

The first ACO algorithm for the GCP was proposed by Costa and Hertz [12]. In their algorithm, pheromones are used to indicate the desirability of assigning the same color to two different vertices. For ants' solution construction, they adapted the heuristics used in two well-known constructive algorithms, the DSATUR heuristics [4] and the Recursive Largest First (RLF) heuristic [33] (both heuristics are described in Section 2.2.2). Each ant colors the entire graph using a modification of one of these algorithms and then the trail is updated.

The authors experimentally compared eight variants of their ACO algorithm on a set of randomly generated GCP instances with up to 300 vertices. With good parameter settings, all the considered variants significantly improved over the underlying DSATUR and RLF heuristics. But unfortunately, the performance of this ACO algorithm, in general, appeared to be worse than other good graph coloring algorithms such as that of Fleurent and Fernand [21] or Costa et al. [13] which use the GA and HEA heuristics (described in Section 2.2.4).

After Costa and Hertz, a new view on the problem was proposed by Commelas and Ozón. In their paper [10] the authors propose a graph coloring algorithm for assignment problems in radio networks, in which the ants do not produce the whole solution, but affect only a small portion of the graph instead. This algorithm outperformed classical methods used in that time, such as simulated annealing or genetic algorithms.

They futher extended the algorithm for the original GCP in [11]. In the presented algorithm, the ants color one vertex at a time and do not use any trail system. The ants are moving from one vertex to another using the edges of the graph, therefore better simulating the natural ants, and trying to minimize the current conflict of the vertex. As the ants perform only small changes to the coloring, it may happen that the algorithm gets stuck with a coloring that cannot be improved by such a small change, but which is however still far from optimal. Such a coloring is called a *local optimum.* To escape from a local optimum, a probabilistic feature is used. When the ants move from one vertex to another, with certain probability p_v they pick the most suitable vertex, otherwise they pick an adjacent vertex at random. When picking the color for a given vertex, a similar probabilistic feature is applied. The algorithm outperformed the heuristic proposed by Costa and Hertz.

The idea by Comellas and Ozón was futher developed by Patel and Bui in [7]. They modified the way the ants choose a color for the vertex they are coloring, and the way the ants move from one vertex to another. The probabitistic feature was avoided and another ideas were introduced instead. The authors included the tabu mechanism in a way that ants are avoiding vertices that they have visited earlier. To escape the local optimum, they use a perturbation procedure that randomly recolors a given subset of the vertices. This algorithm was futher enhanced by Bui, Nguyen, Patel and Phan [6].

Another view on the problem was proposed by Hertz and Zufferey in [30]. In their algorithm they use several ants of different colors. They move these ants around the graph by switching two ants of different colors on two different vertices. The coloring procedure is derived from DSATUR with a modification, that a vertex can only get a color that is represented by at least one ant on it.

Another ACO heuristic was indroduced by Dowsland and Thompson in [18]. They build on the algorithm proposed by Costa and Hertz [12], proposing a number of enhancements and modifications for better performance. In their work they focused on the examination scheduling problem and therefore only on a particular class of graphs. They futher enhanced their algorithm and proposed more modifications for the general Graph Coloring Problem in [19]. The modifications included strenghtening the construction phase, improving the trail system and introducing the local improvement function based on the tabu search (described in Section 2.2.3).

There are many other ACO algorithms for the problems related to GCP, such as scheduling or radio assignment. We would also like to remark that ACO algorithms are not strictly defined. Therefore, many other algorithms that work in a similar way (i.e., greedy force combined with the central memory) can be taken as ant colony optimizations, despite the fact that they do not work with ants or a trail.

In the following chapter we will describe in detail three algorithms we selected for implementation in our program.

Chapter 3

Implemented algorithms

Out of the existing ACO algorithms for the GCP, we chose those of Bui et al. [6], Hertz and Zufferey [30] and Dowsland and Thompson [19]. We tried to choose such algorithms that work in a different way, so that we could better observe their advantages and their drawbacks.

3.1 Overview

The implemented algorithms have some similar principles. We are searching for legal k-colorings, decreasing the value of k as much as possible. When trying to find a legal k-coloring, we are dealing with infeasible solutions, such that have conflicts, and are trying to minimize the number of conflicts.

At the beginning of the algorithm, the upper bound of the chromatic number is computed and the graph is colored initially using this number of colors. We store the initial legal coloring as the *bestColoring*, which is a legal coloring using best number of colors we found so far. The initial number of colors is stored as *bestNumColors*, which is the best number of colors we found a legal coloring with. The value of *availableColors*, which corresponds to the value of k when searching for a k-coloring, is then initialized.

The algorithms run in cycles. In each cycle a portion of the graph is recolored using only *availableColors* colors. If the total conflict of the graph is lower than the value of *bestConflict*, which is the best value of conflict we found so far (using current number of colors), the *bestConflict* is updated.

When the total conflict of the graph is equal to zero and we used less colors than *bestNumColors*, we store this coloring as the *bestColor-*

ing, update the value of *bestNumColors* and decrease the value of *availableColors*. The algorithms run for a predefined number of cycles, or a predefined number of cycles during which no improvement was done.

3.2 Algorithm of Bui, Nguyen, Patel and Phan

The first algorithm is an implementation of the algorithm presented by Bui et al. in [6]. The algorithm is specific in the way it does not use any trail system—the authors suggest that this helps reduce the running time and in their set of experiments they did not found a significant effect on the quality of solution when using the trail. The algorithm is also specific in the way it uses the ants. Each ant does not build the whole solution, but colors only a portion of the graph close to the vertex the ant is on. Therefore, the algorithm is suitable for distributed implementation.

In the following we provide a short description of the algorithm. More technical details as well as exact parameter settings are specified in [6]. As some parts of the algorithm were described unclear, we provide the description as we understand it. This problem is addressed in Chapter 6.

3.2.1 Overview

Let G be the input graph. At first, we find a legal coloring of G using a heuristic called *MXRLF*. Let k be the number of colors we used. Note that k is an upper bound on the chromatic number of G. An initial coloring of G, which may not be a legal coloring, is then derived from this legal k-coloring. We then randomly distribute a colony of ants on the vertices of the graph. The algorithm then proceeds in a number of cycles, *nCycles*.

In each cycle, every ant colors the vertex it is at, using only the set of currently available colors, and moves to another vertex. Each ant does *nMoves* such moves. Then the next ant continues. In addition, every ant has a list of vertices it cannot revisit during next cycles.

At the end of a cycle, if there are no conflicts in the current coloring, we save current coloring as the *bestColoring*, update the value of *bestNumColors*, the number of available colors is reduced by one and we recolor all vertices having color with value larger than *availableColors*. Then we start another cycle.

If the value of *availableColors* has not been changed for the last *nChangeCycles*, we increment *availableColors* by 1 before starting another cycle. If the value of *availableColors* has not been improved in last *nJoltCycles*, we perform *Jolt* (described in Section 3.2.5).

The algorithm stops after *nCycles* cycles or after *nBreakCycles* cycles during which no improvement of *bestNumColors* was done.

3.2.2 Initial coloring

We now descibe the *MXRLF* algorithm. The algorithm runs in cycles. In every cycle it colors a set of vertices with current color. The algorithm stops when all vertices are colored. Let c be the color we are using this cycle. Let P be the set of uncolored vertices not adjacent to any vertices colored with color c and let R be the set of uncolored vertices adjacent to at least one vertex colored with color c. *MXRLF* avoids using vertices from R while building a color class. The first vertex to add to the color class is the one with the maximum degree. Then we add the next vertex to the current color class from P having the maximum degree in R. Ties are broken by selecting the vertex with the minimum degree in P. The process is repeated until P is empty or the color class size limit, *MXRLF_SET_LIMIT*, is exceeded. We then increase the value of current color c and repeat the above until the entire graph is colored.

Let k be the number of color classes produced by *MXRLF* and let α, β and γ be parameters such that $0 < \beta \leq \gamma \leq \alpha < 1$. The initial number of colors available to the ants for coloring the graph, i.e., *availableColors*, is set to $\lceil \alpha \cdot k \rceil$. From the k color classes produced by *MXRLF* we select at random $\lceil \beta \cdot k \rceil$ color classes to be kept. The remaining vertices that do not belong to the selected color classes are then distributed randomly into $\lceil \gamma \cdot k \rceil$ color classes. These $\lceil \gamma \cdot k \rceil$ color classes include the $\lceil \beta \cdot k \rceil$ color classes selected earlier. The color classes are renumbered so that all colors are in the set $\{1, ..., \lceil \gamma \cdot k \rceil\}$. This coloring is then used as a starting point for the ants. To summarize, we now have a coloring of G having $\lceil \gamma \cdot k \rceil$ colors. Note that this coloring may not be a legal coloring of G. We also have a total of $\lceil \alpha \cdot k \rceil$ colors available for the ants to use initially.

3.2.3 How ants color

At the beginning of the algorithm, we distribute a colony of *nAnts* ants randomly to the vertices of the graph. The algorithm consists of a number of cycles. In each cycle ants are activated one at a time. When activated, an ant colors several vertices of the graph using only colors from the set of available colors, i.e., the set $\{1, ..., availableColors\}$.

When an ant is at a vertex v, its objective is to color or re-color v so that the conflict at v is zero, if possible. When there is a choice among several available colors that make conflict zero, the ant just picks one at random. If zero conflict is not possible, the ant will select the color with the smallest number from the list of available colors that will minimize the conflict at v. Furthermore, if zero conflict is not possible, the ant will try to select a color that it has not used in the previous location, i.e., this color can be used only if it is the only color minimizing the conflict. The conflict at this vertex (and its neighbourhood) is then updated.

After an ant finishes coloring its current vertex, it moves to another vertex and tries to color it. The ant moves to another vertex by taking a path of length two. The first edge in that path is selected at random among all edges incident with its current vertex. The second edge in the path is selected so that the ant will end up in a vertex that has the maximum conflict among all vertices adjacent to the vertex at the end of the first edge. Ties are broken arbitrarily. Additionally, each ant also has a tabu list containing *nTabu* last visited vertices that it cannot revisit. Each ant will make *nMoves* such moves before it stops.

3.2.4 Main loop

When all the ants finish coloring, at the end of a cycle we have a coloring of G. We then compute the total conflict of the current coloring and number of colors used in that coloring (which would be *availableColors*). If the total conflict of the current coloring is zero and we used less colors than *bestNumColors*, we save this coloring as the *bestColoring*, update the value of *bestNumColors* and reduce the value of *availableColors* by 1. We must also recolor the vertices having color with value larger than *availableColors*. This recoloring is not specified by the authors of the algorithm, so we propose to recolor these vertices to the color with value *availableColors*. Although this makes additional conflicts, they are quickly descreased by the ants.

If *availableColors* has not been changed for the last *nChangeCycles*, we increment *availableColors* by 1. If the value of *availableColors* has not been improved in last *nJoltCycles*, we perform *Jolt*.

The algorithm stops after it has run for *nCycles* cycles, or if it has not made any improvement of *bestNumColors* for a number of *nBreakCycles* consecutive cycles.

3.2.5 Jolt

To assist ants in escaping local optima, we can perturb the current coloring of the graph by a method called *Jolt*. The idea of the *Jolt* is to inject enough disturbances into the current coloring to move it out of the current local optimum but not enough to destroy the coloring that has been built up to that point. More specifically, if there is no reduction in the number of colors used for the last *nJoltCycles* cycles, then the current coloring is perturbed as follows. The vertices in the graph that have conflicts in the top 10% are selected and their neighbors are randomly re-colored using 80% of the current set of available colors.

3.3 Algorithm of Hertz and Zufferey

This algorithm is an implementation of the heuristic proposed by Hertz and Zufferey [30]. Unfortunately, the authors did not describe some parts of their algorithm in detail, so we had to choose which way the algorithm should be implemented. Although we did our best in trying different variants of the algorithm, our implementation does not perform as well as described in the paper. We address this in Chapter 6.

The algorithm was originally created for the k-coloring problem, but it can be used for the original GCP as well. We modified the algorithm as proposed in the paper: at first, we compute the upper bound on the chromatic number of the graph and then we solve the k-coloring problems, decreasing the value of k every time we find a legal coloring.

In the following we provide a short description of the algorithm. More technical details as well as exact parameter settings are specified in [30].

3.3.1 Overview

Let G be the input graph. At the beginning, we compute k, the upper bound of chromatic number, with the SEQ procedure (described in Section 2.2.2), and color the graph legally using k colors. Then we set the number of colors available to ants, *availableColors*, to $k - 1$.

Algorithm then proceeds in stages. At the beginning of the stage, one ant of every color from 1 to *availableColors* is placed on every vertex. Then we perform the following in cycles until legal coloring is found.

We exchange several ants between vertices (we call this exchange a *move*). We then apply the coloring procedure to color a subset of vertices so that each vertex gets a color that is represented by at least one ant on it.

The stage ends when a legal coloring is found. We then update the value of *bestColoring*, decrease *availableColors* by 1 and start another stage.

The algorithm stops after *maxIteration* cycles during which no improvement of *bestConflict* was done.

3.3.2 Main Loop

By A we denote the set of vertices to be colored at current iteration t. At the beginning of the stage, we place one ant of each valid color on every vertex, set $A = V$ and color set A.

Then we proceed in cycles until legal coloring is found. At the beginning of the cycle, we make several *moves* that change the positions of the ants on the graph and, at the end of each cycle, we recolor a subset $A \subseteq V$ of vertices, using the *Color* procedure. The subset to be colored at iteration t is the following:

$$\begin{aligned} A = &\{\text{vertices involved in a move at iteration}\, t\} \bigcup \\ &\bigcup \{\text{conflicting vertices at iteration}\, t-1\}. \end{aligned}$$

3.3.3 Coloring procedure

The coloring procedure *Color* is the following:

Let A be the set of vertices we want to color. At first, we "decolor" these vertices—set their color as uncolored. The remaining vertices, i.e., the vertices in set $V \setminus A$, keep their colors. The coloring procedure then proceeds in steps. At each step, the procedure colors one vertex from A as follows:

1. select the vertex $x \in A$ with the largest saturation degree, which is the number of different colors that are adjacent to x; if more than one vertex maximize the saturation degree, the vertex with the largest number of adjacent vertices is chosen (ties are broken randomly);

2. assign color to x and remove x from A. The color assigned must be represented by at least one ant on x, and among these colors, we choose the one that minimizes the number of conflicts. If several

colors are possible, we give to x the color which is represented by the most ants on x (ties are broken randomly).

3.3.4 Move

We now describe the move that is to be taken in the current iteration t.

Let a_i and a_j be two ants with colors i and j. Suppose that a_i is on vertex x and a_j is on vertex y. A move $m = (x, i) \longleftrightarrow (y, j)$ consists of exchanging the positions of a_i and a_j on vertices x and y. Thus, a_i is moved from x to y and a_j is moved from y to x. At each iteration t of the algorithm, we modify the distribution of the ants on the graph by performing a sequence of such moves.

Our goal is to change the color of at least one of conflicting vertices. Let v be such vertex (the choice of v will be described later). Let $N_i(v, t-1)$ be the number of ants of color i on v at iteration $t-1$. Suppose v has color c at iteration $t-1$, thus $N_c(v, t-1) > 0$ (otherwise the *Color* procedure would not be able to assign color c to v at the end of iteration $t-1$). In order to be sure that we change the color of v using the Color procedure at the end of iteration t, we have to remove all ants of color c from v.

In order to reduce the risk of cycling, when we remove all ants of color c from vertex v, we forbid to put an ant of color c on v during *tab* iterations, where $tab = UNIFORM(0, 9) + 0,6 \cdot NCV(s)$, where $UNIFORM(a, b)$ is random integer between a and b inclusive, s is the current solution and $NCV(s)$ is the number of conflicting vertices in s.

At first, we define the multiset of all possible moves at iteration t as:

$$\begin{aligned}M(t) = \{m = (x, i) \longleftrightarrow (y, j), \text{ such that:}\\ &x \text{ is a conflicting vertex, } i \text{ is the color of } x,\\ &y \neq x,\ N_j(y, t-1) > 0,\ j \neq i,\ m \textit{ is not forbidden}\},\end{aligned}$$

where move m is represented in M $min(N_i(x, t-1), N_j(y, t-1))$ times.

Now we define the *probability* of the move. The *probability* is defined as follows:

$$p(m, t) = \alpha \cdot GF(m, t) + \beta \cdot Tr(m, t),$$

where $GF(m,t)$ means the *greedy force* of the move, $Tr(m,t)$ means the *trail*, and α and β are the parameters that control the balance between the influence of the trail and greedy force.

Let $m' \in M$ be the move that maximizes the value $p(m,t)$ (ties are broken randomly). The vertex v, whose color we want to change, is then chosen as the argument x of m'. Let c be the color of the vertex v.

So, at the iteration t we perform a sequence of $N_c(v,t-1)$ moves maximizing $p(m,t)$ chosen in the multiset:

$$\begin{aligned} M_v(t) = \{m = (v,c) \longleftrightarrow (y,j) :\ & y \neq v,\ N_j(y,t-1) > 0, \\ & j \neq c,\ m \textit{ is not forbidden}\}, \end{aligned}$$

because every move decreases the number of ants of color c on v by 1. Move m is represented in M_v $min(N_c(v,t-1), N_j(y,t-1))$ times.

3.3.5 Definition of the greedy force

We now describe the way the greedy force is computed for a given move.

The greedy force represents the short-term profit of a single ant. At each iteration, the short term profit consists of removing some conflicts. In order to remove a conflict, we should remove some *ant-conflicts*, where an *ant-conflict* occurs when two ants of the same color i are placed on two adjacent vertices x and y. In this case, the *Color* procedure may give the same color i to x and y.

Suppose we aim to change the current color i of vertex x. We thus have to remove all the ants of color i from vertex x. In order to do that, we perform a sequence of moves of type $m = (x,i) \longleftrightarrow (y,j)$. For such a move m, we define the greedy force $GF(m,t)$ by setting

$$GF(m,t) = Adv(m,t) - Disadv(m,t),$$

where $Adv(m,t)$ and $Disadv(m,t)$ are respectively the advantage and disadvantage of performing move m (i.e. exchanging ants a_i and a_j) at iteration t. We always normalize the $GF(m,t)$ in interval $[0,1]$.

Let $S(x,y,i,t)$ be the number of ants of color i on vertices, different from y, which are adjacent to x at iteration t. Then, we set

$$Adv(m,t) = N_i^2(y,t-1) + S(x,y,i,t-1) + N_j^2(x,t-1) + S(y,x,j,t-1),$$

and

$$Disadv(m,t) = N_j^2(y,t-1) + S(x,y,j,t-1) + S(y,x,i,t-1),$$

and we normalize these values in interval $[0,1]$.

3.3.6 Definition of the trail system

We now decribe how to compute the trail of a given move.

At first, we define the way to update the trail left by ants of color i on vertex x at the end of iteration t:

$$tr(x,i,t) = \rho \cdot tr(x,i,t-1) + \Delta tr(x,i,t),$$

where $0 < \rho < 1$ is an evaporation parameter and $\Delta tr(x,i,t)$ is the reinforcement value of color i on vertex x.

We consider several cases. Suppose that during iteration t, the following events occurred:

1. v loses $r = N_c(v,t-1)$ ants of color c;
2. we put r ants on v, and the r ants have colors in the subset $C(v)$ of $\{1,...,k\}$;
3. the r new ants on v came from a subset of vertices $R(v)$ of V.

We want the trail system to incorporate the following information:

1. if an ant of color i leaves vertex x, then $tr(x,i,t)$ should be evaporated (this is especially true for $x = v$ and $i = c$) and not reinforced;
2. if no ant of color i leaves a vertex $x \in R(v)$, then $tr(x,i,t)$ should be evaporated and slightly reinforced;
3. if an ant of color i arrives on vertex x, then $tr(x,i,t)$ should be evaporated and reinforced, and the reinforcement should be especially large if the generated solution s' from s is better than s, or better than $s*$, which is the best solution visited so far during the search process;
4. if nothing occurs relatively to a vertex x and a color i, then $tr(x,i,t)$ should be slightly evaporated and not reinforced.

Therefore, we consider three values of evaporation factor, and three values of reinforcement factor:

- ρ_1 - strong evaporation factor of the trail
- ρ_2 - medium evaporation factor of the trail
- ρ_3 - light evaporation factor of the trail
- δ_1 - light reinforcement factor of the trail
- δ_2 - medium reinforcement factor of the trail
- δ_3 - strong reinforcement factor of the trail

The authors of [30] suggest the following values of evaporation and reinforcement parameters for modifying the trail of color i on vertex x.

- $\rho = \rho_2$ and $\Delta tr(x, i, t) = 2\delta$ if $x = v$, $i \in C(v)$;
- $\rho = \rho_1$ and $\Delta tr(x, i, t) = 0$ if $x = v$, $i = c$;
- $\rho = \rho_2$ and $\Delta tr(x, i, t) = 0$ if $x = v$, $i \notin C(v) \bigcup \{c\}$;
- $\rho = \rho_2$ and $\Delta tr(x, i, t) = 0$ if $x \in R(v)$,
  i such that there is an ant a_i which left x;
- $\rho = \rho_2$ and $\Delta tr(x, i, t) = \delta$ if $x \in R(v)$ and $i = c$;
- $\rho = \rho_2$ and $\Delta tr(x, i, t) = \delta/2$ if $x \in R(v)$,
  i such that there is no ant a_i which left x;
- $\rho = \rho_3$ and $\Delta tr(x, i, t) = 0$ if $x \notin \{v\} \bigcup R(v)$ and $i \in \{1; ...; k\}$;

where

$$\delta = \begin{cases} \delta_1 & \text{if } f(s) - f(s') \geq 0, \\ \delta_2 & \text{if } f(s) - f(s') < 0, \\ \delta_3 & \text{if } f(s) - f(s*) < 0. \end{cases}$$

$f(s)$ denotes the number of conflicts in solution s.
We can now define the trail of a move m at iteration t as follows:

$$\begin{aligned} Tr[m = (x, i) \longleftrightarrow (y, j), t] = {} & tr(x, j, t-1) + tr(y, i, t-1) - \\ & - tr(x, i, t-1) - tr(y, j, t-1). \end{aligned}$$

We reset $tr(x, i, 0) = 1$, $\forall x \in \{1, ..., n\}$, $\forall i \in \{1, ..., k\}$ everytime a legal coloring is found, and we always normalize the tr and Tr values in interval $[0, 1]$.

3.4 Algorithm of Dowsland and Thompson

This algorithm is an implementation of the heuristic proposed by Dowsland and Thompson in [19]. The algorithm is an improvement of the original algorithm created by Costa and Hertz [12].

The authors proposed multiple variants of the algorithm; therefore, for our implementation we chosed the one that came out best in their set of tests.

The algorithm is specific in the way the ants are not represented as individual entities that "color" the vertices (like in previous algorithms), but are just constructive heuristics instead. The algorithm only uses the parameter *nAnts* to determine the number of solutions it makes during each cycle.

The trail system is inspired by the algorithm of Costa and Hertz [12]: As the quality of the coloring depends on the sets of vertices that are placed in the same color class, as opposed to the actual color allocated to each individual vertex, the trail is used to determine which vertices should be in the same color class.

In the following we provide a short description of the algorithm. More technical details as well as exact parameter settings are specified in [19].

3.4.1 Overview

Let $G = (V, E)$ be the input graph. At the beginning, the value of *availableColors* and *bestNumColors* is initiated to $|V|$. The algorithm then proceeds in cycles.

In each cycle, we sequentially create up to *availableColors* legal color classes, potentially leaving some vertices uncolored. While creating a color class, we make *nAttempts* attempts of choosing the set of the vertices to be colored and use the one resulting in the lowest number of edges in the graph induced by the remaining uncolored vertices. Let *nUsedColors* be the number of legal color classes we created. We distribute the uncolored vertices into these color classes, potentially creating some conflicts. Then we apply the local search optimalization, *Tabu search operator*, that recolors some of the vertices trying to minimize the total conflict of G. In the end, we compute the evaluation of current coloring. In one cycle, we create *nAnts* such solutions.

If we found a legal coloring using less colors than *bestNumColors*, we store this coloring as the *bestColoring* and update the value of *bestNumColors*.

At the end of the cycle, the trail is updated according to the evaluations of produced colorings. If in this cycle a legal coloring was found, the value of *availableColors* is decreased to *bestNumColors* -1 and another round begins. The algorithm stops after *nCycles* cycles.

3.4.2 Creating the color classes

In the algorithm, color classes are built up one at a time as maximal independent sets. When building a color class of color k, we maintain a set W of vertices that are still uncolored and non-adjacent to any vertex of color k. We then proceed in stages until W is empty. In each stage we select one vertex from W to be colored with color k. A vertex i is selected with probability

$$P_{ik} = \frac{\tau_{ik}^{\alpha}\eta_{ik}^{\beta}}{\sum_{j \in W} \tau_{jk}^{\alpha}\eta_{jk}^{\beta}} \quad \text{if}\, i \in W$$
$$P_{ik} = 0 \quad \text{otherwise.}$$

where η_{ik} means the *visibility* of vertex i and color k, τ_{ik} means the *trail* of color k on vertex i and α and β are parameters that control the balance between the influence of the trail and visibility.

As the probability depends on the vertices already colored with the current color, the first vertex in the color class is chosen randomly from the feasible vertices.

When creating a color class, we make several attempts and select the one that results in the minimum number of edges in the graph induced by the remaining uncolored vertices. Number of attempts is given by parameter *nAttempts*.

3.4.3 Visibility

Let W be the set of uncolored vertices that can be included in the current color class k and $deg_X(i)$ degree of vertex i in the subgraph induced by the subset of vertices, X.

The visibility is then defined as:

$$\eta_{ik} = |W| - deg_W(i).$$

3.4.4 Trail part

As the quality of the coloring depends on the sets of vertices that are placed in the same color class, the trail matrix is defined in terms of vertex pairs. Let t_{ij} be the trail between vertices i and j.

In the first cycle the trail between each pair of non-adjacent vertices is set equal to 1. At the end of each cycle the trail matrix is updated according to

$$t_{ij} = \rho \cdot t_{ij} + \sum_{\forall a} f(s_a),$$

where ρ is the evaporating factor parameter, $f(s)$ is an evaluation function that rates the quality of a solution and s_a is a solution produced by "ant" a. The evaluation function is defined as follows:

$$f(s) = \frac{1}{\sum \textit{conflicting edges}}$$

when vertices i and j are in the same color class;

$$f(s) = 0 \quad \text{otherwise.}$$

As the authors do not specify the value if $\sum \textit{conflicting edges} = 0$, we propose the following:

$$f(s) = 1.$$

During the construction process the trail associated with coloring vertex i in color k depends on the set of vertices already colored k, denoted V_k, and is given by

$$\tau_{ik} = \frac{\sum_{j \in V_k} t_{ij}}{|V_k|}.$$

3.4.5 Distribution of uncolored vertices

When the algorithm has created r color classes as independent sets (as described in Section 3.4.2), we distribute the uncolored vertices into the color classes as follows:

Let C_1 to C_r be the sets of vertices in the r color classes and let U be the set of uncolored vertices. For $u \in U$ and $k = \{1, ..., r\}$ we define $e(k, u)$ to be the number of vertices in color class C_k that clash with u, i.e., $e(k, u) = |\Gamma(u) \bigcap C_k|$, where Γ denotes the set of neighbouring vertices. For each $u \in U$ in turn we place u in that C_k that minimises $e(k, u)$ and update $e(k, v)$ for any $v \in U$ still unassigned.

In distribution process, the vertices are taken in any order.

3.4.6 Tabu search operator

This local search algorithm consists of making *nLocalCycles* valid moves, one at an iteration. A move is defined as recoloring a vertex. A valid move is a move recoloring a conflicting vertex.

Every vertex has a tabu-list of colors it cannot be recolored to. The tabu-list consists of *(vertex,color)* pairs, and is updated as follows: After recoloring a vertex, we forbid every move returning vertex i to a previously allocated color class C_k for tl iterations, where the tabu tenure, tl, is a random variable defined as follows:

$$tl = \mathit{UNIFORM}(\mathit{10}) + 0.6 \cdot N_{cand},$$

where N_{cand} is the number of conflicting vertices in the current iteration, and *UNIFORM(*10*)* is a random integer between 1 and 10 inclusive.

At every iteration of local search, we make one valid move that minimizes the total conflict of the solution (ties are broken randomly). In addition, we accept any improving solution as soon as we find it, i.e., if we find a move that would improve the current total conflict (and the move is not tabu) we make it and continue to next iteration. Moreover, if we find a valid move that improves the best total conflict we found during the local search, we make it even if the move was tabu.

Chapter 4

Our Algorithm

In this chapter we describe the principles of our ACO algorithm. We were inspired by the algorithm of Bui et al. in the way the ants color only a small portion of the graph. But unlike this algorithm, our ants move using the anti-edges, i.e., the ants can move only to non-adjacent vertices. We also use the trail system, which affects the ants' decision when moving to another vertex.

The coloring mechanism is also different from the mechanisms mentioned in Chapter 3. Our ants color a vertex only if the recoloring does not increase the conflict. The local optimum is escaped by performing a perturbation procedure.

Moreover, our colony consists of different ants. Each ant has its own color and can color the vertex only with this color. The ants of different colors are independent, and are not affecting each other.

4.1 Overview

As in the algorithms described in Chapter 3, we also use variables *bestColoring*, *bestNumColors*, *availableColors* and *bestConflict* with the same meaning.

Let $G = (V, E)$ be the input graph. At the beginning of the computation, the graph is colored legally using the *MXRLF* procedure introduced by Bui et al. [6]. Let k be the number of colors used in this coloring. The value of *bestColoring* is then set to this coloring, the value of *bestNumColors* is set to k, and the value of *availableColors* is set to $k - 1$.

At the beginning of the algorithm, the colony of ants is randomly distributed on the graph. The colony consists of *nAnts* ants of each color from the set $\{1,...,\mathit{availableColors}\}$.

The algorithm runs in cycles. In each cycle, the ants are activated one at time so that the colors of the ants alternate (first all the first ants of each color, then all the second ants of each color, etc.). When activated, an ant makes a move to a non-adjacent vertex, leaves a trail at that vertex and potentially changes its color to its own color (but not necessarily).

After an ant finishes coloring, the total conflict of the graph is computed and the value of *bestConflict* is updated. If the total conflict is equal to zero, a legal coloring has been found. We save current coloring as the *bestColoring*, update the value of *bestNumColors*, decrease the number of *availableColors* by 1 and recolor the vertices having unavailable color.

The recoloring is a simple greedy procedure. We take the vertices in initial order and assign each vertex the lowest number of color that minimizes its conflict.

The ants of non-available color are then removed from the colony and the rest is randomly distributed on the graph.

We then remove the trails of non-available colors from the vertices. The rest of the trails is left unchanged.

Otherwise, if there has been no change in the value of *bestConflict* in last *nJoltCycles*, we perform *ModJolt*, a modified version of *Jolt* described in Section 3.2.5. This procedure randomly recolors a portion of the graph to assist the ants to escape the local optimum.

The *ModJolt* procedure works in a similar way as *Jolt*. We randomly recolor 10% of the neighbours of current conflicting vertices (these also include the conflicting vertices themselves). To create a basis for a better coloring, we use only colors in range $\{1, ..., \lceil 0.8 \cdot \mathit{availableColors} \rceil\}$.

If there has been no improvement of the value of *bestConflict* for *maxIterations*, the algorithm terminates.

4.2 How ants move

In each cycle, an ant moves from its current place to a non-adjacent vertex. The target vertex is chosen according to its *attrativeness*. In addition, every ant has a tabu list of *nTabu* previously visited vertices it cannot revisit during this turn.

Let c be the color of the considered ant. The attractiveness of a vertex v for an ant of color c is computed as:

$$\mathit{Att}(v, c) = \alpha \cdot \mathit{GF}(v, c) + \beta \cdot t_{vc},$$

where $GF(v,c)$ is the value of the greedy force and t_{vc} is the value of trail of color c on vertex v, and α and β are parameters.

We want the ant to improve the conflict of the target vertex, therefore the greedy part of the move consists of computing the potential conflict of the vertex if it was colored with color c. Also, if the vertex already has color c and its conflict is greater than zero, the ant wants to avoid it.

The value of the greedy force is computed as follows: Let $con(v)$ be the conflict at vertex v, $con_c(v)$ the potential conflict at v if the vertex was colored with color c, and $deg(v)$ the degree of v.

If vertex v has color c and the conflict at this vertex is greater than zero, the value of greedy force is the following:

$$GF(v,c) = -\frac{con(v)}{deg(v)}$$

Otherwise, the greedy force is computed as follows:

$$GF(v,c) = \frac{con(v) - con_c(v)}{deg(v)}$$

The target vertex is then chosen randomly from the non-adjacent vertices satisfying the condition:

$$Att(v,c) \geq (maxAtt - minAtt) \cdot \gamma + minAtt,$$

where $minAtt$ is the minimum value of attractiveness among all non-adjacent, non-tabu vertices and $maxAtt$ is the maximum.

After an ant moves to the target vertex, it updates its tabu-list, where it maintains the last $nTabu$ visited vertices.

4.3 Modifying the trail and coloring the vertex

After an ant of color c comes to a vertex v, it modifies the value of the trail of color c there.

The trail system in our algorithm is different than other trail systems in a way that the ants can not only leave a positive pheromone trail, but can also have a "negative" influence on the trail—leave a "negative" trail. The purpose of this modification is to store the infomation about the disadvantage of making a move to a vertex for other ants of color c.

Let $con(v)$ be the conflict of vertex v, $con_c(v)$ the conflict the vertex v would have if it was colored with color c and let $deg(v)$ be the degree of v.

If the color c would make the conflict of v larger, the trail is updated as follows:

$$t_{vc} = t_{vc} - \frac{con_c(v)}{deg(v)}$$

Otherwise, the trail is updated as follows:

$$t_{vc} = t_{vc} + (con(v) - con_c(v))$$

At the end of the cycle the trail is evaporated and all values of trail are normalized into the interval $[0, 1]$. Let ρ be the evaporation parameter. The evaporatation is computed as follows:

$$t_{vc} = \rho \cdot t_{vc} \quad \forall v \forall c$$

Unlike the algorithm of Hertz and Zufferey mentioned in Chapter 3, we do not reset the trail system after finding a legal solution, we simply remove the trails of colors that are not available. This is because it proved useful not to destroy all the trail information gathered by the ants. We assume the evaporating factor is sufficient to supress the old trails left by ants.

After an ant of color c modifies the trail on a vertex v, it attempts to color it with color c.

The vertex is colored only if the color c would not make its conflict larger than it is. A neutral or a positive contribution is accepted.

4.4 Parameters

In this section, we specify the parameters used in the algorithm and propose their values according to set of tests we performed to tune them. In the following, $n = |V|$.

maxIteration The maximum number of cycles during which no improvement of the *bestConflict* was made, before the algorithm is terminated. We set $maxIteration = 5000$.

nAnts The number of ants of one color in the colony. We set $nAnts = min(0.1 \cdot n, 5)$. We have not observed better results with more ants, we prefer making more cycles instead.

α The parameter influencing the greedy force part of the move attractiveness value. We set $\alpha = 7$.

β The parameter influencing the trail part of the move attractiveness value. We set $\beta = 4$.

γ The treshold of the move attractiveness we are considering when making a move. We set $\gamma = 0.8$.

ρ The evaporation factor. We set $\rho = 0.8$.

MXRLF_SET_LIMIT The limit for the size of each color class in MXRLF. We set $\mathit{MXRLF\_SET\_LIMIT} = 0.7 \cdot n$.

nTabu The number of the recently visited vertices the ant cannot revisit. We set $nTabu = n/12$.

nJoltCycles The number of the consecutive cycles during which the value of *bestConflict* was not improved before the *ModJolt* procedure is done. We set $nJoltCycles = 500$.

4.5 Pseudocode

In this section we provide a pseudocode of our algorithm.

```
• create an initial legal coloring using the MXRLF procedure
    – bestColoring := this coloring
    – bestNumColors := number of colors used
    – bestConflict := this conflict
    – availableColors := number of colors used − 1
• create a colony of nAnts ants of colors {1, ..., availableColors}
• distribute ants on the vertices randomly
• iteration := 0
• noChangeCycles := 0
• while (iteration < maxIteration)
    – iteration := iteration + 1
    – noChangeCycles := noChangeCycles + 1
    – foreach $ant_i$ in colony
```

```
        * $ant_i$ move
        * $ant_i$ modify the trail
        * $ant_i$ color
        * if ($currentConflict < bestConflict$)
            · $bestConflict := currentConflict$
            · $bestNumColors := availableColors$
            · $noChangeCycles := 0$
            · $iteration := 0$
        * if ($currentConflict = 0$)
            · $bestColoring := currentColoring$
            · $availableColors := availableColors - 1$
            · recolor vertices having color out of range
            · $bestConflict := currentConflict$
            · remove non-available ants, distribute the rest randomly
              on the vertices
            · remove non-available trail from vertices
            · break
        * endfor
    – evaporate the trail
    – normalize the trail into the interval $[0, 1]$
    – if ($noChangeCycles > nJoltCycles$)
        * perform $ModJolt$
    – endwhile
• end
```

Chapter 5

The Coloring Ants Program

As a part of this thesis, algorithms described in Chapter 3 and Chapter 4 were implemented in a program called Coloring Ants.

The program was written in language C++ and uses the Qt graphic library. It can be compiled both on Linux and Windows.

The program is interactive and offers various visualisations of the run of the algorithm, which can be used for studying the principles and effectivity of these heuristics. The program has also a command line interface for batch processing.

In this chapter, we provide only a brief description of the program. More details can be found in the documentation on the enclosed CD.

5.1 The run of program

At first, the user loads a file with the graph he/she wants to color. The file must be in DIMACS graph format, see [44] or the user documentation on the enclosed CD. This format is text oriented and contains the vertex pairs representing the edges of the graph.

Then the user chooses the coloring algorithm. The program implements the three algorithms described in Chapter 3, as well as our algorithm described in Chapter 4.

The user then sets the parameters for the coloring algorithm and the visualisation settings. The parameters can be set to their default values proposed by the authors of the algorithm. The visualisation settings are described in Section 5.2.

The computation is started by pressing the button “Start”. The other buttons are used to control the computation—pause it, resume it or stop it. As the algorithms run in cycles, we also added the possibility to make the algorithm perform only one cycle and then pause. After each cycle the current information about the run of the computation is displayed.

The left information window provides overview information, i.e., the cycle the algorithm is in, the number of colors the ants are using, the current conflict or the best number of colors we found a legal coloring with so far.

The bottom log is for storing these overview data, so that the user could read the information from previous cycles.

In the center of the main window is the visualisation screen. After every iteration of the run of the algorithm, the program draws the current state of the graph to this screen. The screen shows a drawing of the graph, specified by visualisation parameters. The program can visualise the conflicts, the colors of the vertices or the trail relations.

By default, the edges of the graph are not visualised, as the visualisation is very time-consuming. But in the visualisation settings, the user can switch the edge visualisation on.

To observe the progress of the algorithm conveniently, it is necessary to delay it. The delay is applied after every cycle. The delay is measured in milliseconds.

The visualisation parameters can be changed when the computation is not running (is paused or stopped) and the change takes effect at the next cycle.

In the following section, we will describe possible visualisation modes in detail.

5.2 Visualisation modes

There are three different visualisation modes:

1. Conflict drawing mode

2. Color drawing mode

3. Trail drawing mode

In the conflict drawing mode, the color and the size of a vertex depends on its conflict. We distinguish 8 conflict categories and the main difference between vertices in different categories are colors. There is a specific color for each conflict category so that the more conflicting vertices are painted with brighter color. The vertices in different catogories have also different size, the bigger the conflict, the bigger the vertex.

The edges between conflicting vertices are painted red.

In the color drawing mode, the user can watch the actual colors of the vertices. This mode has most disadvantages. First, while coloring, the vertices often change their colors while the distribution of color classes remains practically unchanged. Second, when coloring large graphs with large chromatic number, it is hard to color vertices with that many colors in such a way that the colors would be still distinguishable by the user.

The colors are picked from the RGB cube according to particular rules.

In this mode, the edges between conflicting vertices are also painted red, like in the conflict drawing mode.

The trail visualisation works only with algorithm of Dowsland and Thompson. In this mode, the program visualises the trail between each pair of vertices. It has an optional parameter that sets the treshold of trail that should be visualized.

The trail is visualised as an edge between the 2 vertices and its value defines the color of this edge. The stronger the trail, the brighter the color, because we want to point out the strong trails.

Chapter 6

Experimental Results

In this section we describe the experiments we performed on the algorithms described in Chapter 3 and 4.

We have chosen two sets of graph instances, first containing instances that are known to be difficult and the second containing easier instances, or our own random generated graphs.

The tests were performed on a computer with the following technical parameters:

- Architecture: PC (x86-64)
- Processor: Intel Core2 Quad Q9550 (4x2.83GHz)
- Memory: 4 GB RAM
- OS: Gentoo Linux

6.1 Data sets

The DIMACS graph coloring instances are benchmark data sets for testing the graph coloring algorithms since the DIMACS second challenge [47]. The instances can be found at webpage of DIMACS [44], at the overview website of Porumbel [46] or the research website of Johnson et al. [45].

For our experiments we have chosen a set of DIMACS graph instances that are known to be difficult, according to the overview webpage of Porumbel [46]. We compared the results of the ACO algorithms with the best known results as of May 2010 listed in [46] or the chromatic number (if known).

We have also made a set of tests on easier instances to provide a wider range of results.

Table 6.1: Difficult instances

Name	$\|V\|$	$\|E\|$
dsjc1000.1	1000	99258
dsjc1000.9	1000	898898
dsjc250.5	250	31366
dsjc500.1	500	24916
dsjc500.5	500	125249
dsjr500.1c	500	242550
flat1000_50_0	1000	245000
flat300_28_0	300	21695
le450_15c	450	16680
r250.5	250	14849

Table 6.2: Easy instances

Name	$\|V\|$	$\|E\|$
100.5	100	2480
100.5b	100	2418
200.5	200	10043
300.5	300	22327
dsjc125.1	125	736
dsjc250.9	250	27897
le_450_5b	450	5734
queen8_12	96	2736

For our experiments, we used the graph instances mentioned in Tables 6.1 and 6.2.

The graphs 100.5a, 100.b, 200.5 and 300.5 are randomly generated graphs with density 0.5, i.e., the probability of adding an edge between each pair of vertices is 0.5.

6.2 The results

In Tables 6.3 and 6.4 we present the results of experiments. The algorithm of Bui et al. is denoted as BNPP, the algorithm of Hertz and Zufferey as HZ and the algorithm of Dowsland and Thompson as DT. We denote our algorithm as V. BNPP* is a variant of algorithm BNPP that will be described later.

Table 6.3: Experiment results - Difficult instances

Graph	K/χ	BNPP	BNPP*	HZ	DT	V
dsjc1000.1	20/?	22.00 (22) *38.75s*	22.00 (22) *34.15s*	— —	23.00 (23) *1805.95s*	22.80 (22) *415.76s*
dsjc1000.9	223/?	240.80 (240) *351.02s*	235.80 (235) *334.83s*	— —	227.40 (227) *3874.62s*	256.20 (252) *3945.82s*
dsjc250.5	28/?	30.20 (30) *32.35s*	30.00 (30) *31.81s*	38.40 (37) *2953.55s*	29.00 (29) *48.04s*	30.60 (30) *89.27s*
dsjc500.1	12/?	13.00 (13) *11.91s*	13.00 (13) *10.87s*	— —	14.00 (14) *280.80s*	13.40 (13) *76.34s*
dsjc500.5	48/?	53.20 (53) *54.14s*	53.00 (53) *45.16s*	— —	49.60 (49) *299.39s*	54.60 (54) *394.44s*
dsjr500.1c	84/84	85.00 (85) *90.51s*	85.00 (85) *77.83s*	— —	90.00 (89) *296.59s*	86.40 (85) *108.62s*
flat1000_50	50/50	93.20 (92) *203.73s*	88.00 (81) *196.55s*	— —	87.00 (87) *2200.54s*	97.00 (96) *3962.94s*
flat300_28	28/28	33.20 (33) *70.99s*	33.40 (33) *64.20s*	42.80 (42) *5267.22s*	32.00 (32) *75.80s*	33.80 (33) *162.44s*
le450_15c	15/15	16.40 (16) *183.39s*	15.00 (15) *121.51s*	— —	16.00 (15) *206.85s*	21.40 (20) *163.10s*
r250.5	65/65	67.40 (67) *33.45s*	67.80 (67) *33.31s*	68.40 (67) *4293.39s*	68.00 (67) *72.40s*	68.60 (68) *139.25s*

Table 6.4: Experiment results - Easy instances

Graph	K/χ	BNPP	BNPP*	HZ	DT	V
100.5	?	15.20 (15) *0.78s*	15.00 (15) *0.75s*	18.60 (18) *144.94s*	15.00 (15) *5.24s*	15.80 (15) *5.25s*
100.5b	?	14.60 (14) *0.79s*	14.00 (14) *0.75s*	18.40 (18) *138.82s*	14.00 (14) *5.40s*	15.00 (15) *7.75s*
200.5	?	25.40 (25) *12.70s*	25.20 (25) *12.12s*	31.60 (31) *1451.82s*	24.80 (24) *27.34s*	25.80 (25) *45.30s*
300.5	?	34.40 (34) *81.97s*	34.80 (34) *69.53s*	43.50 (43) *5550.99s*	33.00 (33) *78.35s*	35.00 (34) *113.24s*
DSJC125.1	5/?	5.60 (5) *0.74s*	5.00 (5) *0.71s*	6.20 (6) *70.88s*	5.00 (5) *10.55s*	6.00 (6) *1.52s*
DSJC250.9	72/?	73.00 (72) *50.08s*	72.80 (72) *50.46s*	— —	72.40 (72) *76.84s*	74.60 (74) *65.38s*
le450_5b	5/5	5.80 (5) *52.97s*	5.00 (5) *45.65s*	10.00 (10) *3420.93s*	5.00 (5) *245.80s*	7.00 (7) *11.11s*
queen8_12	12/12	12.00 (12) *0.71s*	12.00 (12) *0.71s*	13.60 (13) *83.90s*	12.00 (12) *7.63s*	12.00 (12) *4.10s*

We ran each algorithm on each instance 5 times. The tables contain the mean of best number of colors found and the minimum value archieved (in the parenthesis). We also included the infomation about time consumption. The tables contain the mean time length of the computation in seconds (in italics). A dash in a cell means the algorithm did not finish the computation in our time limit of 2 hours. The table also contains the chromatic number of each instance (denoted χ), or its best known upper bound (denoted K).

During the experiments we had to deal with several problems. First, the algorithms were not decribed with all details. Second, not all values of the parameters were strictly defined. Let us now summarize all the unclear parts and the way we dealt with them.

In algorithm BNPP the authors do not specify the recoloring of the vertices having color out of range. Their description of coloring rules for the ants is also a bit unclear. We implemented a simple recoloring ourselves, and the coloring procedure is implemented as close to their description as possible. However, as this implementation did not match the results provided by the authors, we tried another variant of the coloring procedure: When an ant chooses a color for the vertex and the zero conflict is not possible, the ant will not use the color it has used in previous location even if the color would minimize the conflict. We denote this variant as BNPP*, and the results of this variant are close to the ones provided by authors.

In algorithm HZ the authors do not clearly specify the way to pick a vertex candidate to remove the ants from. We therefore tried two variants of the algorithm. One, described in Section 3.3, computes the values of moves affecting all conflicting vertices and picks the best one. Then it considers only moves involving the vertex of picked move. Another variant picks a conflicting vertex at random and considers only moves involving this randomly chosen vertex. It can be easily observed that the first variant is slower than the second one, but has a potential to produce better results. We made a set of experiments and came to the conclusion that the first variant is better, therefore is used in our set of experiments.

In algorithm DT the authors do not provide the exact settings of all parameters, and do not strictly specify the visibility function. We therefore picked the visibility function denoted *RLF(2)* which was used in most of their experiments and the values that produced the best overall results according to their set of experiments, namely: $nAttempts = 4$, $nLocalCycles = 2 \cdot |V|$.

Although the parameters can be tuned for better performance for each individual instance (as described in [19]), we used their default values for our experiments. Improvement of the results of DT is therefore possible, but must be done individually for each instance.

Also, in algorithm DT the authors do not specify the value of the evaluation function when a legal coloring has been found. We therefore tried a variant in which we reset the trail to its default value if a legal coloring has been found. However, this variant produced almost the same results as the one proposed in Chapter 3.

Let us now summarize the results. The best ACO algorithms according to our experiments are the modified algorithm of Bui et al. (BNPP*) and the algorithm of Dowsland and Thompson (DT). Their performance is balanced, on some instances BNPP* outperforms DT and vice versa. However, as mentioned above, these values can be tuned by right parameter settings. The time complexity is different though, DT runs significantly slower than BNPP*.

The algorithm of Hertz and Zufferey, however, ended way behind the other algorithms, and appears to be not suitable for larger instances. It did not finish the computation within the time limit 7 times out of 10, which makes it the the worst one of the examinated algorithms.

Let us now compare the algorithms with the best known results produced by non-ACO algorithms. According to our experiments, only algorithms BNPP* and DT matched the best known results on one difficult instance, namely *le_450_15c*. The algorithms overall produced worse results on the difficult instances than the best known algorithms. However, we only compared the algorithms with default parameter settings, so it is possible that they can perform better with specific values for each instance.

Such experiments were performed by Dowsland and Thompson. In their set of experiments, they tested the best parameter settings for a number difficult instances. With these settings, the algorithm matched the best known results for some difficult instances, although not for all of them. Moreover, they provided such tests only for 5 instances.

Let us now analyse our algorithm (V). Although the algorithm produces overall worse results than BNPP* or DT, on some graph instances it outperformed DT both in the number of colors and in speed. On the other hand, on these instances the BNPP* produced better results that both DT and V.

We believe the algorithm could be futher enhanced. We experimented with following variants:

We tried to enlarge the size of the colony. This variant did not enhance the performance, on the contrary, it slowed down the computation.

We also tried to change the coloring rules. We experimented with a variant in which an ant recolors a vertex if it does not increase the conflict by more than a given constant. This variant produced very poor results as the algorithm quickly increases the total conflict of the graph and is unable to decrease it. We are aware of the fact that our current variant may suffer from stucking in local optimum. However, we believe that the *ModJolt* procedure is sufficient to randomize the coloring if needed.

We tried a variant with no trail system, in which the move of an ant depended only on the greedy force, but this variant was not successfull either, although the results were close to the current ones. We therefore believe that the trail system is useful for the algorithm.

We assume possible enhancement could be archieved by improving the trail system, or by improving the greedy force part of the move attractiveness.

Chapter 7

Conclusion

In this thesis, we provided an overview of the Graph Coloring Problem. We presented existing approaches and focused on the Ant Colony Optimization heuristics. We described in detail three recent ACO algorithms and also presented an original algorithm.

As a part of the thesis, we implemented these algorithms in a program called Coloring Ants and made a set of experiments on them.

From the experiments it followed, that the best ACO algorithms for the GCP are the algorithm of Bui et al. [6] and the algorithm of Dowsland and Thompson [19]. However, our proposed algorithm has not performed that well, so futher improvement is needed.

We also noticed that with good parameter settings, the ACO algorithms matched the best known results for some difficult graph instances, although not all of them. Therefore, we believe that the algorithms could be futher enhanced. Moreover, the algorithm of Bui et al. produces competitive results in a relatively short time.

Bibliography

[1] Avanthay C., Hertz A., Zufferey N.: *A variable neighborhood search for graph coloring*, European Journal of Operational Research (2003);151: 379–88.

[2] Battiti R., Tecchiolli G.: *The reactive tabu search*, ORSA Journal on Computing 1994;6:126–40.

[3] Blöchliger I., Zufferey N., 2008: *A reactive tabu search using partial solutions for the graph coloring problem*, Computers and Operations Research 35, 3, 960–975.

[4] Brélaz, D.: *New methods to color the vertices of a graph*, Communications of the Assoc. of Comput. Machinery 22 (1979), 251-256.

[5] Brown, R.: *Chromatic scheduling and the chromatic number problem*, Management Science 19, 4 (1972), 456–463.

[6] Bui T. N., Nguyen T. H., Patel C. M., Phan K.-A. T.: *An ant-based algorithm for coloring graphs*, Discrete Applied Mathematics 156 (2008), 190-200.

[7] Bui T. N., Patel C. M.(2002): *An ant system algorithm for coloring graphs*, In: D. S. Johnson, A. Mehrotra, M. Trick (Eds.), *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pp. 83-91, Ithaca, New York, USA.

[8] Chams M., Hertz A., de Werra D.: *Some experiments with simulated annealing for coloring graphs*, European Journal of Operational Research (1987), 32: 260–6.

[9] Chiarandini M., Dumitrescu I., Stützle T.: *Stochastic local search algorithms for the graph colouring problem*, In: T. F. Gonzalez (Ed.): *Handbook of Approximation Algorithms and Metaheuristics, Computer & Information Science Series*, (2007), pp. 63.1-63.17, Chapman & Hall/CRC, Boca Raton, FL, USA.

[10] Comellas F., Ozón J.: *Graph coloring algorithms for assignment problems in radio networks*, Applications of Neutral Networks to Telecommunications 2. Edit. J. Alspector, R. Goodman y T. X. Brown, Lawrence Erlbaum Ass., pp. 49-56, 1995. `http://www.mat.upc.es/~commelas/radio/radio.html`

[11] Comellas F., Ozón J.: *An ant algorithm for the graph coloring problem*, ANTS'98—From Ant Colonies to Artificial Ants: First International Workshop on Ant Colony Optimization, Brussels, Belgium, 1998.

[12] Costa D., Hertz A.: *Ants can colour graphs*, J. Oper. Res. Soc. 48 (1997) 295–305.

[13] Costa D., Hertz A., Dubuis O. (1995): *Embedding of a sequential algorithm for coloring problems in graphs*, Journal of Heuristics, 1: 105-158.

[14] Davis L.: *Order-based genetic algorithms and the graph coloring problem*, In: L. Davis, editor, *Handbook of Genetic Algorithms*, pages 72–90. Van Nostrand Reinhold, USA, 1991.

[15] Dorigo M., Maniezzo V., Colorni A.: *Positive feedback as a search strategy*, Technical Report 91-016, Dipartimento di Elettronica e Informazione, Policecnico di Milano, Italy, 1991.

[16] Dorigo M., Maniezzo V., Colorni A.: *The ant system: optimization by a colony of co-operating agents*, IEEE Trans. Systems Man Cybernet. 26 (1996) 29–41.

[17] Dorigo M., Stützle T.: *Ant Colony Optimization*, The MIT Press (2004).

[18] Dowsland K. A., Thompson J. M.: *Ant colony optimisation for the examination scheduling problem*, J. Oper. Res. Soc. 56 (2005) 426–438.

[19] Dowsland K. A., Thompson J. M.: *An improved ant colony optimisation heuristic for graph colouring*, Discrete Applied Mathematics 156 (2008), 313-324.

[20] Eiben A. E., Hauw J. K., and Van Hemert J. I.: *Graph coloring with adaptive evolutionary algorithms*, Journal of Heuristics, 4(1):25–46, (1998).

[21] Fleurent C., Ferland J. A.: *Genetic and hybrid algorithms for graph coloring*, Annals of Operations Research (1996);63: 437–61.

[22] Galinier P., Hao J.-K.: *Hybrid evolutionary algorithms for graph coloring*, Journal of Combinatorial Optimization (1999);3: 379–97.

[23] Galinier P., Hertz A. (2006): *A survey of local search methods for graph coloring*, Computers & Operations Research, vol. 33, pp. 2547-2562.

[24] Garey, M. R., Johnson, D. S., 1979: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co, New York.

[25] Glover F.: *Future paths for integer programming and links to artificial intelligence*, Computers and Operations Research (1986), 13/5, 533–49.

[26] Hansen P.: *The steepest ascent mildest descent heuristic for combinatorial programming*, In: Congress on numerical methods in combinatorial optimization. Capri, Italy, 1986.

[27] Hansen P., Labbé M., Schindl D., 2005: *Set covering and packing formulations of graph coloring: algorithms and first polyhedral results*, Technical Report G-2005-76, GERAD, Université de Montréal.

[28] Hertz, A., de Werra, D.: *Using tabu search techniques for graph coloring*, Computing 39 (1987), 345–351.

[29] Hertz A., Galinier P., Zufferey N.: *An adaptive memory algorithm for the graph coloring problem*, Discrete Applied Mathematics (2005), accepted for publication.

[30] Hertz A., Zufferey N.: *A New Ant Algorithm for Graph Coloring*, Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization, NICSO 2006, Granada, Spain, 51-60.

[31] Johnson D. S., Aragon C. R., McGeoch L. A., Shevon C.: *Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning*, Operations Research (1991); 39: 378–406.

[32] Klotz W.: *Graph Coloring Algorithms*, Mathematik-Bericht 5 (2002), TU Clausthal, 1-9.

[33] Leighton, F. T.: *A graph coloring algorithm for large scheduling problems*, Journal of Research of the National Bureau of Standards 84, 6 (1979), 489–503.

[34] Malaguti E., Monaci M., and Toth P.: *A Metaheuristic Approach for the Vertex Coloring Problem*, INFORMS Journal on Computing, 20(2):302, 2008.

[35] Malaguti E., Toth P. (2009): *A survey on vertex coloring problems*, International Transactions in Operational Research, pp. 1-34.

[36] Mehrotra A., Trick M. A., 1996: *A column generation approach for graph coloring*, INFORMS Journal on Computing 8, 344–354.

[37] Méndez-Díaz, I., Zabala, P., 2006: *A branch-and-cut algorithm for graph coloring*, Discrete Applied Mathematics 154, 5, 826–847.

[38] Méndez-Díaz, I., Zabala, P., 2008: *A cutting plane algorithm for graph coloring*, Discrete Applied Mathematics 156, 159–179.

[39] Mladenovic N., Hansen P.: *Variable neighborhood search*, Computers and Operations Research (1997);24:1097–100.

[40] Morgenstern C.: *Distributed coloration neighborhood search*, In D.S. Johnson and M.A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of DIMACS series in Discrete Mathematics and Theoretical Computer Science, pages 335-358, American Mathematical Society, 1996.

[41] Rochat Y., Taillard E.: *Probabilistic diversification and intensification in local search for vehicle routing*, Journal of Heuristics 1995;1:147–67.

[42] Sewell, E. C., (1996): *An improved algorithm for exact graph coloring*, In: Johnson, D. S., Trick, M. A. (eds): *Cliques, Coloring, and Satisfiability: 2nd DIMACS Implementation Challenge, 1993. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Providence, RI, pp. 359–373.

[43] Zufferey N.: *Heuristiques pour les Problemes de la Coloration des Sommets d'un Graphe et d'Affectation de Fréquences avec Polarités*, PhD Thesis, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne; 2002.

[44] `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/` [cit. 2010-05-25]

[45] `http://mat.gsia.cmu.edu/COLOR04/` [cit. 2010-05-25]

[46] `http://www.info.univ-angers.fr/pub/porumbel/graphs/` [cit. 2010-05-25]

[47] `http://dimacs.rutgers.edu/Challenges/` [cit. 2010-05-25]

[48] `http://www.imada.sdu.dk/~marco/gcp/` [cit. 2010-05-25]