Solving Weighted Constraint Satisfaction Problems (WCSPs) for Shift Scheduling using Fuzzy Graph Coloring in Python

Student Research Project (T3_3101)

from the Course of Studies Computer Science

at the Baden-Wuerttemberg Cooperative State University (DHBW) Stuttgart

by

Ferdinand Daniel König
Jonas Rheiner

June 10, 2022

Time of Project	September 29, 2021 - June 10, 2022
Student IDs, Course	9591527, 6177815, STG-TINF19A
Supervisor	Johannes Staib, B. Sc.

Author's declaration

Hereby I solemnly declare:

1. that this Student Research Project (T3_3101), titled *Solving Weighted Constraint Satisfaction Problems (WCSPs) for Shift Scheduling using Fuzzy Graph Coloring in Python* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Student Research Project (T3_3101) has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Student Research Project (T3_3101) in the past;
5. the printed version is equivalent to the submitted electronic one, if both are required.

I am aware that a dishonest declaration will entail legal consequences.

Ferdinand Daniel König

Jonas Rheiner

Abstract

Shift scheduling is part of resource planning and finding a solution is not trivial. Next to hard constraints (constraints that must be satisfied), soft constraints – e.g., a smaller likelihood for someone to be assigned to shifts on subsequent weekends – can be introduced. These are weighted with a probability or priority to be fulfilled but they can be neglected, too. Thus, shift scheduling poses a Weighted Constraint Satisfaction Problem (WCSP) which is NP-hard. This research project solves those WCSPs by generating a representative fuzzy graph and examines two main techniques to color those graphs. The proposed technique leverages a binary search on alpha-cuts to find an optimal solution. To measure a solution's quality, a metric from literature and an introduced metric to measure fairness are used. The color assignment is then interpreted as the shift assignment. The project's results are the Python package fuzzy-graph-coloring and a Python application that builds on top to generate a shift schedule as a Comma-Separated Value (CSV)-file out of a JavaScript Object Notation (JSON) input file with parameters like the time frame. A performance measurement regarding the time complexity is included.

Contents

Acronyms

BSD	Berkeley Software Distribution
CEX	Conflict Elimination Crossover
CLI	Command-Line Interface
CSP	Constraint Satisfaction Problem
CSV	Comma-Separated Values
CTM	Color Transposition Mutation
CV	Coefficient of Variation
DEAP	Distributed Evolutionary Algorithms in Python
DRY	Don't Repeat Yourself
DTI	Degree of Total Incompatibility
GA	Genetic Algorithm
GDB	GNU Project DeBugger
GUI	Graphical User Interface
HLSGA	Hybrid Local Search Genetic Algorithm
IEX	Incompatibility Elimination Crossover
JSON	JavaScript Object Notation
NP	Non-deterministic Polynomial time
NSP	Nurse Scheduling Problem
PyPI	Python Package Index
TI	Total Incompatibility
UCTP	University Course Timetable Problem
US	Unfairness Score
VRP	Vehicle Routing Problem
WCSP	Weighted Constraint Satisfaction Problem
XML	Extensible Markup Language

List of Figures

Listings

1 Introduction

Combinatorial optimization refers to searching for an optimal solution in a finite set of possible solutions [1]. Famous examples are the Traveling-Salesman Problem [2], the Knapsack Problem [3], [4], and the Minimal-Spanning-Tree [5]. Another subdomain of combinatorial optimization is the Constraint Satisfaction Problem (CSP). A CSP requires a value, selected from a given finite domain, to be assigned to each variable in the problem so that all constraints relating to the variables are satisfied [6]. Many practical challenges can be modeled as CSPs. Simplified practical examples include location assignment of warehouses to supply customers, frequency assignment of radio stations, timetabling classroom lessons, or scheduling processor tasks [6], [7]. However, modeling real-world examples as CSP frequently requires a more nuanced consideration of constraints. In other words, *soft constraints* do not have to be satisfied under all circumstances and only affect the degree of optimality of a solution [8]. Assigning a weight to each constraint in a given problem enables this. Common examples for a so-called Weighted Constraint Satisfaction Problem (WCSP) are the Nurse Scheduling Problem (NSP) and the University Course Timetable Problem (UCTP) featuring soft constraints each [8], [9].

1.1 Motivation and Problem Statement

Shift scheduling is a common challenge across many industry sectors. However, it is crucial in medical care. A clinic has to deploy its medical personnel optimally to ensure that the correct expertise is available at any given time, even around the clock. This problem has attracted significant attention in recent years due to the ongoing COVID-19 pandemic. As a result of the pandemic, many clinics are overloaded and medical staff is overworked. Under such conditions, efficient and optimal shift planning is vital. However, it also is a challenging problem due to the many factors and constraints. For example, the clinic must satisfy contractual obligations such as each staff member working only a maximum of one shift per day. Furthermore, it is possible that staff members' specific skill sets or fields of expertise must be regarded.

In general, shift scheduling is a Constraint Satisfaction Problem (CSP). However, to more accurately represent real-world problems, such as the one described above, it must be treated as Weighted Constraint Satisfaction Problem (WCSP). Some constraints in shift scheduling are soft. To build on the example given above, a soft constraint is that the

working time for each staff member *should be* consistent on all working days. Ideally, a staff member works only early shifts all week or only late shifts. Nevertheless, if the overall schedule turns out considerably better or is only possible if a staff member works early and late shifts in the same week, the soft constraint can be violated.

Finally, creating an optimal schedule without bias becomes increasingly complex and inefficient with the number of staff members and constraints. Therefore, this paper aims to develop an application that outputs an optimal shift schedule based on the input data. The output must consider the given constraints.

1.2 Properties of Desired Solution

To solve the WCSP, the application models it as a graph and finds a graph coloring. It is important to note that the goal is not to find an arbitrary solution but instead an optimal and fair solution. As part of the search for this, the application must estimate how optimal a given solution is. If a solution exists, the application must always output it, even if the only valid solution can be considered sub-optimal. The program's termination is expected if no solution is possible on the given input data and constraints. However, a method returning a less-than-ideal solution is preferred over no valid output for the applications solving algorithms.

1.3 Intended Procedure

The procedure to tackle the previously described problem will be set as follows. First, an introduction to the fundamental theory, including graph coloring and fuzzy graphs, is given. Furthermore, two techniques regarding fuzzy graph coloring are presented. Moreover, previous work on the topic and other algorithmic approaches to CSPs, which are not part of the solution presented in this paper, are considered. In chapter 3, the implemented constraints and the criteria for measuring the quality of a solution are defined. Chapter 4 covers the implementation of the final program. Two different solving methods will be discussed regarding their implementation and the solution's quality. The time complexity will be compared. Furthermore, the presented application is validated by using examples or a benchmark with increasing complexity levels. Finally, the paper concludes the project by providing the final application – e.g., as a python package or via a code repository –, reflecting on the presented work, giving an outlook, and proposing possible future work.

2 Basics and State of the Art

This chapter introduces graph coloring and fuzzy graphs as fundamentals. Furthermore, two different methods for coloring fuzzy graphs are defined. Alternative fuzzy graph colorings are outlined, too. Finally, different algorithmic approaches to the CSP and previous work on the topic – including existing implementations and other solutions – are discussed.

2.1 Graph Coloring

To define graph coloring, it is essential to first introduce general graph theory. A graph G is a pair of sets (V, E), where V is a finite set of elements called vertices, and E is a finite set of elements called edges, each of which has two associated vertices [10, p. 1]. The sets V and E are referred to as vertex-set $V(G)$ and the edge-set $E(G)$ of graph G [10, p. 1]. Two vertices v, w are called adjacent if $(v, w) \in E(G)$. The number of vertices of G is called the order of G and denoted by n or $|G|$. The number of edges is given by m [10, p. 1].

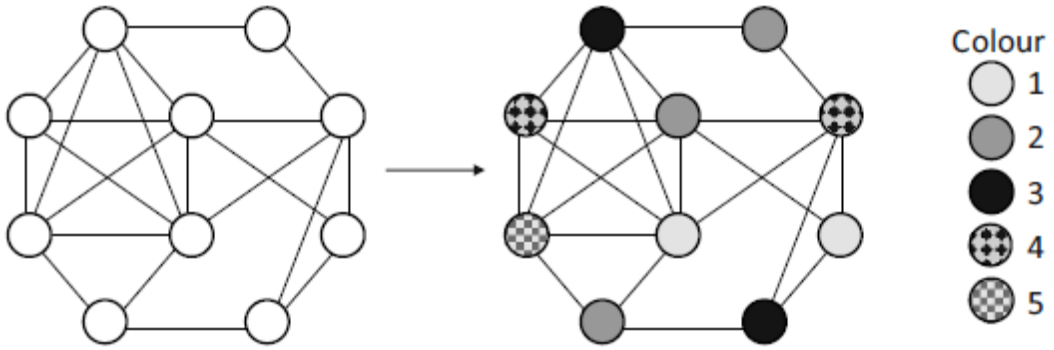

Figure 2.1: Graph (left) and the corresponding 5-coloring (right) [11, Fig. 1.1]

Graph coloring is an assignment of a color to each vertex of a graph such that adjacent vertices receive different colors [10, p. 9]. In literature, the notion of graph coloring can also include or exclusively refer to assigning colors to edges [10, pp. 11f.], [12], [13]. This paper applies graph coloring only to vertices. An example graph and its corresponding coloring are depicted in figure 2.1.

To create a colored graph G, we use the finite set C with k elements and the coloring φ^k which is defined as [10, pp. 9f.]:

$$\varphi^k : V(G) \to C = \{1, \ldots, k\},\ \textit{where}\ \varphi(v) \neq \varphi(w)\ \textit{for}\ (v, w) \in E(G)$$

The set C contains integers which represent different colors in practice [14, p. 1].

A graph G is *k-colorable* if a coloring φ^k with k colors exists. Then φ^k is also called a *k-coloring* of G. The minimum number k, for which G is *k-colorable*, is defined as the chromatic number $\chi(G)$ of G [7, p. 1], [10, p. 9], [14, p. 1]. Finding a *k-coloring* of G with $k = n$ is trivial but computing the chromatic number is an Non-deterministic Polynomial time (NP)-complete problem [10, p. 9]. Therefore, like other NP problems, the correctness of a solution can be verified in polynomial time. Furthermore, this means that computing the chromatic number is a problem that is as hard to solve as any other NP-complete problem (e.g., Traveling-Salesman Problem [2], Knapsack Problem [3], [4]) [15, pp. 1-6].

2.2 Fuzzy Graphs

To define a fuzzy graph, we have to give a definition for a fuzzy set. We denote X as the universe set and x a generic element of it. $\tilde{A}$ is then a fuzzy subset of X. It consists of ordered pairs:

$$\tilde{A} = \{(x, f_{\tilde{A}}(x)) \mid x \in X\},$$

where $f_{\tilde{A}} : X \to [0, 1]$ is the so-called *membership function* of $\tilde{A}$. Then, the value of $f_{\tilde{A}}(x)$ at x is the *degree of membership* and tells how much x is a member of X [16, p. 339]. Fuzzy sets and their properties are further discussed by Zadeh [16].

Generally, $G = (V, E)$ is a graph with the vertex set V and a set of edges $E \subseteq V \times V$. For a fuzzy graph $\tilde{G}$, the fuzzy subsets of V and E are used. For the sake of simple notation, we denote these as their membership function μ and ρ. Other literature prefers the notation $\tilde{G} = (V, \mu, \rho)$ to explicitly define the set of vertices. Since V is implicitly given by μ, $\tilde{G} = (\mu, \rho)$ is also sufficient. Now, $\tilde{G} = (\mu, \rho)$ is called a *fuzzy sub-graph* of G if the restriction

$$\rho(u, v) \leq \min\{\mu(u), \mu(v)\}\ \forall u, v \in V \tag{2.1}$$

holds, because just then, ρ can be a *fuzzy relation* on μ [17, pp. 3f.]. A further elaboration is out of the scope of this paper.

For $\tilde{G}$, there is an underlying crisp graph $G^* = (\mu^*, \rho^*)$, where

$$\mu^* = \{u \in V \mid \mu(u) > 0\} \; and \tag{2.2}$$
$$\rho^* = \{(u, v) \in E \mid \rho(u, v) > 0\}. \tag{2.3}$$

Therefore, G^* is $\tilde{G}$ but without weights [18, p. 885].

Since our use cases imply a crisp vertex set – thus, $\mu(u) = 1 \; \forall u \in V$ – but a fuzzy edge set, these fuzzy graphs are denoted by $\tilde{G} = (V, \rho)$. For more, see section 2.4. Hence, the restriction (2.1) is fulfilled in every case. For constraint-modeling, the degree of membership $\rho(u, v)$ for $u, v \in V$ is interpreted as a measure of incompatibility between those two vertices. In other words, an edge between two vertices is a constraint with a certain weight.

2.3 The α-Cut

Another option to transform a fuzzy graph $\tilde{G} = (\mu, \rho)$ to a crisp graph is the α-cut [19, p. 37]. For any $\alpha \in (0, 1]$, the α-cut graph of $\tilde{G}$ is the crisp graph $G_\alpha = (\mu_\alpha, \rho_\alpha)$ where

$$\mu_\alpha = \{u \in V \mid \mu(u) \geq \alpha\} \; and \tag{2.4}$$
$$\rho_\alpha = \{(u, v) \in E \mid \rho(u, v) \geq \alpha\}. \tag{2.5}$$

Here again, $\forall u \in V : \mu(u) = 1$ and hence $\mu_\alpha = V$. That implies for this problem $G_\alpha = (\mu_\alpha, \rho_\alpha) = (V, \rho_\alpha)$. Conclusively, α is a threshold and all edges with weights equal or above are kept as crisp edges and all other edges are discarded.

2.4 Shift Scheduling using Fuzzy Graph Coloring

To utilize fuzzy graph coloring for shift scheduling, an interpretation for the elements of a fuzzy graph must be given. As mentioned before, there is no advantage of using a fuzzy set for vertices since a vertex is interpreted as one shift. The edges between shifts are constraints, e.g., a team member cannot be assigned to a pair of subsequent shifts. Staff members are colors. By coloring the graph – which is the entirety of shifts and all constraints in a given period – the shifts are assigned to employees. In contrast to vertices, edges are weighted and therefore represented by a fuzzy set. The weights are the importance or priority of a constraint.

2.5 Fuzzy Graph Coloring

The term *fuzzy graph coloring* is not clearly defined. Here, different techniques are presented.

2.5.1 Leveraging Transformation to Crisp Graphs

Since graph coloring is well-defined for crisp graphs, transformations are often used to apply it to fuzzy graphs. The most popular technique is to use α-cuts [13], [20, pp. 213f.], [21], [22] or the underlying crisp graph G^* (formulae (2.2), (2.3)) [21]. On the resulting crisp graph G_α or G^*, ordinary graph coloring can be utilized.

2.5.2 Fuzzy Graph Coloring According to Keshavarz

Fuzzy graph coloring by Keshavarz [18] matches the requirements of the problem on hand the closest. With this algorithm, analogous to the classical graph coloring, a fuzzy graph $\tilde{G} = (V, \rho)$ can be k-colored with a map $\varphi^k : V \to C = \{1, \ldots, k\}$. Special for the fuzzy case is the behavior that two adjacent vertices u, v can share the same color: $\varphi^k(u) = \varphi^k(v)$. We say u and v are incompatible under φ^k with the degree of incompatibility $\rho(u, v)$. $(u, v) \in E$ is an incompatible edge [18, p. 885].

With that, we can define some measures for a k-coloring φ^k of a fuzzy graph $\tilde{G} = (V, \rho)$. Let

$$IE(\varphi^k) := \{(i, j) \in \rho^* \mid \varphi^k(u) = \varphi^k(v)\}$$

be the set of all incompatible edges associated with φ^k.

Building on top of this, we define the *total incompatibility* of φ^k as the sum of each incompatible edge's degree of incompatibility

$$TI(\varphi^k) := \sum_{(u,v) \in IE(\varphi^k)} \rho(u, v)$$

and the *degree of total incompatibility* of φ^k as

$$DTI(\varphi^k) := \frac{TI(\varphi^k)}{\sum_{(u,v) \in \rho^*} \rho(u, v)} = \frac{\sum_{(u,v) \in IE(\varphi^k)} \rho(u, v)}{\sum_{(u,v) \in \rho^*} \rho(u, v)}.$$

As a result, $DTI(\varphi^k)$ is the ratio of the sum of each incompatible edge's weight ($TI(\varphi^k)$) and the sum of all weights. Note that it holds: $DTI(\varphi^k) \in [0, 1]\ \forall k : 1 \leq k \leq n = |V|$ and $DTI(\varphi^1) = 1 \wedge DTI(\varphi^n) = 0$ [18, p. 885].

For a given k, there are different possible k-colorings of $\tilde{G}$, which violate different constraints and result in different sets of incompatible edges $IE(\varphi^k)$. We denote the coloring which minimizes the degree of total incompatibility DTI as $\varphi^k_{\min}$, which is named a minimal k-coloring. The *fuzzy chromatic number* $\tilde{\chi}$ of $\tilde{G}$ itself is a fuzzy set [18, p. 885]:

$$\tilde{\chi}_{\tilde{G}} = \{(k, 1 - DTI(\varphi^k_{\min})) \mid k = \{1, \ldots, n\}\} \tag{2.6}$$

with the corresponding optimal coloring set

$$\Gamma_{\tilde{G}} = \{\varphi^1_{\min}, \ldots, \varphi^n_{\min}\}. \tag{2.7}$$

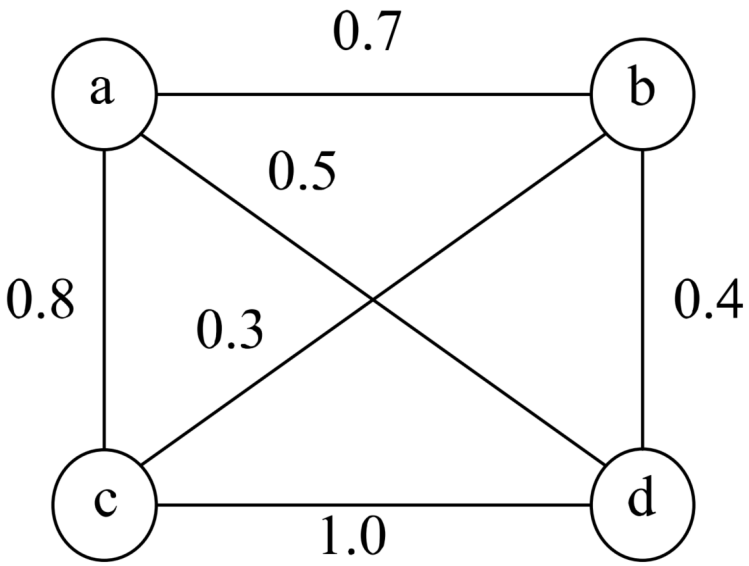

Figure 2.2: Example 1 of a fuzzy graph from [18, p. 886]

Consider the example given in fig. 2.2. It shows the fuzzy graph $\tilde{G} = (V, \rho)$, where $V = \{a, b, c, d\}$ and $\rho = \{((a, b), 0.7), ((a, c), 0.8), ((a, d), 0.5), ((b, c), 0.3), ((b, d), 0.4), ((c, d), 1.0)\}$. If the graph is colored with only one color, then all edges are incompatible and $IE(\varphi^1) = \rho^*$. Therefore, the degree of total incompatibility is $DTI(\varphi^1) = \frac{0.7+0.8+0.5+0.3+0.4+1.0}{0.7+0.8+0.5+0.3+0.4+1.0} = \frac{3.7}{3.7} = 1$. Trivially, for φ^1, there is just one possible coloring [18, p. 886].

There exist multiple different colorings for $k = 2$. For instance, two valid options are

$$\varphi^2_1 = \{(a, 1), (b, 1), (c, 2), (d, 2)\}, \text{ and}$$
$$\varphi^2_2 = \{(a, 1), (b, 2), (c, 2), (d, 1)\}$$

which have a different degree of total incompatibility associated with them:

$$IE(\varphi_1^2) = \{(a,b),(c,d)\} \implies DTI(\varphi_1^2) = \frac{0.7+1.0}{3.7} \approx 0.459$$
$$IE(\varphi_2^2) = \{(a,d),(b,c)\} \implies DTI(\varphi_2^2) = \frac{0.5+0.3}{3.7} \approx 0.216.$$

Verifiably, φ_2^2 minimizes DTI and hence $\varphi_2^2 = \varphi_{\min}^2$.

The minimal 3- and 4-colorings are:

$$\varphi_{\min}^3 = \{(a,1),(b,2),(c,2),(d,3)\} \implies$$
$$IE(\varphi_{\min}^3) = \{(b,c)\} \implies DTI(\varphi_{\min}^3) = \frac{0.3}{3.7} \approx 0.081$$
$$\varphi_{\min}^4 = \{(a,1),(b,2),(c,3),(d,4)\} \implies$$
$$IE(\varphi_{\min}^4) = \emptyset \implies DTI(\varphi_{\min}^4) = \frac{0}{3.7} = 0$$

Finally, the optimal coloring set of $\tilde{G}$ is $\Gamma_{\tilde{G}} = \{\varphi_{\min}^1, \varphi_{\min}^2, \varphi_{\min}^3, \varphi_{\min}^4\}$ and the fuzzy chromatic number is

$$\tilde{\chi}_{\tilde{G}} = \{(1,0),(2,0.784),(3,0.919),(4,1)\}.$$

2.5.3 Other Kinds of Fuzzy Graph Coloring

In this section, other methods of fuzzy graph coloring are commented on in relation to why they are not useful for the presented problem. Papers where the objective was to color edges, are categorically left out.

Samanta et al. [21] proposed fuzzy coloring of fuzzy graphs. As the name suggests, they introduce the concept of fuzzy colors, which is not applicable for using vertices as shifts in a timetable scheduling problem.

Muñoz et al. [20] used two different approaches: α-cuts and *(d,f)-extended coloring*. The first is also used as one of this project's approaches. The latter introduces d, a dissimilarity measure between colors. This is contradictory to this paper's use case. A solution would be two define a constant d between colors. The algorithm colors a graph such that the distance between the colors of two adjacent vertices must be at most the weight of the connecting edge. This makes no sense for the problem on hand, in which a connecting edge is a constraint and the weight is a priority of the constraint, given that d is a constant. In addition, there are cases where no *(d,f)-extended coloring* exists.

2.6 Other Algorithmic Approaches to the CSP

In contrast to graph coloring, there exist other algorithmic approaches to the CSP.

As the CSP features a finite domain, the two most common techniques used in solving algorithms are backtracking and local search combined with heuristics [23], [24]. An algorithm utilizing backtracking will instantiate all variables of the CSP sequentially and check the validity of a constraint as soon as all relevant variables are assigned. If a partial instantiation of variables violates a constraint, backtracking is performed, and the variables are reverted to the most recent valid assignment [23, pp. 33f], [24, pp. 274f]. This method is superior to generating and testing all possible variable combinations because each time a partial variable assignment violates a constraint, a subspace of all possible assignments can be eliminated [23, p. 34]. Still, this method is unpractical for non-trivial problems due to exponential run time complexity in most cases [23, p. 34]. Further inefficiencies of backtracking are proven in [25, pp. 100f].

Other algorithms – which try to find a good approximate solution – use heuristics. One example is tabu search. It was proposed in 1986 [26] with the motivation to allow local search methods to overcome local optima [27, p. 37]. In general, the tabu search technique is used to move step by step from an initial feasible solution towards a solution minimizing an objective function [27, p. 39], [28, p. 346]. A key component during this process is the tabu list which is initialized empty and then filled each iteration with actions that would bring the algorithm back to a solution of a previous iteration [27, p. 37]. Tabu search was successfully deployed in [28] to solve graph coloring limited to crisp graphs. More practical usage is presented in [29] and [30] by solving the UCTP. Another example is given [31] with the Vehicle Routing Problem (VRP).

2.7 Genetic Algorithm

A Genetic Algorithm (GA) is a stochastic algorithm to search for solutions to complicated optimization problems. It was introduced by Holland [32] and is inspired by natural selection and natural genetics. The basic idea is presented in this section.

Prerequisites for the GA are

- a chromosome, which is a genetic representation in the form of a bit string that corresponds to the problem encoding, and
- a fitness function to evaluate how good a chromosome, i.e., a particular solution is [33, pp. 2–4].

The first generation's set of chromosomes – called first population – is generated randomly.

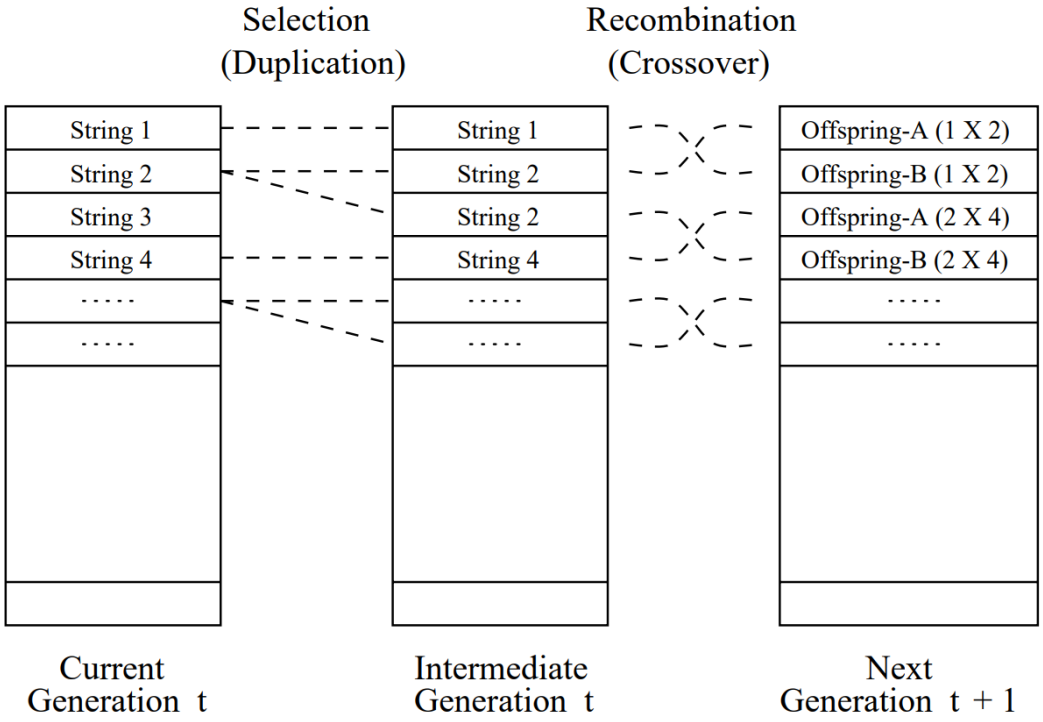

Figure 2.3: Genetic Algorithm: Steps to derive next generation from current generation [33, p. 5]

Figure 2.3 depicts the transition process from a given generation t to its successor $t+1$ applying duplication and crossover. In a population, each chromosome's fitness is measured. Then, selection is applied. It is a stochastic process in which chromosomes with higher fitness have a greater chance of being selected for the generation of the next population. This will be called *offspring*. In figure 2.3, this is accomplished by assigning each string to a chance of duplication based on their fitness score and duplicating them stochastically. The result is an intermediate generation [33, p. 5].

Now, genetic operators are used. The first one is crossover, also known as recombination. Each step uses two parent chromosomes of the intermediate generation and recombines them probabilistically. The second genetic operator is mutation. It is an additional step that can be applied after crossover but is not shown in figure 2.3. The most prevalent version is to flip each bit of a population with a certain low probability of typically below 1% [33, p. 6].

This process of evaluation, selection, crossover, and mutation creates one generation. The population size is kept constant. By repeating, a (locally) optimal solution is approximated. A further explanation of the mathematical details is out of the scope but can be read in Whitley [33].

2.8 Existing Algorithms and Implementations

This part of the paper is about concrete fuzzy graph coloring algorithms and their implementations.

2.8.1 Hybrid Local Search Genetic Algorithm (HLSGA)

Keshavarz [18] discusses how to solve his theoretical, mathematical method of fuzzy graph coloring computationally. The method is considered in subsection 2.5.2.

For a fuzzy graph $\tilde{G} = (V, \rho)$, $\rho = [\rho_{ij}]$ denotes the matrix of incompatibility degrees of vertices, i.e., the weight of the edges. The binary variables x_{ir} and y_{ij} are introduced as follows [18, p. 886].

$$x_{ir} = \begin{cases} 1, & \text{if color } r \text{ is assigned to vertex } i \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

$$y_{ij} = \begin{cases} 1, & \text{if vertices } i \text{ and } j \text{ are assigned to the same color} \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

Then, the problem can be reduced to a binary programming problem which is a combinatorial optimization problem:

$$\min TI_k = \sum_{(i,j)\in\rho^*} \rho_{ij} y_{ij} \tag{2.10a}$$

$$s.t. \quad \sum_{r=1}^{k} x_{ir} = 1, \qquad i \in V \tag{2.10b}$$

$$\sum_{i\in V} x_{ir} \geq 1, \qquad r = 1, \ldots, k \tag{2.10c}$$

$$x_{ir} + x_{jr} - y_{ij} \leq 1, \qquad (i,j) \in \rho^* \tag{2.10d}$$

in which (2.10a) is the objective function to minimize the total incompatibility. (2.10b) ensures that every vertex is assigned to exactly one color. The statement '*for each color r, there exists at least one vertex V that is assigned to it*' is guaranteed by constraint (2.10c). Constraint (2.10d) forces $x_{ir} = x_{jr} = 1 \implies y_{ij} = 1$ and for $x_{ir} + x_{jr} \leq 1$, it allows both values of y_{ij} but minimizing (2.10a) encourages $y_{ij} = 0$ [18, p. 886].

Finding an optimal solution is an NP-hard problem. To approximate a good solution, iterative methods such as Genetic Algorithms (GAs) are used. The following steps are required for a GA:

Encoding: To apply a GA, a feasible solution must be encoded as a chromosome. For a k-coloring $\varphi^k : V \to C = \{1, \ldots, k\}$ $(k \in 1, \ldots, n)$, the corresponding chromosome is $U(\varphi^k) = (u_1, u_2, \ldots, u_n) \in C^n$, where $u_i = \varphi^k(i),\ i = 1, \ldots, n$. That means, u_i is the color of the i$^{\text{th}}$ vertex [18, p. 887].

Initialization: The chromosomes of the initial population are constructed by randomly assigned colors to vertices [18, p. 887].

Fitness Function: The objective function (2.10a), i.e., the total incompatibility is used [18, p. 887].

Selection Operator: Keshavarz decided to use the classic *tournament method*. Here, a small group of chromosomes is randomly drawn from the population. The bit string with the best fitness is chosen for the genetic operations. After the second iteration of drawing, two chromosomes are given as parents [18, p. 887].

Crossover Operator: In [18], the used crossover is the Incompatibility Elimination Crossover (IEX) which is an adaptation of the Conflict Elimination Crossover (CEX) that was designed by Kokosinski et al. [34, pp. 130f.]. In IEX, compatible colors are copied without violating feasibility. The important constraint is (2.10c).

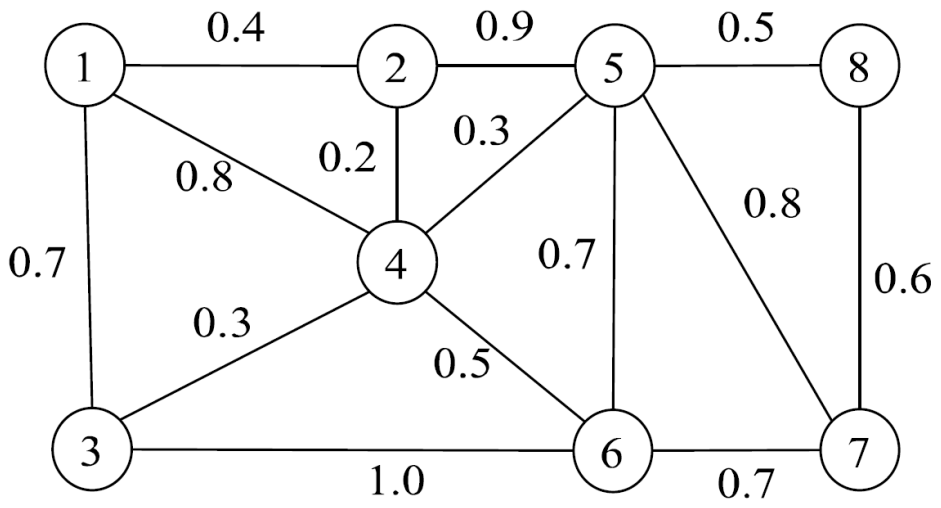

Figure 2.4: Example 2 of a fuzzy graph from [18, p. 888]

All compatible genes, i.e., genes with compatible colors, remain unmodified by IEX. All incompatible genes are replaced by the other parent's genes except for one. Therefore, (2.10c) and hence feasibility is given [18, p. 887]. The mechanism is explained subsequently with an example. Consider the fuzzy graph $\tilde{G} = (V, \rho)$ given by figure 2.4. Let $U_1 = (\underline{\mathbf{1}}, 2, \mathbf{1}, \underline{\mathbf{3}}, 4, \underline{\mathbf{3}}, 2, 1)$ and $U_2 = (\underline{\mathbf{4}}, 3, 1, \underline{\mathbf{4}}, 2, 3, 1, 3)$ be parent chromosomes of a 4-coloring of $\tilde{G}$. Incompatible genes are printed bold and are underlined. The offspring copies all compatible genes and replace the other but one with the genes of the other parents: $S_1 = (\underline{\mathbf{4}}, 2, 1, \underline{\mathbf{4}}, \underline{\mathbf{4}}, 3, 2, 1)$ and $S_2 = (\underline{\mathbf{1}}, 3, \underline{\mathbf{1}}, 4, 2, 3, 1, 3)$. Note that the total incompatibility TI of U_1 to S_1 was reduced from 1.7 to 1.1 and the TI of U_2 and S_2 are 0.8 and 0.7 [18, p. 888].

Mutation Operator: A Color Transposition Mutation (CTM) is applied. It randomly selects two colors and switches them mutually [18, p. 888]. Consider $S_1 = (4, 2, \underline{\mathbf{1}}, 4, 4, \underline{\mathbf{3}}, 2, \underline{\mathbf{1}})$ from above with the randomly selected and marked colors 1 and 3, then the result of CTM is $S_{1,m} = (4, 2, \underline{\mathbf{3}}, 4, 4, \underline{\mathbf{1}}, 2, \underline{\mathbf{3}})$. At first, the mutated solution is equivalent to the original. The desired effect is that in the next generation it will result in a different crossover and adds to variation.

Local Search Operator: The steps above are the components of the GA. According to Keshavarz, a local search on the color space of chromosomes can significantly boost their fitness [18, pp. 890f.]. After crossover and mutation, the local search is performed on a random set of chromosomes. Consider code listing 2.1: 'Procedure: Local Search':

```
1  Input: A k-coloring chromosome U = (u_1, ..., u_n),  u_i ∈ {1, ..., k}
2
3  begin
4      TI* ← TI(U);
5      for i from 1 to n do
6          for r from 1 to k do
7              tempt ← u_i;
8              u_i ← r;
9              if TI(U) < TI* then
10                 TI* ← TI(U);
11                 break;
12             else
13                 u_i ← tempt;
14             end if;
15         end do;
16     end do;
17 end;
```

2.1: Procedure: Local Search (from [18, p. 888])

It iterates over the given chromosome and checks for each position if there is another color that has a better total incompatibility TI. For an example of a valid local search: Let $U = (1, 2, 2, 3, 2, 3, 1, 2)$ be a chromosome for a 3-coloring of the fuzzy graph of fig. 2.4. Note that it holds $TI(U) = 1.9$. By performing algorithm 2.1, a new chromosome with a better TI of 0.9 is obtained: $U' = (1, \underline{\mathbf{1}}, 2, 3, 2, 3, 1, \underline{\mathbf{3}})$ [18, p. 888].
It is not explicitly pointed out in the paper, but it is crucial that the output of the local search is itself a feasible solution, i.e., that constraint (2.10c) is fulfilled.

As the components of the Hybrid Local Search Genetic Algorithm (HLSGA) are given, a complete overview of the algorithm is appropriate. The HLSGA's pseudocode is presented in code listing 2.2. It is important to note that the population is not replaced but appended with the respective offspring as defined in lines 12 and 16. This causes the resulting temporary population to exceed the population size. The next population is then created by copying the temporary population and eliminating the extra members introduced through the previously mentioned steps. Prior to the elimination process, all population chromosomes are sorted by their fitness value. Therefore, the worst chromosomes are discarded at the end of each generation; see lines 17 and 18.

```
1  Input: Fuzzy graph data: n, ρ = [ρ_ij], number of colors GA parameters
2
3  begin
4      t ← 0; // generation counter
5      Initialize a randomly generated population
6      P(t) = {U_1, U_2, ..., U_ps} ⊂ {1, ..., k}^n of chromosomes
7      // ps is the population size
8      Evaluate P(t) using the fitness function;
9      while (not termination condition) do
10         Select the parental population T1(t) from P(t);
11         Create the offspring populations OX(t) and OM(t) from T1(t) by
               IEX and CTM;
12         P_temp(t) ← P(t) ∪ OX(t) ∪ OM(t);
13         Evaluate P_temp(t) using the fitness function;
14         Select the parental population T2(t) from P_temp(t);
15         Create the offspring populations OLS(t) from T2(t) by Local
               Search Procedure;
16         P_temp(t) ← P_temp(t) ∪ OLS(t);
17         Evaluate and sort P_temp(t) members using the fitness function;
18         Select new population P(t+1) from P_temp(t), by eliminating extra
               members;
19         t ← t + 1;
20     end do;
21 end;
22
23 Output: The best solution (k-coloring)
```

Listing 2.2: HLSGA (from [18, p. 888])

Keshavarz implemented his algorithm in MATLAB but does not provide the code. With the program, experiments were conducted and compared to IBM's CPLEX – an optimization software. An interesting insight is that for a fuzzy graph with 100 vertices and 2501 edges and considering a 2- to 17-coloring, with CPLEX, the mean fitness is 101, but for HLSGA it is 79. Remarkable is the required time. CPLEX took 25 minutes whereas HLSGA needs anywhere from 11 seconds to 1 minute 15 seconds [18, p. 889].

In another test, one can conclude that with the local search, the GA is 100 times slower, but yields three times better results [18, Table 6 on p. 891].

2.8.2 Python Packages

There is no package or code-snippet in Python that can perform vertex coloring of a fuzzy graph. Packages to color ordinary crisp graphs do exist.

libcolgraph

`libcolgraph` is a package available on Python Package Index (PyPI). The latest version is 0.0.7 which was released in September, 2020. It is created by the University of Richmond and written for Python 3.5. Python 3.4 and above is supported. The package depends on `NetworkX` [35].

NetworkX

Another package on PyPI is `NetworkX`. As of the time of writing this subsection, the latest release is 2.6.3 from September 2021. It officially supports all Python versions between 3.7 and 3.9. The purpose of this package is the creation and management of graphs and networks [36].

It offers two kinds of graph coloring [37, pp. 292 ff.]. The first type is a greedy coloring algorithm that uses as few colors as possible. The default strategy to accomplish a valid, optimal coloring is *largest-first* which is described by Kosowski & Manuszewski [38].

The second kind is equitable coloring. Here, the number of colors is given and must be one more than the maximum degree of nodes in the graph or higher. It tries to create a coloring such that the sizes of color classes differ by at most one [39].

`NetworkX`'s code is licensed under the 3-clause BSD license. It allows commercial use and a free modification of the code. For a redistribution, the full text of the license and the original copyright notice must be included [40].

2.8.3 Existing Fuzzy Graph Coloring Program

There is a paper with the title *VERTEX COLORING OF A FUZZY GRAPH* which provides a program written in C++ to color a fuzzy graph [41].

We have not been able to use it on fuzzy graphs in general. After fixing bugs including changing `<iostream.h>` to `<iostream>` to make it compliant with any C++-standard beginning with C++98 [42, pp. 319 + 602ff.] – as well as C++11, the latest version at the time when [41] was published [43, pp. 418 + 1001ff.] – and adding commas to match the syntax of functions, the code did execute.

The reworked code was run in a C++11 environment using GDB. The code difference can be found in appendix A.

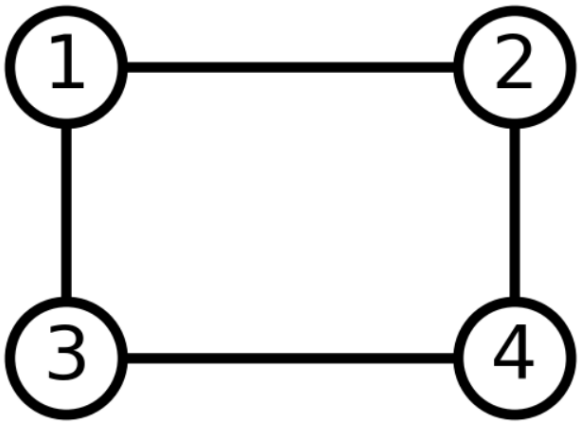

Figure 2.5: Simple undirected, unweighted graph

The example presented in [41, pp. 57f.] does not match the input and output of the application. Lines are missing, and the format is different, too. The recreation of the calculation was possible.

As a next step, a simple unweighted graph (fig. 2.5) was given as input.

The output can be seen in listing 2.3:

```
Enter the number of vertices4
Enter the membership value of vertex 1= 1
Enter the membership value of vertex 2= 1
Enter the membership value of vertex 3= 1
Enter the membership value of vertex 4= 1
Enter the edges 0(0 to 0)1 2
Enter the membership value of this edge1
Enter the edges 1(0 to 0)2 4
Enter the membership value of this edge1
Enter the edges 2(0 to 0)4 3
Enter the membership value of this edge1
Enter the edges 3(0 to 0)3 1
Enter the membership value of this edge1
Enter the edges 4(0 to 0)0 0
Graph is fuzzy graph
chromatic number=4
Vertex 1 is colored 1 Vertex 2 is colored 2 Vertex 3 is colored 3
    Vertex 4 is colored 4
```

Listing 2.3: Output of existing fuzzy graph coloring program

In this example, it is expected to get a chromatic number of 2. Vertices 1 and 4, as well as vertices 2 and 3, can share the same color. However, the program is not able to find this trivial solution. As there are another algorithms we can use, this approach is not further discussed and is dismissed.

3 Requirements and Constraints

In this chapter, the scope and requirements of the final application are defined. Moreover, hard and soft constraints are introduced, and examples for the given domain – shift scheduling – are presented. Finally, the criteria for measuring the quality of a solution are established.

3.1 Requirements

The student research project assignment of our university limits the scope of the project. Therefore, specific requirements are set and outlined in this section.

The final application must deploy graph coloring to find a schedule satisfying multiple constraints. However, the goal is not to find an arbitrary shift assignment fulfilling all given constraints but rather an optimal solution for the given input.

Furthermore, the proposed solving method must be implemented in Python, C, C++, or C#. The application requires no Graphical User Interface (GUI) and should be limited to a Command-Line Interface (CLI) featuring an input file. The calculated solution (i.e., the final shift schedule) must also be output to a file. The file format for input and output should utilize a structured syntax, for example, JavaScript Object Notation (JSON) or Extensible Markup Language (XML).

3.2 Shift Scheduling Constraints

This section defines the constraints considered in the final shift scheduling application. As briefly introduced in section 1.1, we distinguish between hard and soft constraints. A solution is only valid if all hard constraints are satisfied. Soft constraints, on the other hand, are not strictly required. While fulfilling soft constraints improves the quality of a solution, they can be violated.

3.2.1 Hard Constraints

The shift scheduling application considers multiple hard constraints. To be applicable in the real world, as minimal requirement, the following two restrictions must be respected.

First, suppose the schedule features multiple shift positions, i.e., more than one staff member is working per shift. In that case, a staff member cannot be assigned to work multiple shift positions simultaneously. Second, a staff member can only work a single shift per day. It is important to note that the application does not define a shift further. To create a shift assignment over multiple days, details regarding a shift (e.g., working time and breaks) are irrelevant. Instead, the goal of the schedule is to assign all shifts to staff members for each day in a given time frame.

Besides these restrictions, we introduce another hard constraint that limits the succession of late and early shifts. More precisely, a staff member cannot be assigned the last and earliest shifts of two consecutive working days. Of course, this only applies if there are multiple shifts in a single working day.

3.2.2 Soft Constraints

The application considers one soft constraint influencing how shifts are assigned on weekends. Working on weekends is usually valued differently than shifts during the regular working week. The constraint introduces a special relationship between the weekend shift and the shifts on consecutive weekends to factor this in. The idea is that if a staff member is working on a weekend, the same staff member should *ideally* not have to work the following weekends. The technical details for this constraint are that any weekend shift affects the three consecutive weekends. Independent of whether a staff member worked on Saturday or Sunday, the constraint is not restricted to individual days but rather to the following weekends as a whole. Additionally, the three following weekends are given different priorities: For the first consecutive weekend, the constraint (i.e., edge in the graph) has a weight of 0.75, the second consecutive weekend is assigned a weight of 0.5, and the third is 0.25. Thus, weekends closer to the worked shift have a higher priority to be fulfilled.

3.3 Criteria for Measuring Quality and Fairness of a Solution

After setting the constraints, criteria for calculating the quality of a solution should be defined. Basically, a fitness function can feature rewards – which increase the score – and punishments – which decrease it. Related papers use punishments like [8, Eq. (1) on p. 234] and [9, Eq. (2) on p. 108]. There, for each violated constraint, they add up the associated weight times the amount of violations. In this project, two scores are sufficient: the coloring score and the unfairness score.

3.3.1 The Coloring Score

The coloring score focuses solely on the quality of the resulting fuzzy coloring of a given graph. To create a suitable function, one aspect must be considered first: An alternative to the often used linear approach, the equation containing the punishments can be squared. Consider the decision of violating either one constraint A with the weight 0.3 or two constraints B and C with a weight of 0.2 each. With the linear measurement, A would be disregarded because the weights are 0.3 versus $0.2 + 0.2 = 0.4$ for B and C. By using a quadratic punishment, B and C would be violated, since the weight of A is $0.3^2 = 0.09$ and B, C is $0.2^2 + 0.2^2 = 0.08$. In a nutshell, by varying the order of the equation, we can favor keeping one big or two small constraints.

In this project, it is adequate to use the fuzzy chromatic number $(1 - DTI)$ (eq. (2.6)) – from now on, also called *coloring score* – as the measurement for the quality of a solution. It is linear. Here, we use the interpretation presented in section 2.4. In its essence, it is similar to [8, Eq. (1) on p. 234] and [9, Eq. (2) on p. 108] since the sum of the weights of incompatible edges is equivalent to the amount of violated constraints times their associated weight. The fuzzy chromatic number is normalized since DTI is the weighted percentage of how many constraints were ignored compared to all constraints. This is an advantage over the standard ways [8, Eq. (1) on p. 234] and [9, Eq. (2) on p. 108] as it makes the comparison of different problems and their solutions possible without further ado.

3.3.2 The Unfairness Score

The coloring score is great for measuring how well a fuzzy k-coloring fits a given fuzzy graph but it does not state, how equitably colors are distributed. A valid solution could consist of one color, i.e., staff member, being assigned to the majority of nodes, i.e., shifts, whereas the other staff members solely have to work on those occasions where constraints are blocking. We wanted to measure the fair distribution of colors, as well. Hence, the Unfairness Score (US) was introduced. It should punish an unequal distribution of shifts – in other words, every worker should have the same amount of assigned shifts – and should be normalized such that it is comparable for different problems. The Coefficient of Variation (CV) accomplishes this:

$$US := CV := \frac{\sigma}{\mu} \cdot 100\% \tag{3.1}$$

CV is the standard deviation but divided by the mean. As a convention, we will give US as a percentage yet do not explicitly note the percentage symbol. Moving the decimal point makes it easier to read as a human because it is likely to be smaller than 1. The best value is 0.

4 Implementation and Solution

This chapter covers the implementation of the final solution to the problem at hand. The shift scheduling application consists of three major parts: Generating a graph from the given input data and constraints, coloring the graph, and interpreting the colored graph as a shift schedule. Furthermore, the application is validated by demo cases and a performance measurement.

Consistent with the technical requirements outlined in section 3.1, the application is implemented in Python 3.8. We use Python Poetry as a dependency management and packaging tool. Furthermore, the version control system git is deployed. The project and code documentation is generated via Sphinx based on docstrings.

The implementation uses the abstraction of two layers. The shift scheduling layer covers the graph generation, the wrapper for the graph coloring, as well as the graph interpretation. It offers a CLI. The graph coloring algorithms are out-sourced in the underlying layer: the fuzzy graph coloring.

4.1 Graph Generation

Before the graph can be generated, an input file format, that contains all required information, must be defined. We specify the input file to follow the JSON standard. This provides easy input validation utilizing a JSON schema. Moreover, a JSON file features less overhead compared to XML and the syntax closely follows the Python dictionary object. The application parameters required for generating a graph (i.e., a shift schedule) are given in the example configuration file in listing 4.1. The file contains the number of shifts per day, the number of staff members needed per shift, and total staff members. Furthermore, the working days of the week have to be specified, and a time frame is defined via a start and end date. Finally, a boolean value controls whether the soft constraint `balanced_weekends`, introduced in section 3.2.2, is enabled. The application is designed to automatically generate this input file from the example integrated in the JSON schema if it does not exist.

```
{
  "shifts": 2,
  "staff_per_shift": 3,
  "total_staff": 15,
  "work_days": [
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Su"
  ],
  "start_date": "2022-02-01",
  "end_date": "2022-03-01",
  "soft_constraints": {
    "balanced_weekends": true
  }
}
```

Listing 4.1: Example configuration file

After parsing and validating the input file with a JSON schema, a graph representing the shift schedule can be generated. As explained in section 2.4, each shift position is interpreted as a vertex. The edges between vertices are constraints. A graph coloring is the assignment of staff members (colors) to each shift position (vertices). For the creation and management of the graph, the python package `NetworkX` is used. In order to identify which shift is represented by a vertex, we assign labels following the notation: `day.shift.position`. The days are numbered according to the given time frame and starting date. If the shift schedule `start_day` is set to `2022-02-18`, the first day is the 18th of February, the second day is the 19th, and so forth. Days are numbered consecutively independent of whether they are working days or days where no shift should be scheduled. Shift and position in each shift are labeled with increasing integers and are zero-based.

4.1.1 Modeling Hard Constraints

As defined in section 3.2, the graph generation is based on multiple constraints. To model the hard constraints that a staff member cannot work two shifts simultaneously and can only work one shift per day, the `NetworkX` function `complete_graph(number_of_nodes)` can be called. It returns a graph where all pairs of distinct nodes have an edge connecting them. With these constraints, each day can be modeled as a fully connected graph with nodes equal to the number of shifts times the number of staff members working per shift (shift positions). The graph is fully connected because once a shift position is assigned to a staff member, no other shift position on that day can be assigned to the same staff member. In terms of graph coloring, once a vertex has a specific color, no other vertex in the subgraph can have the same color. Finally, the complete subgraphs from each day can be composed into a single overall graph. This can be done with the function `compose_all([graph, subgraph])`. In the implementation, the final graph is gradually built day by day. Alternatively, all days could be generated first and then composed into one graph in a single operation.

The hard constraint preventing the assignment of the same staff member to the last shift on one day and the earliest shift on the consecutive working day can be implemented similarly. Once again, a fully connected subgraph is required. This time, the graph consists of all vertices representing the last shift positions on the previous day and all the first shift positions on the current day. To complete the implementation of this constraint, it is crucial to check if the previous day is a working day according to the input configuration. Only on this condition, the subgraph should be created and composed with the overall graph. Otherwise, no additional edges are needed. For example, if Saturday and Sunday are not working days, a staff member can work the last shift on Friday and the first shift on Monday, even though they are consecutive working days.

4.1.2 Modeling the Soft Constraint

If the soft constraint is enabled in the configuration and Saturday or Sunday are considered working days, there is one more step to the subgraph generation for each day. To integrate the soft constraint, a weighted edge from each node representing a shift on the weekend to the consecutive weekends has to be added. This is done in three steps. First, all affected days of future weekends are calculated and collected in a list. Here, it is important to check if the specific day is still within the configured timeframe for the shift schedule. In the second step, a graph node is created for each day in the future weekends list. Finally, edges connecting each shift on the current day to all shifts on future weekends is added using the `NetworkX` function `add_weighted_edges_from([(u, v, weight)])`. It allows adding multiple edges in a single operation based on a list of tuples. In this case, u is the node of a shift on the current day and v a node representing a shift on a future weekend. The third element of the tuple is the weight of the edge. As defined in section 3.2.2, the weights are 0.75, 0.5, and 0.25 for the first, second, and third consecutive weekend.

4.2 Graph Coloring

To perform graph coloring independent of the practical use case of the final application, we introduce a python package with two major functions: `genetic_fuzzy_color` and `alpha_fuzzy_color`. Both will be able to do the entire graph coloring process, including pre- and post-processing like relabling the graph if required. In this section, the HLSGA, proposed by Keshavarz [18] and discussed in 2.8.1, is implemented (`genetic_fuzzy_color`). Eventually, we point out why it is unsuited for solving the shift scheduling problem. As a consequence, Fuzzy Greedy k-Coloring – using α-cuts – is realized to successfully solve the stated problem (`alpha_fuzzy_color`).

4.2.1 Genetic Algorithm Graph Coloring

There are multiple options to realize a GA in Python. While an implementation without dependencies is possible, utilizing an existing library provides a robust baseline, accelerates the process, and focuses development on the key components specific to the problem in this paper. Two prevalent frameworks are PyGAD [44] and Distributed Evolutionary Algorithms in Python (DEAP) [45]. This implementation utilizes PyGAD because it features a clear lifecycle and provides extensive documentation [46]. As a result, parts of the framework can be easily adapted, extended, and customized to fit the problem at hand. Additionally, PyGAD fulfills all technical requirements being actively developed in Python 3.7.3 [46].

In the following paragraphs, the general structure of the graph coloring process is outlined, and key components of the `fuzzy_color` function are highlighted. The different steps in the computation process are built on top of the PyGAD lifecycle which is depicted in figure 4.1. The lifecycle closely follows the general iterative sequence of GAs presented in section 2.7.

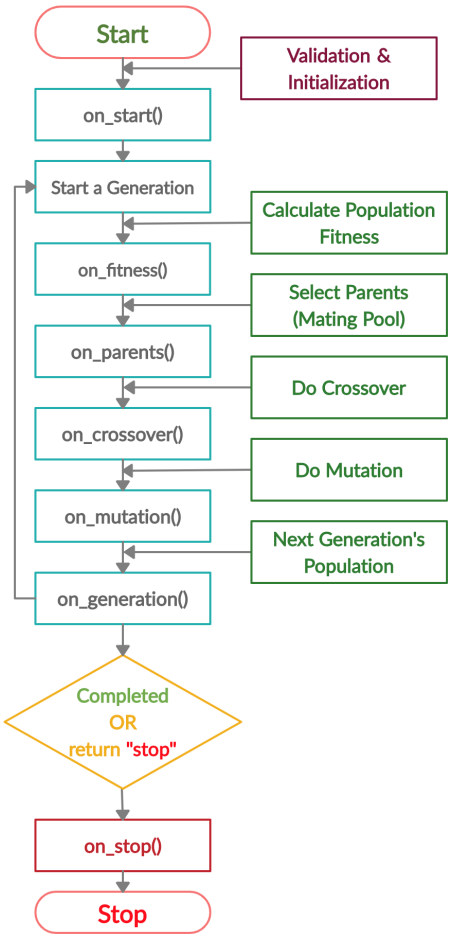

Figure 4.1: PyGAD lifecycle [44, p. 5]

Furthermore, the figure also includes the major PyGAD callback functions [44, p. 4]. For example, `on_generation()` is called at the end of each generation and can be used to implement additional steps (e.g., the local search procedure) apart from the genetic operators. For all genetic operators, PyGAD provides default implementations of common GA methods, for example, single-point crossover. The IEX and CTM operators, used by the HLSGA, are not implemented by default. However, PyGAD allows defining custom genetic operators by passing the function with the correct signature.

Besides valid genetic operators, PyGAD also requires the configuration of hyperparameters. An overview of the implementation's main parameters and their respective default values are given in table 4.1. The table only shows a selection of all available parameters in the PyGAD framework. The input graph and k-coloring determine other parameters. For example, the number of genes in each chromosome equals the number of vertices of the input graph. Similarly, the available integer values for each gene depend on k (i.e., the available colors). Furthermore, the HLSGA introduces the need for an additional hyperparameter: `local_search_probability`. It defines the probability of executing a local search for each chromosome. By default, the local search probability is set to 20%. The default values are based on the configuration used in [18].

Table 4.1: Implementation main hyperparameters and default values

Parameter	**Comment**	**Default value**
`num_generations`	Number of generations	15
`solutions_per_pop`	Number of chromosomes within the population	100
`keep_parents`	Number of parents to keep in the next population	$\lfloor \frac{solutions\_per\_pop}{2} \rfloor$
`crossover_probability`	Probability of applying the crossover operation to a parent	0.8
`mutation_probability`	Probability of mutating a chromosome	0.3

Input

In order to execute the fuzzy graph coloring function, the minimum required input parameter is a weighted graph (i.e., fuzzy graph). As the implementation uses NetworkX, a weighted graph refers to a NetworkX graph object where each edge carries the key-value attribute `weight` with a valid value. Furthermore, the function features optional parameters. Therefore, it is possible to compute a specific k-coloring. This limits the calculation to the given k instead of computing a fuzzy coloring for all possible k (i.e., 1 to n). Moreover, the function also offers a verbose flag, which outputs progress information such as total elapsed time if set to true. Finally, the GA hyperparameters and default values previously presented in section 4.2 can be overwritten by passing a new value as a keyword argument.

Validation and Initialization

Following the lifecycle, before PyGAD starts the iterative steps of the GA, input parameters must be validated, and the initial population must be generated. While PyGAD handles the validation of default GA parameters, the input graph must be verified independently. Validating the input graph has two steps. First, the weight attribute of all edges is checked. If not all edges carry a weight attribute, the graph is transformed to a weighted graph by assigning each edge missing the weight attribute the key-value pair `weight = 1`. An exception is raised if an edge holds an invalid weight. In the second step, the vertices of the input graph are relabeled using consecutive integers to provide a consistent naming pattern for the algorithm. This is required because the consecutive integer values represent the order of the vertices in a chromosome. To not modify the input graph by the user, a copy is relabeled. Additionally, the original vertex labels are saved to a variable. Then, the output color assignment can get the same naming as the input graph.

```
1  def _initial_population_generator(k: int, sol_per_pop: int, num_genes:
       int):
2      """
3      Generates the initial Genetic Algorithm population.
4      :param k: Number of available colors (k-coloring)
5      :param sol_per_pop: Number of solutions/chromosomes per population
6      :param num_genes: Number of genes in the solution/chromosome
7      :return: Initial population as nested numpy array
8      """
9      initial_population = np.empty((0, num_genes), int)
10     for _ in range(sol_per_pop):
11         chromosome = np.zeros(num_genes, int)
12         for color, gene_idx in enumerate(random.sample(range(num_genes
               ), k)):
13             chromosome[gene_idx] = color + 1
14         zero_mask = (chromosome == 0)
15         chromosome[zero_mask] = default_rng().choice(range(1, k + 1),
               zero_mask.sum())
16         initial_population = np.append(initial_population, [chromosome
               ], axis=0)
17     return initial_population
```

Listing 4.2: Initial population generation function

After validating and, if needed, pre-processing the input, the initial population must be generated. The generator function is given in listing 4.2. All chromosomes of the starting population are randomly generated with the population size, the number of available colors, and the number of genes per chromosome depending on the GA hyperparameters. To ensure that each chromosome contains all k colors at least once, a chromosome is initialized as an array filled with zeros (line 11). Then, in lines 12 and 13, each available color is assigned a random position in the empty chromosome. Finally, in lines 14 and 15, each remaining zero is replaced with a random color. This process is repeated until the population size is reached to produce the initial population.

It is important to note that generating the initial population allows potential duplicate chromosomes and initially equivalent colorings. We disregard the issue of duplicate chromosomes in the starting population as it is increasingly unlikely for large colorings and has little impact for simple cases. The initially equivalent colorings, for example, `[1 2 2 1]` and `[2 1 1 2]`, are wanted as they provide a different baseline for the crossover and mutation operations.

Calculation of the Population's Fitness

The next necessary function is the fitness function to measure the quality of a feasible solution [44, p. 4]. PyGAD specifies it to have two parameters: `solution` and `solution_idx`. The latter is the index of the given solution in the population at hand [46, sec. 'Preparing the `fitness_func` Parameter'].

In the case of using a GA for graph coloring, we need another parameter: The graph that should be colored. To set an additional parameter for a function with a fixed header, the factory design pattern can be leveraged.

The factory in code listing 4.3 is able to dynamically generate `_fitness_function` instances that are loosely coupled to a given graph. This is a kind of dependency injection. The factory's return type is a function: the fitness function.

```
1  def _fitness_function_factory(graph: nx.Graph):
2      """
3      Factory to generate fitness-function. Loosely couples graph-
           instance.
4      :param graph: graph-instance (nx.Graph)
5      :return: fitness_function
6      """
7
8      def _fitness_function(solution: tuple, solution_idx: Any):
9          """
10         Fitness function to measure the quality of a given chromosome,
               i.e., solution.
11         :param solution: Chromosome: tuple with length equal to number
               of vertices. Each item, i.e., gen is a color
12         :param solution_idx: Required by PyGAD interface but is not
               used
13         :return: fitness: 1 - (Degree of Total Incompatibility (DTI))
14         """
15         return _get_coloring_score(graph, solution)
16
17     return _fitness_function
```

Listing 4.3: The fitness function factory

```python
def _get_coloring_score(graph: nx.Graph, coloring) -> float:
    """
    Calculates the score for a given graph and coloring.
    Coloring can either be a tuple of colors or a dict (node: color assignment).
    :param graph: NetworkX Graph
    :param coloring: Node coloring
    :return: Coloring score (1 - degree of total incompatibility)
    """
    total_incompatibility = 0
    for (i, j) in graph.edges():
        y_ij = _y_ij(i, j, coloring) if isinstance(coloring, dict) else _y_ij(i - 1, j - 1, coloring)  # eq. (2.10a)
        total_incompatibility += graph[i][j]["weight"] * y_ij
    score = 1 - (total_incompatibility / graph.size(weight="weight"))  # 1 - DTI
    return score
```

Listing 4.4: The function calculating the fitness

The fitness function (listing 4.4) is using TI – equation (2.10a) – but computes another step to obtain the Degree of Total Incompatibility (DTI). Remember, $DTI(\varphi^k) \in [0, 1]$ and a higher value implies more violated edges. Since the fitness function – as the name suggests – is maximized, we use $1 - DTI$ which itself is in $[0, 1]$ and if maximized, will result in better solution. It is connected to the quality of a solution for the time scheduling problem as described in section 3.3.

Genetic Operators

As described previously, with the interpretation of feasible solutions as chromosomes, operations analogous to real-world genetics are performable.

As a **parent selection** method, we use the tournament method introduced in section 2.8.1. The size of the group that is randomly drawn to get the fittest parent is set to `K_tournament = 10`, which is 10%. Now, `keep_parents = int(solutions_per_pop / 2)` are selected to be used in the next generation. Since the population size is 100, 50 parents are kept. The other slots of the next generation are filled by the offspring. There, another 50 parents are selected. It is defined by `num_parents_mating`. The built-in method of PyGAD is used.

The offspring is created by applying crossover and mutation as well as the local search which is discussed in the following subsection. As a **crossover**-method, the Incompatibility Elimination Crossover (IEX) (see 2.8.1) is implemented. Considering IEX's dependency on the input graph – as incompatible edges are checked – the factory pattern is used. This pattern is elaborated in the previous subsection. The **mutation** utilized is the Color Transposition Mutation (CTM) (see 2.8.1). For the implementation, no factory pattern was used because the swapping of colors can be calculated without knowledge about the underlying graph. The code follows the exact instructions as defined. `genetic_fuzzy_color` has parameters to manipulate the probabilities of occurring crossover and occurring mutation: `crossover_probability` and `mutation_probability` as described in table 4.1.

Local Search

In section 2.8.1, a local search around feasible solutions is brought in. The local search implemented in the project builds on top of listing 2.1. Note that in the code, the fitness function $(1 - DTI)$ is used instead of TI. It is important to point out that a direct implementation of 2.1: 'Procedure: Local Search' will result in a bug: One after another gene is considered and varied for a better version. The edge case that a gene represents the only occurrence of one color could lead to the deletion of a particular color if changed. Hence, the requested k-coloring will turn into a $(k-1)$-coloring. This is caught in the code.

Output

The complete optimal coloring set is computed if just the graph-parameter is provided. This includes all possible minimal k-colorings for the given graph. The function then outputs a nested dictionary which holds the color assignment in the form `vertex_label: color` and the respective coloring score of the given coloring for each k. If a specific k is set via the optional parameters, a tuple with the vertex color assignment and score for the minimal k-coloring is returned. Enabling the verbose flag prints additional information to the console but does not change the function return values.

HLSGA's Flaws

The Genetic Algorithm (GA) as used cannot correctly deal with hard constraints. Therefore, it can violate constraints with the weight of 1, too. That poses a problem since hard constraints must be fulfilled as they represent legal and moral duties, e.g., no consecutive shifts as well as natural limitations. For instance, for a given person, it is not possible to work in two different positions at the same time.

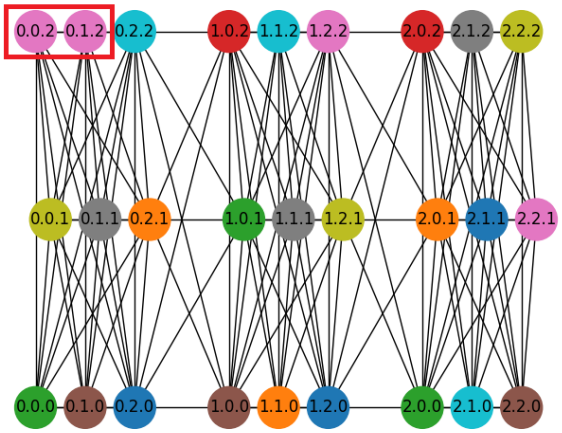

Figure 4.2: GA violating a hard constraint (marked in red)

Figure 4.2 shows a simple, colored graph with 27 vertices and 126 edges which are solely hard constraints. k was set to 9. The edge between `0.0.2` and `0.1.2` is violated with the following color assignment created by the algorithm.

```
1: ['0.2.0', '1.2.0', '2.1.1'],     2: ['0.2.1', '1.1.0', '2.0.1'],
3: ['0.0.0', '1.0.1', '2.0.0'],     4: ['1.0.2', '2.0.2'],
5: ['0.1.0', '1.0.0', '2.2.0'],
6: ['0.0.2', '0.1.2', '1.2.2', '2.2.1'],
7: ['0.1.1', '1.1.1', '2.1.2'],     8: ['0.0.1', '1.2.1', '2.2.2'],
9: ['0.2.2', '1.1.2', '2.1.0']
```

Listing 4.5: Coloring for fig. 4.2 where the key is the color and its values are assigned nodes

Remedies could be to define the search space such that it only consists of valid, feasible solutions or – since HLSGA is a stochastic algorithm – roll several times until a valid solution is found. Both could be complex to implement. In addition, with a turned-on local search, the coloring takes 28.5 seconds with a standard deviation of 8.7s to color the example given in fig. 4.2 ($n = 10$). Measured times in seconds: $[39.2, 21.4, 19.3, 23.4, 37.1, 27.4, 38.8, 39.8, 18.0, 20.6]$ on a laptop with an Intel(R) Core(TM) i7-8650U CPU. For the approach presented in the next part, the measured time is effectively 0 seconds. Considering hard constraints are essential for solving the problem, this implementation cannot be used. Hence, the need for a time complexity comparison planned in section 1.3 is void.

4.2.2 Fuzzy Greedy k-Coloring

After recognizing, that the HLSGA does not match the requirements, another algorithm was needed. Various literature (e.g., [13], [22]) propose using an ordinary graph coloring after performing an α-cut. The same technique was presented before in section 2.5.1. This approach is extended because in many use cases, the minimal k is wanted, but here, k is defined, as it is the number of members in a team.

The previously presented package `libcolgraph` is tagged with `manylinux` [35], meaning the wheel, i.e., the package is only available for a group of linux-distributions like the Debian family [47], [48]. Though it is possible to compile the code for Microsoft Windows, we decided to not take the risk of creating a package that is not platform-independent out of the box.

Greedy k-Coloring

`NetworkX` has a function that is able to color crisp graphs. There, a minimal k is calculated. In addition, the greedy color function of `NetworkX` assigns the first available color. Therefore, since colors are integer numbers, smaller colors are used more often than bigger ones. That makes shift scheduling less fair as shifts are not assigned equally among all staff members. Moreover, `NetworkX` does not take a k as input parameter but instead computes a coloring using the minimal number of colors. This is unsuited for shift scheduling as the available colors k are given by the number of staff members. To solve these issues, we implement the `greedy_k_color()` function which is discussed in this section.

```
1  def greedy_k_color(graph: nx.Graph, k: int, fair: bool = False) ->
       dict:
2      if k > graph.number_of_nodes():
3          raise InvalidKColoringError(f"Graph has no {k}-coloring as it
               only has {graph.number_of_nodes()} vertices")
4      coloring = {}
5      available_colors = {c: 0 for c in range(k)}
6      nodes = sorted(graph, key=graph.degree, reverse=True)
7      if fair:
8          for u in nodes:
9              # Set to keep track of colors of neighbours
10             neighbour_colors = {coloring[v] for v in graph[u] if v in
                   coloring}
11             for color in dict(sorted(available_colors.items(), key=
                   lambda item: item[1])):
12                 if color not in neighbour_colors:
13                     available_colors[color] = available_colors[color]
                           + 1
14                     break
15             else:
16                 raise NoSolutionException("No more colors")
17             coloring[u] = color
```

Listing 4.6: Function to perform a greedy k-coloring pt. 1

The algorithm introduced in code listing 4.6 is based on `NetworkX`'s implementation [49], where the algorithm iterates over all nodes and assigns the first color that is not violating any edge. `NetworkX` uses the strategy 'largest_first', in which the list of nodes is in decreasing order by degree. The procedure given in listing 4.6 utilizes the same order, as defined in line 6. A notable difference to the `NetworkX` implementation of the greedy color algorithm is the way colors are chosen. In the original, colors are given by `itertools.count()`. This means there is no upper boundary, and the algorithm can use as many colors as required. That is a problem if a fixed k is given. To guarantee that only colors in $[0, k)$ are used, line 5 limits the number of colors. Furthermore, the dictionary `available_colors` fulfills another task: It tracks the frequency of how often a specific color has been assigned. Each time a color is assigned, its key value increases (line 13). With this information, the algorithm can choose the first color that does not violate any edge using a color list sorted by least frequency (line 11). As a result, colors are assigned more evenly, compared to the `NetworkX` implementation. We can ensure that a coloring uses at least k colors because of line 2. If k is larger than the number of nodes in the graph, the function raises an exception. If k is less or equal to this number, the first k nodes are colored by all available colors consecutively as the color is chosen by least frequency. Finally, if the algorithm loops over all available colors without finding a valid coloring, a `NoSolutionException` is raised. This

does not mean that no solution exists but that the algorithm could not find one given the implemented node order and color assignment method.

```
1     else:
2         coloring = nx.greedy_color(graph)
3         if max(coloring.values()) + 1 > k:
4             raise NoSolutionException(f"Minimal solution needs more
                  colors than k={k} < {max(coloring.values()) + 1}")
5
6         for c in range(max(coloring.values()) + 1, k):
7             color_dist = Counter(coloring.values())
8             most_used_color = max(color_dist, key=color_dist.get)
9             if color_dist[most_used_color] <= 1:
10                raise NoSolutionException("Not possible to replace
                      colors")
11            nodes_with_most_used_color = [k for k, v in coloring.items
                  () if v == most_used_color]
12            replace_color_nodes = default_rng().choice(
13                nodes_with_most_used_color,
14                size=math.floor(color_dist[most_used_color] / 2),
15                replace=False
16            )
17            for node in replace_color_nodes:
18                coloring[node] = c
19
20    return coloring
```

Listing 4.7: Function to perform a greedy k-coloring pt. 2

There are cases where the execution of the fair greedy k-color algorithm, given in listing 4.6, does not result in a solution but executing the `NetworkX` coloring function produces a valid coloring with an equal or smaller k. To provide a solution in these cases, we implement a second algorithm in `greedy_k_color()` given in listing 4.7. It starts by computing a coloring using the `NetworkX` greedy coloring function. If the coloring uses fewer colors than the given k, the most used colors are replaced using new colors. This produces a coloring using exactly k colors where the color frequency is balanced to a certain degree.

If the `NetworkX` coloring uses more colors than available a `NoSolutionException` is raised in line 4. To extend a given `NetworkX` coloring, the algorithm iterates over all available new colors (line 6). The number of available new colors is given by the difference between the number of colors used in the `NetworkX` solution and the fixed k. In each iteration, the most used color is calculated (lines 7 and 8). Then, in line 9, a list of all nodes assigned the most used color is computed. Half of these nodes are assigned a new color to achieve a more even color distribution. Which nodes' colors are replaced is chosen at random in lines 12 to 16.

Which of the two presented methods the `greedy_k_color()` function uses is controlled by the `fair` parameter. The default value for is `fair = False`.

The effect of the parameter can also be observed when visualizing the resulting graph coloring. Figure 4.3 shows the colored shift schedule graph for three days with two shifts per day, three staff members per shift, and nine staff members in total. The color and node assignments are also given in listing 4.8. For this output, the fair parameter was set to `False`. As a result, the produced staff assignment has the shift distribution `[2, 2, 1, 1, 2, 3, 3, 3, 1]` and an unfairness score of 40.82. While some staff members have to work three shifts, others are only assigned one.

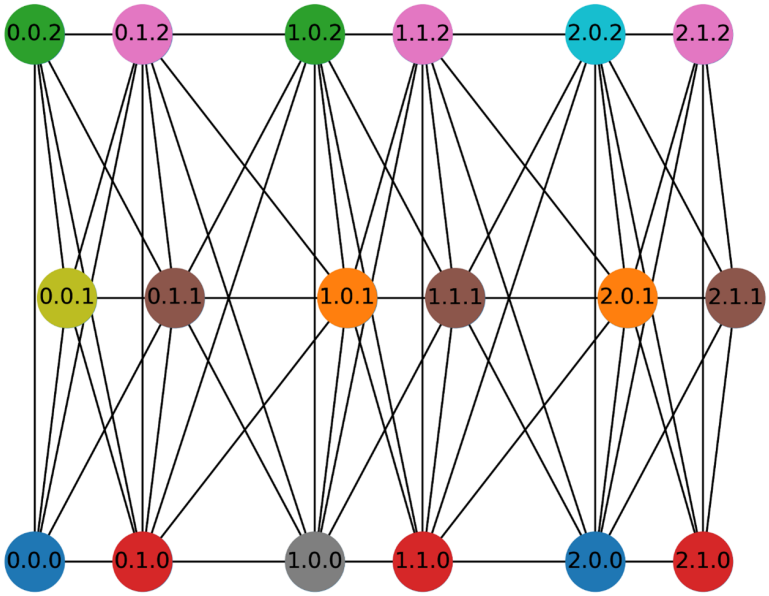

Figure 4.3: Greedy k-coloring with `fair = False`

```
1: ['0.0.2', '1.0.2'],
2: ['0.0.0', '2.0.0'],
3: ['0.0.1'],
4: ['1.0.0'],
5: ['1.0.1', '2.0.1'],
6: ['0.1.2', '1.1.2', '2.1.2'],
7: ['0.1.0', '1.1.0', '2.1.0'],
8: ['0.1.1', '1.1.1', '2.1.1'],
9: ['2.0.2']
```

Listing 4.8: Coloring for fig. 4.3 where the key is the color and its values are assigned nodes

Executing the same configuration with the fair parameter set to `True` produces a different result. The colored graph is depicted in figure 4.4, alternatively, the color node assignments are given listing 4.9. Evaluating the staff assignments shows the shift distribution `[2, 2, 2, 2, 2, 2, 2, 2]` and therefore an unfairness score of 0. Compared to figure 4.3, the greedy k-coloring with the parameter `fair = True` creates a timetable where every staff member is assigned an equal number of shifts. Furthermore, a pattern in the color assignments emerges. As expected, colors are chosen by their frequency of occurrence. Due to this, traversing the graph day by day (in the figure left to right) demonstrates that every available color is used once before an already assigned color is chosen a second time. This is different from running greedy k-coloring with `fair = False` as colors are picked in a fixed order. As seen in figure 4.3 and listing 4.8 the colors 3 (lime), 4 (grey), and 9 (cyan) are assigned only once while others are assigned three times.

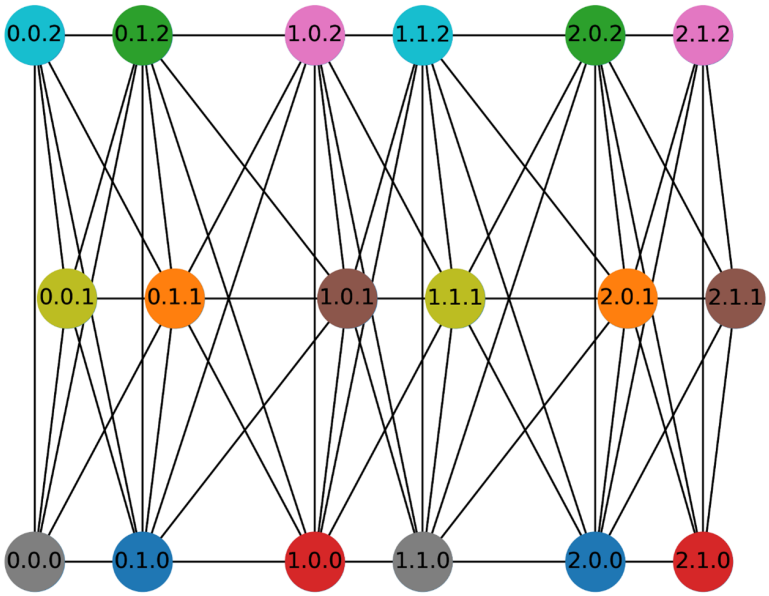

Figure 4.4: Greedy k-coloring with `fair = True`

```
1: ['0.0.2', '1.1.2'],
2: ['0.0.0', '1.1.0'],
3: ['0.0.1', '1.1.1'],
4: ['0.1.2', '2.0.2'],
5: ['0.1.0', '2.0.0'],
6: ['0.1.1', '2.0.1'],
7: ['1.0.0', '2.1.0'],
8: ['1.0.2', '2.1.2'],
9: ['1.0.1', '2.1.1'],
```

Listing 4.9: Coloring for fig. 4.4 where the key is the color and its values are assigned nodes

In the final shift scheduling application the function is called in a way to first try the fair coloring while keeping a fallback (see listing 4.10). As explained in section 1.1, it is preferred that the application returns a less-than-ideal solution over no valid output.

```
def fuzzy_color(graph: nx.Graph, k: int, verbose: bool = False):
    try:
        return fgc.alpha_fuzzy_color(graph, k, fair=True, return_alpha
            =True)
    except fgc.NoSolutionException:
        if verbose:
            print("Failed to use a fair order of colors, i.e., team
                members. Try to use best-fit.")
        return fgc.alpha_fuzzy_color(graph, k, return_alpha=True)
```

Listing 4.10: Alpha-fuzzy-color-function with fallback

The usage of the package `fgc` is elaborated later on. See that `alpha_fuzzy_color` is a function to perform fuzzy graph coloring presented in the next paragraph.

Fuzzy k-Coloring Using α-Cuts Combined With Binary Search

With the newly introduced function, creating a k-coloring is possible. Now, we want fuzzy graphs to be colorable, too. As explained in section 2.5.1, the simplest way to achieve this is to make an α-cut of a fuzzy graph to obtain a crisp graph, which can then be colored. Remember, a bigger α implies less edges and therefore less constrains. The challenge is to find an α that is

1. big enough such that a solution does exist and
2. as small as possible to take lots of constraints into account.

To meet the requirements, the following algorithm is implemented. First, it is checked if the solution is trivial, i.e., k is equal to the number of nodes and accordingly returned. At all times, there are two variables: `latest_alpha` to store the current best alpha and the related coloring `coloring`.

The first alpha to be checked is $\alpha = 1$. That implies only hard constraints, i.e., constraints with the weight of 1 are kept. It is equivalent to a check whether a solution exists. If no solution exists, an exception will be raised. After that, the next edge case is considered: $\alpha = 0$. Note that mathematically speaking, the limit $\lim_{\alpha \to 0}$ would be correct since an α-cut of $\alpha = 0$ is not defined. To simplify that, in the application the function is defined for $\alpha = 0$ and will act like $\lim_{\alpha \to 0}$. That means the underlying crisp graph is used. If there is a solution for this graph, it is a solution without violations. If there is no solution, we know that $\alpha \in (0, 1]$, where the minimal α is an optimum. It is possible to limit the search space: Use only values that really add or remove edges, i.e., take the (ordered) set of the edges' weights. Consider the previously introduced graph in figure 2.4. The search space is $S = \{.2, .3, .4, .5, .6, .7, .8, .9, 1.0\}$. To get the smallest α, a linear search could be performed. It has a time complexity of $\mathcal{O}(n)$.

A better approach is leveraging a binary search with time complexity of $\mathcal{O}(\log n)$. Three pointers on S are defined: a lower and an upper boundary idx_{low}, idx_{upper} to specify the subset that must be still searched, as well as a pointer to the current α to be checked idx_{α}. The boundary values are included. In each iteration, the third pointer is set to the middle value. It will be the alpha for the next check. There, it will be tested if a solution exists.

- If an exception is raised, the considered α was too big.
- If a solution exists, the optimal α is found or below the considered α.

The boundary values are adjusted accordingly and the next iteration starts. The break condition is met if the search space has eliminated all values except for one. This is the minimal α.

Consider a 3-coloring of the graph given by figure 2.4. An iteration is unambiguously defined by a triple with the three pointers $(idx_{low}, idx_{\alpha}, idx_{upper})$. The binary search follows the following constructive solution:

1. (0, 3, 7): Initial triple. Solution is found $\Longrightarrow$ New upper boundary: 3.
2. (0, 1, 3): $\alpha = 0.3$ is too small $\Longrightarrow$ New lower boundary must be at least the next bigger value of α. Therefore, the next idx_{low} is set to the value one higher.
3. (2, 2, 3): There is a solution for $idx_{\alpha} = 2$.
4. (2, 2, 2): Final triple and break condition: Lowest $\alpha = 0.4$.

The resulting coloring is `{1: 2, 2: 1, 3: 0, 4: 0, 5: 0, 6: 1, 7: 2, 8: 1}` and is depicted by fig. 4.5. It has a coloring score of approximately 0.929.

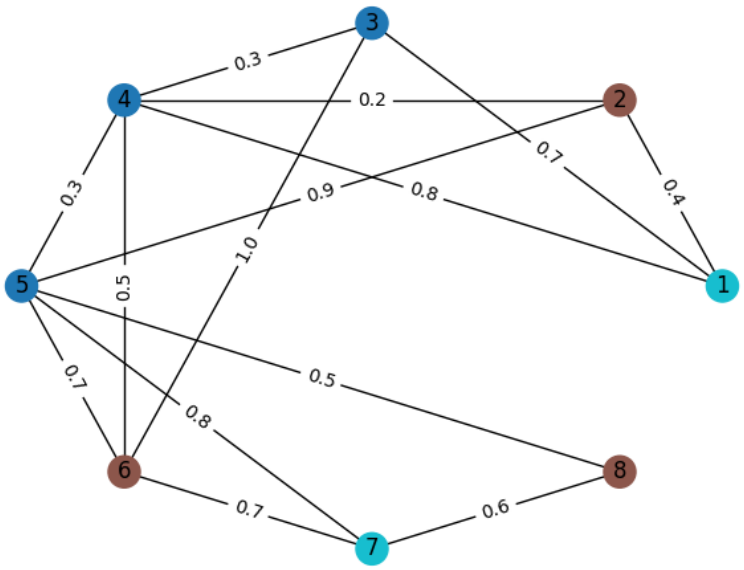

Figure 4.5: 3-coloring of graph presented in fig. 2.4

Verifiably, a fuzzy k-coloring can be now calculated that maximizes the coloring score.

4.3 Graph Interpretation

In the last step of the application, a colored graph is interpreted to produce a shift schedule. Similar to the graph generation process discussed in section 4.1, it is important to define an output format. The final shift schedule is output as a Comma-Separated Values (CSV) file together with the quality measurements established in section 3.3. The file contains the fields `Day, Date, Shift, Position, Staff`. An example is given in listing 4.11. A CSV file provides a human-readable format while also conserving the ability to further process the output in another application. Additionally, a CSV file can be easily presented as table to the user. Utilizing JSON syntax is unsuited as the output contains no advanced data structures such as tuples or arrays and the entire data follows a consistent, repeating format.

```
Day,Date,Shift,Position,Staff
Fr,2022-02-18,1,1,2
Fr,2022-02-18,1,2,3
Fr,2022-02-18,2,1,4
Fr,2022-02-18,2,2,5
Mo,2022-02-21,1,1,0
Mo,2022-02-21,1,2,1
Mo,2022-02-21,2,1,2
Mo,2022-02-21,2,2,3
Tu,2022-02-22,1,1,0
Tu,2022-02-22,1,2,1
Tu,2022-02-22,2,1,4
Tu,2022-02-22,2,2,5
```

Listing 4.11: Example shift schedule output file

Generating a shift schedule based on a colored graph can be done in a single loop. The code is shown in code listing 4.12. For every vertex in the graph, the label contains all important information about the shift. As introduced in section 4.1, each label follows the notation: `day.shift.position`. The staff member assignment is then given by the coloring. The `coloring` dictionary object in line 6 is returned by the `fuzzy_color` function discussed in section 4.2. The date given in the output file is calculated based on the numbered day and the starting date. In line 8, all information is formatted and written to the output file as a final step.

```
1 for node in sorted(graph):
2     d, s, p = node.split(".")
3     d = int(d)
4     s = int(s) + 1
5     p = int(p) + 1
6     assigned_staff = coloring.get(node)
7     date = input_data["start_day"] + datetime.timedelta(days=d)
8     csv_writer.writerow([_get_weekday(date), date.strftime("%Y-%m-%d")
          , s, p, assigned_staff])
```

Listing 4.12: Shift schedule generation from graph

4.4 Validation and Performance Measurement

To validate the final application, we consider various demo cases and a performance measurement. This section covers the validation of both layers – the shift scheduling application and the fuzzy graph coloring package. The execution time is measured using the native Python module `timeit` [50].

4.4.1 Shift Scheduling Demo Cases

The application offers multiple parameters that can be set via the configuration file. We propose various demo cases to demonstrate the functionality and manually validate the program's correct execution with different inputs. Each demo case consists of an input file.

The first and simplest demo case is given in the `introduction.json` file. It covers a timeframe of three days, two shifts per day, three workers per shift, and nine staff members in total. The resulting colored graph is depicted in figure 4.6, and the corresponding shift schedule output is portrayed in figure 4.7.

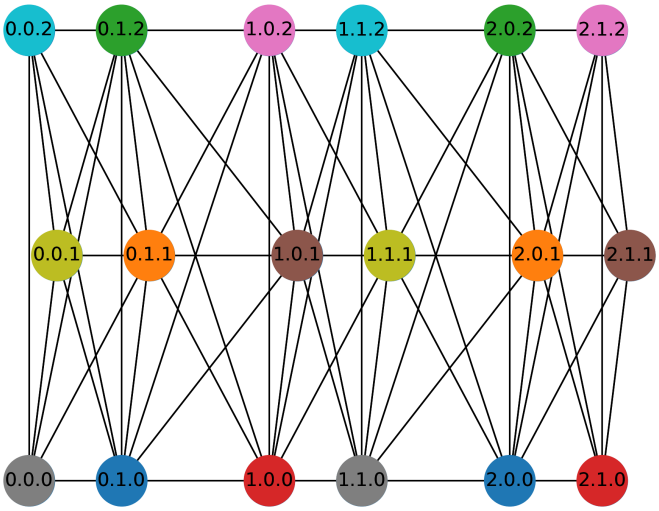

Figure 4.6: Introduction demo case graph coloring

	Day	Date	Shift	Position	Staff
1	Mo	2022-02-07	1	1	6
2	Mo	2022-02-07	1	2	7
3	Mo	2022-02-07	1	3	8
4	Mo	2022-02-07	2	1	0
5	Mo	2022-02-07	2	2	1
6	Mo	2022-02-07	2	3	2
7	Tu	2022-02-08	1	1	3
8	Tu	2022-02-08	1	2	4
9	Tu	2022-02-08	1	3	5
10	Tu	2022-02-08	2	1	6
11	Tu	2022-02-08	2	2	7
12	Tu	2022-02-08	2	3	8
13	We	2022-02-09	1	1	0
14	We	2022-02-09	1	2	1
15	We	2022-02-09	1	3	2
16	We	2022-02-09	2	1	3
17	We	2022-02-09	2	2	4
18	We	2022-02-09	2	3	5

Figure 4.7: Introduction demo case shift schedule

A more complex demo case is considered in `days_off.json`. This configuration creates a shift schedule over 14 days with two shifts, four workers per shift, and a total of 16 staff members. As opposed to the previous example, in `days_off.json`, the soft constraint `balanced_weekends` is enabled. Another difference, as the file name implies, lies in the configuration of the working days. Here only Monday, Wednesday, Thursday, Friday, and Saturday are set as working days. This can be noticed when considering the visualization of the resulting coloring, as presented in figure 4.8. The gaps in the graph are caused by the days off where no shift is scheduled. Nodes are connected across the second and third gaps due to the soft constraint connection shifts on consecutive weekends, as explained in section 3.2.2.

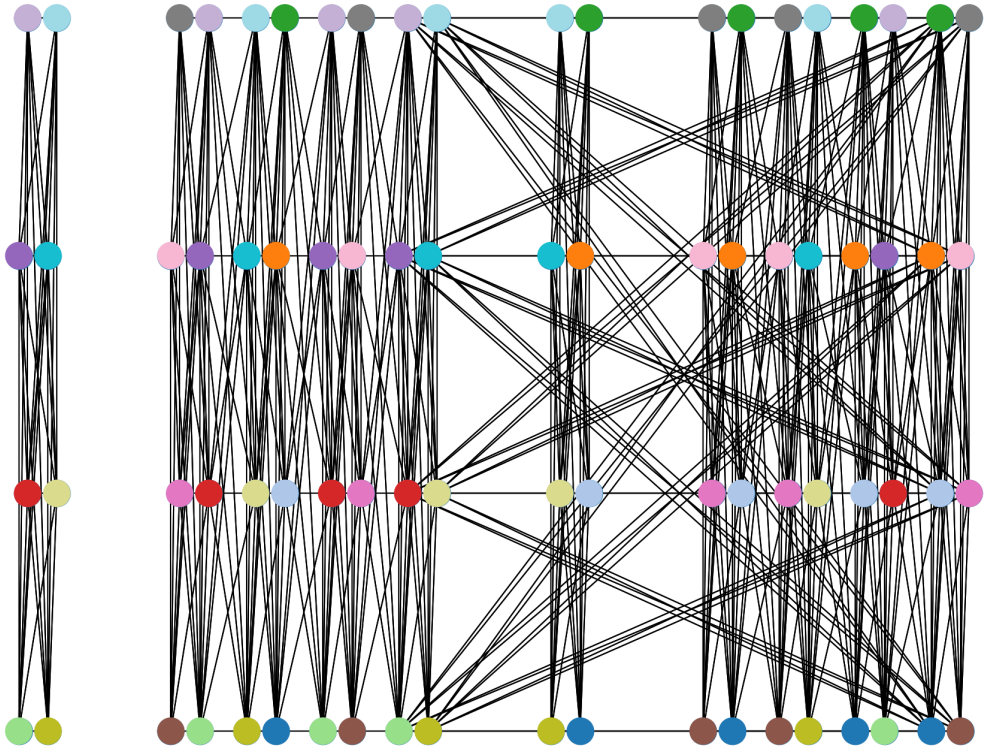

Figure 4.8: Days off demo case graph coloring

More demo cases include examples of the `balanced_weekends` soft constraint and larger time frames. To manually test the execution with larger graphs, we consider two cases with roughly 1000 vertices, respectively, with and without the soft constraint. While the application handles the configuration well, the resulting visualization is unclear due to the number of nodes and edges in the limited space for the given image resolution.

The full configuration files for all demo cases are listed in appendix B.

4.4.2 Performance Measurement

In this part, the performance of the application is measured. For one given input, the complete procedure is executed in five groups containing of five runs. Of these five runs only the minimal execution time is considered. Thus, there are five minimums in total: the smallest execution time per group. Of these groups, the mean and standard deviation are calculated. The background of using the minimums is that we consider the execution time for a given input to be stable. Fluctuations are introduced by the operating system and environment, e.g., when the scheduler prioritizes another program. Hence, the most isolated and purest result are the minimums where the pace only depends on the algorithm.

Table 4.2: Size of graph with respect to period

Considered weeks	**Number of nodes**	**Number of edges (BW off)**	**Number of edges (BW on)**
4	168	663	1527
8	336	1335	3927
12	504	2007	6327
16	672	2679	8727
20	840	3351	11127
24	1008	4023	13527
28	1176	4695	15927
32	1344	5367	18327
36	1512	6039	20727
40	1680	6711	23127
44	1848	7383	25527
48	2016	8055	27927
52	2184	8727	30327

The most interesting metric is how long the program needs to create a solution with respect to the considered period. That is directly connected to the number of nodes and edges of the graph. First, the following parameters are set as a constant: There are two shifts per day with 3 staff member per shift. 15 workers are available. The start date is January, 1st 2022. The end date is variable as stated in table 4.2 along with the size of the graph, i.e., the number of nodes and edges. There are more edges with `balanced_weekends` turned on which is abbreviated as 'BW'.

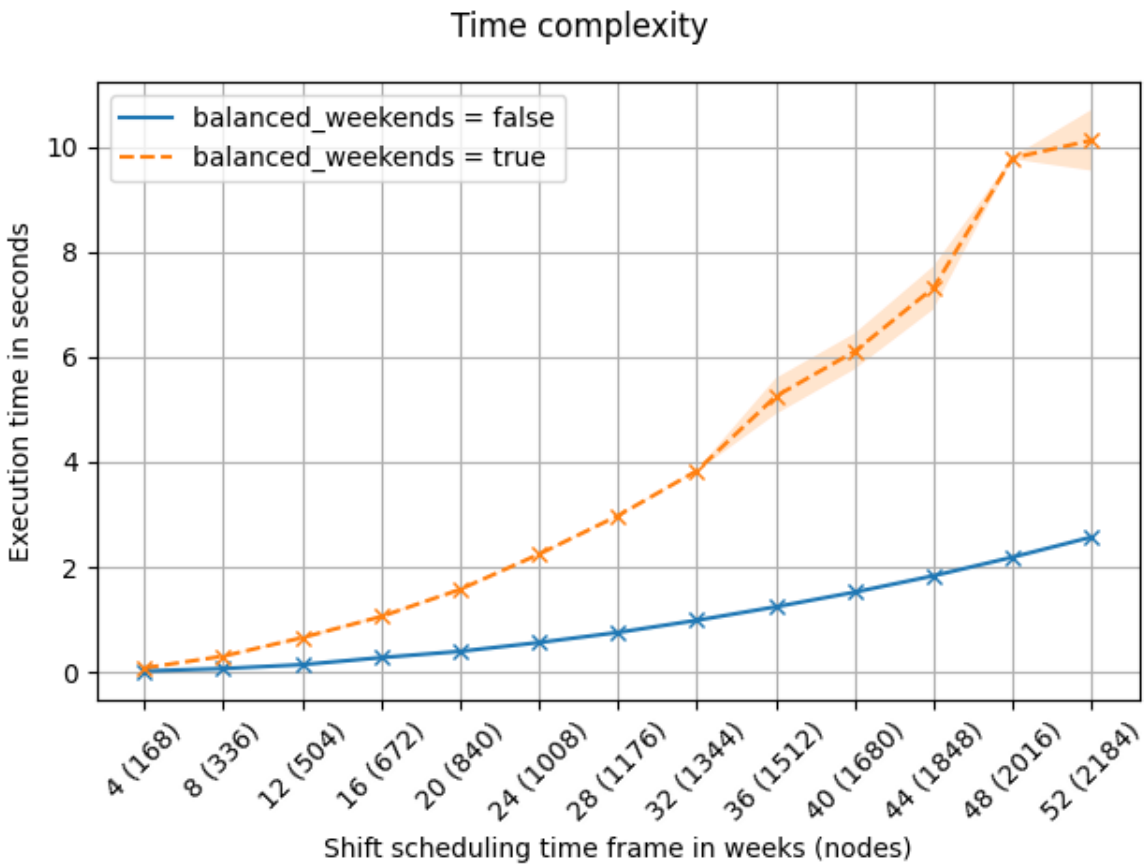

Figure 4.9: Time complexity with respect to weeks / nodes.

The performance measurement was run on a laptop with an Intel(R) Core(TM) i7-8650U CPU. Figure 4.9 shows the results. The top line is with activated `balanced_weekends`. The shaded area around the line is the standard deviation whereas the line itself is the mean. The curve shows that the complexity of the application is in $\mathcal{O}(n^2)$ where n is the number of nodes. The more edges there are due to the soft constraint, the steeper the ascend is.

One case we want to point out is a graph with 1008 nodes and 4023 respectively 13527 edges as it is the first test with over 1000 nodes. There, the mean execution time was 0.57s respectively 2.25s. The standard deviation is 0.01s for both. Keep in mind that these numbers refer to the minimums as described before.

In a nutshell, the algorithm scales well and has a good performance for the use case. Waiting a couple of seconds until the time schedule is calculated is reasonable.

4.5 Python Package and GitHub Repositories

"Don't reinvent the wheel" is a famous quote and is also implied by the Don't Repeat Yourself (DRY) software design principle. It means, one should reuse pre-existing code. We provide the code of this student research project as open source such that others can built upon.

4.5.1 The Fuzzy Graph Coloring Python Package

Python uses so-called wheels to distribute packages. A wheel is a well-defined archive containing code. Though, the name's origin is Python Package Index (PyPI) formerly being called *CheeseShop* due to its inspiration from Monty Python's Cheese Shop Sketch [51]. Nowadays, using a python wheel is often metaphorically referred to not reinvent the wheel and hence to not re-invent code that already exists.

As mentioned earlier, the project uses the dependency management tool Poetry. It enables the creation of a Python wheel and the deployment on PyPI. The fuzzy graph coloring layer is published. See pypi.org/project/fuzzy-graph-coloring/. Conclusively, it is installable via `pip install fuzzy-graph-coloring` in any Python environment. The package name's abbreviation is `fgc`. The source code is available on GitHub (github.com/ferdinand-dhbw/fuzzy-graph-coloring).

4.5.2 The Shift Scheduling Project

The second layer which is the shift scheduling project itself is provided on GitHub, too. Visit github.com/jrheiner/shift-scheduling. Just as `fgc`, the shift scheduling project is licensed under GNU General Public License v3.0.

5 Conclusion

This chapter summarizes the paper and reflects on the final result concerning goals formulated at the beginning of the project. Moreover, an outlook and possible future work is presented.

5.1 Reflection

In this project paper, we implement and validate an application to find optimal shift schedules. As required, the program uses graph coloring to solve a given WCSP. It is implemented in Python 3 and provides a CLI. Further, the project source code is open-source and available on GitHub. Our fuzzy graph coloring solving algorithm is published as Python Package on PyPI.

5.1.1 Final Results

The final application considers three hard, one soft constraint, and additional parameters set in a JSON input file. The calculated shift schedule is written to a CSV output file. As demanded in the problem statement, we provide two metrics to measure the quality of a solution: The coloring and unfairness score. Furthermore, the implementation produces a less-than-ideal solution over no valid output. If no solution is possible, the program terminates. To solve a given problem, we use two approaches, trying to calculate a fair solution first. If this fails, a fallback is executed to find a solution based on a different strategy (node order). A binary search procedure is implemented to perform the α-cut operation - even on larger inputs efficiently.

5.1.2 Limitations

The final application satisfies all requirements. Nevertheless, the project also has limitations. As already discussed in subsection 4.2.1, the first proposed technique for coloring fuzzy graphs cannot correctly deal with the concept of hard constraints. Therefore, it is unsuited for the shift scheduling use case. Due to this, the plan for comparing the solving methods is void.

Furthermore, the functionality of the program is limited to some extent. The application does not consider constraints based on individual staff members, for example, skill sets and personal holiday wishes. Additionally, all shifts are treated equally, and there is no possibility of modeling special shifts. Moreover, while the proposed metrics to measure a solution's quality include the unfairness score, the interpretation is basic. As defined in section 3.3, the fairness of a solution is measured solely based on the distribution of shifts, i.e., the quantity per staff member.

Finally, in the current state, soft constraints with the same weight are treated as one unit during the computation of a solution. Consequently, there is no fine-grained consideration for edges carrying the same weight. For instance, if the generated graph has five edges with a weight of 0.7, the α-cut either includes all five edges or removes all five edges. This is a limitation concerning the solution quality in some cases.

5.1.3 Assessment

This student research project is a success. The application follows the software engineering principle separation of concerns with the clear separation between the fuzzy graph coloring layer and the use case specific shift scheduling part. Hence, the shift scheduling program and the fuzzy graph coloring package can be re-used independently.

While the project has some limitations, as expressed in the two previous subsections, the final application fulfills all assignment requirements and solves the problem. Furthermore, it is important to point out that the program scales well with the final implementation of the solving algorithm. As shown in the performance benchmark in section 4.4.2, creating a simple shift schedule for 52 weeks takes 2.25 seconds. For that case, the underlying graph consists of 2184 nodes and 8727 edges.

5.2 Outlook

For future work, the rework to a user-friendly application is suitable. The goal was not to create a good Graphical User Interface (GUI) but to be used by potentially non-technical users – people who actually plan shifts – an easy and comprehensive interface is relevant.

The application does only cover a fraction of real-life requirements. To depict the real world, more constraints can be added such as a reward if the alternation of early and late shifts is minimized, or working for five days in a row is favored. Additionally, constraints on an individual basis are imaginable. This is not trivially model-able via edges in the graph but could be considered in an advanced coloring algorithm. Then, staff members could have distinct skill sets or preferences.

Specifying fairness could be the subject of another research project. The Definition in this project is simple but one can argue that different or more factors should be valued. This could be connected with the same individual-based approach introduced in the previous paragraph such that a solution is fairer concerning personal needs.

This work and the results are not reviewed by someone who really plans shift schedules. It is not examined whether this project has the potential to have a positive impact on the workflow or the quality of timetables. In the future, one could get the feedback of an expert in this area and introduce user acceptance tests.

The final application deploys fuzzy graph coloring based on a binary search of α-cuts. If a threshold of say $\alpha = 0.7$ is chosen, all edges with a weight of 0.7 are removed. It follows that possibly a whole, immense group of edges are discarded, and hence, a solution exists that takes a fraction of the removed edges into account. The idea is to bring in a more fine granular gradation of the removal of edges.

As a practical part of this research project, the python package `fuzzy-graph-coloring` was published. With that, other Weighted Constraint Satisfaction Problems (WCSPs) are tack-able, too. It is yet to be examined, to which other use cases it is applicable and where it will be employed.

Bibliography

[1] A. Schrijver, *Combinatorial optimization: polyhedra and efficiency*, ser. Algorithms and combinatorics 24. Berlin ; New York: Springer, 2003, ISBN: 9783540443896.

[2] M. M. Flood, "The traveling-salesman problem," *Operations Research*, vol. 4, no. 1, pp. 61–75, 1956, ISSN: 0030-364X. DOI: 10.1287/opre.4.1.61.

[3] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN: 978-3-642-07311-3. DOI: 10.1007/978-3-540-24777-7.

[4] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*, ser. Wiley-Interscience series in discrete mathematics and optimization. Chichester ; New York: J. Wiley & Sons, 1990, ISBN: 9780471924203.

[5] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *IEEE Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985, ISSN: 1058-6180. DOI: 10.1109/MAHC.1985.10011.

[6] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557–581, 1999, ISSN: 03772217. DOI: 10.1016/S0377-2217(98)00364-6.

[7] D. Marx, "Graph colouring problems and their applications in scheduling," *Periodica Polytechnica Electrical Engineering*, vol. 48, pp. 11–16, 2004.

[8] D. Santos, P. Fernandes, H. L. Cardoso, and E. Oliveira, "A weighted constraint optimization approach to the nurse scheduling problem," in *2015 IEEE 18th International Conference on Computational Science and Engineering*, IEEE, 2015, pp. 233–239, ISBN: 978-1-4673-8297-7. DOI: 10.1109/CSE.2015.46.

[9] A. Chaudhuri and K. De, "Fuzzy genetic heuristic for university course timetable problem," *International Journal of Advances in Soft Computing and Its Applications*, vol. 2, Mar. 2010. [Online]. Available: https://www.researchgate.net/publication/50590446_Fuzzy_Genetic_Heuristic_for_University_Course_Timetable_Problem (visited on 02/13/2022).

[10] L. W. Beineke and R. J. Wilson, *Topics in Chromatic Graph Theory*. Cambridge: Cambridge University Press, 2015, ISBN: 9781139519793. DOI: 10.1017/CBO9781139519793.

[11] R. Lewis, *A Guide to Graph Colouring*. Cham: Springer International Publishing, 2016, ISBN: 978-3-319-25728-0. DOI: 10.1007/978-3-319-25730-3.

[12] R. Mahapatra, S. Samanta, and M. Pal, "Applications of edge colouring of fuzzy graphs," *Informatica*, pp. 313–330, 2020, ISSN: 0868-4952. DOI: 10.15388/20-INFOR403.

[13] Nivethitha V., "Graph theory- fuzzy graph coloring and its application," vol. 6, no. 2, pp. 629–641, 2019. DOI: 10.5281/zenodo.3895708.

[14] T. R. Jensen and B. Toft, *Graph Coloring Problems*. Hoboken, NJ, USA: John Wiley & Sons, Inc, 1994, ISBN: 9781118032497. DOI: 10.1002/9781118032497.

[15] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, ser. A Series of books in the mathematical sciences. San Francisco: W. H. Freeman, 1979, ISBN: 9780716710448.

[16] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90241-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001999586590241X.

[17] J. N. Mordeson and P. S. Nair, *Fuzzy graphs and fuzzy hypergraphs*. Physica, 2012, vol. 46.

[18] E. Keshavarz, "Vertex-coloring of fuzzy graphs: A new approach," *Journal of Intelligent & Fuzzy Systems*, vol. 30, pp. 883–893, 2016, ISSN: 1875-8967. DOI: 10.3233/IFS-151810.

[19] L. Prasad and R. Sattanathan, "Fuzzy total coloring of fuzzy graphs," *International Journal of Information Technology and Knowledge Management*, vol. Volume 2, p 37–39, 2009.

[20] S. Muñoz, M. T. Ortuño, J. Ramírez, and J. Yáñez, "Coloring fuzzy graphs," *Omega*, vol. 33, no. 3, pp. 211–221, 2005, ISSN: 0305-0483. DOI: 10.1016/j.omega.2004.04.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305048304000660.

[21] S. Samanta, T. Pramanik, and M. Pal, "Fuzzy colouring of fuzzy graphs," *Afrika Matematika*, 2015. DOI: 10.1007/s13370-015-0317-8.

[22] M. Ananthanarayanan and S. Lavanya, "Fuzzy graph coloring using alpha cuts," *International Journal of Engineering*, vol. 4, no. 10, p. 8269, 2014.

[23] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI Magazine*, vol. 13, no. 1, p. 32, Mar. 1992. DOI: 10.1609/aimag.v13i1.976. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/976.

[24] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," *Computational Intelligence*, vol. 9, no. 3, pp. 268–299, 1993, ISSN: 0824-7935. DOI: 10.1111/j.1467-8640.1993.tb00310.x.

[25] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977, ISSN: 00043702. DOI: 10.1016/0004-3702(77)90007-8.

[26] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & Operations Research*, vol. 13, no. 5, pp. 533–549, 1986, ISSN: 03050548. DOI: 10.1016/0305-0548(86)90048-1.

[27] M. Gendreau and J.-Y. Potvin, "Tabu search," in *Handbook of Metaheuristics*, ser. International Series in Operations Research & Management Science, M. Gendreau and J.-Y. Potvin, Eds., vol. 272, Cham: Springer International Publishing, 2019, pp. 37–55, ISBN: 978-3-319-91085-7. DOI: 10.1007/978-3-319-91086-4_2.

[28] A. Hertz and D. de Werra, "Using tabu search techniques for graph coloring," *Computing*, vol. 39, no. 4, pp. 345–351, 1987, ISSN: 0010-485X. DOI: 10.1007/BF02239976.

[29] G. Kendall and N. M. Hussin, "A tabu search hyper-heuristic approach to the examination timetabling problem at the mara university of technology," in *Practice and Theory of Automated Timetabling V*, ser. Lecture Notes in Computer Science, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, E. Burke, and M. Trick, Eds., vol. 3616, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 270–293, ISBN: 978-3-540-30705-1. DOI: 10.1007/11593577_16.

[30] G. M. White, B. S. Xie, and S. Zonjic, "Using tabu search with longer-term memory and relaxation to create examination timetables," *European Journal of Operational Research*, vol. 153, no. 1, pp. 80–91, 2004, ISSN: 03772217. DOI: 10.1016/S0377-2217(03)00100-0.

[31] J.-F. Cordeau and G. Laporte, "Tabu search heuristics for the vehicle routing problem," in *Metaheuristic Optimization via Memory and Evolution*, ser. Operations Research/Computer Science Interfaces Series, R. Sharda, S. Voß, C. Rego, and B. Alidaee, Eds., vol. 30, Boston: Kluwer Academic Publishers, 2005, pp. 145–163, ISBN: 1-4020-8134-0. DOI: 10.1007/0-387-23667-8_6.

[32] F. Hayes-Roth, "Review of 'Adaptation in Natural and Artificial Systems by John H. Holland', The U. of Michigan Press, 1975," *SIGART Bull*, no. 53, p. 15, 1975, ISSN: 0163-5719. DOI: 10.1145/1216504.1216510.

[33] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, no. 2, pp. 65–85, 1994, ISSN: 1573-1375. DOI: 10.1007/BF00175354. [Online]. Available: http://cobweb.cs.uga.edu/~potter/CompIntell/ga_tutorial.pdf (visited on 01/29/2022).

[34] Zbigniew Kokosinski, Krzysztof Kwarciany, and Marcin Kolodziej, "Efficient graph coloring with parallel genetic algorithms," *Comput. Artif. Intell.*, vol. 24, pp. 123–147, 2005. [Online]. Available: https://riad.pk.edu.pl/~zk/pubs/CAI2005.pdf (visited on 01/29/2022).

[35] *Wheelodex.org: Libcolgraph*. [Online]. Available: https://www.wheelodex.org/projects/libcolgraph/ (visited on 01/22/2022).

[36] *Wheelodex.org: Networkx*. [Online]. Available: https://www.wheelodex.org/projects/networkx/ (visited on 01/22/2022).

[37] Aric Hagberg, Dan Schult, and Pieter Swart, "Networkx reference: Release 2.6.2," 2021. [Online]. Available: https://networkx.org/documentation/stable/_downloads/networkx_reference.pdf (visited on 01/22/2022).

[38] K. M. Adrian Kosowski, *Classical Coloring of Graphs*. 2004, ISBN: 0-8218-3458-4. [Online]. Available: https://fileadmin.cs.lth.se/cs/Personal/Andrzej_Lingas/k-m.pdf (visited on 01/22/2022).

[39] H. A. Kierstead, A. V. Kostochka, M. Mydlarz, and E. Szemerédi, "A fast algorithm for equitable coloring," *Combinatorica*, vol. 30, no. 2, pp. 217–224, 2010, ISSN: 0209-9683. DOI: 10.1007/s00493-010-2483-5.

[40] Aric Hagberg, Dan Schult, and Pieter Swart, *License of networkx: 3-clause bsd license*, 2022. [Online]. Available: https://github.com/networkx/networkx/blob/main/LICENSE.txt (visited on 01/22/2022).

[41] A. Dey and A. Pal, "Vertex coloring of a fuzzy graph," 2012. [Online]. Available: https://www.researchgate.net/publication/280316647_VERTEX_COLORING_OF_A_FUZZY_GRAPH (visited on 01/15/2022).

[42] *Programming Languages - C++: ISO/IEC 14882, 1998-09-01*, ser. International standard. ISO/IEC, 1998.

[43] *Programming Languages - C++: ISO/IEC 14882, 2011-09-01*, ser. International standard. ISO/IEC, 2011.

[44] A. F. Gad, *Pygad: An intuitive genetic algorithm python library*, 2021. arXiv: 2106.06158 [cs.NE].

[45] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012.

[46] A. F. Gad, *Pygad - python genetic algorithm! pygad 2.16.3 documentation*, 2020. [Online]. Available: https://pygad.readthedocs.io/ (visited on 02/13/2022).

[47] Coloring Graphs lab, University of Richmond, *Pypi: Libcolgraph 0.0.7*. [Online]. Available: https://pypi.org/project/libcolgraph/ (visited on 02/25/2022).

[48] Python Software Foundation, *Pep 513: A platform tag for portable linux built distributions*. [Online]. Available: `https://www.python.org/dev/peps/pep-0513/#bundled-wheels-on-linux` (visited on 02/25/2022).

[49] NetworkX developers, *Networkx: Greedy_coloring.py*. [Online]. Available: `https://github.com/networkx/networkx/blob/main/networkx/algorithms/coloring/greedy_coloring.py` (visited on 02/25/2022).

[50] Python Software Foundation, *Timeit — measure execution time of small code snippets — python 3.8.12 documentation*, 2022. [Online]. Available: `https://docs.python.org/3.8/library/timeit.html` (visited on 02/13/2022).

[51] *Python wiki: Cheeseshop*. [Online]. Available: `https://wiki.python.org/moin/CheeseShop` (visited on 03/20/2022).

Appendix

A. Code Difference fuzzyvertexcoloring.cpp

B. Demo Cases Configuration Files

A. Code Difference fuzzyvertexcoloring.cpp

```
   1    1://File name: fuzzyvertexcoloring.cpp
   2    2:#include <conio.h>
   3    3:#include <stdio.h>
   4    4:#include <stdlib.h>
-  5     :#include <iostream.h>
+       5:#include <iostream>     // https://stackoverflow.com/
     questions/13103108/why-cant-g-find-iostream-h
   6    6:
   7    7:void input();
   8    8:int MaxDegreeVertex ();
   9    9:void testfuzzygraph();
  10   10:int chromaticnumber();
  11   11:void UpdateNN(int ColorNumber);
  12   12:int FindSuitableY(int ColorNumber, int& VerticesInCommon);
  13   13:int MaxDegreeVertex();
  14   14:int MaxDegreeInNN();
  15   15:
  16   16:const int MAX = 100;   // of which number of vertices is
       smaller or equal to 100
  17   17:int n;                                  // necessary variables
        n is the number of vertices of the graph
  18   18:float a[MAX][MAX];             // a[n][n] is the adjacency
       matrix of the graph
  19   19:int color[100];                        // array color[n]
       stores the colors of the vertices and color[i] = 0 if we 've not
        colored it yet
  20   20:int degree[100];               // array degree[n] stores the
       degrees of the vertices
  21   21:int NN[100];                   // array NN[] stores all the
       vertices that is not adjacent to current vertex
  22   22:int NNCount;                   // NNCount is the number of
       that set
```

```
23  23:int unprocessed;                 // unprocessed is the number
    of vertices with which we 've not worked
24  24:float d[100];                    //store the membership value
    of vertices
25  25:
26  26:// testing the condition of the fuzzy graph
27  27:void testfuzzygraph() {
28  28:         int m=0;
29  29:         float c;
30  30:         for(int i=0; i < n; i++) {
31  31:                 for(int j=0; j < n; j++) {
32  32:                         if(d[i]>=d[j]) c=d[j];
33  33:                         else c=d[i];
34  34:                         if(a[i][j]>c) m++;
35  35:                 }
36  36:         }
37  37:         if(m==0) printf("Graph is fuzzy graph\n");
38  38:         else printf("The graph is not fuzzy graph\n");
39  39:}
40  40:
- 41    :testadject(){
+   41:void testadject(){
42  42:         float c,m;
43  43:         for(int i=0;i<=n-1;i++){
44  44:                 for(int j=0;j<=n-1;j++){
45  45:                         if(d[i]>d[j]) c=d[j];
46  46:                         else c=d[i];
47  47:                         m=c/2.0;
48  48:                         if(m>a[i][j]){
49  49:                                 a[i][j]=0.0;
50  50:                                 a[j][i]=0.0;
51  51:                         }
52  52:                 }
53  53:         }
54  54:}
55  55:
56  56:// testing the graph is path fuzzy graph or not
57  57:int testpathgraph (){
58  58:         if(a[0][n-1]==0&&degree[0]==1&&degree[n-1]==1){
59  59:                 printf("The Graph is Fuzzypathgraph\n");
60  60:                 printf("Chromatic number is 2");
61  61:                 getch();
62  62:                 exit(0);
63  63:         } else
64  64:                 return 1;
65  65:}
66  66:
```

```
67  67:// coloring function of the graph
68  68:void Coloring() {
69  69:       int x,y, ColorNumber = 0, VerticesInCommon = 0;
70  70:       while (unprocessed>0) // while there is an uncolored
    vertex
71  71:       {
72  72:               x = MaxDegreeVertex(); // find the one
    vertices with maximum degree
73  73:               ColorNumber ++;
74  74:               color[x] = ColorNumber; // give it a new color
75  75:               unprocessed --;
76  76:               UpdateNN(ColorNumber); // find the set of non-
    neighbors of x
77  77:               while (NNCount>0) { // find y, the vertex has
    the maximum neighbors in common with x
78  78:                                                       //
    VerticesInCommon is this maximum number
79  79:                       y = FindSuitableY(ColorNumber,
    VerticesInCommon); // in case VerticesInCommon = 0 and y is
    determined that the vertex with max degree in NN
80  80:                       if (VerticesInCommon == 0)
81  81:                               y = MaxDegreeInNN();
82  82:                       color [y] = ColorNumber; // color y
    the same to x
83  83:                       unprocessed --;
84  84:                       UpdateNN(ColorNumber); // find the new
     set of non-neighbors of x
85  85:               }
86  86:       }
87  87:}
88  88:
89  89:// initializing function
90  90:void Init(){
91  91:       for (int i=0; i<n; i++){
92  92:               color[i] = 0; // be sure that at first, no
    vertex is colored
93  93:               degree[i] = 0; // count the degree of each
    vertex
94  94:               for (int j = 0; j < n; j++)
95  95:                       if (a[i][j]>0) // if i-th vertex is
    adjacent to another
96  96:                               degree[i] ++; // increase its
    degree by 1
97  97:       }
98  98:       NNCount = 0; // number of element in NN set
99  99:       unprocessed = n;
```

```
 100 100:} // this function finds the unprocessed vertex of which
     degree is maximum
 101 101:int MaxDegreeVertex(){
 102 102:        int max_i, max = -1;
 103 103:        for (int i=0; i < n; i++)
 104 104:                if (color[i] == 0)
 105 105:                        if (degree[i]>max){
 106 106:                                max = degree[i];
 107 107:                                max_i = i;
 108 108:                        }
 109 109:        return max_i;
 110 110:}
 111 111:
 112 112:// this function is for UpdateNN function it reset the value
     of scanned array
 113 113:void scannedInit(int scanned[MAX]){
 114 114:        for (int i=0; i < n; i++)
 115 115:                scanned[i] = 0;
 116 116:}
 117 117:
 118 118:// this function updates NN array
 119 119:void UpdateNN(int ColorNumber){
 120 120:        NNCount = 0; // firstly, add all the uncolored
     vertices into NN set
 121 121:        for (int i=0; i < n; i++) if (color[i] == 0){
 122 122:                NN[NNCount] = i;
 123 123:                NNCount ++; // when we add a vertex, increase
     the NNCount
 124 124:        }
-125     :        for (i=0; i < n; i++) // then remove all the vertices
     in NN that is adjacent to the vertices colored ColorNumber
+    125:        for (int i=0; i < n; i++) // then remove all the
     vertices in NN that is adjacent to the vertices colored
     ColorNumber
 126 126:        {
 127 127:                if (color[i] == ColorNumber) // find one
     vertex colored ColorNumber
 128 128:                for (int j=0; j < NNCount; j++)
 129 129:                        while(a[i][NN[j]]>0) // remove vertex
     that adjacent to the found vertex
 130 130:                        {
 131 131:                                NN[j] = NN[NNCount - 1];
 132 132:                                NNCount --; // decrease the
     NNCount
 133 133:                        }
 134 134:        }
 135 135:}
```

```
136 136:
137 137:// this function will find suitable y from NN
138 138:int FindSuitableY(int ColorNumber, int& VerticesInCommon){
139 139:       int temp,tmp_y,y;
140 140:       int scanned[MAX]; // array scanned stores uncolored
    vertices except the vertex is being processing
141 141:       VerticesInCommon = 0;
142 142:       for (int i=0; i < NNCount; i++) // check the i-th
    vertex in NN
143 143:       {
144 144:               tmp_y = NN[i]; // tmp_y is the vertex we are
    processing
145 145:               temp = 0; // temp is the neighbors in common
    of tmp_y and the vertices colored ColorNumber
146 146:               scannedInit(scanned);
147 147:               for (int x=0; x < n; x++)       //reset
    scanned array in order to check all
148 148:

    //the vertices if they are adjacent to i-th vertex in NN
149 149:                       if (color[x] == ColorNumber) // find
    one vertex colored ColorNumber
150 150:                               for (int k=0; k < n; k++)
151 151:                                       if (color[k] == 0 &&
    scanned[k] == 0)
152 152:                                               if (a[x][k] >0
     && a[tmp_y][k]>0) // if k is a neighbor in common of x and tmp_y
153 153:                                               {
154 154:                                                       temp
    ++;
155 155:
    scanned[k] = 1; // k is scanned
156 156:                                               }
157 157:                                               if (temp >
    VerticesInCommon){
158 158:
    VerticesInCommon = temp;
159 159:                                                       y =
    tmp_y;
160 160:                                               }
161 161:       }
162 162:       return y;
163 163:} // find the vertex in NN of which degree is maximum
164 164:
165 165:int MaxDegreeInNN(){
166 166:       int tmp_y; // the vertex has the current maximum
    degree
```

```
 167 167:        int temp, max = 0;
 168 168:        for (int i=0; i < NNCount; i++){
 169 169:                temp = 0;
 170 170:                for (int j=0; j < n; j++)
 171 171:                        if (color[j] == 0 && a[NN[i]][j]>0)
 172 172:                                temp ++;
 173 173:                        if (temp>max) // if the degree of
     vertex NN[i] is higher than tmp_y's one
 174 174:                        {
 175 175:                                max = temp; // assignment NN[i
     ] to tmp_y
 176 176:                                tmp_y = NN[i];
 177 177:                        }
 178 178:        }
 179 179:        if (max == 0) // so all the vertices have degree 0
 180 180:                return NN[0];
 181 181:        else // exist a maximum, return it
 182 182:                return tmp_y;
 183 183:}
 184 184:
 185 185:// print out the output
 186 186:void PrintOutput(){
 187 187:        for (int i=0; i < n; i++)
-188     :                printf("Vertex %d is colored %d " i+1,color[i]
     ); // element i-th of array color stores the color of (i+1)- th
     vertex
+    188:                printf("Vertex %d is colored %d ", i+1, color[
     i]); // element i-th of array color stores the color of (i+1)- th
     vertex
 189 189:}
 190 190:
 191 191:void input(){
 192 192:        float membership;
 193 193:        int maxedges, origin, destin;
 194 194:        printf("Enter the number of vertices");
 195 195:        scanf("%d",&n);
 196 196:        for(int i=0;i<=n-1;i++){
 197 197:                printf("Enter the membership value of vertex %
     d= ",i+1);
 198 198:                scanf("%f",&d[i]);
 199 199:        }
 200 200:        maxedges=n*(n-1)/2;
-201     :        for( i=0;i<maxedges;i++) {
+    201:        for(int i=0;i<maxedges;i++) {
 202 202:                printf("Enter the edges %d(0 to 0)",i);
 203 203:                scanf("%d%d",&origin,&destin);
 204 204:                if((origin==0)&&(destin==0))
```

```
 205 205:                        break;
 206 206:                if(origin>n||destin>n||origin<=0||destin<=0){
 207 207:                        printf("invalid edge");
 208 208:                        i--;
 209 209:                } else {
 210 210:                        printf("Enter the membership value of
     this edge");
 211 211:                        scanf("%f",&membership);
 212 212:                        a[origin-1][destin-1]=membership;
 213 213:                        a[destin-1][origin-1]=membership;
 214 214:                }
 215 215:        }
 216 216:}
 217 217:
 218 218://Find the chromatic number
 219 219:int chromaticnumber(){
 220 220:        int max = -1, max_i;
 221 221:        for (int i=0; i < n; i++)
 222 222:                if (color[i]>max){
 223 223:                        max = color[i];
 224 224:                        max_i = i;
 225 225:                }
 226 226:        printf("chromatic number=%d\n",max);
 227 227:        return max;
 228 228:}
 229 229:
 230 230:// main function
-231     :void main(){
-232     :        clrscr();
+    231:int main(){
+    232:        // clrscr();
 233 233:        input(); // read the input adjacency matrix from file
 234 234:        testfuzzygraph(); // testing the graph is fuzzy graph
     or not
 235 235:        testadject(); // testing the adjacency of the edge
 236 236:        Init(); // initialize the data : degree, color array
     ..
 237 237:        testpathgraph(); // test path graph or not
 238 238:        Coloring(); // working function
 239 239:        chromaticnumber(); // find the chromatic number
 240 240:        PrintOutput(); // print the result onto the console
     lines
 241 241:        getch();
+    242:        return 0;
 242 243:}
```

Listing 1: Code difference `fuzzyvertexcoloring.cpp`

B. Demo Cases Configuration Files

```
{
  "shifts": 2,
  "staff_per_shift": 3,
  "total_staff": 9,
  "work_days": [
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Su"
  ],
  "start_date": "2022-02-7",
  "end_date": "2022-02-9",
  "soft_constraints": {
    "balanced_weekends": false
  }
}
```

Listing 2: Demo case file `introduction.json`

```
{
  "shifts": 2,
  "staff_per_shift": 4,
  "total_staff": 16,
  "work_days": [
    "Mo",
    "We",
    "Th",
    "Fr",
    "Sa"
  ],
  "start_date": "2022-02-7",
  "end_date": "2022-02-20",
  "soft_constraints": {
    "balanced_weekends": true
  }
}
```

Listing 3: Demo case file `days_off.json`

```
{
  "shifts": 2,
  "staff_per_shift": 3,
  "total_staff": 18,
  "work_days": [
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Su"
  ],
  "start_date": "2022-02-7",
  "end_date": "2022-02-27",
  "soft_constraints": {
    "balanced_weekends": true
  }
}
```

Listing 4: Demo case file `sc-introduction-fair.json`

```
{
  "shifts": 2,
  "staff_per_shift": 3,
  "total_staff": 15,
  "work_days": [
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Su"
  ],
  "start_date": "2022-02-7",
  "end_date": "2022-08-17",
  "soft_constraints": {
    "balanced_weekends": false
  }
}
```

Listing 5: Demo case file `5k-edges-1k-nodes.json`

```
{
  "shifts": 2,
  "staff_per_shift": 3,
  "total_staff": 15,
  "work_days": [
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Su"
  ],
  "start_date": "2022-02-7",
  "end_date": "2022-08-17",
  "soft_constraints": {
    "balanced_weekends": true
  }
}
```

Listing 6: Demo case file `15k-edges-1k-nodes-sc.json`