Faculty of Mathematics and Computer Science
Institute of Computer Science
Chair of Advanced Computing

– Master's Thesis –

Solving Coloring Problems on Bipartite Graphs Using Linear Algebra

Submitted in partial fulfillment of the requirements for the degree

Master of Science (M.Sc.)

Computational and Data Science

1. Supervisor: Prof. Dr.-Ing Martin Bücker
2. Supervisor: M. Sc. Johannes Schoder

Submitted by:

Niklas Rausch

Lutherstraße 161

07743 Jena

niklas.rausch@uni-jena.de

Date of submission: 02.04.2024

Zusammenfassung

Zum Lösen von d-1 und d-2 Graphfärbungsproblemen werden C Programme, welche die GraphBLAS API nutzen, präsentiert. Die C Implementation *SuiteSparse:GraphBLAS* von Timothy A. Davis wird dazu genutzt. Die d-1 Färbung färbt beliebige ungerichtete Graphen und die d-2 Färbung färbt ungerichtete bipartite Graphen. Die GraphBLAS API nutzt Objekte und Operationen der linearen Algebra, um Graphalgorithmen darzustellen. Der Zusammenhang zwischen Graphen und Operationen der linearen Algebra wird erläutert und die theoretischen Grundlagen der GraphBLAS API werden besprochen. Die gefärbten Graphen werden mithilfe des Python Pakets NetworkX visualisiert. Anhand dieser Bilder wird an kleinen Graphen überprüft, dass es sich um valide Färbungen handelt. Die Funktion $fast\_gnp\_random()$ des Python Pakets NetworkX erzeugt zufällige Graphen mit 10, 20, 40 und 80 Knoten zum Testen der Implementationen. Die Anzahl benötigter Farben für die entsprechende Färbung und die Laufzeit der Programme wird ausgewertet. Der Programmcode, der dieser Arbeit zugrunde liegt, und die erstellten Plots werden in einem privaten Git-Repository gespeichert und den Betreuern zur Verfügung gestellt.

Abstract

To solve d-1 and d-2 graph coloring problems, C programs using the GraphBLAS API are presented. The C implementation *SuiteSparse:GraphBLAS* by Timothy A. Davis is used for this purpose. The d-1 coloring colors arbitrarily undirected graphs, and the d-2 coloring colors undirected bipartite graphs. The GraphBLAS API uses linear algebra objects and operations to represent graph algorithms. The relation between graphs and linear algebra operations is explained, and the theoretical foundations of the GraphBLAS API are discussed. The colored graphs are visualized using the Python package NetworkX. Through these images, small graphs are used to check that the colorings are valid. The function $fast\_gnp\_random()$ of the Python package NetworkX generates random graphs with 10, 20, 40 and 80 nodes for testing the implementations. The number of colors required for the corresponding coloring and the runtime of the programs are evaluated. The program code on which this work is based and the created plots are stored in a private Git repository and made available to the supervisors.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

Coloring problems in graphs are an important part of graph theory. There are many different coloring problems for different classes of graphs. For example, the well-known four-color theorem states that every planar graph, i.e. graphs that can be drawn in the plane without edge intersections, can be colored with four colors [10, p. 2]. This thesis deals with the distance-1 (d-1) coloring problem and the distance-2 (d-2) coloring problem. A d-1 coloring of a graph $G = (V, E)$ assigns a color to each of its nodes so that adjacent nodes always have different colors [8, p. 253]. The aim is to find the smallest number of colors to color all nodes of a graph. Furthermore, bipartite graphs are considered. These are graphs in which the set of nodes can be divided into two sets, with no edges running between nodes within the sets. Here the d-1 coloring problem supplies two colors, one color for each set. Therefore, the d-2 coloring problem for bipartite graphs is considered, which states that two nodes must not have the same color if there is a node between them that is directly connected to the two nodes [11, p. 10].

This work aims to solve the d-2 coloring problem for bipartite graphs using the GraphBLAS API, in particular the *SuiteSparse:GraphBLAS* C implementation from Timothy A. Davis [6] which was introduced in May 2017. The LAGraph Github repository [2] collects previously developed algorithms using GraphBLAS. Among others, the algorithms breadth-first search, triangle count and PageRank are listed, but no algorithms that solve coloring problems are available. The target of this thesis is to close this gap. This goal is approached by first solving the d-1 coloring problem for arbitrary undirected graphs and then solving the d-2 coloring problem for undirected bipartite graphs. Using the GraphBLAS API, graph algorithms can be formulated in the language of linear algebra. The underlying insight behind the GraphBLAS API is that when a graph is represented by an adjacency matrix, each step of the breadth-first search consists of a matrix-vector multiplication [6, p. 18]. Sparse matrix and vector operations are defined by the GraphBLAS API on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms [7, p. 1]. In general, it is important to understand how

the GraphBLAS API works and how graph theoretical problems are solved by using the GraphBLAS API. The motivation for an approach using linear algebra are the decades of experience and optimization of linear algebra calculation methods such as matrix-vector products and the like.
Among other things, graph coloring is used for automatic differentiation. During the preconditioning the coloring is used so that preconditioning is combined with automatic differentiation in the best possible way [5]. Automatic differentiation is then used to solve non-linear systems of equations, such as in fluid dynamics, ice-sheet or glacier modeling [1].
The following chapter introduces the basic concepts of graph theory and explains the relationship between linear algebra objects and graphs. In addition, the central functionalities of the GraphBLAS API are discussed, which are used in the implementations of d-1 and d-2 coloring. In the following chapter, the use of custom functions is shown using the implementation of the matrix-vector product with functions of the GraphBLAS API. A pseudocode of the d-1 coloring algorithm using the GraphBLAS API is then discussed, and the C implementations of d-1 coloring and d-2 coloring are reviewed using code snippets. Finally, the written programs are evaluated using different graphs, whereby the number of colors required for different graphs and different numbers of nodes are compared. In the conclusion, an outlook is given on possible future questions and ideas for extensions and improvements to the previous implementation.

Chapter 2

Graph Algorithms in the Language of Linear Algebra

The chapter consists of three sub-chapters. First, a brief overview of the basic concepts of graph theory is given and the representation of graphs by adjacency matrices is described. Subsequently, a first example of a pseudo code of a graph coloring algorithm is presented. In addition, an introduction to the basic ideas behind the GraphBLAS API and typical program components are given.

2.1 Duality between Graphs and Matrices

Firstly, a few basic terms of graph theory are defined which are needed for future considerations.

Definition 2.1.1 (***graph***)**.** A graph G is a tuple $G = (V, E)$ consisting of a finite set $V \neq \emptyset$ and a set E of two-element subsets of V. The elements of V are called *nodes*. An element $e = \{v_1, v_2\}$ of E is called an *edge* with *end nodes* v_1 and v_2. We say that v_1 and v_2 are *incident* with e and that v_1 an v_2 are *adjacent* or *neighbors* of each other [8, p. 2].

Graphs are often illustrated with pictures in the plane, whereby the nodes of a graph $G = (V, E)$ are represented by bold type points and the edges by straight lines connecting the endpoints [8, p. 2].

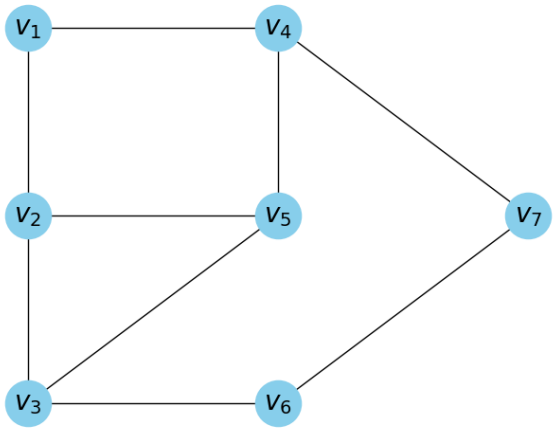


Figure 2.1: Example of an undirected graph with $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and several edges connecting the nodes.

In graph theory, a distinction is made between directed and undirected graphs. A graph is called directed if the edges have a direction and an edge can be used to get from one node to another but not to return via the same edge [8, p. 25]. Here, the edges are defined as an ordered tuple $e = (v_1, v_2)$ with v_1 as the start node and v_2 as the end node. The edges are shown as arrows. In undirected graphs, the edges have no direction. As soon as an edge runs between two nodes, you can get from one node to the neighboring node and back again [10, p. 12]. We only consider undirected graphs here. In Figure 2.1 an example of an undirected graph with $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and several edges between the nodes is shown. This exemplary graph will be used later in Chapter 3.2 for d-1 coloring.

Definition 2.1.2 (***bipartite graph***)**.** A graph G is called bipartite if there is a partition $V = A \cup B$ and $A \cap B = \emptyset$ of the node set V such that each edge $e \in E$ is incident with a node in A as well as with a node in B. This implies that G has no loops. Loops are edges from a node to itself [10, p. 42].

Bipartite graphs are used for d-2 coloring. If a bipartite graph is selected as input for d-1 coloring, the optimal coloring, i.e. the coloring with the fewest colors used, is always a coloring with two colors where each set is colored with one color. In Figure 2.2 an example of a bipartite graph with two sets of equal size is given.

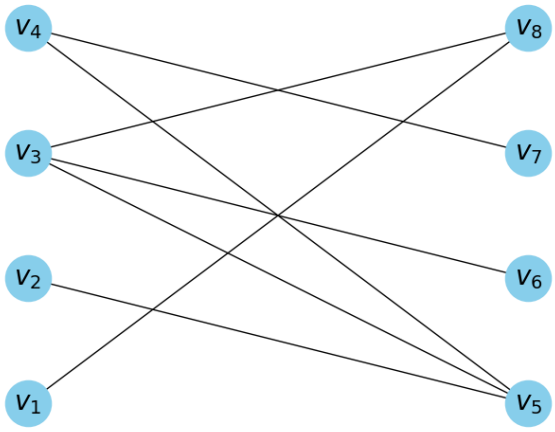
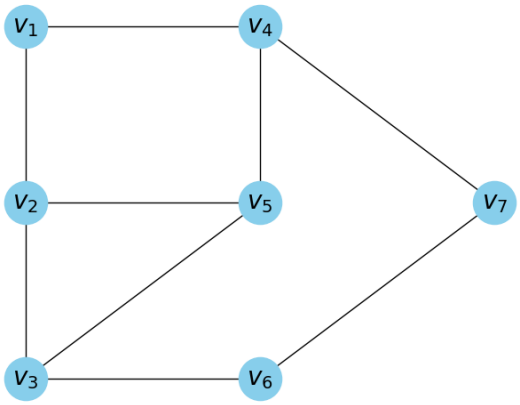


Figure 2.2: Example of a bipartite graph with two sets of nodes $A = \{v_1, v_2, v_3, v_4\}$ and $B = \{v_5, v_6, v_7, v_8\}$.

The core concept in an array-based graph algorithm is the duality between a graph and its adjacency representation. For a graph $G = (V, E)$ with N nodes and M edges, the $N \times N$ adjacency matrix A has the value $A(i, j) = 1$ if there is an edge e_{ij} from node v_i to node v_j and is zero otherwise. This duality allows graph algorithms to be simply expressed as a sequence of linear algebraic operations. Therefore many relations between fundamental linear algebraic operations and graph operations can be made [9, pp. 4f]. In Figure 2.3 the adjacency matrix A related to the example graph G from Figure 2.1 is shown.



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 2.3: The duality between a graph G and its adjacency matrix A.

Because of the undirected structure of graph G the adjacency matrix A is symmetric. Another observation is that the main diagonal of A is exclusively occupied by zeros, since the graph G has no loops.

2.2 Example of a Graph Coloring Algorithm

In Algorithm 1 the pseudocode of a graph coloring algorithm [13, p. 231] is shown.

Algorithm 1 Example of a graph coloring algorithm.

Input: Graph $G = (V, E)$
Output: Array C of colors for each $v \in V$

```
1: procedure COLOR(G)
2:     U ← V
3:     while |U| ≥ 0 do
4:         Choose an independent set I from U
5:         Color the nodes in I and put in C
6:         U ← U − I
7:     return C
```

An independent set in a graph $G = (V, E)$ is a subset I of the node set V such that no two nodes in I are adjacent [8, p. 55]. This approach uses independent sets for coloring the graph. In every iteration an independent set of nodes is found and then colored with one color. The colored nodes are removed from the set of nodes to be colored and the algorithm terminates as soon as there are no more nodes left to be colored.

2.3 GraphBLAS API Basics

The development of the GraphBLAS API began in 2013 with the publishing of the position paper *Standards for Graph Algorithm Primitives* [12], with the intention to define a common API using a standard set of primitives. The usage of a common API was motivated by the large amount of graph algorithms in terms of linear algebraic operations. Based on the *Basic Linear Algebra Subprograms* (BLAS) specification, GraphBLAS is intended to be an extension that satisfies the needs of graph algorithms [12]. A variety of interesting properties can be found in linear algebraic approaches to fundamental graph algorithms. These properties include the graph/adjacency matrix duality, corresponding with semiring operations, and extensions to tensors for representing multiple-edge graphs. [9, p. 4]
As shown in Chapter 2.1 graphs can be represented in terms of matrices, where the values stored in these matrices correspond to attributes of edges in the graph. The nodes of a graph are saved in a vector. Weights can be assigned to the individual nodes and are stored in a vector. Operations defined by the GraphBLAS API operate on these matrices and vectors. This duality is used, among other things, to express a breadth-first search step performed on a Graph G from a starting node s by a fundamental linear algebra operation, the matrix-vector product. In the following equation, the expression $v(s) = 1$ means that the vector v has the value 1 at the position s and is 0 otherwise.

$$BFS(G, s) \quad \Leftrightarrow \quad A^T v, \quad v(s) = 1 \tag{2.1}$$

Hence, graph algorithms can simply be recast as a sequence of linear algebraic operations. [9, p. 5]

Programs that use the GraphBLAS API use predefined and user-defined semirings. In mathematical terms, semirings are an algebraic structure consisting of a set of allowed values called domain, a commutative and associative binary operator called addition and a binary associative operator called multiplication. The multiplication distributes over addition from left and right. The identity elements are 0 for addition and 1 for multiplication. The additive identity is an annihilator over multiplication [4, p. 18]. The real numbers $\mathbb{R}$ and the set of all integers $\mathbb{Z}$ are semirings with the known operations of addition and multiplication. If $\mathbb{R}$ and $\mathbb{Z}$ are extended with $+\infty$, they become semirings with min as "addition" and + for "multiplication". Linear algebra on the $(min, +)$ semiring is often useful to solve different types of shortest path problems [9, p. 15]. In the context of GraphBLAS programs the definition of a semiring diverges from the mathematically rigorous definition since certain combinations of domains, operators and identity elements are useful in graph algorithms even if they don't strictly match the mathematical definition of a semiring [4, p. 18].

Objects defined in the GraphBLAS standard include types (the domains of elements), collections of elements (matrices, vectors, and scalars), operators on these elements (unary, index unary, and binary operators), algebraic structures (semirings and monoids), and descriptors. GraphBLAS objects are defined as opaque types which means that they can only be managed, manipulated, and accessed through the GraphBLAS API. The lifetime of each GraphBLAS object consists of the sequence of instructions executed in programmatic order between its creation and destruction. The GraphBLAS C API defines a number of these objects which are created during the initialization of the GraphBLAS context by executing GrB_init and released when the GraphBLAS context is terminated by calling $GrB\_finalize$. Before GraphBLAS objects can be used, they must be initialized by calling one of the constructor methods of the object. Each object type has an explicit constructor method of the form GrB_*_new, where '*' is replaced by the type of the object (e.g. GrB_Vector_new). Objects created explicitly by calling a constructor should be released by a call to GrB_free. In addition GraphBLAS objects can be manipulated by using a mask to control which computed values are stored in the output operand of a GraphBLAS operation. [4, pp. 23ff] The following chapter shows an implementation of a GraphBLAS C program that calculates a matrix-vector product to illustrate the use of GraphBLAS objects.

To get started with using GraphBLAS functions the installation of *SuiteSparse:GraphBLAS* C implementation is needed. For detailed installation instructions, see Tim Davis' Github

repository [3]. To use GraphBLAS objects and functions the header file *GraphBLAS.h* has to be included.

Chapter 3

Solving Coloring Problems by Using the GraphBLAS API

This chapter describes the implementation of the GraphBLAS C programs for solving the d-1 and d-2 coloring problems. First, the required custom functions for printing and initializing GraphBLAS objects are introduced and demonstrated using a C implementation with GraphBLAS objects for calculating the matrix-vector product. The d-1 coloring algorithm is described using pseudocode and the individual iterations are visualized through an example graph. The C implementations of the d-1 and d-2 coloring algorithm are then discussed in detail by applying code snippets.

3.1 Custom Functions and Matrix-Vector Product

Firstly, a simple C program which uses the GraphBLAS API is provided. The C program called *matrix-vector-mul.c* performs a matrix-vector multiplication for 3×3 matrices and vectors of size 3. In order to have a better overview of the *main-function*, auxiliary functions are implemented at the beginning. To check the results of matrix-vector multiplications, it is useful to write custom functions for printing matrices and vectors. The GraphBLAS API also offers predefined functions for this purpose, but there is no possibility to represent matrices as 2-dimensional objects. In Listing 3.1 the output of the predefined version from the GraphBLAS API for printing 4×4 matrices is shown on the left and the output of the self-defined print function is shown on the right.

```
    (0,0)    0                    0 1 0 1
    (0,1)    1                    1 0 1 0
    (0,2)    0                    0 1 0 1
    (0,3)    1                    1 0 1 0
    (1,0)    1
    (1,1)    0
    (1,2)    1
    (1,3)    0
    (2,0)    0
    (2,1)    1
    (2,2)    0
    (2,3)    1
    (3,0)    1
    (3,1)    0
    (3,2)    1
    (3,3)    0
```

Listing 3.1: Comparison of the output of the GraphBLAS and the custom function for printing a 4×4 matrix.

The custom function *printMatrix()* requires a matrix of type *GrB_Matrix* as an input parameter and is shown in Listing 3.2. To iterate through the matrix, the indices *nrows* and *ncols* of type *GrB_Index* are defined. These are assigned the number of rows and columns of the input matrix using *GrB_Matrix_nrows()* and *GrB_Matrix_ncols()*. Now a nested *for-loop* iterates over the matrix so that each entry is extracted and printed using the function *GrB_Matrix_extractElement_FP64()*. This prints the rows of the matrix one below the other. Similarly, the function *printVector()* is implemented to print vectors.

```c
// Custom function to print a GraphBLAS matrix
void printMatrix(GrB_Matrix matrix) {
    GrB_Index nrows, ncols;
    GrB_Matrix_nrows(&nrows, matrix);
    GrB_Matrix_ncols(&ncols, matrix);

    for (GrB_Index i = 0; i < nrows; i++) {
        for (GrB_Index j = 0; j < ncols; j++) {
            double value;
            GrB_Matrix_extractElement_FP64(&value, matrix, i, j);
            printf("%f ", value);
        }
        printf("\n");
    }
}
```

Listing 3.2: Custom function to print a GraphBLAS matrix.

Next, functions for filling matrices and vectors with random values are required. This is done by using the custom functions *fillMatrixRandom()* and *fillVectorRandom()*. The *fillMatrixRandom()* function is shown in Listing 3.3.

```
28 // Custom function to fill a matrix with random values
29 void fillMatrixRandom(GrB_Matrix matrix, double range) {
30     GrB_Index nrows, ncols;
31     GrB_Matrix_nrows(&nrows, matrix);
32     GrB_Matrix_ncols(&ncols, matrix);
33 
34     for (GrB_Index i = 0; i < nrows; i++) {
35         for (GrB_Index j = 0; j < ncols; j++) {
36             GrB_Matrix_setElement_FP64(matrix, ((double)rand() / RAND_MAX) *
     range, i, j);
37         }
38     }
39 }
```

Listing 3.3: Custom function to fill a GraphBLAS matrix with random values.

The input parameters of *fillMatrixRandom()* are a matrix of type *GrB_Matrix* and the value *range* of type *double*. To specify the scope of the random value, utilize the *range* parameter. Analogously to Listing 3.2, the indices *nrows* and *ncols* for iterating over the matrix are defined. By iterating row by row in a nested *for-loop*, a random value within the specified range for each element using the *GrB_Matrix_setElement_FP64()* function is set. For this purpose, include the C library *stdlib.h* header file at the beginning of the program and use the *rand()* function.
All auxiliary functions are now implemented and the main function can be viewed in Listing 3.6. Firstly, the GraphBLAS context is initialized using the function *GrB_init()*. There are two possible input parameters for this function *GrB_BLOCKING* and *GrB_NONBLOCKING*, which define the mode. In *GrB_BLOCKING* mode, the methods are executed in program order. In *GrB_NONBLOCKING* mode, the methods can be executed in parallel and in any order. The effects of the choice of mode are not investigated in this thesis and all programs are implemented in *GrB_BLOCKING* mode. In Line 78 to 83 of Listing 3.6 the 3×3 matrix A and vectors x and y of size 3 are created by using the *GrB_Matrix_new()* and *GrB_Vector_new()* methods. Next, the matrix A and vectors x and y are filled with random values in the range from 0 to 10 and printed by using the earlier discussed auxiliary functions. Finally, the matrix-vector multiplication is performed using the GraphBLAS operation *GrB_mxv()*, which is shown in Listing 3.4.

```
96     // Perform matrix-vector multiplication: y = A * x
97     GrB_mxv(y, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP64, A, x,
    GrB_NULL);
```

Listing 3.4: Matrix-vector multiplication using the GraphBLAS function *GrB_mxv()*.

The matrix A and the vector x are taken as input and the matrix-vector product of these variables is calculated by using the semiring *GrB_PLUS_TIMES_SEMIRING_FP64*. The result is written to the vector y. The second parameter is the mask. The default

value *GrB_NULL* is used here, which means that all results of the matrix-vector operation are saved in the output vector y. The third parameter specifies whether the values are to be accumulated or assigned in the existing output vector y. The default value *GrB_NULL* is selected here, whereby the values are assigned. The last parameter corresponds to the descriptor. The descriptor is used in GraphBLAS to modify the behavior of a GraphBLAS method and appears as the last argument in the signature of a method. For example, the descriptor can specify that an input matrix must be transposed or that the complement of the mask must be used. As in all following implementations, the descriptor is set to the default value *GrB_NULL*. This means that the input matrix is not transposed and the mask is used as it is. The result vector y is printed and at the end all GraphBLAS objects are released using the *GrB_free()* method. The GraphBLAS context is ended with *GrB_finalize()*. Listing 3.5 illustrates the output of the matrix-vector product implementation.

```
Matrix A:
1.980521 7.232950 1.670938
2.188172 6.585775 5.861900
7.163243 8.878914 1.404522

Vector x:
[3.047422, 1.259153, 9.200778]

Result Vector y:
[30.516801, 68.894826, 45.932035]
```

Listing 3.5: Output of the matrix-vector-mul.c program.

```
int main() {
    // Initialize GraphBLAS
    GrB_init(GrB_BLOCKING);

    // Seed the random number generator
    srand((unsigned int)time(NULL));

    // Create matrix A and vectors x,y
    GrB_Matrix A;
    GrB_Vector x;
    GrB_Vector y;
    GrB_Matrix_new(&A, GrB_FP64, 3, 3);
    GrB_Vector_new(&x, GrB_FP64, 3);
    GrB_Vector_new(&y, GrB_FP64, 3);

    // Fill matrix A and vector x
    fillMatrixRandom(A, 10.0);
    fillVectorRandom(x, 10.0);

    // Print matrix A and vector x
```

```
90      printf("\nMatrix A:\n");
91      printMatrix(A);
92      printf("\n");
93      printVector(x, "Vector x");
94      printf("\n");
95
96      // Perform matrix-vector multiplication: y = A * x
97      GrB_mxv(y, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP64, A, x,
        GrB_NULL);
98
99      // Print the result vector y
100     printVector(y, "Result Vector y");
101
102     // Free matrices and finish
103     GrB_Matrix_free(&A);
104     GrB_Vector_free(&x);
105     GrB_Vector_free(&y);
106     GrB_finalize();
107
108     return 0;
109 }
```

Listing 3.6: Main function to perform a matrix-vector-product.

3.2 GraphBLAS d-1 Coloring Algorithm

This section explores the d-1 coloring algorithm, starting with a pseudocode solution. The particular iterations are traced and visualized on an example graph. A C program designed to solve the d-1 coloring problem is discussed in which the *SuiteSparse:GraphBLAS* C implementation is used.

3.2.1 Pseudocode of a d-1 Coloring Algorithm

Following the explanations of Muhammad Osama et al. in the paper *Graph Coloring on the GPU* [13], Algorithm 2 illustrates a d-1 coloring algorithm in pseudocode using functions of the GraphBLAS API.

Algorithm 2 Pseudocode of a GraphBLAS d-1 coloring algorithm.

Input: Adjacency matrix A of graph $G = (V, E)$ and already built empty vectors C, $weight$, $frontier$.

Output: Array C of colors for each $v \in V$.

```
1:  procedure GRAPHBLASCOLOR(A, C)
2:      ▷ Initialize colors to 0
3:      GrB_assign(C, GrB_NULL, 0, GrB_ALL, nrows(A), desc);
4:      ▷ Assign random weight to each node
5:      GrB_apply(weight, GrB_NULL, GrB_NULL, set_random(), weight, desc);
6:      for each color = 1, ..., n do
7:          ▷ Find max of neighbours
8:          GrB_vxm(max, GrB_NULL, GrB_NULL, GrB_INT32MaxTimes,
            weight, A, desc);
9:          GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL, GrB_INT32GT,
            weight, max, desc);
10:         ▷ Find all largest nodes that are uncolored
11:         GrB_reduce(succ, GrB_NULL, GrB_INT32Plus, frontier, desc);
12:         ▷ Stop when frontier is empty
13:         if succ = 0 then
14:             break;
15:         ▷ Assign new color
16:         GrB_assign(C, frontier, GrB_NULL, color, GrB_ALL, nrows(A), desc);
17:         ▷ Get rid of colored nodes in candidate list
18:         GrB_assign(weight, frontier, GrB_NULL, 0, GrB_ALL, nrows(A), desc);
```

Algorithm 2 requires an already created adjacency matrix A of a graph $G = (V, E)$ and empty vectors C, $weight$ and $frontier$. The size of the vectors corresponds to the number of nodes in V. First, the values of the coloring vector C are set to 0 using the operation *GrB_assign()* from the GraphBLAS API. The *GrB_apply()* operation is then used to fill the vector *weight* with distinct random values which are greater than 0. A previously implemented auxiliary function *set_random()* is required for this. In the following, the loop iterations of Algorithm 2 are reproduced step by step. Therefore we take a look at the example graph in Figure 2.1 from Chapter 2.1. For every node, a random value as weight is assigned, as shown in Figure 3.1.

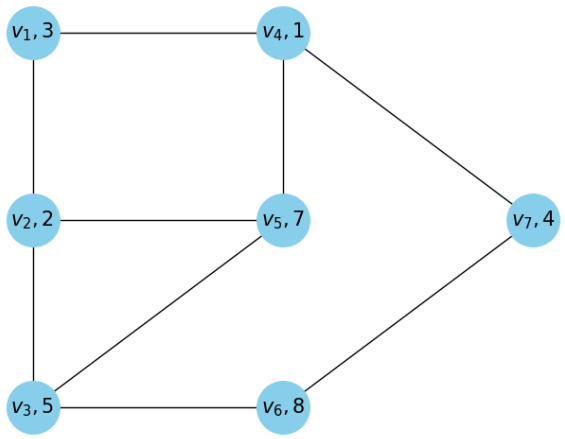


Figure 3.1: Graph for coloring process with random weights.

The maximum of the neighbors weights for every node is computed by using $GrB\_vxm()$ with the $GrB\_INT32MaxTimes$ predefined semiring in line 8. Afterwards the $frontier$ vector is filled with ones if the weight of the associated node is greater than the maximum weight of its neighbors. This is done by using the $GrB\_INT32GT$ comparison function. The values of the variables after the first iteration are displayed in Table 3.1.

$Node$	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$max_{(neighbours)}$	2	7	8	7	5	5	8
$weights$	3	2	5	1	7	8	4
$frontier_{(weight>max)}$	1	0	0	0	1	1	0

Table 3.1: First iteration of the d-1 coloring algorithm.

Using the $GrB\_assign()$ function in line 16, a color is assigned to each node whose $frontier$ variable is 1. It follows from the $frontier$ values in Table 3.1 that color 1 (e.g. blue) is selected for v_1, v_5 and v_6. The $GrB\_assign()$ function in line 18 sets the weights of the colored nodes to 0 which means that these nodes are not candidates for coloring in any following iteration. For the next iteration the graph in Figure 3.2 needs to be colored.

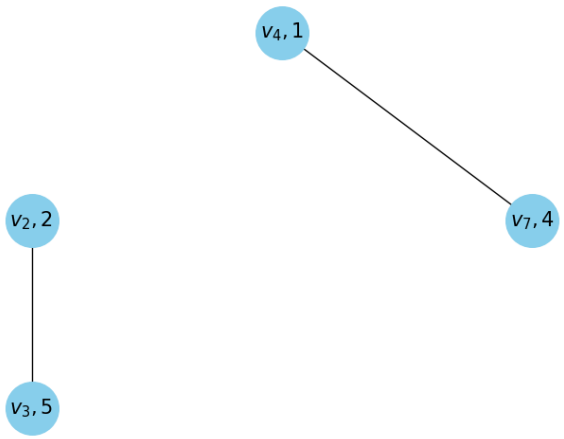


Figure 3.2: Graph for coloring process with random weights after the first iteration.

The second iteration of Algorithm 2 starts and the following values are assigned.

$Node$	v_2	v_3	v_4	v_7
$max_{(neighbours)}$	5	2	4	1
$weights$	2	5	1	4
$frontier_{(weight>max)}$	0	1	0	1

Table 3.2: Second iteration of the d-1 coloring algorithm.

This results in the next color 2 (e.g. green) being selected for the nodes v_3 and v_7. Therefore only two nodes v_2 and v_4 are remaining for the third iteration. The values of the third iteration are shown in Table 3.3.

$Node$	v_2	v_4
$max_{(neighbours)}$	0	0
$weights$	2	1
$frontier_{(weight>max)}$	1	1

Table 3.3: Third iteration of the d-1 coloring algorithm.

In a final step v_2 and v_4 are colored with the color 3 (e.g. yellow). The termination criterion applies because the $GrB\_reduce()$ function in line 11 sets the *succ* variable to zero, as the $frontier$ vector now only consists of zeros. This results in the final coloring vector $C = (1, 3, 2, 3, 1, 1, 2)$. The now fully colored graph is shown in Figure 3.3.

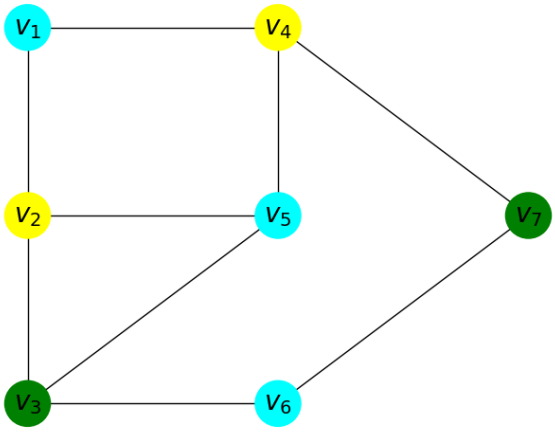


Figure 3.3: Final colored graph with coloring $C = (1, 3, 2, 3, 1, 1, 2)$.

3.2.2 C Implementation of a GraphBLAS d-1 Coloring Algorithm

The complete implementation is shown in Appendix A.1. The main components of the program are discussed below with the use of code snippets. Further auxiliary functions are required, before we get to the *main-function* of the program. First, a custom function called *createAdjacencyMatrix()*, as shown in Listing 3.7, is implemented, which creates and initializes the adjacency matrix of a graph.

```
18 GrB_Matrix createAdjacencyMatrix() {
19     GrB_Matrix adjacency_matrix;
20     GrB_Matrix_new(&adjacency_matrix, GrB_INT32, MATRIX_SIZE, MATRIX_SIZE);
21
22     // Define the adjacency matrix data
23     int32_t data[MATRIX_SIZE][MATRIX_SIZE] = {
24         {0, 1, 0, 1, 0, 0, 0},
25         {1, 0, 1, 0, 1, 0, 0},
26         {0, 1, 0, 0, 1, 1, 0},
27         {1, 0, 0, 0, 1, 0, 1},
28         {0, 1, 1, 1, 0, 0, 0},
29         {0, 0, 1, 0, 0, 0, 1},
30         {0, 0, 0, 1, 0, 1, 0}
31     };
32
33     // Populate the adjacency matrix
34     for (int i = 0; i < MATRIX_SIZE; ++i) {
35         for (int j = 0; j < MATRIX_SIZE; ++j) {
36             GrB_Matrix_setElement_INT32(adjacency_matrix, data[i][j], i, j);
37         }
38     }
39     return adjacency_matrix;
```

Listing 3.7: Auxiliary function for adjacency matrix creation and initialization.

For this purpose, an empty GraphBLAS matrix is created and the *GrB_Matrix_new()* function is used to select the number of rows and columns as *MATRIX_SIZE* and *GrB_INT32* as the type of entries. Here, *MATRIX_SIZE* is a macro which is defined by the preprocessor directive $\#define$. The matrix data is created as a two-dimensional array named *data* with 0s and 1s. Using nested loops, the values from the array *data* are inserted into the created GraphBLAS matrix. The *GrB_Matrix_setElement_INT32* function is used to set the value at a specific position in the matrix. To assign random weights to the nodes, the custom function *generateDistinctRandom()* is required, which generates different random integer values between 1 and 100. The range of values can be adjusted for larger graphs. In addition there are custom functions for printing GraphBLAS matrices, vectors and scalars. These are based on the function *printMatrix()* already discussed in Listing 3.2 from Chapter 3.1.
The *main-function* begins with the initialization of GraphBLAS using *GrB_init()* and choosing the $GrB\_BLOCKING$ mode. The previously implemented function *createAdjacencyMatrix()* creates the adjacency matrix named *adjacency_matrix*. The GraphBLAS vectors C, $weight$, $frontier$, max of size *MATRIX_SIZE* and type *GrB_INT32* and a GraphBLAS scalar *succ* of type *GrB_INT32* are created and initialized using *GrB_Vector_new()* and *GrB_Scalar_new()* as shown in Listing 3.8. In addition, there is an auxiliary variable of type integer called $Intsucc$.

```
114     GrB_Vector C;
115     GrB_Vector weight;
116     GrB_Vector frontier;
117     GrB_Vector max;
118     GrB_Scalar succ;
119     GrB_Vector_new(&C, GrB_INT32, MATRIX_SIZE);
120     GrB_Vector_new(&weight, GrB_INT32, MATRIX_SIZE);
121     GrB_Vector_new(&frontier, GrB_INT32, MATRIX_SIZE);
122     GrB_Vector_new(&max, GrB_INT32, MATRIX_SIZE);
123     GrB_Scalar_new(&succ, GrB_INT32);
124     int32_t Intsucc;
```

Listing 3.8: Definition and initialization of GraphBLAS objects and auxiliary variable.

The values of the coloring vector C are set to 0 using the function *GrB_assign()*. Different random integer values are assigned to the *weight* vector using the self-defined auxiliary function *generateDistinctRandom()* and the function *GrB_Vector_setElement_INT32()* as shown in Listing 3.9.

```
135     GrB_assign(C, GrB_NULL, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE, GrB_NULL);
136     printVector(C, "Colors initialized to 0");
137
138     // Assign distinct random integer as weight to each vertex
139     for (GrB_Index i = 0; i < MATRIX_SIZE; i++) {
140         int random_value = generateDistinctRandom();
141         GrB_Vector_setElement_INT32(weight, random_value, i);
142     }
143     printVector(weight, "weight with random values");
```

Listing 3.9: Setting the coloring vector C to zero and filling the *weight* vector with different random integer values.

Now the coloring process begins, which is described by a for loop as shown in Listing 3.10. The function *GrB_vxm()* forms the matrix-vector product of the weight vector and the adjacency matrix, whereby the standard arithmetic semiring $(\times, +, \mathbb{R})$ is overloaded with the predefined semiring *GrB_MAX_TIMES_SEMIRING_INT32* and the result is written to the vector *max*. This calculates the maximum of the weights of its neighbors for each node. The *frontier* vector is now assigned a 1 at each position if the weight of a node is greater than the maximum of the weights of its neighbors, and 0 otherwise. The function *GrB_eWiseAdd()* and the predefined semiring *GrB_GT_INT32* are used for this purpose. To check the termination criterion, the scalar value *succ* is calculated using the function *GrB_reduce()*. To do this, all values in *frontier* are added up with the predefined semiring *GrB_PLUS_INT32* and saved in *succ*. As soon as the value for *succ* is 0, the for loop is exited as no further nodes can be colored. In the next step, the nodes in the coloring vector C are colored with the first color. This is guaranteed by the function *GrB_assign()*, which uses *frontier* as a mask and thus assigns the first color to exactly those nodes for which the *frontier* vector has the value 1. Finally, the *GrB_assign()* function sets the weights of the colored nodes to 0, which means that they are not eligible for coloring in the next iteration step. The for loop is exited and the d-1 coloring vector is printed as soon as the termination criterion is met. In the end, all objects used are released by *GrB_free()* and the GraphBLAS context is ended with *GrB_finalize()*.

```
145     for(GrB_Index color = 1; color < MATRIX_SIZE; color++) {
146       // Find maximum of neighbors
147       GrB_vxm(max, GrB_NULL, GrB_NULL, GrB_MAX_TIMES_SEMIRING_INT32, weight,
        adjacency_matrix, GrB_NULL);
148       printVector(max, "max");
149
150       GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL, GrB_GT_INT32, weight, max,
      GrB_NULL);
151       printVector(frontier, "frontier after Greater Than operation");
152
153       // Find number of nodes that will be colored
154       GrB_reduce(succ, GrB_NULL, GrB_PLUS_INT32, frontier, GrB_NULL);
155
```

```
156       printScalarInt32(succ, "succ");
157
158       // Gets integer value from GrB_Scalar Type succ to be able to check
      the following break condition
159       GrB_Scalar_extractElement_INT32(&Intsucc, succ);
160
161       // Stop when frontier is empty
162       if (Intsucc == 0) {
163          break;
164       }
165
166       // Assign new color
167       GrB_assign(C, frontier, GrB_NULL, color, GrB_ALL, MATRIX_SIZE,
      GrB_NULL);
168       printVector(C, "C after iteration");
169       printf("\n");
170
171       // Get rid of colored nodes in candidate list
172       GrB_assign(weight, frontier, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE,
      GrB_NULL);
173       printVector(weight, "weight after iteration");
174    }
```

Listing 3.10: Calculation of the coloring vector C.

In Appendix A.1 print statements are included at several points in order to monitor the steps of the algorithm and visually trace them at the end. The results of the implementation are discussed in the Chapter 4. For the plots of the colored graphs discussed there, a reduced C program is used, which only outputs the input matrix and the coloring vector C.

3.3 GraphBLAS d-2 Coloring Algorithm

Another coloring problem is the distance-2 coloring of a graph, called d-2 coloring. Given a graph $G = (V, E)$, in the d-2 coloring problem on G, the objective is to assign a color i_v to each node $v \in V$ such that any two nodes v_1 and v_2 at distance 2 in G are assigned different colors $i_{v_1} \neq i_{v_2}$. The d-2 coloring is traced back to the d-1 coloring. Thus, the following assertion is made and subsequently proven.

Theorem 3.3.1. Let $G = (V, E)$ be a graph with n nodes represented by the adjacency matrix A. Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be the graph represented by A^2, where the diagonal of A^2 is set to 0 and all values greater than 1 are set to 1. Then the following applies:
The d-1 coloring of $\tilde{G}$ corresponds to the d-2 coloring of G.

Proof. Let $k \in V$ and $\Phi(i)$ be the color of node i. It suffices to show that the following equivalence holds:
Let $(i,j) \in \tilde{E}$ with $\Phi(i) \neq \Phi(j)$
$\Leftrightarrow \exists$ path from i via k to j in G with $\Phi(i) \neq \Phi(j)$
Let $(i,j) \in \tilde{E}$ with $\Phi(i) \neq \Phi(j)$. Then follows with the summation notation of the matrix-matrix product:

$$\begin{aligned} A^2(i,j) &= \sum_{k=1}^{n} A(i,k) \cdot A(k,j) \\ &= \sum_{k=1}^{n} A(i,k) \cdot A(k,j) \qquad \forall k \text{ with } A(i,k) \neq 0 \text{ and } A(k,j) \neq 0 \\ &= s \end{aligned}$$

$\Leftrightarrow s$ is the number of paths of length 2 in G from i to j with $\Phi(i) \neq \Phi(j)$
Since $(i,j) \in \tilde{E}$ it follows that s is at least 1. □

3.3.1 C Implementation of a GraphBLAS d-2 Coloring Algorithm

The complete implementation is shown in Appendix A.2. In many parts, d-2 coloring is similar to the d-1 coloring program from Chapter 3.2.2. Therefore, only the most important new components of the program are explained below using code snippets. To calculate the d-2 coloring of a graph, the input adjacency matrix is modified using the auxiliary function *matrixManipulation()*, which is shown in Listing 3.11.

```
68 GrB_Matrix matrixManipulation(GrB_Matrix adjacency_matrix) {
69     GrB_Matrix adjacency_matrix_manipulated;
70     GrB_Matrix_new(&adjacency_matrix_manipulated, GrB_INT32, MATRIX_SIZE,
       MATRIX_SIZE);
71
72     // Perform matrix multiplication: adjacency_matrix_manipulated =
       adjacency_matrix * adjacency_matrix
73     GrB_mxm(adjacency_matrix_manipulated, GrB_NULL, GrB_NULL,
       GxB_PLUS_TIMES_INT32, adjacency_matrix, adjacency_matrix, GrB_NULL);
74     printf("\nAdjacency matrix square:\n");
75     printMatrix(adjacency_matrix_manipulated);
76
77     // Set diagonal entries to zero
78     setDiagonalToZero(adjacency_matrix_manipulated);
79
80     // Display the matrix with diagonal entries set to zero
81     printf("\nMatrix with diagonal entries set to zero:\n");
82     printMatrix(adjacency_matrix_manipulated);
```

```
83
84      // Apply the condition: set entries greater than 1 to 1
85      GrB_apply(adjacency_matrix_manipulated, GrB_NULL, GrB_NULL,
        GrB_MIN_INT32, 1, adjacency_matrix_manipulated, GrB_NULL);
86
87      // Display the modified matrix
88      printf("\nElements greater 1 set to 1:\n");
89      printMatrix(adjacency_matrix_manipulated);
90
91      // Return the manipulated matrix
92      return adjacency_matrix_manipulated;
93  }
```

Listing 3.11: Function to manipulate the adjacency matrix and return the resulting matrix.

First, the square of the adjacency matrix is calculated using the GraphBLAS function *GrB_mxm*() and stored in the previously initialized GraphBLAS matrix called *adjacency_matrix_manipulated*. The diagonal of *adjacency_matrix_manipulated* is set to 0 using a predefined auxiliary function, which can be seen in Listing 3.12. Here, the matrix is iterated over in a for loop and the values on the diagonal are set to 0 by the GraphBLAS function *GrB_Matrix_setElement_INT32()*.

```
41  void setDiagonalToZero(GrB_Matrix matrix) {
42      GrB_Index nrows, ncols;
43      GrB_Matrix_nrows(&nrows, matrix);
44      GrB_Matrix_ncols(&ncols, matrix);
45
46      for (GrB_Index i = 0; i < nrows; ++i) {
47          GrB_Matrix_setElement_INT32(matrix, 0, i, i);
48      }
49  }
```

Listing 3.12: Function to set diagonal entries of a GraphBLAS matrix to zero.

All values of the matrix that are greater than 1 are set to 1 using the GraphBLAS function *GrB_apply()*. The resulting matrix has the form of an adjacency matrix which represents an undirected graph without loops. This graph is colored using the d-1 coloring algorithm. The resulting coloring C corresponds to a valid d-2 coloring of the input graph as shown in Theorem 3.3.1. In the *main-function*, the previously discussed auxiliary function *matrixManipulation()* is applied to the input adjacency matrix and the resulting matrix is used as the input matrix in the *GrB_vxm()* function, where the maximum weight of neighbors for each node is calculated. The two changes are shown in Listing 3.13.

```
145     GrB_Matrix manipulated_matrix = matrixManipulation(adjacency_matrix);
146       GrB_vxm(max, GrB_NULL, GrB_NULL, GrB_MAX_TIMES_SEMIRING_INT32, weight,
        manipulated_matrix, GrB_NULL);
```

Listing 3.13: Changes in the main-function between d-1 and d-2 coloring.

Chapter 4

Evaluation

The programs from Chapter 3.2.2 and 3.3 are tested and evaluated in three ways. Firstly, directly via the generated executable of the respective C program, whose output is printed in the terminal. Another way to evaluate the C program is to use the Python package NetworkX to plot the colored graph. For small graphs, the coloring can be observed visually. Finally, the number of colors required for d-1 and d-2 coloring are compared for graphs of different sizes. To generate the graphs, the function $fast\_gnp\_random\_graph()$ of the Python package NetworkX is used, which generates random Erdős-Rényi graphs.

4.1 Output Observation

Firstly, the output of the d-1 coloring C Program shown in Listing A.1 is discussed. Here the example graph from Figure 2.1 is colored. The output starts with the adjacency matrix, which represents the graph to be colored. The vectors C, $weight$ and $frontier$ are then printed before their initialization as shown in Listing 4.1.

```
1  Adjacency matrix:
2  0 1 0 1 0 0 0
3  1 0 1 0 1 0 0
4  0 1 0 0 1 1 0
5  1 0 0 0 1 0 1
6  0 1 1 1 0 0 0
7  0 0 1 0 0 0 1
8  0 0 0 1 0 1 0
9
10 Vectors:
11 C:
12 [22058, 22058, 22058, 22058, 22058, 22058, 22058]
13 weight:
14 [22058, 22058, 22058, 22058, 22058, 22058, 22058]
15 frontier:
```

```
[22058, 22058, 22058, 22058, 22058, 22058, 22058]
```

Listing 4.1: First part of the output of the d-1 coloring C program.

The vector C set to 0 and the *weight* vector with different, random integer values between 1 and 100 are then printed. Next, the vectors *weight*, *max*, *frontier* and the coloring vector C are printed one below the other for each loop pass. The output shown in Listing 4.2 follows the same pattern as the Tables 3.1 - 3.3 from Chapter 3.2.1.

```
Colors initialized to 0:
[0, 0, 0, 0, 0, 0, 0]
weight with random values:
[86, 14, 22, 69, 99, 79, 73]
max:
[69, 99, 99, 99, 69, 73, 79]
frontier after Greater Than operation:
[1, 0, 0, 0, 1, 1, 0]
succ: 3
C after iteration:
[1, 0, 0, 0, 1, 1, 0]

weight after iteration:
[0, 14, 22, 69, 0, 0, 73]
max:
[69, 22, 14, 73, 69, 73, 69]
frontier after Greater Than operation:
[0, 0, 1, 0, 0, 0, 1]
succ: 2
C after iteration:
[1, 0, 2, 0, 1, 1, 2]

weight after iteration:
[0, 14, 0, 69, 0, 0, 0]
max:
[69, 0, 14, 0, 69, 0, 69]
frontier after Greater Than operation:
[0, 1, 0, 1, 0, 0, 0]
succ: 2
C after iteration:
[1, 3, 2, 3, 1, 1, 2]

weight after iteration:
[0, 0, 0, 0, 0, 0, 0]
max:
[0, 0, 0, 0, 0, 0, 0]
frontier after Greater Than operation:
[0, 0, 0, 0, 0, 0, 0]
succ: 0
```

Listing 4.2: Second part of the output of the d-1 coloring C program.

Finally, the calculated d-1 coloring is printed in the form of the vector C.

```
58 d1-coloring:
59 [1, 3, 2, 3, 1, 1, 2]
```

Listing 4.3: Third part of the output of the d-1 coloring C program.

For various runs of the program, distinct colorings are calculated because of the different settings of the random weight vector. The coloring calculated in Listing 4.3 matches the coloring in Figure 3.3. Further valid colorings are shown in Listing 4.4.

```
1 [3, 2, 1, 1, 3, 2, 3], [1, 2, 3, 2, 1, 4, 3], [1, 3, 2, 2, 1, 3, 4],
2 [1, 2, 3, 2, 1, 1, 3], [2, 1, 3, 1, 2, 1, 2], [1, 3, 1, 3, 2, 2, 1],
3 [2, 1, 2, 1, 3, 3, 4]. [4, 1, 2, 3, 4, 1, 2], [2, 1, 2, 1, 3, 3, 2]
```

Listing 4.4: Valid colorings of the d-1 coloring.

The expected d-1 coloring with two different colors is obtained by choosing an adjacency matrix that represents a bipartite graph. For this purpose, the adjacency matrix of the bipartite graph in Figure 2.2 is selected, which is shown in Listing 4.5 with the resulting coloring.

```
1 Adjacency matrix:
2 0 0 0 0 0 0 0 1
3 0 0 0 0 1 0 0 0
4 0 0 0 0 1 1 0 1
5 0 0 0 0 1 0 1 0
6 0 1 1 1 0 0 0 0
7 0 0 1 0 0 0 0 0
8 0 0 0 1 0 0 0 0
9 1 0 1 0 0 0 0 0
10
11 d1-coloring:
12 [2, 2, 2, 2, 1, 1, 1, 1]
```

Listing 4.5: Result of the d-1 coloring for a bipartite graph.

Below, the output of the d-2 coloring C program from Listing A.2 is discussed, whereby the complete output is shown in Appendix A.3. The input adjacency matrix is squared, the diagonal is set to 0 and all values greater than 1 are set to 1. These modifications are shown in lines 1-49. The modified input matrix is then colored using the d-1 coloring algorithm. The resulting coloring corresponds to a valid d-2 coloring of the original input graph according to Therorem 3.3.1. The results of d-2 coloring for different graphs are shown in Listing 4.6 and Listing 4.7.

```
Adjacency Matrix:
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 1
0 0 0 0 1 0 1 0
0 1 1 1 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
1 0 1 0 0 0 0 0

d-2 coloring:
[1, 1, 2, 3, 1, 2, 2, 3]
```

Listing 4.6: Result of the d-2 coloring for a bipartite graph with 8 nodes.

```
Adjacency Matrix:
0 0 0 0 0 0 1 0 1 1 1 0
0 0 0 0 0 0 1 0 1 1 0 1
0 0 0 0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 1 0 1 0 0 1
0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 1 1 1 0 1 1
1 1 1 1 1 1 0 0 0 0 0 0
0 0 1 0 1 1 0 0 0 0 0 0
1 1 1 1 1 1 0 0 0 0 0 0
1 1 0 0 1 0 0 0 0 0 0 0
1 0 0 0 1 1 0 0 0 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0

d-2 coloring:
[3, 5, 2, 1, 6, 4, 3, 4, 5, 2, 1, 6]
```

Listing 4.7: Result of the d-2 coloring for a bipartite graph with 12 nodes.

4.2 Graph Visualization

Another way to evaluate the C programs is to use the Python package NetworkX to plot the colored graphs. For this purpose, a Python wrapper is implemented that takes the *executable* files of the d-1 and d-2 coloring C programs as an input. The output of the *executable* files is reduced so that it only contains the adjacency matrix of the graph to be colored and the coloring vector, which assigns a color to each node. The nodes of a graph with n nodes are labeled v_1 to v_n. The graph is plotted according to its adjacency matrix and corresponding colors are set for the calculated coloring. For the d-1 coloring, the graphs in Listing 4.8, represented by their adjacency matrices, are plotted.

$$
A = \begin{matrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{matrix}
\quad
B = \begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0
\end{matrix}
\quad
C = \begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{matrix}
$$

Listing 4.8: d-1 coloring input matrices for evaluation with NetworkX.

The plots of the graphs, which are represented by the adjacency matrices A, B and C, are shown in Figure 4.1.

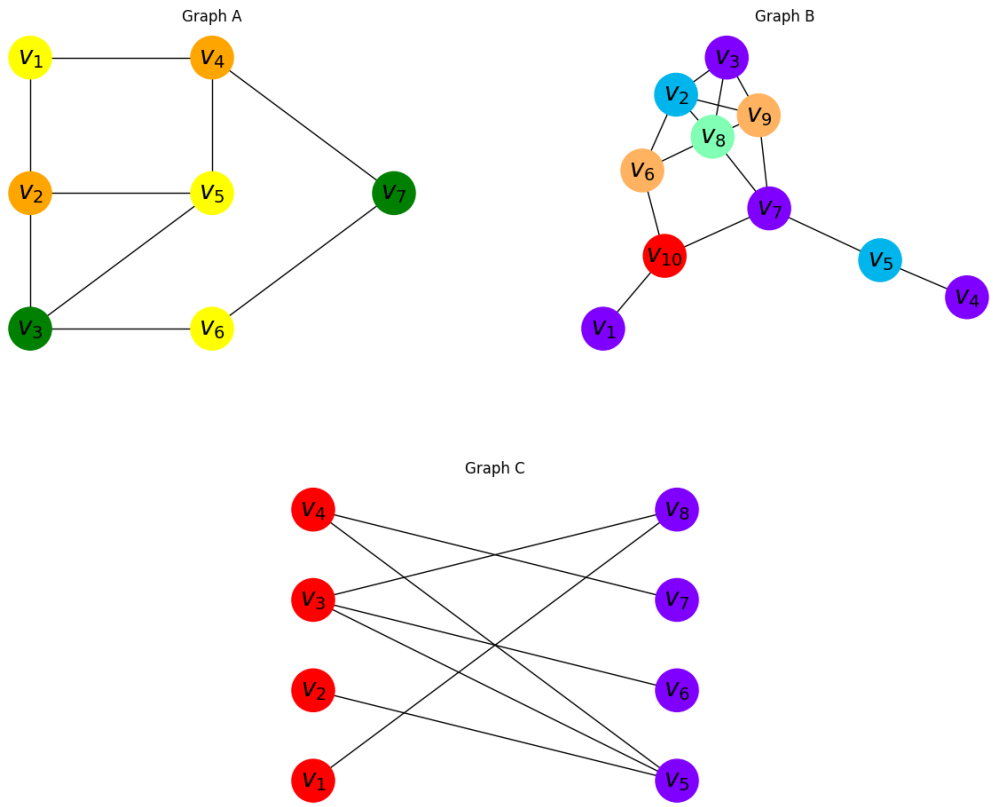


Figure 4.1: Plots of the d-1 colored graphs A, B and C using a Python wrapper and NetworkX.

The Python wrapper that generates the graph B is shown in Appendix A.4. For small graphs, it can be observed visually that the d-1 colorings are valid. In all three graphs, neighboring nodes are colored differently. Graph A corresponds to the example graph in

Figure 2.1 and has the same coloring as in chapter 3.2.1 in Figure 3.3. Graph B is another undirected graph with 10 nodes, which is colored with 5 different colors. Graph C is a bipartite graph with the two node sets v_1 - v_4 and v_5 - v_8 within which there are no edges. This graph is colored with 2 colors, i.e. the lowest possible number of colors.
The d-2 coloring is shown visually by using the graphs represented by the adjacency matrices A, B and C in Listing 4.9.

```
      0 0 0 0 0 0 0 1          0 0 0 0 0 0 1 0 0 1 1 0          0 0 0 0 1 0
      0 0 0 0 1 0 0 0          0 0 0 0 0 0 1 0 0 1 0 1          0 0 0 0 1 1
      0 0 0 0 1 1 0 1          0 0 0 0 0 0 0 1 1 0 0 0          0 0 0 0 0 1
A =   0 0 0 0 1 0 1 0    B =   0 0 0 0 0 0 1 0 1 0 0 1    C =   0 0 0 0 1 0
      0 1 1 1 0 0 0 0          0 0 0 0 0 0 1 1 0 1 1 0          1 1 0 1 0 0
      0 0 1 0 0 0 0 0          0 0 0 0 0 0 0 1 1 0 1 0          0 1 1 0 0 0
      0 0 0 1 0 0 0 0          1 1 0 1 1 0 0 0 0 0 0 0
      1 0 1 0 0 0 0 0          0 0 1 0 1 1 0 0 0 0 0 0
                               0 0 1 1 0 1 0 0 0 0 0 0
                               1 1 0 0 1 0 0 0 0 0 0 0
                               1 0 0 0 1 1 0 0 0 0 0 0
                               0 1 0 1 0 0 0 0 0 0 0 0
```

Listing 4.9: d-2 coloring input matrices for evaluation with NetworkX.

The plots of the graphs, which are represented by the adjacency matrices A, B and C are shown in Figure 4.2.

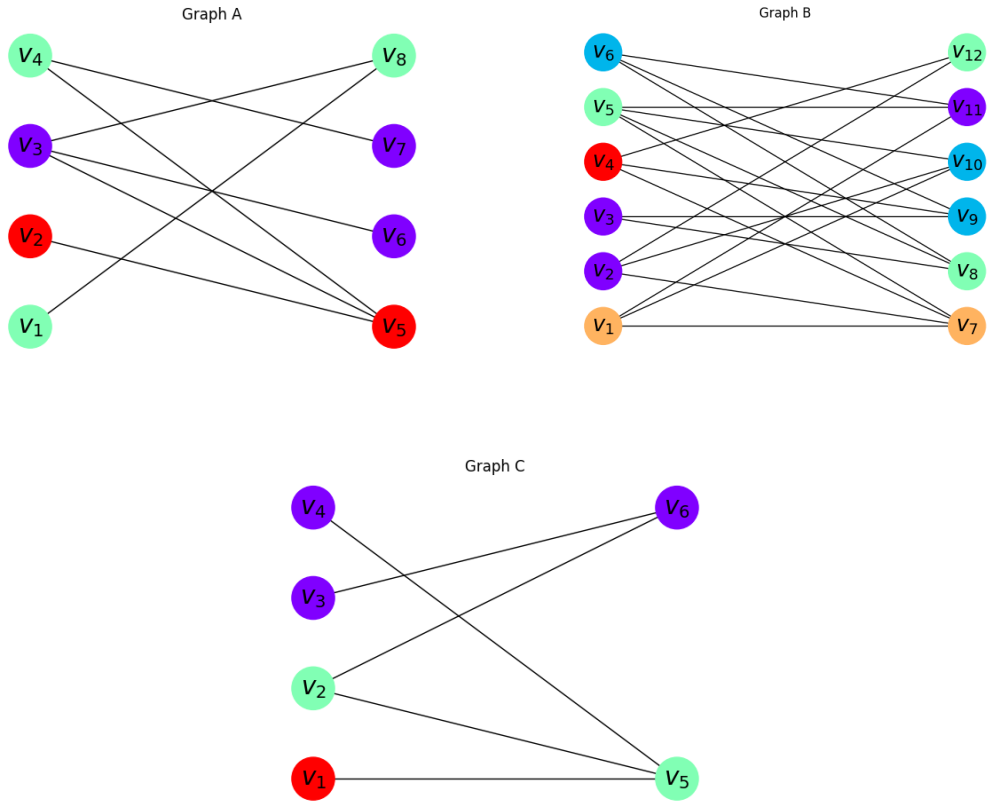


Figure 4.2: Plots of the d-2 colored graphs A and B using a Python wrapper and NetworkX.

The first graph A corresponds to the bipartite graph with 8 nodes from Figure 2.2. The d-2 coloring requires 3 colors here. Graph B has 12 nodes and is colored in 5 colors.
Graph C shows that the d-2 coloring C program can also be used to color bipartite graphs with two unequally sized sets of nodes.
The d-2 coloring C program allows the colors of both sets of nodes not to be disjoint. Often this convention is undesired, and its preferred to have a color number l for the left set of nodes and a color number r for the right set of nodes, which results in the number of colors $l + r$ for the entire graph.

4.3 Comparison of the Number of Colors for different Erdős-Rényi Graphs

The function $fast\_gnp\_random(n, p, seed = None, directed = False)$ of the Python package NetworkX is used to create Erdős-Rényi graphs. This function creates a graph with n nodes, in which each of the $[n(n-1)]/2$ possible edges is created with probability p.

A seed is used as an indicator for the state of the random number generation. In addition, graphs can be selected to be directed or undirected.
The functions implemented in chapters 3.2.2 and 3.3 for calculating the d-1 and d-2 coloring of a graph require input matrices in the form of a 2-dimensional array of the format shown in Listing 4.10. It is not the row or column dimensions that are relevant here, but the enclosure of the individual rows with curly brackets and the placement of the commas.

```
24          {0, 1, 0, 1, 0, 0, 0},
25          {1, 0, 1, 0, 1, 0, 0},
26          {0, 1, 0, 0, 1, 1, 0},
27          {1, 0, 0, 0, 1, 0, 1},
28          {0, 1, 1, 1, 0, 0, 0},
29          {0, 0, 1, 0, 0, 0, 1},
30          {0, 0, 0, 1, 0, 1, 0}
```

Listing 4.10: Format of the input adjacency matrix for d-1 and d-2 coloring.

To generate the Erdős-Rényi graph and save it as an adjacency matrix in the required format, the steps shown in Listing 4.11 are necessary. The formatted adjacency matrix is saved in a .txt file and is passed to the C program to calculate the d-1 coloring.

```
5  # Generate a Erdoes-Renyi graph
6  G = nx.fast_gnp_random_graph(10, 0.3, None, False)
7
8  # Save the adjacency matrix as a 2D array
9  adj_matrix = nx.adjacency_matrix(G)
10 adj_array = adj_matrix.toarray()
11
12 # Convert the adjacency matrix to the desired format
13 formatted_matrix = "{" + "},\n{".join([", ".join(map(str, row)) for row in
       adj_array]) + "}"
14
15 # Save the formatted adjacency matrix to a .txt file
16 with open("fast_gnp_random-10x10.txt", "w") as file:
17     file.write(formatted_matrix)
```

Listing 4.11: Erdős-Rényi graph generation and storage of the formatted adjacency matrix.

For the evaluation of the d-1 coloring, undirected graphs with 10, 20, 40 and 80 nodes are examined. This can be achieved by selecting the parameter n of the function $fast\_gnp\_random(n, p, seed = None, directed = False)$. The parameter p, which sets the probability for the creation of each possible edge, is kept constant at 0.3. The default value $None$ is selected as $seed$ and the fourth input parameter is set to $False$ to obtain undirected graphs. In order to receive the number of colors required for coloring, the C implementation from chapter 3.2.2 is extended by a function that calculates the maximum of a GraphBLAS vector. The lines of code to be added are shown in Listing 4.12.

```c
GrB_Monoid max_monoid;
GrB_Monoid_new(&max_monoid, GrB_MAX_INT32, (int32_t)0);

GrB_Vector_reduce_INT32(&max_value, NULL, max_monoid, C, NULL);
printf("Number of colors: %d\n", max_value);

GrB_Monoid_free(&max_monoid);
```

Listing 4.12: Function to calculate the maximum of a GraphBLAS vector.

To do this, the monoid *max_monoid* is created using the predefined binary operator GrB_MAX_INT32 and the identity element 0. Applied to the coloring vector C, the maximum value, which corresponds to the number of required colors, is stored in the variable *max_value* and then printed.
The number of colors required for the respective graphs are shown in Table 4.1. The different programs are executed several times and the lowest number of colors achieved is taken. The number of required colors doubles with each doubling of the number of nodes. In addition, the required runtime of the programs is measured in milliseconds (ms). The mean value of five runtime measurements for each program was calculated. No specific growth pattern can be derived from the measured times. In order to obtain a clear tendency towards growth behavior, more data points are required.

Graph	Number of nodes	Number of colors	Required time in ms
$fast\_gnp\_random$	10	3	0.179
$fast\_gnp\_random$	20	6	0.2658
$fast\_gnp\_random$	40	12	0.7734
$fast\_gnp\_random$	80	24	1.854

Table 4.1: Evaluation of the C implementation of the d-1 coloring algorithm using the $fast\_gnp\_random$ function from NetworkX for graph generation with $p = 0.3$.

To evaluate the C program of the d-2 coloring algorithm, adjacency matrices are required, which represent bipartite graphs. In general, these adjacency matrices have the block matrix form shown in Figure 4.3, where B is an arbitrary adjacency matrix and B^T is the transposed matrix of B. The remaining values of the matrix are 0s.

$$A = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$$

Figure 4.3: General block form of an adjacency matrix A of a bipartite graph.

The matrix B used for evaluation corresponds to a square adjacency matrix of a directed graph. The function $fast\_gnp\_random(n, p, seed = None, directed = True)$ to generate the matrix is used as before to evaluate the d-1 coloring, with the difference that the fourth input parameter is set to True to obtain a directed graph. The parameter p, which

sets the probability for the creation of each possible edge, is kept constant at 0.3 and the default value *None* is selected as seed. To create a 10×10 adjacency matrix of a bipartite graph, a 5×5 adjacency matrix B of an Erdős-Rényi graph is taken and set as the lower left block matrix. The transposed matrix B^T is set as the upper right block and the rest of the matrix is filled with 0s. The resulting graph is called *bip_fast_gnp_random*. In addition, the format of the resulting matrix is set to the appropriate form (see Listing 4.10). As only undirected graphs are colored, the matrix is checked for symmetry. When the matrix is symmetrical, the adjacency matrix represents an undirected graph. The number of colors required to color the respective graph is shown in Table 4.2. Compared to d-1 coloring, significantly more colors are required for the larger graphs with 40 and 80 nodes. The runtime is measured in milliseconds (ms) and is the mean value of five runtime measurements for each program. It is remarkable that the runtime when coloring graphs with 80 nodes shows a clear jump, which does not occur in this order of magnitude with d-1 coloring. The required time indicates that there may be polynomial growth. Again, more data points are needed to be able to determine a clear growth behavior.

Graph	Number of nodes	Number of colors	Required time in ms
bip_fast_gnp_random	10	3	0.2094
bip_fast_gnp_random	20	5	0.3254
bip_fast_gnp_random	40	16	0.6504
bip_fast_gnp_random	80	38	11.5374

Table 4.2: Evaluation of the C implementation of the d-2 coloring algorithm using the *fast_gnp_random* function from NetworkX for graph generation with $p = 0.3$.

Chapter 5

Conclusion

This work aims to become familiar with the GraphBLAS API and to implement programs not yet publicly available that solve the graph coloring problems of d-1 and d-2 coloring using the GraphBLAS API. Looking back at the content of this thesis, Chapter 2 explained the relation between graphs and linear algebra operations and discussed the theoretical foundations of the GraphBLAS API. Chapter 3 presented the implementation of the d-1 coloring C program, which colors arbitrary undirected graphs. The pseudocode presented by Muhammad Osama et al. in the paper *Graph Coloring on the GPU* [13] was chosen as the approach and the most important components of the implementation were discussed in Chapter 3.2.2. In addition, a C program for the d-2 coloring of bipartite graphs was implemented. The procedure for tracing back to d-1 coloring was theoretically demonstrated and the implementation was subsequently discussed. In Chapter 4, the implementations were evaluated in three different ways. First, the individual steps of the algorithms were traced using the custom functions for printing GraphBLAS objects. The results of d-1 and d-2 coloring of different graphs were presented. Another possibility for evaluation was the use of the Python package NetworkX for plotting graphs. Here, the d-1 and d-2 colorings were visually reproduced on small example graphs using Python wrappers, which use the *executable* files of the d-1 and d-2 coloring programs as input. Finally, using the Python package NetworkX and the function *fast_gnp_random_graph*, random graphs with 10, 20, 40 and 80 nodes were generated on which the implementations were tested. The number of calculated colors and the runtime were compared.

The implementations of the d-1 and d-2 coloring using the GraphBLAS API still offers scope for extensions and improvements. The inclusion of a function for calculating maximum independent node sets is a useful extension that could lead to better colorings, i.e. colorings with a smaller number of colors. In the current implementation, only an arbitrary independent node set is calculated in each iteration, but not the maximum inde-

pendent node set. An implementation of such a function is given as example code in the GraphBLAS C API [4, p. 289] and could be integrated into the existing implementation in a suitable form. This example code introduces the idea that the random values assigned to each node as weights are scaled by the inverse of the respective node degree. This reduces the probability that nodes with a lower node degree are selected and thus larger sets of nodes per iteration are selected for coloring. In transportation networks, biological networks or even communication networks, graphs with a much higher number of nodes occur. Therefore, it is a possible future task to measure the performance of the implementations on larger graphs and to compare them with other graph coloring algorithms. The programs are tested with up to 80×80 adjacency matrices, which corresponds to graphs with 80 nodes. In order to be able to use larger graphs, an extension of the previous implementation is conceivable, which can read in graphs in various formats, such as CSV or JSON. Up to now, it is only possible to enter the input adjacency matrix manually in the program code.

Bibliography

[1] *Ice-sheet and Sea-level System Model (ISSM).* `https://issm.jpl.nasa.gov`. (accessed: 09.02.2024).

[2] *LAGraph - library plus a test harness for collecting algorithms that use GraphBLAS.* `https://github.com/GraphBLAS/LAGraph`. (accessed: 01.03.2024).

[3] *SuiteSparse:GraphBLAS implementation of the GraphBLAS standard Github Repository.* `https://github.com/DrTimothyAldenDavis/GraphBLAS`. (accessed: 23.02.2024).

[4] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. Moreira. *The GraphBLAS C API Specification. GraphBLAS. org, Tech. Rep*, 2021.

[5] H. M. Bücker, M. Lülfesmann, and M. A. Rostami. *Enabling Implicit Time Integration for Compressible Flows by Partial Coloring: A Case Study of a Semi-matrix-free Preconditioning Technique*, pages 23–32. 2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, jan 2016.

[6] T. A. Davis. *Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. ACM Trans. Math. Softw.*, 45(4), dec 2019.

[7] T. A. Davis. *Algorithm 1037: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra. ACM Trans. Math. Softw.*, 49(3):1, sep 2023.

[8] D. Jungnickel. *Graphs, Networks and Algorithms*, volume 3. Springer, 2005.

[9] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, 2011.

[10] S. O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen.* Springer, 2009.

[11] M. Lülfesmann. *Full and partial Jacobian computation via graph coloring: Algorithms and applications.* Cuvillier Verlag, 2012.

[12] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. *Standards for Graph Algorithm Primitives.* In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2013.

[13] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens. *Graph coloring on the GPU.* In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 2019.

Appendix A

Appendix

A.1 C Program of the d-1 Coloring Algorithm

```c
// d1 coloring of the example graph from Figure 2.1

// takes a fixed matrix which can be set before compiling so d-1 coloring from any undirected graph is possible

// to compile and run use:
// gcc -o d-1-coloring d-1-coloring.c -I/path/to/graphblas/include -L/path/to/graphblas/lib -lgraphblas
// ./d-1-coloring

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <GraphBLAS.h>

#define MATRIX_SIZE 7

// Function to create and initialize a GraphBLAS matrix
GrB_Matrix createAdjacencyMatrix() {
    GrB_Matrix adjacency_matrix;
    GrB_Matrix_new(&adjacency_matrix, GrB_INT32, MATRIX_SIZE, MATRIX_SIZE);

    // Define the adjacency matrix data
    int32_t data[MATRIX_SIZE][MATRIX_SIZE] = {
        {0, 1, 0, 1, 0, 0, 0},
        {1, 0, 1, 0, 1, 0, 0},
        {0, 1, 0, 0, 1, 1, 0},
        {1, 0, 0, 0, 1, 0, 1},
        {0, 1, 1, 1, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 1},
```

```
        {0, 0, 0, 1, 0, 1, 0}
    };

    // Populate the adjacency matrix
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            GrB_Matrix_setElement_INT32(adjacency_matrix, data[i][j], i, j);
        }
    }
    return adjacency_matrix;
}

// Custom function to generate distinct random integer values
int generateDistinctRandom() {
    static bool initialized = false;
    if (!initialized) {
        srand(time(NULL));
        initialized = true;
    }

    static bool usedValues[100] = {false};  // Assuming a range of values
    from 1 to 100

    int random_value;
    do {
        random_value = rand() % 100 + 1;  // Adjust the range as needed
    } while (usedValues[random_value]);

    usedValues[random_value] = true;

    return random_value;
}

// Custom function to print a GraphBLAS scalar of type GrB_INT32
void printScalarInt32(GrB_Scalar scalar, const char* name) {
    int32_t value;
    GrB_Scalar_extractElement_INT32(&value, scalar);
    printf("%s: %d\n", name, value);
}

// Custom function to print a GraphBLAS vector
void printVector(GrB_Vector vector, const char* name) {
    GrB_Index size;
    GrB_Vector_size(&size, vector);

    printf("%s:\n[", name);

    for (GrB_Index i = 0; i < size; i++) {
        int32_t value;
```

```
78          GrB_Vector_extractElement_INT32(&value, vector, i);
79          printf("%d", value);
80
81          if (i < size - 1) {
82              printf(", ");
83          }
84      }
85      printf("]\n");
86  }
87
88  // Custom function to print a GraphBLAS matrix
89  void printMatrix(GrB_Matrix matrix) {
90      GrB_Index nrows, ncols;
91      GrB_Matrix_nrows(&nrows, matrix);
92      GrB_Matrix_ncols(&ncols, matrix);
93
94      for (GrB_Index i = 0; i < nrows; ++i) {
95          for (GrB_Index j = 0; j < ncols; ++j) {
96              int32_t value;
97              GrB_Matrix_extractElement_INT32(&value, matrix, i, j);
98              printf("%d ", value);
99          }
100         printf("\n");
101     }
102 }
103
104 int main() {
105     // Initialize GraphBLAS
106     GrB_init(GrB_BLOCKING);
107
108     // Create and print the adjacency matrix
109     GrB_Matrix adjacency_matrix = createAdjacencyMatrix();
110     printf("\nAdjacency matrix:\n");
111     printMatrix(adjacency_matrix);
112
113     // Create empty vectors, scalar and auxiliary variable
114     GrB_Vector C;
115     GrB_Vector weight;
116     GrB_Vector frontier;
117     GrB_Vector max;
118     GrB_Scalar succ;
119     GrB_Vector_new(&C, GrB_INT32, MATRIX_SIZE);
120     GrB_Vector_new(&weight, GrB_INT32, MATRIX_SIZE);
121     GrB_Vector_new(&frontier, GrB_INT32, MATRIX_SIZE);
122     GrB_Vector_new(&max, GrB_INT32, MATRIX_SIZE);
123     GrB_Scalar_new(&succ, GrB_INT32);
124     int32_t Intsucc;
125
126     // Print the three vectors
```

```
127     printf("\nVectors:\n");
128     printVector(C, "C");
129     printVector(weight, "weight");
130     printVector(frontier, "frontier");
131     printf("\n");
132 
133     // See GraphBLAS API p. 167 for right set of input variables
134     // Initialize colors to 0
135     GrB_assign(C, GrB_NULL, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE, GrB_NULL);
136     printVector(C, "Colors initialized to 0");
137 
138     // Assign distinct random integer as weight to each vertex
139     for (GrB_Index i = 0; i < MATRIX_SIZE; i++) {
140         int random_value = generateDistinctRandom();
141         GrB_Vector_setElement_INT32(weight, random_value, i);
142     }
143     printVector(weight, "weight with random values");
144 
145     for(GrB_Index color = 1; color < MATRIX_SIZE; color++) {
146       // Find maximum of neighbors
147       GrB_vxm(max, GrB_NULL, GrB_NULL, GrB_MAX_TIMES_SEMIRING_INT32, weight,
      adjacency_matrix, GrB_NULL);
148       printVector(max, "max");
149 
150       GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL, GrB_GT_INT32, weight, max,
     GrB_NULL);
151       printVector(frontier, "frontier after Greater Than operation");
152 
153       // Find number of nodes that will be colored
154       GrB_reduce(succ, GrB_NULL, GrB_PLUS_INT32, frontier, GrB_NULL);
155 
156       printScalarInt32(succ, "succ");
157 
158       // Gets integer value from GrB_Scalar Type succ to be able to check
     the following break condition
159       GrB_Scalar_extractElement_INT32(&Intsucc, succ);
160 
161       // Stop when frontier is empty
162       if (Intsucc == 0) {
163          break;
164       }
165 
166       // Assign new color
167       GrB_assign(C, frontier, GrB_NULL, color, GrB_ALL, MATRIX_SIZE,
     GrB_NULL);
168       printVector(C, "C after iteration");
169       printf("\n");
170 
171       // Get rid of colored nodes in candidate list
```

```
172         GrB_assign(weight, frontier, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE,
    GrB_NULL);
173         printVector(weight, "weight after iteration");
174     }
175
176     printVector(C, "\nd-1 coloring");
177
178     // Free resources
179     GrB_Matrix_free(&adjacency_matrix);
180     GrB_Vector_free(&C);
181     GrB_Vector_free(&weight);
182     GrB_Vector_free(&frontier);
183     GrB_Vector_free(&max);
184     GrB_Scalar_free(&succ);
185     GrB_finalize();
186
187     return 0;
188 }
```

Listing A.1: C program of the d-1 coloring algorithm.

A.2 C Program of the d-2 Coloring Algorithm

```
1  // d-2 coloring of the example graph from Figure 2.2
2
3  // to compile and run use:
4  // gcc -o d-2-coloring d-2-coloring.c -I/path/to/graphblas/include -L/path/
       to/graphblas/lib -lgraphblas
5  // ./d-2-coloring
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <time.h>
10 #include <stdbool.h>
11 #include <GraphBLAS.h>
12
13 #define MATRIX_SIZE 8
14
15 // Function to create and initialize a GraphBLAS matrix
16 GrB_Matrix createAdjacencyMatrix() {
17     GrB_Matrix adjacency_matrix;
18     GrB_Matrix_new(&adjacency_matrix, GrB_INT32, MATRIX_SIZE, MATRIX_SIZE);
19
20     // Define the adjacency matrix data
21     int32_t data[MATRIX_SIZE][MATRIX_SIZE] = {
22         {0, 0, 0, 0, 0, 0, 0, 1},
23         {0, 0, 0, 0, 1, 0, 0, 0},
24         {0, 0, 0, 0, 1, 1, 0, 1},
25         {0, 0, 0, 0, 1, 0, 1, 0},
```

```
        {0, 1, 1, 1, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0},
        {1, 0, 1, 0, 0, 0, 0, 0}
    };
    // Populate the adjacency matrix
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            GrB_Matrix_setElement_INT32(adjacency_matrix, data[i][j], i, j);
        }
    }
    return adjacency_matrix;
}

// Function to set diagonal entries of a GraphBLAS matrix to zero
void setDiagonalToZero(GrB_Matrix matrix) {
    GrB_Index nrows, ncols;
    GrB_Matrix_nrows(&nrows, matrix);
    GrB_Matrix_ncols(&ncols, matrix);

    for (GrB_Index i = 0; i < nrows; ++i) {
        GrB_Matrix_setElement_INT32(matrix, 0, i, i);
    }
}

// Custom function to print a GraphBLAS matrix
void printMatrix(GrB_Matrix matrix) {
    GrB_Index nrows, ncols;
    GrB_Matrix_nrows(&nrows, matrix);
    GrB_Matrix_ncols(&ncols, matrix);

    for (GrB_Index i = 0; i < nrows; ++i) {
        for (GrB_Index j = 0; j < ncols; ++j) {
            int32_t value;
            GrB_Matrix_extractElement_INT32(&value, matrix, i, j);
            printf("%d ", value);
        }
        printf("\n");
    }
}

// Function to manipulate the adjacency matrix and return the manipulated
    matrix
GrB_Matrix matrixManipulation(GrB_Matrix adjacency_matrix) {
    GrB_Matrix adjacency_matrix_manipulated;
    GrB_Matrix_new(&adjacency_matrix_manipulated, GrB_INT32, MATRIX_SIZE,
    MATRIX_SIZE);

```

```
72      // Perform matrix multiplication: adjacency_matrix_manipulated =
        adjacency_matrix * adjacency_matrix
73      GrB_mxm(adjacency_matrix_manipulated, GrB_NULL, GrB_NULL,
        GxB_PLUS_TIMES_INT32, adjacency_matrix, adjacency_matrix, GrB_NULL);
74      printf("\nAdjacency matrix square:\n");
75      printMatrix(adjacency_matrix_manipulated);
76
77      // Set diagonal entries to zero
78      setDiagonalToZero(adjacency_matrix_manipulated);
79
80      // Display the matrix with diagonal entries set to zero
81      printf("\nMatrix with diagonal entries set to zero:\n");
82      printMatrix(adjacency_matrix_manipulated);
83
84      // Apply the condition: set entries greater than 1 to 1
85      GrB_apply(adjacency_matrix_manipulated, GrB_NULL, GrB_NULL,
        GrB_MIN_INT32, 1, adjacency_matrix_manipulated, GrB_NULL);
86
87      // Display the modified matrix
88      printf("\nElements greater 1 set to 1:\n");
89      printMatrix(adjacency_matrix_manipulated);
90
91      // Return the manipulated matrix
92      return adjacency_matrix_manipulated;
93  }
94
95  // Custom function to generate distinct random integer values
96  int generateDistinctRandom() {
97      static bool initialized = false;
98      if (!initialized) {
99          srand(time(NULL));
100         initialized = true;
101     }
102     static bool usedValues[100] = {false};  // Assuming a range of values
        from 1 to 100
103     int random_value;
104     do {
105         random_value = rand() % 100 + 1;  // adjust the range as needed
106     } while (usedValues[random_value]);
107     usedValues[random_value] = true;
108     return random_value;
109 }
110
111 // Custom function to print a GraphBLAS scalar of type GrB_INT32
112 void printScalarInt32(GrB_Scalar scalar, const char* name) {
113     int32_t value;
114     GrB_Scalar_extractElement_INT32(&value, scalar);
115     printf("%s: %d\n", name, value);
116 }
```

```
117
118 // Custom function to print a GraphBLAS vector
119 void printVector(GrB_Vector vector, const char* name) {
120     GrB_Index size;
121     GrB_Vector_size(&size, vector);
122     printf("%s:\n[", name);
123
124     for (GrB_Index i = 0; i < size; i++) {
125         int32_t value;
126         GrB_Vector_extractElement_INT32(&value, vector, i);
127         printf("%d", value);
128         if (i < size - 1) {
129             printf(", ");
130         }
131     }
132     printf("]\n");
133 }
134
135 int main() {
136     // Initialize GraphBLAS
137     GrB_init(GrB_BLOCKING);
138
139     // Create and print the adjacency matrix
140     GrB_Matrix adjacency_matrix = createAdjacencyMatrix();
141     printf("\nAdjacency matrix:\n");
142     printMatrix(adjacency_matrix);
143
144     // Call the matrix manipulation function and get the manipulated matrix
145     GrB_Matrix manipulated_matrix = matrixManipulation(adjacency_matrix);
146
147     // Display the manipulated matrix
148     printf("\nAdjacency matrix used for d1 coloring:\n");
149     printMatrix(manipulated_matrix);
150
151     // Create empty vectors, scalar and auxiliary variable
152     GrB_Vector C;
153     GrB_Vector weight;
154     GrB_Vector frontier;
155     GrB_Vector max;
156     GrB_Scalar succ;
157     GrB_Vector_new(&C, GrB_INT32, MATRIX_SIZE);
158     GrB_Vector_new(&weight, GrB_INT32, MATRIX_SIZE);
159     GrB_Vector_new(&frontier, GrB_INT32, MATRIX_SIZE);
160     GrB_Vector_new(&max, GrB_INT32, MATRIX_SIZE);
161     GrB_Scalar_new(&succ, GrB_INT32);
162     int32_t Intsucc;
163
164     // Print the three vectors
165     printf("\nVectors:\n");
```

```
166     printVector(C, "C");
167     printVector(weight, "weight");
168     printVector(frontier, "frontier");
169     printf("\n");
170 
171     // see GraphBLAS API for right set of input variables for GraphBLAS
    functions
172     // Initialize colors to 0
173     GrB_assign(C, GrB_NULL, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE, GrB_NULL);
174     printVector(C, "Colors initialized to 0");
175 
176     // Assign distinct random integer as weight to each vertex
177     for (GrB_Index i = 0; i < MATRIX_SIZE; i++) {
178         int random_value = generateDistinctRandom();
179         GrB_Vector_setElement_INT32(weight, random_value, i);
180     }
181     printVector(weight, "weight with random values");
182 
183     for( GrB_Index color = 1; color < MATRIX_SIZE; color++) {
184       // find maximum of neighbors
185       GrB_vxm(max, GrB_NULL, GrB_NULL, GrB_MAX_TIMES_SEMIRING_INT32, weight,
     manipulated_matrix, GrB_NULL);
186       printVector(max, "max");
187 
188       GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL, GrB_GT_INT32, weight, max,
    GrB_NULL);
189       printVector(frontier, "frontier after Greater Than operation");
190 
191       // Find all largest nodes that are uncolored
192       GrB_reduce(succ, GrB_NULL, GrB_PLUS_INT32, frontier, GrB_NULL);
193 
194       printScalarInt32(succ, "succ");
195 
196       // gets integer value from GrB_Scalar Type succ to be able to check
    the following break condition
197       GrB_Scalar_extractElement_INT32(&Intsucc, succ);
198 
199       // stop when frontier is empty
200       if (Intsucc == 0) {
201          break;
202       }
203 
204       // Assign new color
205       GrB_assign(C, frontier, GrB_NULL, color, GrB_ALL, MATRIX_SIZE,
    GrB_NULL);
206       printVector(C, "C after iteration");
207       printf("\n");
208 
209       // Get rid of colored nodes in candidate list
```

```
210         GrB_assign(weight, frontier, GrB_NULL, 0, GrB_ALL, MATRIX_SIZE,
        GrB_NULL);
211         printVector(weight, "weight after iteration");
212       }
213
214       printVector(C, "\nd-2 coloring");
215
216       // Free resources
217       GrB_Matrix_free(&adjacency_matrix);
218       GrB_Matrix_free(&manipulated_matrix);
219       GrB_Vector_free(&C);
220       GrB_Vector_free(&weight);
221       GrB_Vector_free(&frontier);
222       GrB_Vector_free(&max);
223       GrB_Scalar_free(&succ);
224       GrB_finalize();
225
226       return 0;
227 }
```

Listing A.2: C program of the d-2 coloring algorithm.

A.3 Output of the d-2 Coloring C Program

```
1  Adjacency matrix:
2  0 0 0 0 0 0 0 1
3  0 0 0 0 1 0 0 0
4  0 0 0 0 1 1 0 1
5  0 0 0 0 1 0 1 0
6  0 1 1 1 0 0 0 0
7  0 0 1 0 0 0 0 0
8  0 0 0 1 0 0 0 0
9  1 0 1 0 0 0 0 0
10
11 Adjacency matrix square:
12 1 0 1 0 0 0 0 0
13 0 1 1 1 0 0 0 0
14 1 1 3 1 0 0 0 0
15 0 1 1 2 0 0 0 0
16 0 0 0 0 3 1 1 1
17 0 0 0 0 1 1 0 1
18 0 0 0 0 1 0 1 0
19 0 0 0 0 1 1 0 2
20
21 Matrix with diagonal entries set to zero:
22 0 0 1 0 0 0 0 0
23 0 0 1 1 0 0 0 0
24 1 1 0 1 0 0 0 0
25 0 1 1 0 0 0 0 0
```

```
0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0

Elements greater 1 set to 1:
0 0 1 0 0 0 0 0
0 0 1 1 0 0 0 0
1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0

Adjacency matrix used for d1 coloring:
0 0 1 0 0 0 0 0
0 0 1 1 0 0 0 0
1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0

Vectors:
C:
[22011, 22011, 22011, 22011, 22011, 22011, 22011, 22011]
weight:
[22011, 22011, 22011, 22011, 22011, 22011, 22011, 22011]
frontier:
[22011, 22011, 22011, 22011, 22011, 22011, 22011, 22011]

Colors initialized to 0:
[0, 0, 0, 0, 0, 0, 0, 0]
weight with random values:
[39, 21, 38, 77, 3, 32, 19, 82]
max:
[38, 77, 77, 38, 82, 82, 3, 32]
frontier after Greater Than operation:
[1, 0, 0, 1, 0, 0, 1, 1]
succ: 4
C after iteration:
[1, 0, 0, 1, 0, 0, 1, 1]

weight after iteration:
[0, 21, 38, 0, 3, 32, 0, 0]
max:
[38, 38, 21, 38, 32, 3, 3, 32]
```

```
frontier after Greater Than operation:
[0, 0, 1, 0, 0, 1, 0, 0]
succ: 2
C after iteration:
[1, 0, 2, 1, 0, 2, 1, 1]

weight after iteration:
[0, 21, 0, 0, 3, 0, 0, 0]
max:
[0, 0, 21, 21, 0, 3, 3, 3]
frontier after Greater Than operation:
[0, 1, 0, 0, 1, 0, 0, 0]
succ: 2
C after iteration:
[1, 3, 2, 1, 3, 2, 1, 1]

weight after iteration:
[0, 0, 0, 0, 0, 0, 0, 0]
max:
[0, 0, 0, 0, 0, 0, 0, 0]
frontier after Greater Than operation:
[0, 0, 0, 0, 0, 0, 0, 0]
succ: 0

d1-coloring:
[1, 3, 2, 1, 3, 2, 1, 1]
```

Listing A.3: Output of the d-2 coloring C program.

A.4 Python Program for plotting Graph B using NetworkX

```python
import subprocess
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

def run_c_program():
    # Run the compiled C program and capture its output
    result = subprocess.run(["./graphblas_example_v5"], capture_output=True, text=True)
    return result.stdout

def is_symmetric(matrix):
    return (matrix == matrix.T).all()

def parse_output(output):
    # Split the output into lines
    lines = output.strip().split('\n')

```

```
    # Parse the adjacency matrix, skipping empty lines
    matrix_lines = [list(map(int, line.split())) for line in lines[:-1] if
    line.strip()]

    # Check if the adjacency matrix is symmetric
    if is_symmetric(np.array(matrix_lines)):
        print("Adjacency Matrix is symmetric.")
    else:
        print("Adjacency Matrix is not symmetric.")

    # Print the adjacency matrix for debugging
    print("Adjacency Matrix:")
    for row in matrix_lines:
        print(row)

    # Parse the coloring vector and convert it to a 1D NumPy array
    coloring_line = lines[-1]
    coloring = np.ravel(np.array(list(map(int, coloring_line.split()))))

    # Print the shape of the coloring array for debugging
    print("Coloring Vector:")
    for color in coloring:
        print(color)

    return np.array(matrix_lines), coloring


def plot_colored_graph(matrix, coloring):
    # Create a graph from the adjacency matrix
    G = nx.Graph(np.array(matrix))

    # Create a mapping from node indices to node names
    node_names = {i: f'$v_{{{i+1}}}$' for i in range(len(G.nodes()))}

    # Create a colormap for node colors
    cmap = plt.cm.rainbow

    # Create a figure and axis for the plot
    fig, ax = plt.subplots()

    # Plot the graph with colored nodes
    nx.draw(G, labels=node_names, node_size=1200,
            node_color=coloring, cmap=cmap, font_color='black', font_size
    =20, ax=ax)

    plt.title("Graph B")

    # Display the plot
    plt.show()
```

```
65
66 if __name__ == "__main__":
67     # Run the compiled C program and capture the output
68     c_program_output = run_c_program()
69
70     # Parse the output to get the adjacency matrix and coloring vector
71     adjacency_matrix, coloring = parse_output(c_program_output)
72
73     # Plot the graph with coloring using NetworkX
74     plot_colored_graph(adjacency_matrix, coloring)
```

Listing A.4: Python program for plotting graph B using NetworkX.

A.5 Python Program for generating Erdős-Rényi Graphs

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  # Generate a Erdoes-Renyi graph
6  G = nx.fast_gnp_random_graph(10, 0.3, None, False)
7
8  # Save the adjacency matrix as a 2D array
9  adj_matrix = nx.adjacency_matrix(G)
10 adj_array = adj_matrix.toarray()
11
12 # Convert the adjacency matrix to the desired format
13 formatted_matrix = "{" + "},\n{".join([", ".join(map(str, row)) for row in
       adj_array]) + "}"
14
15 # Save the formatted adjacency matrix to a .txt file
16 with open("fast_gnp_random-10x10.txt", "w") as file:
17     file.write(formatted_matrix)
18
19 # Draw the graph
20 nx.draw(G)
21 plt.show()
```

Listing A.5: Python Program for generating Erdős-Rényi graphs.

A.6 Additions to the Declaration of Academic Integrity

For this master thesis the following type of an allowed AI tool was used:
Translations generated by AI tools: AI-generated translations may be copied directly.
In agreement with my supervisor Prof. Bücker, I am allowed to translate individual text passages from German into English in the period from October 2023 to April 2024 with the help of DeepL.com, an online translation service, and not to specify them individually.

Declaration of Academic Integrity

I hereby confirm that this work — or in case of group work, the contribution for which I am responsible and which I have clearly identified as such — is my own work and that I have not used any sources or resources other than those referenced.
I take responsibility for the quality of this text and its content and have ensured that all information and arguments provided are substantiated with or supported by appropriate academic sources. I have clearly identified and fully referenced any material such as text passages, thoughts, concepts or graphics that I have directly or indirectly copied from the work of others or my own previous work. Except where stated otherwise by reference or acknowledgement, the work presented is my own in terms of copyright.

I understand that this declaration also applies to generative AI tools which cannot be cited (hereinafter referred to as 'generative AI').
I understand that the use of generative AI is not permitted unless the examiner has explicitly authorized its use (Declaration of Permitted Resources). Where the use of generative AI was permitted, I confirm that I have only used it as a resource and that this work is largely my own original work. I take full responsibility for any AI-generated content I included in my work.
Where the use of generative AI was permitted to compose this work, I have acknowledged its use in a separate appendix. This appendix includes information about which AI tool was used or a detailed description of how it was used in accordance with the requirements specified in the examiner's Declaration of Permitted Resources.
I have read and understood the requirements contained therein and any use of generative AI in this work has been acknowledged accordingly (e.g. type, purpose and scope as well as specific instructions on how to acknowledge its use).

I also confirm that this work has not been previously submitted in an identical or similar form to any other examination authority in Germany or abroad, and that it has not been previously published in German or any other language.

I am aware that any failure to observe the aforementioned points may lead to the imposition of penalties in accordance with the relevant examination regulations. In particular, this may include that my work will be classified as deception and marked as failed. Repeated or severe attempts to deceive may also lead to a temporary or permanent exclusion from further assessments in my degree programme.

Jena, 02.04.2024