Bachelor thesis

Evolutionary Graph Coloring

Marvin Williams

Date: January 9, 2017

Supervisors: Prof. Dr. Peter Sanders
Dr. Christian Schulz
Dr. Darren Strash

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Acknowledgments

I gratefully acknowledge everyone who supported me during my work on the thesis. First of all I want to thank my supervisors Prof. Peter Sanders, Dr. Christian Schulz and Dr. Darren Strash for the frequent and insightful conversations about the topic and for the lots of helpful advice. I also want to thank the people who worked with me at the institute during the time I worked on the thesis. They provided a very friendly working environment and gave new input whenever needed. Additionally, I would like to thank Dr. Pablo San Segundo for the source code of their algorithm.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 9. Januar 2017

Marvin Williams

Abstract

The Graph Coloring Problem (GCP) asks for the minimum number of colors required to color the vertices of a graph such that no two adjacent vertices have the same color.

In this thesis we present an evolutionary algorithm for the GCP with novel crossover operations using graph partitioning. Our population contains only legal colorings and we use various greedy coloring algorithms to initialize it. In each generation, two colorings of the population function as parents. We combine them using one of the proposed crossover operations. Our first crossover uses a graph partitioning as the crossing point and generates a new coloring by selecting a different coloring for each block. The second crossover works in a similar fashion but uses a vertex separator instead of a partitioning. Our third crossover computes a new coloring from the overlap of the parents. Finally, we improve the new coloring with a local search algorithm. As the goal for our algorithm is to perform well on large graphs, we use a simple tabu search as well as fast crossovers. We evaluate our proposed algorithm by comparing it to the state-of-the-art algorithm PASS by San Segundo [36] on graph instances found in the literature. We are able to outperform PASS on almost 50 % of the graph instances.

Zusammenfassung

Eine Graphfärbung ist eine Zuordnung von den Knoten eines Graphen zu Farben. Bei einer gültigen Färbung dürfen adjazente Knoten nicht die gleiche Farbe haben. In dieser Arbeit beschäftigen wir uns mit dem Problem, wie viele Farben mindestens benötigt werden, um einen Graphen gültig zu färben.

Wir stellen hiefür einen evolutionären Algorithmus vor, der neuartige Kreuzungsoperatoren verwendet, die auf Graphpartitionierung beruhen. Die Population unseres Algorithmus besteht ausschließlich aus gültigen Färbungen und wird mit verschiedenen Greedy-Algorithmen initialisiert. Anschließend werden in jeder Generation zwei Färbungen als Eltern ausgewählt und mit einander gekreuzt, um eine neue Lösung, das *Kind*, zu erzeugen. Wir stellen drei unterschiedliche Kreuzungsoperatoren für unseren Algorithmus vor. Der Erste partitioniert den Graphen in zwei Blöcke und färbt jeden Block entsprechend einem Elternteil. Der zweite Operator folgt einem ähnlichen Prinzip, verwendet jedoch einen Separator anstelle der Partitionierung. Unser letzte Operator konstruiert eine neue Färbung, indem er gleich gefärbte Teile in den Eltern als Basis übernimmt. Bevor das Kind eine Färbung in der Population ersetzt, wird es mithilfe von lokaler Suche verbessert. Da unser Algorithmus für große Graphen ausgelegt ist, verwenden wir eine simple Tabu-Suche und schnelle Kreuzungsoperatoren. Abschließend vergleichen wir die Resultate unseres Algorithmus auf Graphen, die in der Literatur verwenden werden, mit dem aktuellen Algorithmus PASS von San Segundo [36]. Wir erreichen auf fast 50 % der getesteten Graphen bessere Ergebnisse.

Table of Contents

1 Introduction

The Graph Coloring Problem (GCP) asks for a coloring of each vertex of a graph with as few colors as possible such that no adjacent vertices have the same color. We are interested in the GCP as many practical problems which involve the partitioning of objects into disjoint classes can be solved using graph coloring.

These problems range from everyday life to specific technologies. One textbook example is timetable scheduling [10, 26], where a number of lectures are to be assigned to different time slots. Some of the lectures might not be allowed in the same time slot because they are held by the same teacher or take place in the same room. In a graph where these conflicting lectures are connected, the colors of a coloring represent the time slots of a valid assignment. Graph coloring is used in practice by processors for register allocation [6]. Each register can hold one value at a time and a value has a timespan in which it is accessed. Processors construct an interference graph in which the loadable values are connected if their timespans overlap. A coloring of the interference graph yields an assignment of values to registers with each color representing one register. Tests for short circuits in printed circuit boards [16] can be sped up with graph coloring. Nets on the board correspond to vertices that are connected if there is a possible short cut between the nets. Thus, all the nets with the same color can be tested simultaneously. Besides these practical applications, graph coloring also finds use in a number of mathematical problems [18].

Generally, the GCP is NP-complete [20] and we currently know of no efficient algorithm for large graphs. The wide range of applications along with its combinatorial complexity elevates the GCP to one of the most famous and most researched problems in graph theory.

Probably the most well-known result related to graph coloring is the four color theorem. Initially proposed as a conjecture, it states that four colors suffice to color an arbitrary planar graph. The theorem gained much publicity not only because of many false alleged proofs [4] but also because it was the first to be proved extensively computer-aided [1]. While we are able to find colorings using only four colors for any planar graph in quadratic time [35], it is already NP-hard to decide whether three colors suffice for the same graph [8]. Since the GCP is NP-hard even with these restrictions, we need advanced heuristics like evolutionary algorithms and local search to tackle the problem on general graphs.

1.1 Our Results

Many algorithms for the GCP have been proposed most of which are tailored for specific graph subclasses. They exploit specific properties of those graphs and achieve optimal or near optimal results. Metaheuristics make few assumptions about the graph instance itself, even though they may perform better on particular graph classes. We start by introducing local search variants for graph coloring including tabucol. Our algorithm combines the robustness of evolutionary algorithms with the improvements tabucol achieves. We then explain our choices and approaches for the components of our algorithm in all its detail. This covers initialization of solutions, crossover operations, local search parameters and diversity control. We use the KaHIP framework by Sanders and Schulz [37] to create novel crossovers and introduce enhancements to tabucol. In order to classify the quality of our implementation, we perform tests with various modifications and finally compare it in detail to a state-of-the-art DSATUR-based implementation[1]. While on small graphs no algorithm has a clear advantage over the other, our algorithm performs better on large instances.

1.2 Structure of the Thesis

In Chapter 2 we give an overview of the notations and definitions we use throughout this thesis. We also introduce the concepts our algorithm implements. Chapter 3 covers previous on the topic and presents powerful techniques. The chapter additionally suggests more in-depth work on the individual topics for further reading. We then present our algorithm in detail in Chapter 4. Subsequently, we perform parameter tuning and an experimental analysis in Chapter 5. Chapter 6 concludes this thesis with a summary and gives an outlook on possible improvements and suggestions.

[1] we use the algorithm described by San Segundo [36]

2 Preliminaries

In this chapter, we give an overview of definitions and algorithms we use in this thesis. This includes basic graph notation and techniques used by our algorithm. We further present the concepts of evolutionary algorithms and local search.

2.1 Graphs

A graph $G = (V, E)$ is a tuple of vertices V and edges $E \subseteq V \times V$. It is *undirected*, if $\forall v, w \in V, (v, w) \in E \Leftrightarrow (w, v) \in E$. We then write edges as 2-element sets instead of tuples. The number of vertices and edges are denoted as $n = |V|$ and $m = |E|$, respectively. For each vertex $v \in V$ the *neighborhood* is defined as $N(v) = \{w \in V \mid \{v, w\} \in E\}$ and $d(v) = |N(v)|$ is called the *degree* of v. Each vertex $w \in N(v)$ is said to be *adjacent* to v. The maximum degree of a graph is $\Delta = \max\{d(v) \mid v \in V\}$. A *path* of length n in G is a sequence of distinctive vertices such that there exists an edge connecting each pair of consecutive vertices; that is $v_0, \ldots, v_n$ is a path if $\forall i < n, \{v_i, v_{i+1}\} \in E$ and $v_0, \ldots, v_n$ are distinct. Then, v_0 and v_n are called *connected*. A graph is connected if $\forall v, w \in V, v$ and w are connected. Given a *weight* function $\omega : E \to \mathbb{R}$, we can define a *weighted* graph by assigning a *weight* to each edge. Throughout this thesis we only consider connected, undirected graphs without self-loops (called *simple* graphs).

Independent Set. An independent set (IS) is a subset of vertices that are pairwise nonadjacent. It is NP-complete to decide whether a graph contains an independent set of a size k [24]. Hence, finding a maximum independent set is NP-hard.

Matching. A matching M is a subset of the edges of a graph such that each vertex is contained in at most one edge in the matching. If no edge can be added to a matching M, it is said to be *maximal*. A *maximum matching* includes the most edges among all possible matchings and is always maximal. A special case of maximum matchings are *perfect* matchings, where $|M| = \frac{|V|}{2}$ and therefore every vertex is contained in an edge of the matching. In a weighted graph we can define a *maximum weighted matching* M with $\omega(M) = \sum_{m \in M} \omega(m)$ maximal among all matchings.

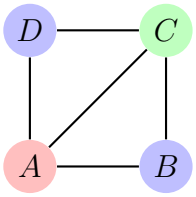
Bipartite Graphs. A graph is bipartite if its vertices can be divided into two disjoint sets such that no two vertices in the same set are adjacent. Hence, each set forms an independent set. The assignment problem [33] can naturally be expressed with weighted bipartite graphs. A maximum weighted matching in the graph corresponds to an optimal assignment.

Subgraph. A subgraph $G' = (V', E')$ of a graph contains a subset of the edges: $E' \subseteq E$. The vertices V' are all endpoints of E', so $V' = \{v \in V \mid \{v, w\} \in E'\}$.

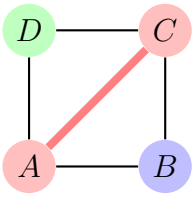
2.2 Graph Coloring

A *graph coloring* is a mapping of vertices or edges of a graph to natural numbers, referred to as colors. In our thesis we solely consider *vertex coloring* as a function $c : V \to \mathbb{N}$ but refer to it as graph coloring throughout the thesis. Consequently, a k-coloring of graph G is a function $c : V \to \{1, \ldots, k\} \subset \mathbb{N}$. The vertices with color i form a color class $V_i = \{v \in V \mid c(v) = i\}$. The edge $\{v, w\}$ is a conflict and both the edge and the corresponding vertices v and w are said to be conflicting in color i if vertices v and w have the same color i (see Figure 2.1). If assigning a certain color to a vertex induces no conflict, the said color is *free* for this vertex. The set of free colors for vertex v is denoted by $F(v)$. The number of distinct colors used in the neighborhood of vertex v is its saturation degree $\rho(v) = |\{i \in \{1, \ldots, k\} \mid \exists w \in N(v) : c(w) = i\}|$. A k-coloring is *legal* if no two vertices are conflicting and all vertices are colored. Hence, a coloring is legal if and only if the color classes form independent sets and are a partition of V. The fewer colors a legal coloring uses, the better is its *quality*.

A graph G is *k-colorable* if a legal k-coloring for G exists. The k-Graph Coloring Problem (k-GCP) asks if a graph is k-colorable. The smallest k for which a graph G is k-colorable is referred to as the *chromatic number* $\mathcal{X}(G)$. Finding this k is known as the general *Graph Coloring Problem* (GCP) and often associated with finding a legal coloring with k colors.



(a) a coloring with no conflicts



(b) the conflict is highlighted red

Figure 2.1: Different colorings of the same graph

2.3 Local Search

Local search is a common metaheuristic to improve a given solution for an optimization problem by modifying the solution iteratively. Given a problem, the set of all parameter configurations the local search can possibly traverse, is the search space S. A move changes a configuration into another within the search space. The set of all possible configurations reachable from a solution with one move is its neighborhood $\mathcal{N}$. Hence, the neighborhood is a function $\mathcal{N} : S \rightarrow 2^S$. In each iteration, the local search scans the neighborhood for the best possible move according to an *objective function* f and applies it (see Algorithm 1). A move is applied even if it deteriorates the current configuration.

Algorithm 1: Local Search

Input: $InitialSolution\ I$
Result: the best solution visited

```
1 begin
2     s ← I
3     for i ← 1 to numIter do
4         s' ← min_{s'∈N(s)}(f(s'))
5         Move (s, s')
```

Local search suffers from two major drawbacks: the convergence towards local optima and cycling through few configurations. To overcome these issues, many techniques to guide the search direction have been suggested. In the following we examine the *tabu search* variant of local search, which was proposed by Glover [17]. The basic idea behind tabu search is to not revisit configurations that have been visited in the recent past. To accomplish this goal, the algorithm stores the last τ configurations in a *tabu list* and marks them tabu, where τ is called the *tabu tenure*. Moves that lead to a tabu configuration are not considered in the current iteration.

Four different local search strategies exist according to Galinier and Hertz [15], namely the *legal*, the *k-fixed partial legal*, the *k-fixed penalty* and the *penalty* strategy.

Legal. As the name suggests, this strategy only considers legal colorings in the search space. Simple moves like changing the color of one vertex in a legal coloring will generally not lead to legal colorings and therefore cannot be used in this strategy. A more sophisticated move like the *Kempe Chain Interchange* proposed by Morgenstern and Shapiro [32] is required to maintain a legal coloring. We refer the interested reader to their paper about this move, as we do not go into detail here. The objective of this strategy is to minimize the number of colors needed, but as a move rarely reduces the

number of colors, Morgenstern and Shapino suggest to minimize $-\sum_{i \in C} |V_i|^2$ [32]. This function encourages the tabu search to remove vertices from small color classes and eventually extinguish them.

k-fixed partial legal. In this strategy the number of colors is fixed and the search space contains all partially colored legal solutions. The tabu search tries to assign a color not greater than k to each vertex. On the one hand, the Kempe Chain Interchange is again a possible move but on the other hand one can simply assign a color to an uncolored vertex and uncolor adjacent conflicting vertices. The objective with this strategy is to decrease the number of uncolored vertices and eventually color all vertices.

k-fixed penalty. The search space contains all possible k-colorings which are not necessarily legal. Here, a move consists in changing the color of one vertex into another not greater than k. The advantage of this move over the Kempe chain interchange is its simplicity and the straightforward neighborhood but in return it is not as powerful. The objective is to decrease the number of conflicts, that is to minimize $|\{\{v, w\} \in E \mid c(v) = c(w)\}|$.

Penalty. The penalty strategy is similar to the k-fixed penalty strategy except it allows an arbitrary number of colors. Therefore the objective is to not only decrease the number of conflicts but also to reduce the number of colors used. It can be shown that

$$\sum_{i \in \{1,\dots,k\}} 2|V_i||E_i| - \sum_{i \in \{1,\dots,k\}} |V_i|^2$$

where $E_i = \{\{v, w\} \in E \mid v, w \in V_i\}$ consists of the conflicting edges in color class V_i has optima in legal colorings. The second term minimizes the number of colors as seen in the legal strategy.

2.4 Evolutionary Algorithms

Inspired by nature, evolutionary algorithms capture the idea of natural selection and create new solutions based on favorable properties of existing solutions. This technique has been successfully applied in many fields of research (a survey is given in [12]). Evolutionary algorithms draw their strength from the simplicity of the individual operations and the robustness their combination yields.

It turns out that they are especially well suited for hard optimization problems like the GCP. Evolutionary algorithms are *metaheuristics* because they do not specify a certain implementation but rather an abstract idea. In its simplest form, the algorithm starts

with a set of *individuals*, which are elements of the search space, forming the *population*. We denote the population as P and its individuals as $I_1, \ldots, I_p$, with the population size $p = |P|$. In terms of graph coloring, each individual represents a coloring. In each *generation*, some individuals are chosen via a certain selection rule and function as *parents*. A sensible *crossover* operator extracts favorable properties from the parents and hands them down to the offspring. After some *mutations* are applied to the offspring, it is reinserted into the population and the routine of reproduction and mutation repeats. As the crossover operator is responsible for generating new individuals it is considered a key component of evolutionary algorithms. An overview of this process is given in Figure 2.2.

The *diversity* of the population measures the similarity of the individuals to each other and is an important aspect of evolutionary algorithms. If the individuals are too similar, it becomes difficult to overcome the population's locality in the search space. The reason for this effect is that the offspring is similar to its parents, thus if the parents are similar to each other we cannot traverse new areas of the search space. Therefore the eviction of an individual in order to replace it with the offspring is crucial for the algorithm to succeed.

2.5 Graph Partitioning

The graph partitioning problem is to divide the vertices of a graph into k almost equally sized subsets such that an objective function is minimized. Typically, one tries to have the number of edges connecting vertices of different subsets as low as possible.

KaHIP. We use the Karlsruhe High Quality Partitioning Framework (KaHIP) by Schulz [39] to generate graph partitions and vertex separators, which are utilized by our crossover operators. The framework consists of the following programs:

- KaFFPa, a multilevel graph partitioning implementation
- KaFFPaE, a distributed evolutionary algorithm
- KaBaPE, guarantees balancing constraints

Further information on KaHIP and a user guide can be found in [38].

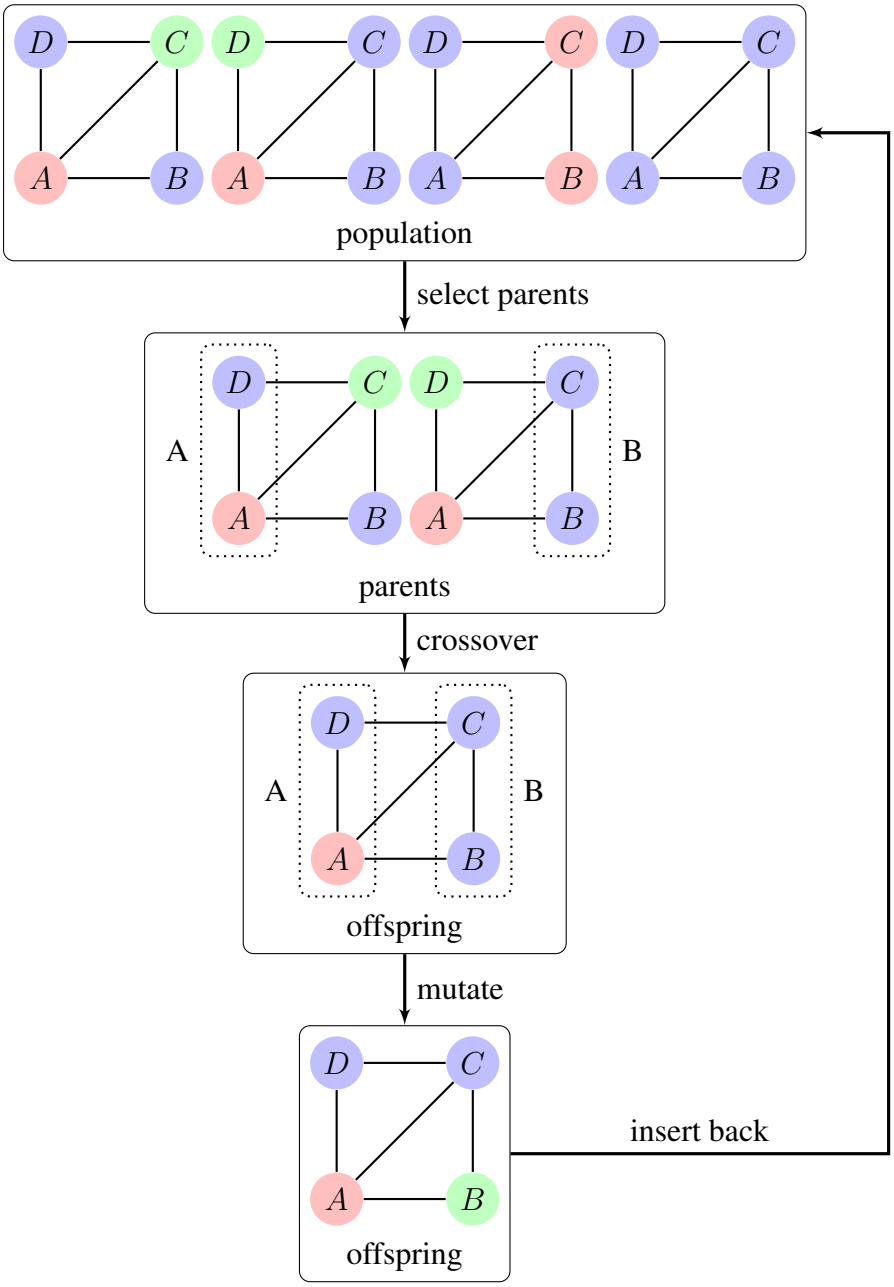


Figure 2.2: Example iteration of an evolutionary algorithm

3 Related Work

Graph coloring is one of the most studied NP-hard problems and therefore many algorithms tackling the GCP have been developed and published. Due to the combinatorial complexity of this problem only a few exact algorithms have been proposed. Even approximating the chromatic number within a constant factor is NP-hard [28]. In this chapter, we give an overview of existing approaches for solving the GCP. We cover both exact and heuristic algorithms.

3.1 Exact Algorithms

Exhaustive search can solve the k-GCP by looking at every possible assignment in $\mathcal{O}(k^n)$. Thus, $\mathcal{X}(G)$ can be obtained by solving the k-GCP for increasing k until a legal coloring is found. This procedure is infeasible even for reasonably small graphs.

Alternatively, branch and bound algorithms can be used to achieve optimal colorings. For this purpose, one uncolored vertex is selected and a free color is assigned to it recursively. Each time the algorithm reaches a leaf by coloring the last remaining vertex, it adjusts the upper bound accordingly. The algorithm branches for each possible color assignment in the new subproblem to color the remaining vertices without exceeding the upper bound given by prior branches. This method always finds $\mathcal{X}(G)$ but the required number of subproblems strongly depends on the choice of the branch vertex. Also, good bounds prior to the branching can reduce the number of subproblems drastically [40]. Tight upper bounds allow to prune many subproblems early and a lower bound can stop the search if it is met. Björklund and Husfeldt [2] showed that the GCP can be solved in $\mathcal{O}(2.4423^n)$ with polynomial space using inclusion-exclusion.

3.2 Heuristics

Heuristics can be divided into various classes. In this section we discuss *greedy* methods, *tabu search* and *hybrid evolutionary algorithms*.

3.2.1 Greedy Coloring

Greedy coloring algorithms look at the vertices of a graph in a particular order and successively assign the smallest free color to each vertex (see Figure 3.1).

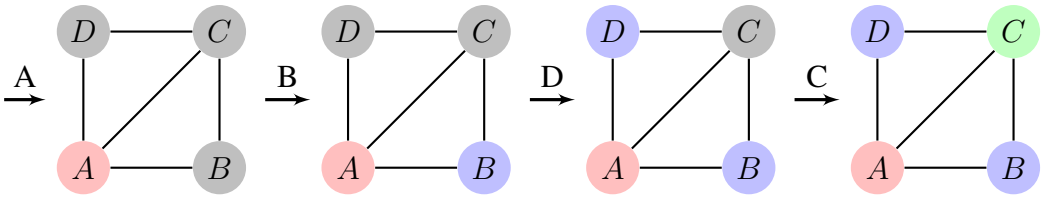


Figure 3.1: Greedy coloring with the ordering A, B, D, C

This method may find an optimal coloring for any graph with the proper order. Hence, finding a proper vertex ordering solves the GCP and is NP-hard. A vertex gets the *maximal* possible color assigned if all its neighbors have distinct colors, therefore $\forall v \in V, c(v) \leq d(v) + 1$. Consequently, greedy coloring algorithms use at most $\Delta + 1$ colors which is an upper bound for the GCP. According to Brooks' theorem, this upper bound can be reduced to Δ except for the K_n and C_n for odd n [27].

A prominent greedy heuristic, namely DSATUR (degree saturation), was proposed by Brélaz [3]. DSATUR colors the vertex with the highest degree first. Next, the vertex with the highest saturation degree is chosen until all vertices are colored. If there is a tie, DSATUR colors the vertex with highest degree in the uncolored subgraph; further ties are broken lexicographically. For pseudocode, see Algorithm 2. Brélaz proved that DSATUR is exact for bipartite graphs and yields the best results compared to other greedy heuristics in 84 % of the tested graphs (random graphs with $n \leq 100$). The algorithm is often used to generate initial solutions in algorithms that try to iteratively improve a single solution [7, 11, 29].

Brélaz [3] also provided an exact algorithm using the DSATUR heuristic to select the branch vertex. Sewell [40] introduced a new tiebreaking strategy for choosing the vertex (called SEWELL) based on DSATUR that minimizes the number of free colors for the remaining subgraph. Among all vertices with maximum saturation degree, the one with the most mutual free colors with each uncolored neighbor is selected:

$$v_{\text{next}} = \arg\max_{v \in T}\Big(\sum_{\substack{w \in N(v),\\ w \in U}} |F(v) \cap F(w)|\Big),$$

where U is the set of all uncolored vertices and $T \subset U$ is the set of all uncolored vertices with maximum saturation degree. Sewell proposed a new exact algorithm based on this improved tiebreaking rule and used tabucol to compute an initial upper bound. Reportedly, the speedup of this algorithm was between 0 % and 600 % compared to

Algorithm 2: DSATUR

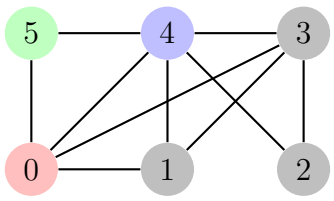
Input: graph G
Output: legal coloring c

```
1  begin
2      G' ← G
3      for i ← 1 to n do
4          ρ_max, d_max ← −1
5          for v ∈ V do
6              if ρ_G(v) > ρ_max then
7                  ρ_max ← ρ_G(v)
8                  d_max ← d_G'(v)
9                  v_next ← v
10             else if ρ_G(v) = ρ_max ∧ d_G'(v) > d_max then
11                 d_max ← d_G'(v)
12                 v_next ← v
13
14         c(v_next) ← SmallestFreeColor (v_next)
15         G' ← G' − {v_next}
```

exact DSATUR on randomly generated graphs. San Segundo [36] further improved the tiebreaking strategy suggested by Sewell. Instead of all uncolored neighbors, his tiebreaking rule (called PASS) considers only uncolored neighbors with maximum saturation:

$$v_{\text{next}} = \arg\max_{v \in T}\Big(\sum_{\substack{w \in N(v),\\ w \in T}} |F(v) \cap F(w)|\Big).$$

The different behavior of DSATUR, SEWELL and PASS is illustrated in Figure 3.2. In this scenario the vertices $\{0, 4, 5\}$ are already colored and now the heuristic selects the next vertex to color. The uncolored vertices are $U = \{1, 2, 3\}$ with the saturation



(a) Subproblem

DSATUR	3
SEWELL	3
PASS	1

(b) The selected vertex

Figure 3.2: Determining the next vertex to color

degrees $\rho(1) = \rho(3) = 2, \rho(2) = 1$, thus the vertices with maximum saturation degree are $T = \{1, 3\}$. DSATUR selects vertex 3 as it has the maximum degree of 2 in the uncolored subgraph. The free colors of the vertices are $F(1) = F(3) = \{\text{green}\}$ and $F(2) = \{\text{green}, \text{red}\}$. For vertex 1 it is $|F(1) \cap F(3)| = 1$ and for vertex 3 it is $|F(3) \cap F(1)| + |F(3) \cap F(2)| = 2$, so SEWELL also selects vertex 3. As vertex $2 \notin T$, PASS omits the second term for vertex 3 and vertices 1 and 3 tie, so PASS chooses vertex 1 lexicographically. The exact version with the PASS selection rule applies greedy DSATUR prior to the branching to obtain a tight upper bound of colors. We refer to both the selection rule and the exact algorithm as PASS. San Segundo reported that PASS outperforms the other two algorithms and can prove optimality up to three times faster [36].

Parallel greedy heuristics also have been studied. A very successful strategy is to iteratively extract independent sets and color them in parallel [23].

3.2.2 Tabu Search

Tabucol is a k-fixed penalty implementation of tabu search proposed by Hertz and de Werra [19] to solve the k-GCP. The search space contains all k-colorings and the objective function f is the number of conflicts in the coloring. Instead of making whole configurations tabu, undoing moves from the recent past is forbidden and the tabu tenure is individually determined for each move. The algorithm uses a conflicting k-coloring c as the initial configuration. A move $m = (i, v)$ with $i \in \{1, \ldots, k\}, i \neq c(v)$ changes the color of vertex v to i and we write $m(c) = c'$. Hence, tabucol generates a neighbor c' from c by changing the color to a single vertex. The complement of m is $\overline{m} = (c(v), v)$ as it restores the previous color of v. The gain $\gamma(m) = f(c) - f(m(c))$ of a move is its improvement in the objective function. After generating all considered neighbors, the move with maximum gain is picked and applied. The tabu tenure τ determines for how many iterations the complement of the selected move is tabu. This procedure is repeated until a stopping criterion, such as a maximum number of iterations, is met or no conflicts are left. In the latter case tabucol finds a legal k-coloring and therefore solves the k-GCP.

Hertz and Werra introduced an aspiration criterion that allows the search to consider moves that are tabu: Tabucol considers a move to a tabu neighbor c' that is tabu if $f(c') < f(c)$. See Algorithm 3 for pseudocode.

Additionally, they remove some large independent sets from the graph as a preprocessing step to obtain a smaller graph for coloring. The algorithm finds near optimal colorings on random graphs with up to 1000 vertices. Furthermore, tabucol finds feasible colorings not necessarily close to optimum very fast. Since tabucol was introduced, many improvements like a dynamic tabu tenure and more sophisticated aspiration criteria have been proposed [5, 11]. Despite its age, tabucol is still widely used as

Algorithm 3: Tabucol

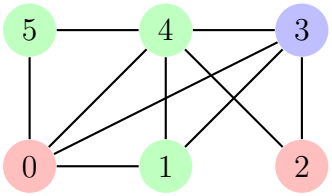
Input: Graph G
number of colors k
maximum number of iterations $maxIter$
Output: c if $f(c) = 0$, otherwise no feasible coloring has been found

```
1 begin
2     c ← generateSolution (G, k)        // The coloring is likely to be infeasible
3     T ← empty tabu list
4     i ← 0
5     while f(c) > 0 and i < maxIter do
6         m ← arg max_{m'}(γ(m') | m' ∉ T ∨ f(m'(c)) < f(c))
7         updateTabuList (T, m̄, i)  // Append m̄ to T and remove expired moves
8         c ← m(c)
9         i ← i + 1
```

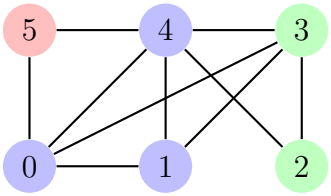
a subroutine in hybrid graph coloring algorithms, for example in [10, 14]. Galinier and Hertz [15] gathered many local search methods for graph coloring in a comprehensive overview.

3.2.3 Hybrid Evolutionary Algorithms

The results achieved by evolutionary algorithms are among the best for many combinatorial problems; Davis [9] was the first to apply them to graph coloring. Fleurent and Ferland [13] extensively studied replacing mutations with tabu search in genetic algorithms for the GCP. With the introduction of a population and powerful crossovers they founded *hybrid graph coloring algorithms* and were able to improve known results for the benchmarks of the second DIMACS challenge [21]. Since then, most of the established algorithms employ heuristics like tabu search or DSATUR as subroutines in evolutionary algorithms. We refer the interested reader to the survey by Preux and Talbi [34]. Most hybrid evolutionary algorithms found in literature try to find feasible k-colorings to gradually solve the GCP. Generally, the crossovers found in literature can be divided into assignment crossovers [13] and partition crossovers [10, 14]. Galinier and Hao [14] proposed a hybrid evolutionary algorithm with a highly specialized partition crossover operator. This crossover (called GPX) alternately picks the largest color class from one of the parents. This color class is handed down to the offspring and all vertices in the picked color class are removed from both parents. This procedure is repeated k times, and remaining vertices are colored randomly in the offspring. An example is shown in Figure 3.3.

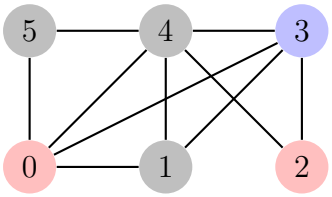


p_1 (current)

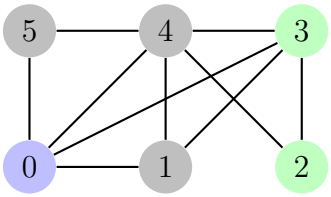


p_2

(a) largest color class in p_1 is $\{1, 4, 5\}$

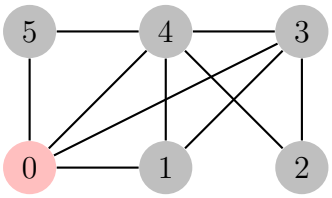


p_1

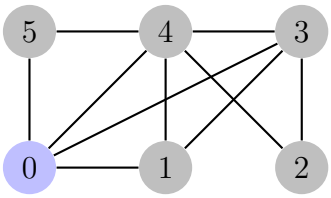


p_2 (current)

(b) largest color class in p_2 is $\{2, 3\}$

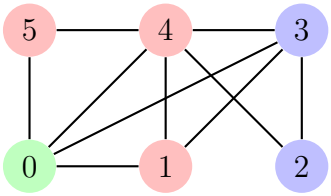


p_1 (current)



p_2

(c) largest color class in p_1 is $\{0\}$



(d) offspring with the selected color classes

Figure 3.3: Example GPX crossover

After the crossover generates a new offspring, a round of tabu search improves the offspring and tries to minimize the number of conflicts. This hybrid algorithm matched the best known results on most of the graphs from the second DIMACS challenge and even found new best results for four graphs. It became a popular basis for crossovers and eviction rules. For example, Lü and Hao [30] generalized the GPX for an arbitrary number of parents.

4 EvoCol

In this chapter we introduce our novel hybrid evolutionary algorithm and describe all its components in detail. We use the previously described tabucol and build sophisticated crossovers with graph partitioning. At first, we outline our evolutionary framework, and then describe each of its components. Also, we describe critical routines and justify their use in the algorithm.

4.1 Outline

Unlike other evolutionary algorithms, our population only contains legal colorings. We aim to gradually increase the quality of the population. This approach is different to many proposed algorithms that solve the k-GCP repeatedly for decreasing k (for example [10, 14, 30]). The colorings in our population do not necessarily have the same number of colors. Consequently, the crossover operators must be able to handle parents with different numbers of colors. The generated offspring however is not necessarily legal in the first place hence we need to resolve the conflicts. We then apply tabucol to improve the quality of the offspring by reducing the number of colors. Finally, we replace the least fit individual in the population with the offspring. In line with evolutionary algorithms, we repeat this procedure for a given number of generations and eventually output the best individual among the population.

4.2 Initialization

We need an initial population to start the evolution. Typically, fast greedy algorithms are used for this purpose. To assure high diversification, these algorithms heavily rely on randomness and produce varying solutions. We implement three different initialization methods that produce legal colorings. In Chapter 5 we compare these methods in terms of speed, solution quality and influence on the evolutionary algorithm.

Random. Our simplest method colors the vertices in a random order. That is, it permutes the vertices and assigns the smallest free color to them one after another. This generally produces poor results but the coloring depends heavily on the order in

which the vertices are colored. Nonetheless, this method is very fast and leads to a high diversification among the population.

DSATUR. We apply DSATUR (see Algorithm 2) to initialize our population. It is our slowest method but produces the best initial colorings. A disadvantage is the low diversity achieved since the order of vertices is restricted to some extent. DSATUR offers randomization only when two or more vertices tie according to the deterministic selection rule.

Degeneracy. The degeneracy order is a popular choice for greedy coloring. We iteratively select the vertex with minimum degree and remove it from the graph and reverse the resulting order. With this strategy we obtain a coloring with at most $l + 1$ colors if every subgraph contains one vertex v with $d(v) \leq l$ [41]. The smallest such l is the *degeneracy* of the graph.

PASS. We initialize our population with colorings generated by the exact PASS algorithm with a time limit. PASS eventually computes optimal colorings but we apply a time limit that depends on the time our random initialization needs.

4.3 Selection

A crucial part of the evolutionary algorithm is the selection of parents to produce offspring. The selection effects both the diversity and the solution quality, so an unfit method impairs the entire evolutionary algorithm. We use tournament selection as it assures both fairly good parents and variety. Tournament selection consists of multiple rounds each of which nominates one parent. In each round, two randomly selected individuals compete against each other. The number of selected individuals is the *tournament size*. The individual with the best quality wins the round and becomes the designated parent. As in our algorithm all crossovers require two parents, two tournament rounds suffice in each selection.

A major benefit of tournament selection is the adaptability of the tournament size. On the one hand, a large tournament size assures parents of high quality but reduces the probability for a single individual to become parent. On the other hand, a small tournament size increases the said probability but decreases the quality of the parents. In fact, a tournament size of 1 is equivalent to random selection. For more information on tournament selection, we refer the interested reader to [31].

4.4 Crossovers

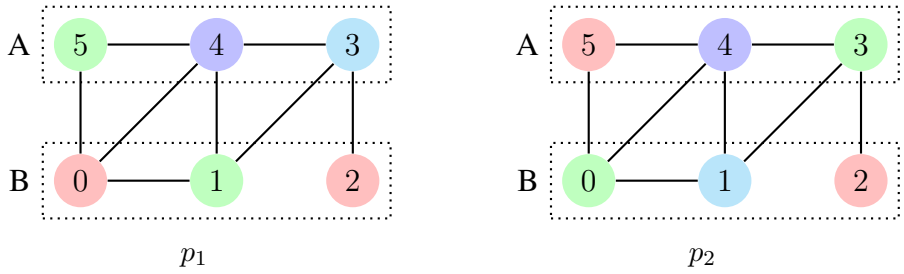
Crossovers combine two parents to create a new individual. In an evolutionary sense, they imitate the natural reproduction. Crossovers try to pass certain structures of the parents down to the offspring. We propose three different crossovers for our evolutionary algorithm:

- **Partition crossover**, which uses graph partitioning
- **Separator crossover**, which uses a vertex separator
- **Overlap crossover**, which uses mutual structures of the parents

All crossovers require exactly two legal colorings as parents. We denote $\{p_1, p_2\} \subseteq P$ as the selected parents and o as the generated offspring. All the presented crossovers generate a single, legal offspring. We now discuss each crossover type in detail.

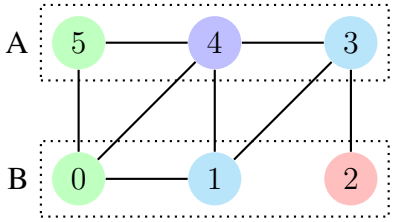
4.4.1 Partition Crossover

First we split the vertices of the graph into the *blocks* $A, B \subseteq V$. The sets A and B form a partitioning if $A \cup B = V$ and $A \cap B = \emptyset$. To generate the offspring, we color the vertices of the blocks each according to one parent (illustrated in Figure 4.1).



p_1 p_2

(a) the parents p_1 and p_2



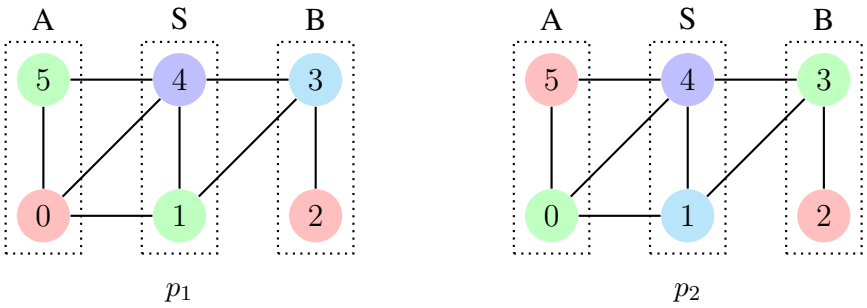
(b) the offspring with p_1 in A and p_2 in B

Figure 4.1: Partition crossover before resolving the conflicts. Partitions A and B are framed

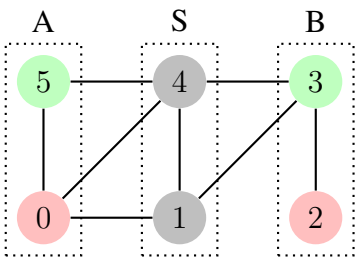
Since the parents are legal colorings, if a vertex and all its neighbors are in the same partition, then this vertex cannot conflict. However, any edge connecting the two blocks can indeed conflict. For each conflicting edge we look at both its vertices and recolor the one with the smaller smallest free color accordingly.

4.4.2 Separator Crossover

The separator crossover is strongly related to the partition crossover but uses, as the name suggests, a vertex separator. The separator $S \subset V$ divides the remaining vertices into two distinct subsets A and B such that no edge connects vertices from distinct partitions. Our crossover colors the partitions A and B in a similar manner to the partition crossover. As the separator is still uncolored and no edges connect the partitions A and B, the produced partial coloring is legal (see Figure 4.2). We color the vertices in the separator with the greedy DSATUR heuristic. With this method the generated offspring remains a legal coloring.



(a) the parents p_1 and p_2

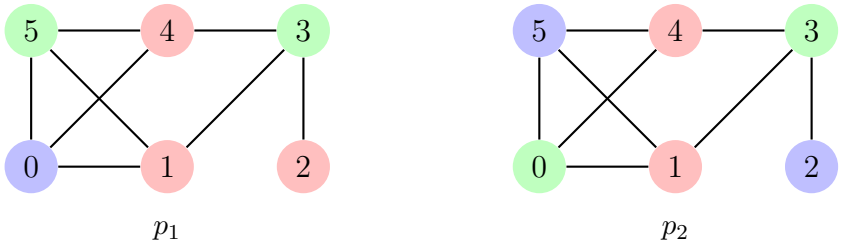


(b) the offspring with p_1 in A and p_2 in B

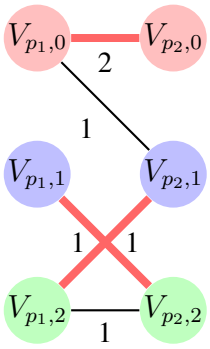
Figure 4.2: Separator crossover with separator S and partitions A and B, separator still uncolored

4.4.3 Overlap Crossover

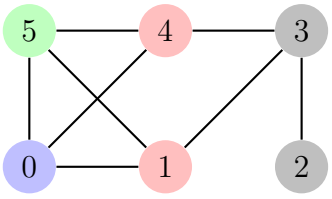
The overlap crossover detects similar color classes in the parents and takes mutual vertices of similar color classes as new color classes for the offspring. For this purpose, we intersect each color class of parent p_1 with each color class of parent p_2 and count the *mutual* vertices: $\forall i \in \{1, \dots, k_{p_1}\}, j \in \{1, \dots, k_{p_2}\} : \omega_{ij} = |V_{p_1,i} \cap V_{p_2,j}|$, where $V_{c,i}$ is color class i in coloring c. A bipartite graph with the color classes $V_{p_1,i} \cup V_{p_2,j}$ as vertices and the edges $\{V_{p_1,i}, V_{p_2,j}\}$ with weights $\omega(\{V_{p_1,i}, V_{p_2,j}\}) = \omega_{ij}$ represents the overall overlap of the color classes. A matching M with maximum weight consequently is the maximum global overlap of color classes in the two parents. We compute a maximum weighted matching by iteratively finding augmenting paths in the bipartite graph in $\mathcal{O}(|V|^2|E|)$. For each edge $\{V_{p_1,i}, V_{p_2,j}\} \in M$ we add the color class $V_{p_1,i} \cap V_{p_2,j}$ to the offspring (see Figure 4.3). Note that edges with weight 0 are left out; including those leads to the problem of finding the maximum matching with maximal weight. The remaining uncolored vertices are then colored using the DSATUR heuristic. The complete procedure is given in Algorithm 4.



(a) the parents p_1 and p_2



(b) overlap graph with highlighted maximum matching



(c) the offspring with mutual vertices of the matched color classes colored

Figure 4.3: Overlap crossover with the color classes $V_0 = V_{p_1,0} \cap V_{p_2,0}$, $V_1 = V_{p_1,1} \cap V_{p_2,2}$ and $V_2 = V_{p_1,2} \cap V_{p_2,1}$ in the offspring

Algorithm 4: Overlap Crossover

Input: Parents p_1, p_2
Output: offspring o

```
1 begin
2     B ← ConstructOverlapGraph (p_1, p_2)
3     M ← MaximumMatching (B)
4     color ← 0
5     foreach {V_{p_1,i}, V_{p_2,j}} ∈ M do
6         foreach v ∈ V_{p_1,i} ∩ V_{p_2,j} do
7             o(v) ← color
8         color ← color + 1
9     Dsatur (o)
```

4.5 Mutation

Usually, random mutations are applied to the offspring after crossing to maintain the diversity. Our approach is to use local search instead of random mutations to improve the offspring's quality in the evolutionary process. As mentioned before, tabu search and in particular tabucol is a commonly used heuristic to improve given solutions. For our algorithm we use an improved version of the tabucol presented in Section 3.2.2.

A move with a non-conflicting vertex never has a positive gain. Following Galinier and Hao [14], we therefore use the simple heuristic to only consider moves with conflicting vertices. On the one hand, this heuristic speeds up the search, on the other hand we occasionally do not pick the best possible move. Our tabu tenure τ depends on the current number of conflicts $f(c)$ and is randomized to prevent cycling through configurations. We define it as follows: $\tau = \alpha * f(c) + \mathit{rand}[0, \beta]$ with parameters α and β. The best values for these parameters are empirically determined in Chapter 5.

Tabucol is a k-fixed penalty local search. As such, it aims to resolve conflicts in a coloring with no more than k colors. The crossovers guarantee to produce non-conflicting offspring to be consistent with our population. Starting with the legal coloring, we repeatedly reduce the number of colors by removing one color class and apply tabucol to resolve the induced conflicts. More precisely, we put each vertex of the removed color class into an existing color class in such a way that as few conflicts as possible are induced. As the removed color class is an independent set, we can recolor the vertices in an arbitrary order. To keep the number of new conflicts low, we heuristically always remove the smallest color class. We stop this procedure if tabucol fails to resolve all conflicts within a given number of iterations. No algorithm in the literature to our knowledge applies tabu search in this fashion but naturally deploys the legal

strategy with sophisticated moves to reduce the number of colors. We prefer our approach because the simpler moves offer a good trade-off between computational effort and quality.

Originally, tabucol makes the move tabu that reverts the applied move. We use a simpler strategy and make the entire vertex of the applied move tabu. This simplification allows us to use priority queues for a faster search for the best move. The priority queues hold the vertices to consider for the next move prioritized by the maximum gain of a move with this vertex. The effect of this change is evaluated in Chapter 5.

4.6 Diversification

After we improve the offspring with our tabu search, we insert it back into the population. This step involves selecting an individual to replace. Besides the solution quality, we want to keep the diversification of the population high. In line with metaheuristics, we define diversity and provide a method to choose the least fit individual to evict. We therefore define $\delta : (V \to \mathbb{N}) \times (V \to \mathbb{N}) \to \mathbb{N}$ to measure the similarity of two individuals and call $\delta(c_1, c_2)$ the *distance* between the colorings c_1 and c_2.

Naturally, two colorings are considered equal if they can be transformed into each other by relabeling color classes. The minimum number of vertices one has to recolor to transform one of the two colorings into the other is an intuitive yet powerful metric to constitute similarity. In fact, we can use the same weighted overlap graph constructed by the overlap crossover for arbitrary colorings to measure their similarity. A matching M in the overlap graph with maximal weight represents which color class in c_1 corresponds to which color class in c_2 such that as few vertices as possible are not in any mutual color class. Each vertex that is in exactly one of the two corresponding color classes needs to be put in the proper color class to make the colorings equal. Hence, we set the number of vertices that are not in a mutual color class equal to the distance δ and define

$$\delta(c_1, c_2) = n - \omega(M) = \sum_{\{V_i, V_j\} \in M} |V_i \setminus V_j \cup V_j \setminus V_i|.$$

Where $\delta(c_1, c_2) \in [0, n-1]$ with $\delta(c_1, c_2) = 0$ if c_1 equals c_2 and $\delta(c_1, c_2) = n - 1$ if in one coloring every vertex has a unique color and the other coloring uses only one color.

In the following we describe how we use δ to select the individual to replace our offspring with: If every coloring in the population uses less colors than the offspring, the average solution quality would decline in case of replacement and the offspring is discarded. Otherwise, we evict the coloring using the most colors. As a tiebreaking strategy we use the distance metric and select the coloring with the smallest distance to the offspring. Algorithm 5 shows the replacement in pseudocode.

Algorithm 5: Insert Back

Input: Offspring o, Population P

```
1  begin
2      I_worst ← o
3      I_max ← NumberColors(o)
4      δ_min ← n
5      foreach I_current ∈ P do
6          if NumberColors(I_current) ≥ I_max then
7              if NumberColors(I_current) > I_max or δ(I_current, o) < δ_min then
8                  I_worst ← I_current
9                  I_max ← NumberColors(I_current)
10                 δ_min ← δ(I_current, o)
11     if I_worst ≠ o then
12         Replace(P, I_worst, o) // Replace I_worst with o in P
```

5 Experimental Results

We evaluate the performance of our proposed evolutionary algorithm EvoCol for the GCP. At first we present the hardware we used for the experiments as well as the chosen graph instances. We then improve our algorithm by tuning various parameters and discuss our choices. Finally, we compare the tuned EvoCol to a recent DSATUR implementation.

5.1 Setup

We implemented the algorithm in C++ and compiled it using g++ v.4.8.5 with the -O3 flag switched on for optimization. The computer we used was equipped with an Intel Xeon CPU E5-4640 CPU with 32 cores, 2.4 GHz and 512 GB of DDR3 RAM. Each run was executed on a dedicated core and repeated three times with different random seeds. The operating system was Ubuntu 14.04.5 LTS and ran on a linux kernel v.3.13.0-95.

KaHIP provides an interface to compute partitions and separators. Our crossovers invoke KaHIP via the interface with a random seed and a random imbalance within 0.05 and 0.5 to generate 2-way partitions and separators. We show possible tuning parameters of EvoCol and constitute their default settings as well as the impact they have on the solution quality. We compare the results of the tuned EvoCol to a modified version of the exact DSATUR implementation by San Segundo [36].

For the parameter tuning as well as the final comparisons we used a time limit of ten hours. The values presented in the convergence plots are computed as follows: Each time the algorithm finds a new lowest number of colors (also called solution), it outputs the current time stamp, the solution and the seed used. The outputs for one instance with different seeds are merged and sorted by time stamp in ascending order. We then generate a sequence of tuples with the time stamps and the average of the best solutions of all runs found prior to that time stamp. The tuples prior to the time stamp at which all runs have output their first solution are discarded. If we plot graphs for graph families, we add the graph instance to each tuple in each sequence and again merge the sequences. We then compute the final sequence similar to the above described method but with the geometric mean of the solutions for the different instances. An example of this procedure is given in Table 5.1.

	Instance A			Instance B			
Time stamp	Run 1	Run 2	Merge	Run 1	Run 2	Merge	Final
1		10	-	30		-	-
2	11	9	10		28	29	17.03
3	9		9	24	26	25	15

Table 5.1: Example solution output and convergence sequences for the instances A and B with two runs for each instance. The column "Merge" is the sequence for the single instances and "Final" is the final sequence.

Modifications to PASS

We briefly describe some modifications we made to the PASS implementation by San Segundo [36]. The given implementation was not designed to handle large graphs, as an adjacency matrix was used to store the graph. This graph representation requires much memory storage and exceeds the resources available to us for many graphs. Hence, we mainly contributed the modifications to the internal graph representation and graph access. The changes neither alter the time complexity nor the functionality of any routine but reduce the required memory storage significantly. More precisely, we replaced the adjacency matrix with an adjacency list. The original PASS implementation is not able to process graphs with more than $50\,000$ vertices on our benchmark hardware.

5.2 Instances

We used graphs from various sources for our experiments which can be divided into different graph families. The graphs from the second DIMACS challenge are a major part of our tests. These graphs are difficult to color and widely used as benchmark instances. Therefore we have results for various algorithms on these graphs for comparison. The DIMACS family consists of the following graphs:

- **dsjcX.Y, dsjrX.Y**, random graphs with X vertices by Johnson et al. [22]
- **rX.Y**, X random points in the unit square pairwise connected if they lay within a certain distance, taken from [21].
- **leX.Y**, Leighton [25] graphs with X vertices and chromatic number Y
- **CX.Y**, dense graphs with X vertices and up to 4 million edges
- **latin_square, school1**, latin square and class scheduling graphs respectively

We also included graphs of the families SOCIAL, NETWORK, SIMULATION and ROADMAP. They are larger than the DIMACS graphs and arise from real world data. Graphs of the SOCIAL family originate from social networks, collaboration relations and communication logs. They often contain large cliques and dense subgraphs but

generally have an unbalanced edge distribution. Hence, they are interesting to investigate in relation to randomly generated graphs. The NETWORK family contains graphs that represent websites and links between them, hosts in peer-to-peer networks and autonomous systems. In the SIMULATION family are spare matrix graphs that arise from physical experiments and simulations. Finally, the ROADMAP family contains graphs representing street maps from various countries.

5.3 Parameter Tuning

EvoCol offers many parameters, hence we have to find a reasonable configuration prior to our experiments and comparisons. The parameters we choose to tune impact the performance of our algorithm.

- Population size
- Initialization method
- Tabu search iterations
- Tabu search parameters
- Crossover

We perform tests varying these parameters and evaluate the influence they have. The initial parameter configuration of the tuning parameters is given in Table 5.2a. To assure that we set the parameters in a reasonable way, we select graphs from different graph families for the tuning. The graphs vary in the number of vertices and have different structures, so the parameters we set eventually suit a wide range of graphs. Table 5.2b shows the graphs we selected for parameter tuning. We perform the parameter tuning incrementally, that is we use the best setting for a parameter before we examine the next parameter. The plots in this chapter show the geometric mean of colors computed as described above with all test graphs as one family.

Population size	20
Initialization method	random
Tabu search iterations	100 000
Tabu search parameters	$\alpha = 1, \beta = 50$
Crossover	partition

(a) Initial parameter configuration

Graph	Family
caidaRouterLevel	NETWORK
dsjc500.5	DIMACS
flat300_28_0	DIMACS
r1000.5	DIMACS
soc-Epinions1	SOCIAL

(b) Selected graph instances

Table 5.2: Parameter tuning information

5.3.1 Initialization Methods

We first look at different initialization methods in Figure 5.1. The different methods heavily impact the final solution quality. While the random initialization is the fastest, the results achieved remain the worst among all initialization methods. We observe that at first DSATUR performs best and is much faster than Degeneracy, which even computes a worse initial value. PASS produces by far the best initial values but takes very long to find them. Hence, EvoCol is not able to further improve on that initialization within the remaining time, which renders it impractical if used solely. The degeneracy based initialization takes longer than DSATUR but the solution quality improves faster. Eventually it passes all other methods after about 20 000 seconds and performs best. Finally we tested the combination of the DSATUR, Degeneracy and PASS initialization by selecting one method for each individual randomly, exploiting the advantages of all methods. This strategy outperforms each individual method and we use it as initialization method for the subsequent tests.

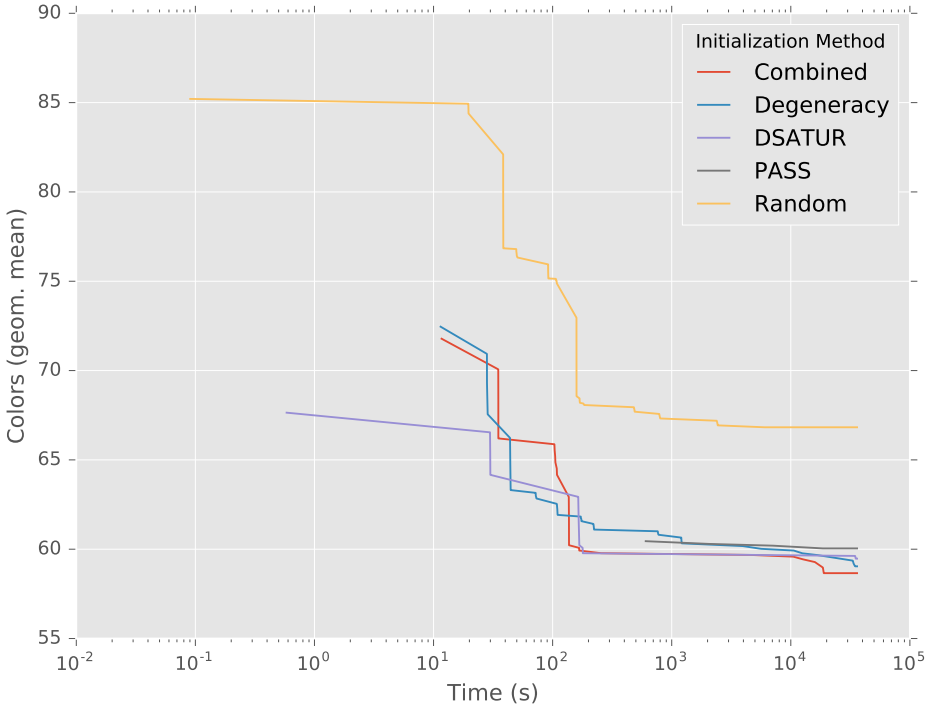


Figure 5.1: Different initialization methods

5.3.2 Population Size

Next, we examine different population sizes. Our initial population size is 20, we test smaller as well as larger population sizes. Figure 5.2 shows how the population size effects EvoCol. All variants initialize with the same solution quality in the early stages of the algorithm. With large populations we cross a single individual less frequently, so at first the average solution quality improves slower than with a small population. On the other hand, a small population has a low variety among the individuals so local optima are hard to escape and quality improvements are less frequent. After 1000 seconds the solution quality stagnates with 4 individuals, while it further improves with all other variants. With 50 individuals we end up only slightly better than with 4. In the late stage of the algorithm at about 10 000 seconds the variants 20 and 30 leave 4 and 50 behind. Eventually, a population size of 20 yields the best result, so we do not alter our initial setting.

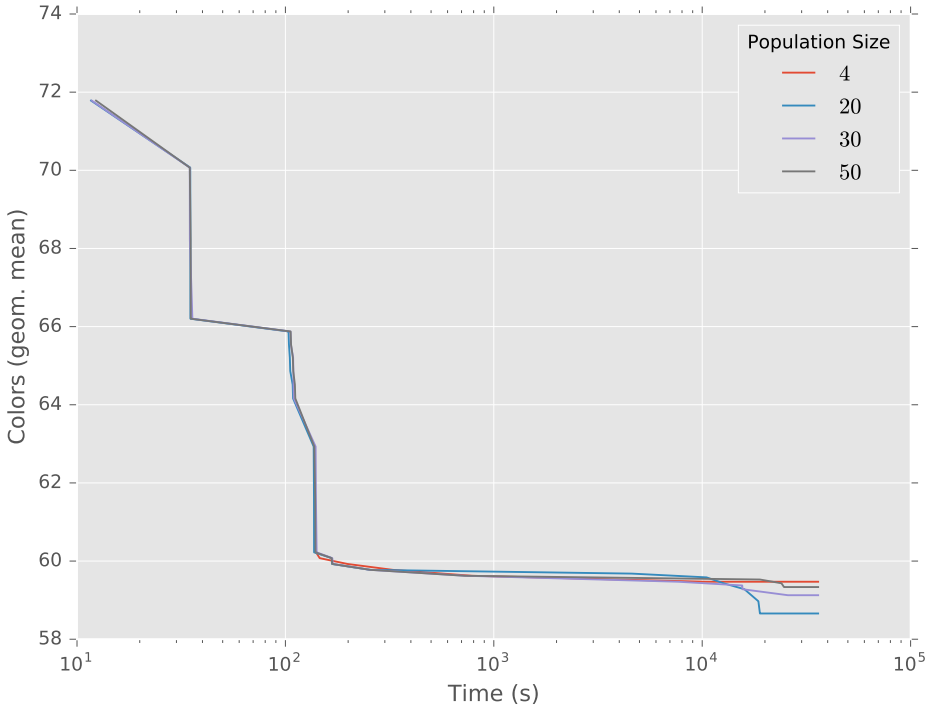


Figure 5.2: Impact of different population sizes

5.3.3 Tabu Search Parameters

The number of iterations for the tabu search to improve the offspring is crucial. Apparently a higher number of iterations leads to offspring of better quality. This improvement is at the expense of time and diversity, as the tabu search might converge to the same local optimum for similar colorings. The number of tabu search iterations effects the number of generations; the more tabu search iterations the less generations can be performed within the same time. In the following we examine different parameter settings for our tabu search.

Tabu Tenure. We first look at the effect of the tabu tenure parameters and set them appropriately. In Figure 5.3 we observe that EvoCol starts off best with α set to 0 (no influence of the number of conflicts). The improvement stagnates earlier with $\beta = 100$ than with $\beta = 50$. As a too long tabu list affects the tabu search negatively and involving the number of conflicts has a positive influence, we choose $\alpha = 1$ and $\beta = 50$. Note that the way we utilize tabu search to reduce the number of colors influences this result. We introduce very few conflicts with each color reduction compared to the number of

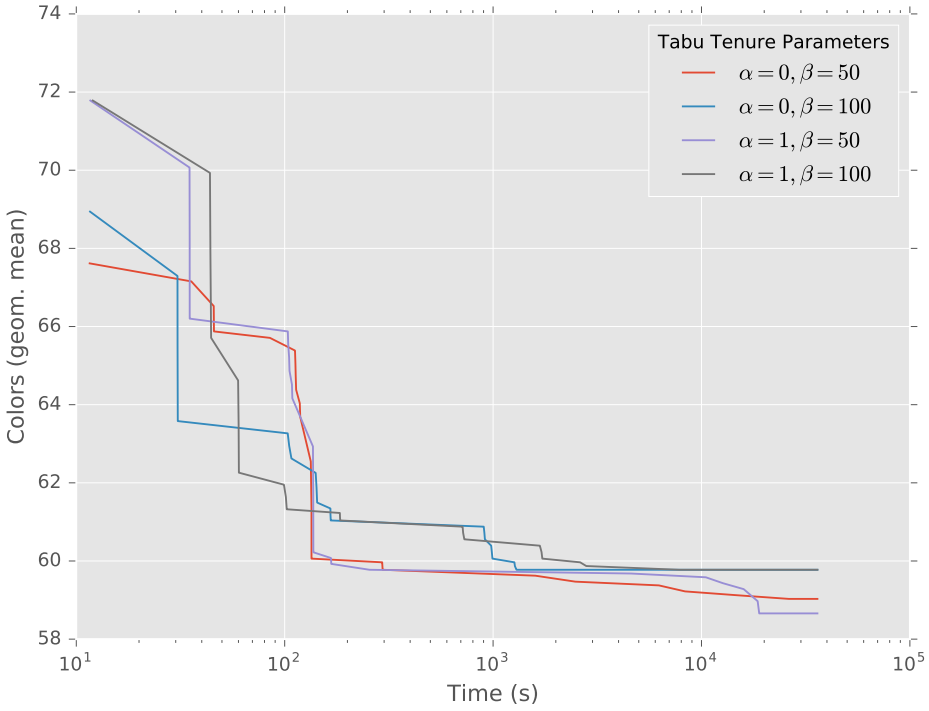


Figure 5.3: Impact of tabu tenure adjustment

conflicts in a random k-coloring. Also, other initialization methods might require other tabu tenure settings.

Iterations. Figure 5.4 shows different numbers of iterations for the tabu search in order to improve the colorings. The time spent on one generation increases with the number of iterations the tabu search performs. On the one hand, a high number of iterations yields better offspring, on the other hand we spend less time crossing the individuals. With $50\,000$ iterations we obtain better results faster than with the other variants. However, the solution quality almost stagnates after 100 seconds and ends up worst compared to the other variants. While $200\,000$ and $300\,000$ iterations are slightly better in the late stage of the algorithm, the run with the $100\,000$ iterations passes the others iterations and becomes the best. Consequently, we select $100\,000$ tabu search iterations for our algorithm.

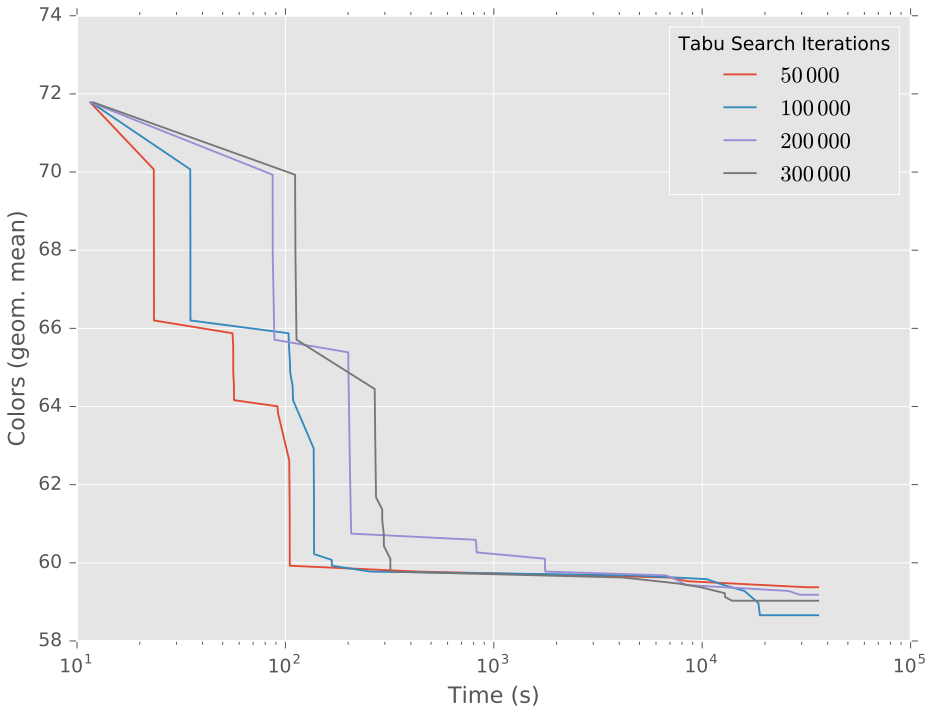


Figure 5.4: Impact of different tabu search iterations

Tabu Mode. Lastly, we examine the effect of our choice to make the entire vertex instead of the complement of the last move tabu. The plot in Figure 5.5 shows the performance of both variants. We clearly see the advantage of making the whole vertex tabu. For small instances the more fine-grained option to make moves tabu is affordable and outperforms the other option. Making vertices tabu simplifies the search and prevents cycling in a small subset of vertices. To overcome the latter, one could increase the tabu tenure. On the other hand, the additional effort does not pay off for large graphs and making vertices tabu eventually performs better. This result justifies our choice to simplify *tabucol* as described.

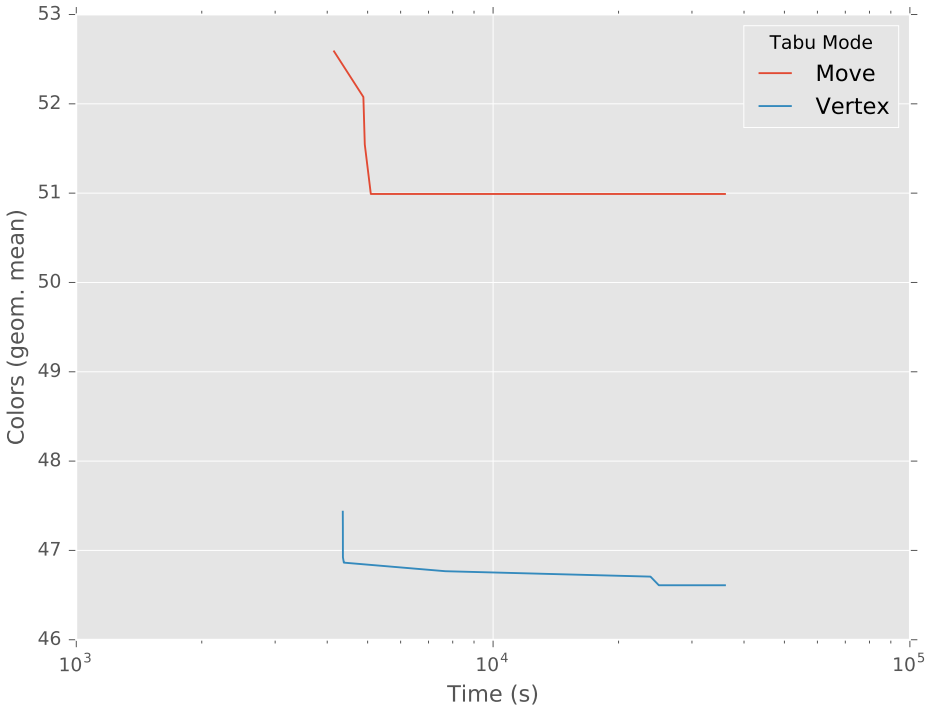


Figure 5.5: Behavior of the different tabu modes

5.3.4 Crossovers

In Figure 5.6 we analyze the performance of the three proposed crossover operators separately. The crossovers are evaluated by running the evolutionary algorithm using each of the crossovers solely. We also examine the benefit of crossovers in general by testing our algorithm without crossing. For this purpose we choose one individual with tournament selection and improve it with our tabu search. Additionally, we test selecting one crossover randomly in each generation. For the first 1000 seconds the three crossovers show roughly the same behavior. Thereafter the separator crossover becomes superior and steadily improves the solution quality while the other crossovers stagnate. In the end, the partition crossover catches up with the separator crossover and almost equally. This behaviour is expected and due to their similarity. Despite its domain knowledge (it uses the color classes explicitly), the overlap crossover performs worst among the individual crossovers. We observe that using crossovers is beneficial for our algorithm as any crossover improves the result with respect to using no crossover. Finally, the combination of all three crossovers yields the best result and therefore we use this variant for our algorithm.

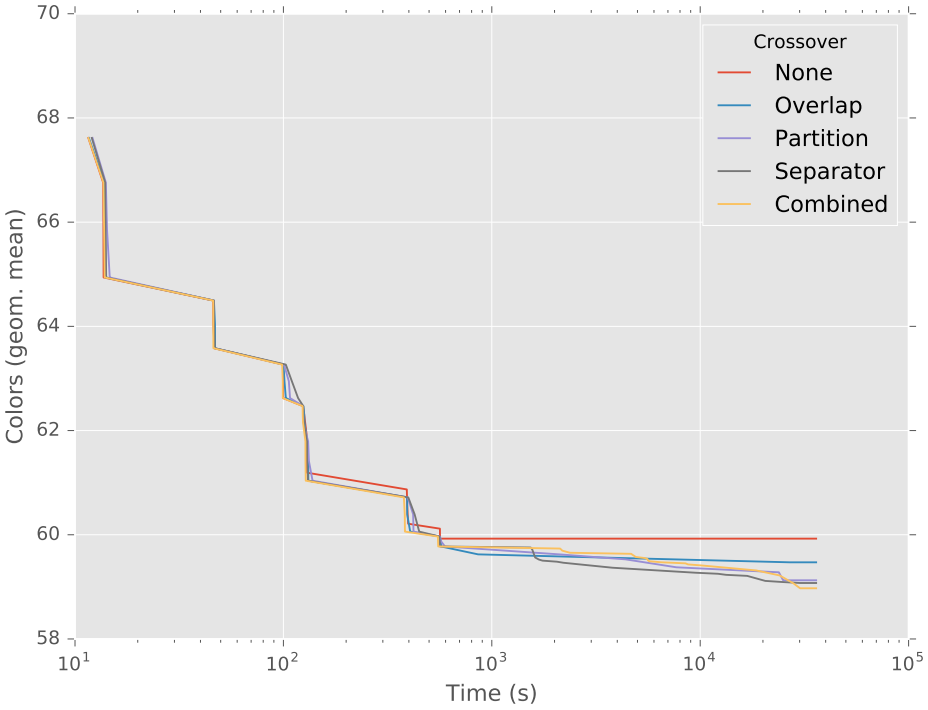


Figure 5.6: Performance of the different crossovers

5.4 Comparison with PASS

In this section we compare EvoCol to PASS on the graph families listed above and analyze the results. For this purpose we present convergence plots and detailed tables for both algorithms. The tables contain the minimum, maximum and the average number of colors for each graph obtained with different seeds and show the best known result (best k) as well as the chromatic number for the graphs if available[1].

We start with the graphs from the DIMACS family. They help us to classify the performance of both algorithms as the chromatic number is known for most graphs and we can compare our results to the best known results. We clearly outperform PASS on some of the graphs, most notably on `C2000.5`, `flat1000_50_0` and `latin_square`. As seen in Figure 5.7 for the latter graph, PASS is not able to improve its initial coloring any further shortly after the start. On many graph instances EvoCol and PASS perform

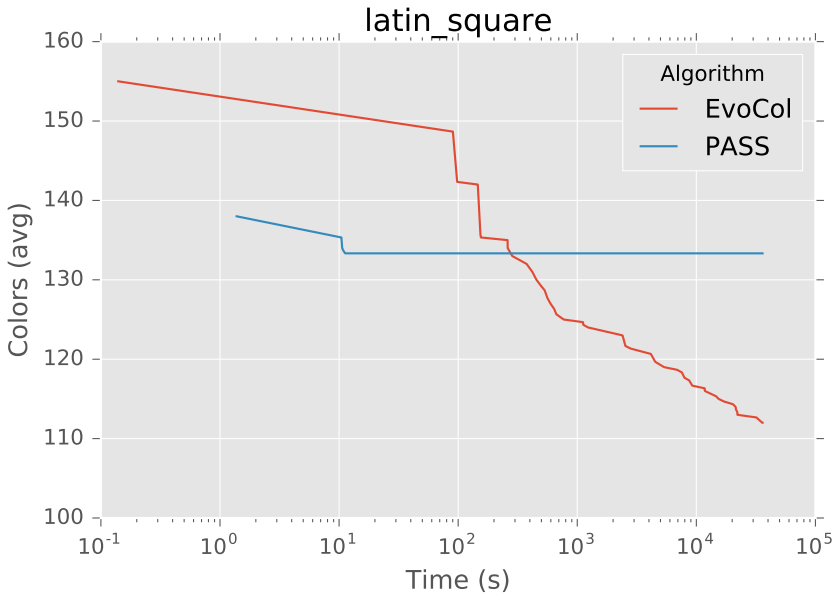


Figure 5.7: Convergence plots of EvoCol and PASS on the `latin_square` graph

similar, as for example on `le450_25c` and `r1000.1c`. Figure 5.8 and Figure 5.9 show the convergence plots for these graphs, respectively. EvoCol is slow at the beginning but increases the solution quality drastically at the late stage of the algorithm. For both graphs EvoCol surpasses PASS at the end, whereas PASS produces significantly better solutions in the beginning (up to 22 colors less). EvoCol needs much time to initialize the population before it starts crossing the individuals, while PASS only initializes and improves a single coloring. While we obtain optimal colorings on seven

[1] We refer to the table given at `http://www.info.univ-angers.fr/pub/porumbel/graphs/`

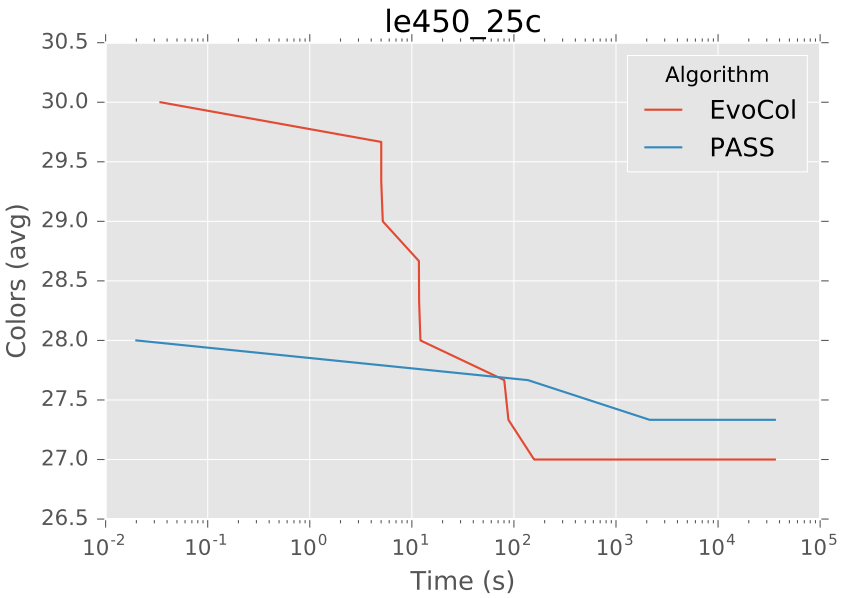


Figure 5.8: Convergence plots of EvoCol and PASS on the `le450_25c` graph

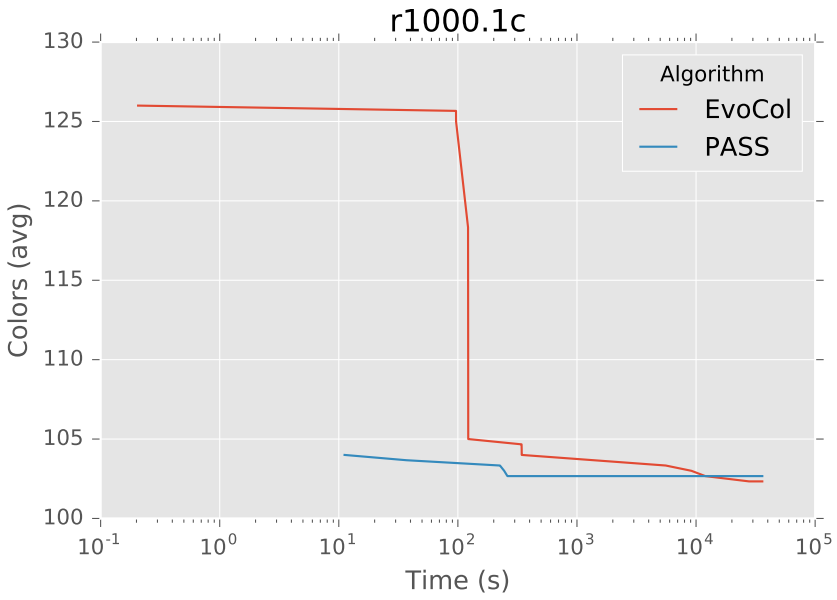


Figure 5.9: Convergence plots of EvoCol and PASS on the `r1000.1c` graph

graphs, both EvoCol and PASS deviate up from the optimum colorings by up to 50 % on others, for example `flat1000_76_0`. EvoCol also falls short of PASS on one graph but has an overall advantage, as seen in Figure 5.10. A detailed list of all graphs in the DIMACS family is presented in Table 5.3.

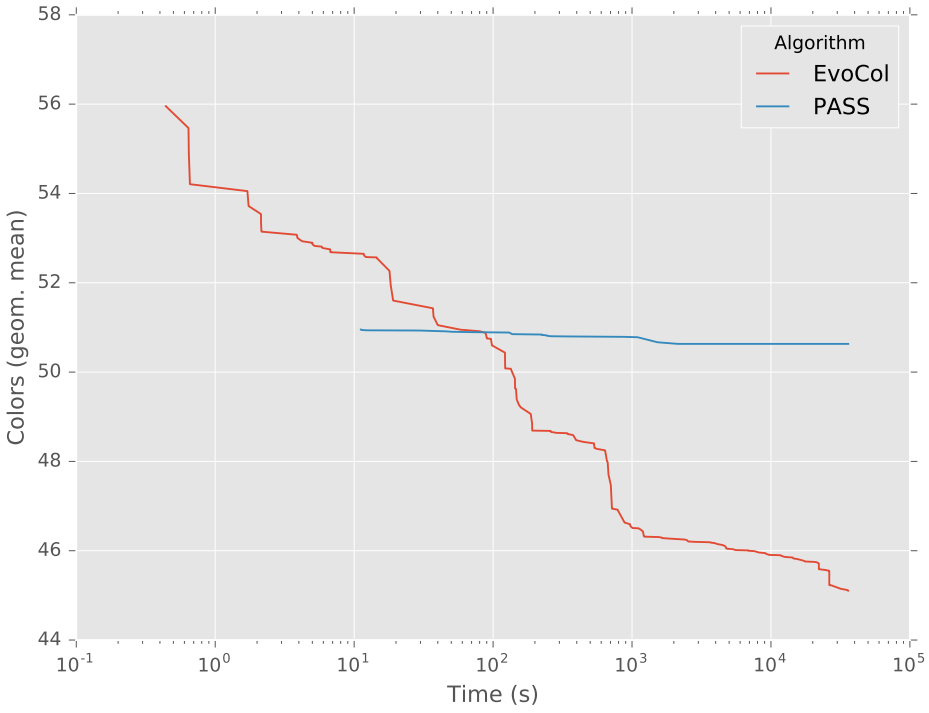


Figure 5.10: Convergence plots of EvoCol and PASS on the DIMACS graphs

Graph				EvoCol			PASS		
Name	n	best k	$\mathcal{X}$	Avg	Min	Max	Avg	Min	Max
C2000.5	2000	146	?	**171.00**	*171*	171	205.33	204	207
dsjc1000.1	1000	20	?	**22.00**	*22*	22	25.00	25	25
dsjc1000.5	1000	83	?	**93.00**	*93*	93	112.67	112	113
dsjc250.1	250	8	8	9.00	9	9	9.00	9	9
dsjr500.5	500	122	122	**124.00**	*124*	124	133.00	133	133
flat1000_50_0	1000	50	50	**73.33**	*50*	87	111.00	111	111
flat1000_60_0	1000	60	60	**91.00**	*91*	91	111.67	110	114
flat1000_76_0	1000	82	76	111.67	111	113	**111.33**	111	112
latin_square	900	97	?	**112.00**	*110*	113	133.33	130	136
le450_25a	450	25	25	25.00	25	25	25.00	25	25
le450_25c	450	25	25	**27.00**	27	27	27.33	27	28
le450_5a	450	5	5	**5.00**	*5*	5	8.67	8	9
r1000.1	1000	20	20	20.00	20	20	20.00	20	20
r1000.1c	1000	98	?	**102.33**	101	104	102.67	101	104
r250.1c	250	64	64	**64.00**	*64*	64	65.00	65	65
r250.5	250	65	65	**65.67**	*65*	66	67.00	67	67
school1	385	14	14	14.00	14	14	14.00	14	14

Table 5.3: The results of both algorithms on the DIMACS graphs. Bold and italic entries indicate a distinct advantage of the corresponding algorithm in the average and minimum number of colors, respectively.

On the SOCIAL family EvoCol outperforms PASS for `soc-Slashdot0811`, `ca-CondMat` and `wiki-Talk`. Figure 5.11 and Figure 5.12 show that in these cases both algorithms start off equal, but PASS is not able to improve the coloring within the time limit. The graphs are larger than the DIMACS graphs, thus PASS can only

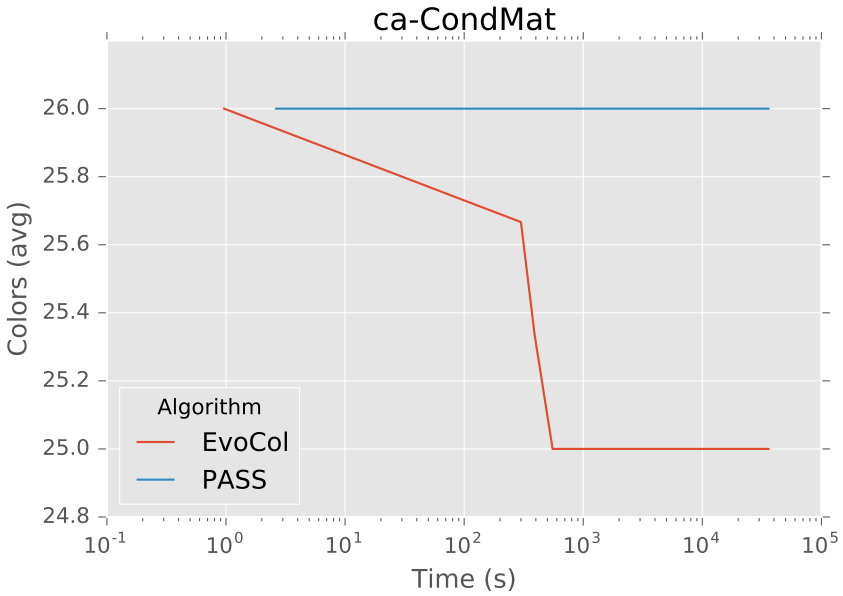


Figure 5.11: Convergence plots of EvoCol and PASS on the `ca-CondMat` graph

traverse a smaller portion of the search tree and it becomes harder to improve on an initial coloring. Also, the greedy DSATUR initialization requires a significant amount of the time. The `coPapers` graphs contain large cliques and therefore have large chromatic numbers; PASS is not even able to compute an initial coloring on these graphs. The advantage of a sophisticated initialization on the other hand makes PASS better on the graph `soc-Slashdot0902`, as can be seen in Figure 5.13. In general, both algorithms rarely improve on initial solutions in this family. The complete list of the SOCIAL family is shown in Table 5.4.

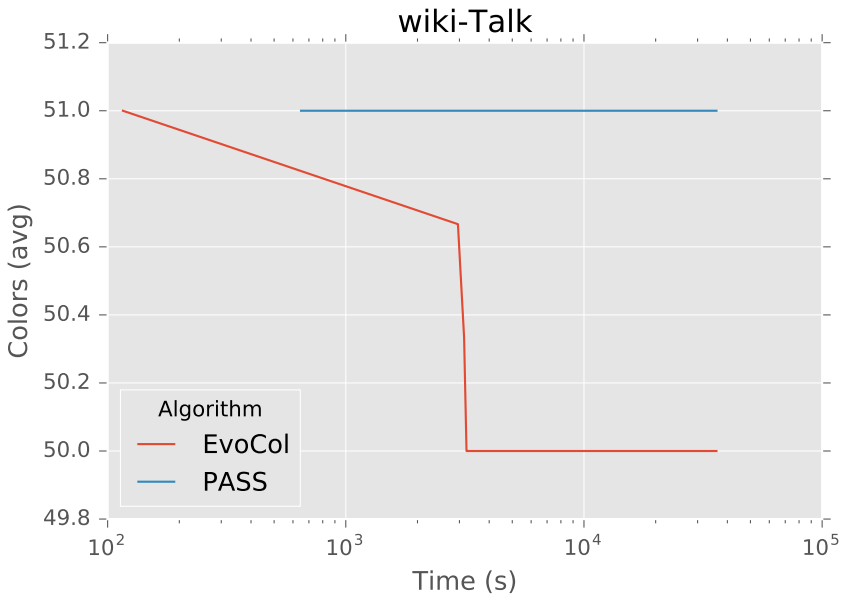


Figure 5.12: Convergence plots of EvoCol and PASS on the `wiki-Talk` graph

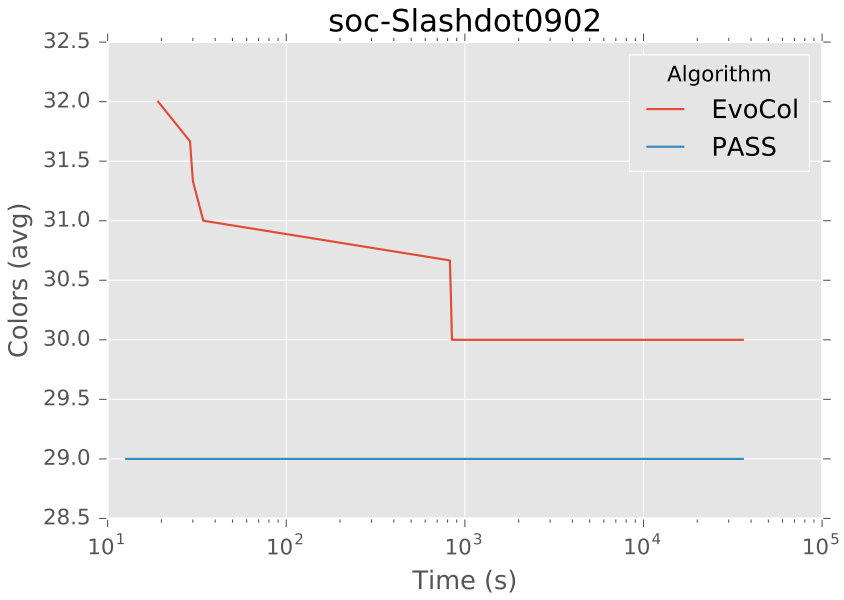


Figure 5.13: Convergence plots of EvoCol and PASS on the `soc-Slashdot0902` graph

Graph		EvoCol			PASS		
Name	n	Avg	Min	Max	Avg	Min	Max
astro-ph	16 706	57.00	57	57	57.00	57	57
ca-CondMat	23 133	**25.00**	*25*	25	26.00	26	26
citationCiteseer	268 495	13.00	13	13	13.00	13	13
coAuthorsCiteseer	227 320	87.00	87	87	87.00	87	87
coAuthorsDBLP	299 067	115.00	115	115	115.00	115	115
coPapersCiteseer	434 102	**845.00**	*845*	845	-	-	-
coPapersDBLP	540 486	**337.00**	*337*	337	-	-	-
email-Enron	36 692	25.00	25	25	25.00	25	25
soc-Slashdot0811	77 360	**29.00**	*29*	29	31.00	31	31
soc-Slashdot0902	82 168	30.00	30	30	**29.00**	*29*	29
wiki-Talk	2 394 385	**50.00**	*50*	50	51.00	51	51

Table 5.4: The results of both algorithms on the SOCIAL graphs. Bold and italic entries indicate a distinct advantage of the corresponding algorithm in the average and minimum number of colors, respectively. Missing entries indicate that the corresponding algorithm finds no coloring within the time limit.

The performance on the NETWORK graphs is similar to what we observed on the SOCIAL family. EvoCol maintains its advantage over PASS on these graphs, as PASS cannot finish initialization for `as-skitter` and performs equally on the other graphs. The results for all NETWORK graphs is given in Table 5.5.

Graph		EvoCol			PASS		
Name	n	Avg	Min	Max	Avg	Min	Max
as-skitter	1 696 415	**68.00**	*68*	68	-	-	-
p2p-Gnutella05	8846	5.00	5	5	5.00	5	5
web-Google	875 713	44.00	44	44	44.00	44	44

Table 5.5: The results of both algorithms on the NETWORK graphs. Bold and italic entries indicate a distinct advantage of the corresponding algorithm in the average and minimum number of colors, respectively. Missing entries indicate that the corresponding algorithm finds no coloring within the time limit.

The SIMULATION graphs are sparse compared to the other graphs in our test, hence their chromatic numbers are low and the graphs are easier to color. Our algorithm never performs worse than PASS and outperforms it on the graph with the highest number of colors in this family, `audikw1`. Figure 5.14 shows that PASS starts off with more colors than EvoCol despite the more sophisticated initialization method but cannot improve its initial coloring. The behaviour is similar to what we observed in

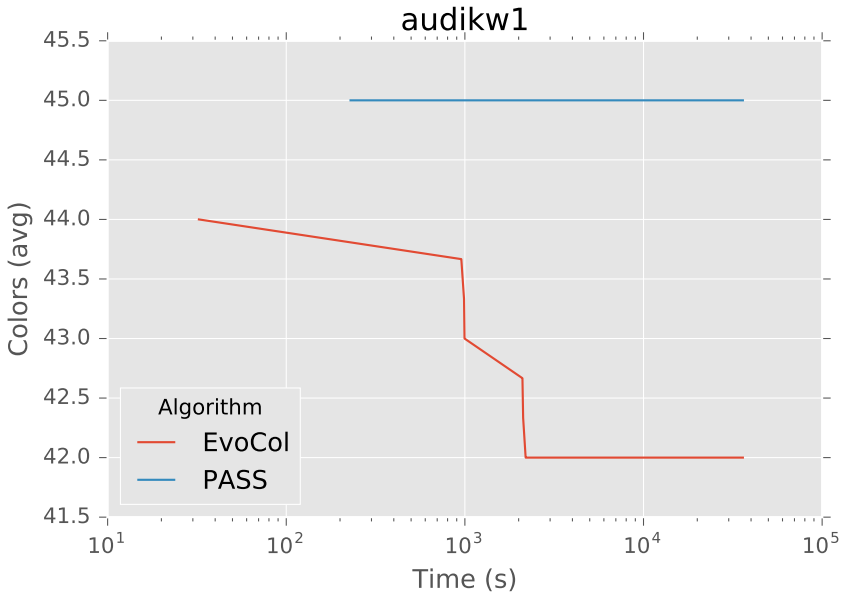


Figure 5.14: Convergence plots of EvoCol and PASS on the `audikw1` graph

Figure 5.11. All results for this family are given in Table 5.6.

Graph		EvoCol			PASS		
Name	n	Avg	Min	Max	Avg	Min	Max
af_shell9	504 855	22.00	22	22	22.00	22	22
audikw1	943 695	**42.00**	*42*	42	45.00	45	45
ecology1	1 000 000	2.00	2	2	2.00	2	2
ecology2	1 000 000	2.00	2	2	2.00	2	2
ldoor	952 203	34.00	34	34	34.00	34	34
thermal2	1 227 087	5.00	5	5	5.00	5	5

Table 5.6: The results of both algorithms on the SIMULATION graphs. Bold and italic entries indicate a distinct advantage of the corresponding algorithm in the average and minimum number of colors, respectively.

Finally, we perform a test on a large road map. Road maps typically have a very low chromatic number as they are sparse compared to the other graphs in our test set. Therefore, they are easy to color and as both algorithms found a 4-coloring for the graph `roadNet-CA` (see Table 5.7), we did not further investigate roadmaps.

Graph		EvoCol			PASS		
Name	n	Avg	Min	Max	Avg	Min	Max
roadNet-CA	1 965 206	4.00	4	4	4.00	4	4

Table 5.7: The results of both algorithms on the ROADMAP graphs

6 Discussion

6.1 Conclusion

This thesis introduced the graph coloring problem and covers notable work on this field. We further presented the evolutionary algorithm EvoCol for the GCP. We proposed new crossover operators as well as a novel approach to utilize tabu search to improve the offspring generated by our crossovers. Some of our crossovers require graph partitioning or vertex separators as crossover points. We used the KaHIP framework by Schulz [39] to generate partitions and separators of high quality. Additionally, we developed a new way to utilize tabucol to improve the solutions generated by our crossovers. Therefore we adjusted tabucol for our needs and analyzed the impact of our modifications.

Then we investigated in finding reasonable parameters for our algorithm and showed the effect of different initialization methods and crossovers. Finally, we measured the performance of EvoCol and the PASS algorithm by San Segundo [36]. We were able to surpass the results of PASS on most instances of the DIMACS family and outperformed it on nearly 50 % of the SOCIAL family. Our algorithm was designed to color large graphs and profits by the good partitions of KaHIP rather than complex operations on single vertices.

6.2 Further Work

Parallelization of our evolutionary algorithm is interesting in terms of performance. Major parts such as the initialization and the crossovers could be executed simultaneously. Also, multi-way separators could be investigated in combination with multi-parent crossovers.

The current way of diversity control can possibly be improved by more advanced replacement routines. We want to shed light on the effect of diversity among the population and build sensible replacement and selection methods. Also, techniques to refresh the population if the diversity becomes too low could be added. One could improve the performance by storing a set of partitions in a pool and select one randomly for the crossover. Currently a new partitioning is generated whenever the corresponding crossover is invoked. We further want to find parameter settings especially suited for certain graph families.

Bibliography

[1] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. part i: Discharging. *Illinois J. Math.*, 21(3):429–490, 09 1977.

[2] Andreas Björklund and Thore Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Algorithmica*, 52(2):226–249, 2008.

[3] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22 (4):251–256, 1979.

[4] Kimberly Ann Calton. Four color theorem. 2009.

[5] Massimiliano Caramia and Paolo Dell'Olmo. *A Fast and Simple Local Search for Graph Coloring*, pages 316–329. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48318-2.

[6] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, 2004.

[7] Marco Chiarandini and Thomas Stützle. An application of iterated local search to graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[8] David Dailey. Uniqueness of colorability and colorability of planar 4-regular graphs are np-complete. *Discrete Mathematics*, 30(3):289 – 293, 1980.

[9] Lawrence Davis. Order-based genetic algorithms and the graph coloring problem. 1991.

[10] Raphaël Dorne and Jin-Kao Hao. *A new genetic local search algorithm for graph coloring*, pages 745–754. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-49672-4.

[11] Raphaël Dorne and Jin-Kao Hao. Tabu search for graph coloring, t-colorings and set t-colorings. In *Meta-heuristics*, pages 77–92. Springer, 1999.

[12] Anna Esparcia-Alcazar. *Applications of Evolutionary Computation*. Springer, 2010.

[13] Charles Fleurent and Jacques Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63(3):437–461, 1996.

[14] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

[15] Philippe Galinier and Alain Hertz. A survey of local search methods for graph coloring. *Computers & Operations Research*, 33(9):2547 – 2562, 2006.

[16] Michael Garey, David Johnson, and Hing So. An application of graph coloring to printed circuit testing. *IEEE Transactions on Circuits and Systems*, 23(10): 591–599, 1976.

[17] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533 – 549, 1986.

[18] Pierre Hansen and Odile Marcotte. *Graph colouring and applications*. Number 23. American Mathematical Soc., 1999.

[19] Alain. Hertz and Dominique de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[20] Robert Irving. Np-completeness of a family of graph-colouring problems. *Discrete Applied Mathematics*, 5(1):111 – 117, 1983.

[21] David Johnson and Michael Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, 1996. ISBN 0821866095.

[22] David Johnson, Cecilia Aragon, Lyle McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

[23] Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[24] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[25] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489–506, 1979.

[26] Vahid Lotfi and Sanjiv Sarin. A graph coloring algorithm for large scale scheduling problems. *Computers & Operations Research*, 13(1):27 – 32, 1986.

[27] László Lovász. Three short proofs in graph theory. *Journal of Combinatorial Theory, Series B*, 19(3):269–271, 1975.

[28] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 286–293. ACM, 1993. ISBN 0-89791-591-7.

[29] Zhipeng Lü and Jin-Kao Hao. Adaptive tabu search for course timetabling. *European Journal of Operational Research*, 200(1):235 – 244, 2010.

[30] Zhipeng Lü and Jin-Kao Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203(1):241 – 250, 2010.

[31] Brad Miller and David Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[32] Craig Morgenstern and Harry Shapiro. Coloration neighborhood structures for general graph coloring. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 226–235. Society for Industrial and Applied Mathematics, 1990. ISBN 0-89871-251-3.

[33] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[34] Philippe Preux and El-Ghazali Talbi. Towards hybrid evolutionary algorithms. *International transactions in operational research*, 6(6):557–570, 1999.

[35] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 571–575. ACM, 1996. ISBN 0-89791-785-5.

[36] Pablo San Segundo. A new dsatur-based algorithm for exact vertex coloring. *Comput. Oper. Res.*, 39(7):1724–1733, 2012.

[37] Peter Sanders and Christian Schulz. *Think locally, act globally: Highly balanced graph partitioning*, pages 164–175. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38527-8.

[38] Peter Sanders and Christian Schulz. Kahip v0.53 - karlsruhe high quality partitioning - user guide. *CoRR*, abs/1311.1714, 2013.

[39] Christian Schulz. *High Quality Graph Partitioning*. epubli, 2013.

[40] Edward Sewell. An improved algorithm for exact graph coloring. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:359–373, 1996.

[41] Dominic Welsh and Martin Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1): 85–86, 1967.