UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Vložitev ne-ravninskih grafov: Hramba in predstavitev

(Embedding non-planar graphs: Storage and Representation)

Ime in priimek: Đorđe Klisura
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Matjaž Krnc
Somentor: dr. Katja Berčič

Koper, september 2020

Ključna dokumentacijska informacija

Ime in PRIIMEK: Đorđe KLISURA

Naslov zaključne naloge: Vložitev ne-ravninskih grafov: Hramba in predstavitev

Kraj: Koper

Leto: 2020

Število listov: 27	Število slik: 9	Število tabel: 2
Število prilog: 2	Število strani prilog: 2	Število referenc: 40

Mentor: doc. dr. Matjaž Krnc

Somentor: dr. Katja Berčič

Ključne besede: ravninski graf, prikaz grafa, prehodna številka, baza podatkov grafov

Math. Subj. Class. (2010): 05C10, 05C62, 05C85, 68P20

Izvleček:
V zaključni nalogi predlagamo način, kako kanonično predstaviti neplanarne grafe, skupaj z ravninsko vložitvijo, ki premore najmanj sekajočih se povezav. To dosežemo z uporabo najsodobnejših orodij, kot so kanonično označevanje grafov, Nautyjev niz Graph6 in kombinatorno predstavitvijo ravninskih grafov. Kolikor nam je znano, tega do sedaj še nihče ni naredil. Poleg tega omenjeni postopek implementiramo v jeziku SageMath in izračunamo vložitve v ravnino za nekatere razrede kubičnih grafov. Glavni prispevek naše naloge je (i) razširitev ene izmed zbirk grafov, ki se nahaja na MathDataHub-u, in (ii) razširitev izvorne kode orodja SageMath.

Key document information

Name and SURNAME: Đorđe KLISURA

Title of final project paper: Embedding non-planar graphs: Storage and Representation

Place: Koper

Year: 2020

Number of pages: 27 Number of figures: 9 Number of tables: 2

Number of appendices: 2 Number of appendix pages: 2

Number of references: 40

Mentor: Assist. Prof. Matjaž Krnc, PhD

Co-Mentor: Katja Berčič, PhD

Keywords: planar graph, graph representation, crossing number, graph database

Math. Subj. Class. (2010): 05C10, 05C62, 05C85, 68P20

Abstract: In the final project paper, we propose a convention to canonically represent non-planar graphs together with their least-crossing embedding. We achieve this by using state-of-the-art tools such as canonical labelling of graphs, Nauty's Graph6 string and combinatorial representation for planar graphs. To the best of our knowledge, this has not been done before. Besides, we implement the mentioned procedure in a SageMath language and compute embeddings for certain classes of cubic graphs. Our main contribution is (i) an extension of one of the graph data sets hosted on MathDataHub, and (ii) towards extending the SageMath codebase.

Acknowledgements

First, I would like to express my very great appreciation to my mentor and professor Matjaž Krnc for his guidance and suggestions during the planning, development and writing of this final project paper.

I would also like to thank my co-mentor Katja Berčič for given advice and support during the writing of this thesis.

I would like to thank Ivan Smirnov for good discussion and help [34].

I want to thank my friends Aleksandar Avdalović, Đorđe Mitrović and Mirza Krbezlija, for their selfless help during studies and the strong influence on my development as an academic.

And last but not least, I want to thank my family, in particular my parents, for supporting me in all conceivable ways from when I was young, up to now.

List of Contents

List of Tables

List of Figures

List of Appendices

List of Abbreviations

i.e.	that is
e.g.	for example
etc	et cetera
s.t.	such that
iff	if and only if

1 Introduction

1.1 Motivation of mathematical databases

Mathematicians increasingly use computers to support their research. This includes most aspects of a researcher's work, from publishing and reading papers to computations in mathematical software. Perhaps surprisingly, mathematicians also generate and use data, and in some areas of mathematics, the production and manipulation of large datasets is becoming increasingly important.

The main uses for these mathematical datasets and databases are of an exploratory nature. Researchers use them to test hypotheses, or to find patterns and counterexamples. It is not too hard to find such "datasets" that even predate computers: the Atlas of Graphs and the Foster census are two such examples from graph theory. The Online Encyclopedia of Integer Sequences (OEIS) is perhaps the most successful modern mathematical database, which represents an online database of integers. Today, the OEIS contains more than 334000 sequences, which are of use to both professional mathematicians and amateurs and is the largest database of its kind. The sequences in the database act as fingerprints for their associated records. A somewhat similar project in graph theory is the House of Graphs.

An important notion is that of using mathematical objects, such as integer sequences, or graphs, to search for mathematical theorems. This has been introduced as theorem fingerprinting by Billey and Tenner [4] as a way to make searching for mathematical knowledge more efficient. Fingerprints, in the more general sense of the word, are used in many areas of science, from computer science to chemistry, archaeology and genetics. Some examples include computer documentation, reducing duplication in web search results, and DNA fingerprints.

1.2 Storing graphs in the database

In this thesis, we focus on storing graphs (embeddings) in databases. The two most commonly used data structures for representing graphs are adjacency lists and two-dimensional boolean adjacency matrices.

An *adjacency list* is implemented as an array containing linked list for each of the source vertices. Each of these list contains all of the destination vertices to which the source vertex, that the list represents, is connected to by an edge. *Two-dimensional boolen adjacency matrices*, as the name suggests, are matrices which encode adjacency of the vertices in the graph via their boolean entries. The rows of these matrices represent source vertices, while the columns are the destination vertices. In particular, the value of the entry is indicative of the existence of an edge between the vertices the entry is indexed by. It is 1 if such an edge exists and 0 otherwise. For further investigation, we suggest Cormen, Leiserson, Rivest and Stain [9].

Another way of implementing graphs in the database is by using *flags*. For each vertex, we need to store an element, its visited flag, its list of edges, and a link to the next vertex. Then, for each edge, we need a link to the vertex it connects to and a link to the next edge. Finally, we need to define what information is needed to keep track of the whole graph, which is a pointer to the initial vertex.

On graph databases: It is worth noting here that this thesis does not deal with graph databases, but databases or datasets which contain graphs as entries. A graph database (GDB), on the other hand, uses the structure of graphs to represent and store data. Such databases are often used for storing heavily interconnected data. Relationships between data items are represented as edges between nodes. This does not only allow for linking the items directly, but often enables one to retrieve data with a single operation. For more about graph databases, refer to Bourbakis [5], Byoung-Ha, Seon-Kju and Seon-Young [19] and Renzo and Claudio [29]. We next discuss the concept of graph labeling crucial to such databases.

1.3 Canonical labeling

Both in mathematics and computer science, the term canonical form (or normal, standard form) of an object refers to a standard way of presenting this object mathematically. The desirable properties for such a form to posses are simplicity, so the expression is easier to manipulate with, and uniqueness, needed for proper identification of the object. In the case of graphs, canonical form is called canonical labeling and the problem of finding it is referred to as graph canonization or graph canonicalization. For computing canonical forms of graphs we use Bliss algorithm by Junttila and Kaski [17]. For more regarding this topic, refer to Vikraman, Bireswar and Kobler [38], [36], Valut [37], and Babai and Luks [22]. Nauty and Traces is a tool that was written by B. McKay and which is used by a lot of commerical and non-commerical mathematical tools, including SageMath, where we can directly generate graphs based on his code. For further reading, refer to McKay and Piperno [23].

1.4 Graph embeddings

In the broad sense, graph embeddings are understood as transformations of graphs into a vector or a set of vectors, which serve as their representations. The key idea of an embedding is to offer a simplified, less abstract description of a graph, which still captures its crucial properties, as well as the properties of its subgraphs, edges and vertices. In general, the more successful an embedding is in preserving information, the more useful it will be.

Most embeddings fall into one of the two following categories: vertex embeddings and graph embeddings.

By vertex embeddings we mean a vector representation of each of the graph's vertices. Such embeddings are usually used when one is interested in analyzing or making use of the local structure of a graph, in particular at the level of a vertex. One of the examples where these embeddings occur naturally is the problem of visualization of a graph and its vertices as objects on the 2D plane.

On the other hand, graph embeddings refer to vector representations of a graph as a whole. This approach is useful for visualizations, analysis and making predictions on the global level of the whole graph. An interesting example of their application is the study of chemical structures. For more about graph embeddings, we refer the reader to Goyal and Ferrara [13].

When discussing embeddings in graph theory, a particularly important family of graphs is formed by the planar graphs. A planar graph is a graph which can be embedded in the plane. Less formally, a graph is planar if and only if it is possible to draw it on the plane in such a way that no two edges intersect, except possibly at their common endpoints. A particular embedding (a drawing) is called a plane graph or a planar embedding of a graph. For more about planar graphs and embeddings, look at Trudeau and Richard [35] and Barthelemy [3].

The idea of combinatorial embeddings first appeared in the work of Heffter(1891), but has only been properly addressed and formalized later by Edmonds(1960) and Youngs(1963). We will discuss them in more detail in the second chapter. Their importance arises from, but is not limited to, the fact that together with canonical labeling, combinatorial embeddings can be used to obtain a unique representation of (planar) embeddings for (planar) graphs.

1.5 Crossing numbers

The crossing number of a graph is defined to be lowest number of edge crossings of a plane drawing of the graph. This means that there cannot exists a drawing of a graph on the plane with fewer edge crossings than the graph's crossing number. From this definition it is easy to see that a graph is planar if and only if its crossing number is zero. Pál Turán initiated the study of crossing numbers in his brick factory problem, where he asked for a factory plan that minimized the number of crossings between tracks connecting brick kilns to storage sites. Scientific studies have shown that graphs drawings with fewer crossing are more easily understood by individuals. For this and many other reasons, determining the crossing number remains a problem of both interest and importance. More regarding this topic, refer to Turan [27] and Purchase, Cohen and Murray [7].

The *genus* of a graph is the minimal integer n such that the graph can be drawn without crossing itself on a sphere with n handles (i.e. an oriented surface of genus n). Thus, a planar graph has genus 0, because it can be drawn on a sphere without self-crossing. For more, refer to Mohar and Thomassen [25].

A more general concept compared to a planar embedding is the *book embedding*. A book embedding of a graph is an embedding into a book, which is a collection of half-planes all having the same line as their boundary. Intuitively, we can understand the situation as follows: given a graph G and a book of k pages, we look at embeddings that map the vertices of G into the common line along the spine of the book, the edges of G onto the pages in such a way that each edge is contained by one page and no three edges cross in one point. We are interested in finding the embedding which minimizes the sum of crossings on all pages and we call this minimum the book embedding number of a graph G. For more about book embeddings and the book crossing number, refer to a book by Ernst and Mayr [39] and to Persinger [1].

Thus, the focus of the thesis will be mainly oriented towards the canonical representation of embeddings which realizes the crossing number of certain graphs.

2 Preliminaries

We start by describing basic notions of graph theory in section 2.1, where we cover the definition of graphs, drawing of graphs, crossing number and planar graphs together with planarity testing. In section 2.2 we talk about algorithms, we define asymptotic notations $\mathcal{O}$, Ω and Θ and we say a few words about P and NP-hard problems. Then we describe the time and space complexity of an algorithm and how to calculate it. At the end of section 2.2. we talk about the programming environment we used. We proceed in section 2.3 with describing some useful notions related to representation theory, such as JSON data formats, the idea of combinatorial embedding and canonical forms.

2.1 Graph Theory

A graph G is an ordered pair $G = (V, E)$ comprising:

- V a set of vertices (also called nodes or points)
- $E \subseteq \{\{x, y\} | (x, y) \in V^2 \wedge x \neq y\}$ a set of edges (also called links or lines), which are unordered pairs of vertices (i.e., an edge is associated with two distinct vertices).

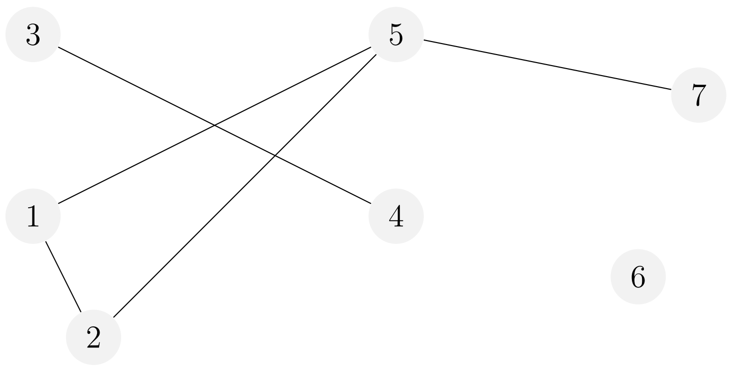


Figure 1: Graph on 7 vertices with edge set E= $\{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 4\}, \{5, 7\}\}$.

To avoid ambiguity, this type of object may be called precisely an undirected simple graph. For further reading, refer to Anderson [2].

A graph with vertex set V is said to be a graph on V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$. The number of vertices of a graph G is its *order*, written as $|G|$ and the number of edges is denoted by $||G||$. Graphs are *finite* and *infinite* according to their order. Unless otherwise stated, the graphs we consider are all finite. *Empty graph* is denoted by $\emptyset$. A graph of order 0 or 1 is called *trivial.* For further reading on the topic of graph theory check out the book by Diestel [10].

As presented in Figure 1 usual way to picture a graph is by drawing a dot for each vertex and joining two of these dots by a line if the corresponding two vertices form an edge. Just how these dots and lines are drawn is considered irrelevant: all that matters is the information which pairs of vertices form an edge and which do not. For more about graph drawings, refer to Diestel [10].

If xy is an edge, then we say that x and y are *adjacent* or that y is *neigbour* of x and denote this by $x \sim y$. A vertex is *incident* with an edge if it is one of the two vertices of the edge. If all the vertices of G are pairwise adjacent, then G is complete. A complete graph on n vertices is a K^n. Pairwise non-adjacent vertices or edges are called independent. More formally, a set of vertices or of edges is independent (or stable) if no two of its elements are adjacent. For further reading, refer to Diestel [10] and Godsil and Royle [12].

Two graphs X and Y are equal if and only if they have the same vertex set and the same edge set. This motivates the following:

Definition 2.1. Two graphs X and Y are isomorphic if there is a bijection φ from $V(X)$ to $V(Y)$ such that $x \sim y$ in X if and only if $\varphi(x) \sim \varphi(y)$ in Y. We say that φ is an *isomorphism* from X to Y.

Since φ is a bijection, it has an inverse, which is an isomorphism from Y to X. If X and Y are isomorphic, then we write $X \cong Y$. It is normally appropriate to treat isomorphic graphs as if they were equal. For more about isomorphisms, refer to a book by Godsil and Royle [12].

2.1.1 Crossing number

The crossing number, $cr(G)$, of a graph G is the minimum number of pairs of crossing edges in a depiction of G. Obviously, planar graphs have crossing number 0. For more about crossing number of a graph, including the theorem below, refer to Schaefer [31].

Theorem 2.2. *For any graph G on n vertices and m edges, we have*

$$cr(G) \geq c\frac{m^3}{n^2}$$

for $m \geq 4n$ and some constant c.

2.1.2 Planar graphs

When a connected graph can be drawn without any edges crossing, it is called planar. Even though sometimes the graph does not look like planar, it still might be if we can redraw it in such a way that we do not have any edge crossings. E.g. the graph in Figure 2a is a planar graph because we can redraw it like in Figure 2b. Since both graphs are the same, if one is planar, the other must be too.

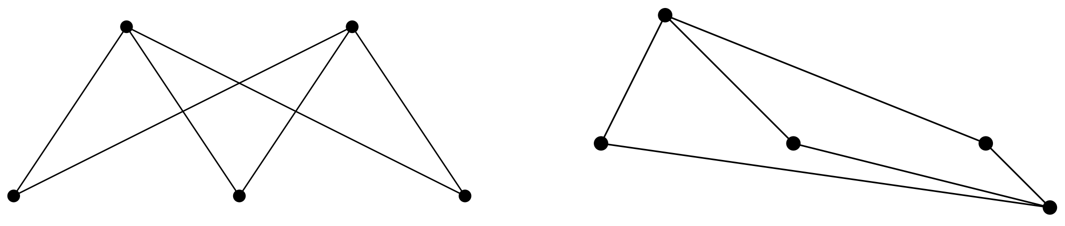

(a) Graph G. (b) Planar drawing of G.

Figure 2: Graph G drawn in two ways.

When a planar graph is drawn without edges crossing, the edges and vertices of the graph divide the plane into regions, called faces. The graph presented in Figure 2b has three faces (we include 'outside' region as a face as well). The number of faces does not change no matter how we draw the graph (as long as we do so without edge crossings), so it makes sense to ascribe the number of faces as a property of planar graphs.
A connection between the number of vertices (v), the number of edges (e) and the number of faces (f) in any connected planar graph is called Euler's formula.

Theorem 2.3. *For a connected planar simple graph $G = (V, E)$ with $e = |E|$ and $v = |V|$, if we let f be the number of faces (regions) that are created when drawing a planar representation of the graph, then $v - e + f = 2$.*

Another important theorem is Kuratowski's theorem that represents a mathematical forbidden graph characterization of planar graphs. For more reading refer to Kuratowski [18].

Theorem 2.4. *A finite graph G is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or of $K_{3,3}$*

A subdivision of a graph is a graph formed by subdividing its edges into paths of one or more edges. Kuratowski's theorem states that a finite graph G is planar, if it is not possible to subdivide the edges of K_5 or $K_{3,3}$ and then possibly add additional edges and vertices, to form a graph isomorphic to G. Equivalently, a finite graph is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 or

$K_{3,3}$. Some of the indirect application of Kuratowski's theorem is: if an algorithm can find a copy of K_5 or $K_{3,3}$ within a given graph, the input graph is for sure not planar and we don't do any additional computation. For more reading on this topic, refer to a book by Levin [21].

Planarity testing: The planarity testing problem is the algorithmic problem of testing whether a given graph is a planar graph. It represents a very complex and well-studied problem in computer science for which many practical algorithms have developed. Most of these methods operate in $\mathcal{O}(n)$ time, where n is the number of edges (or vertices) in the graph, which is asymptotically optimal. More about time complexity we talk in Section 2.2. The output of a planarity testing algorithm may be an embedding of a planar graph if the graph is planar, or an obstruction to planarity such as a Kuratowski subgraph if it is not. For more reading on this topic, refer to Hopcroft and Tarjan [14].

2.2 Algorithms

An algorithm represents a finite series of unambiguous, computer-implementable instructions to solve a specific set of estimable problems. An algorithm can be expressed within a limited amount of space and time, and in a well-defined formal language for calculating a function. In particular, algorithms are created to simplify the process of finding a solution to a set of problems, and also to help us in the process of thinking and understanding the problem. For more reading on this topic, refer to Knuth [11]. We proceed by defining asymptotic notations and algorithm space and time complexity.

Big $\mathcal{O}$ notation: Big $\mathcal{O}$ notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. We give a formal definition: Let f be a real or complex valued function and g a real valued function. Let both functions be defined on some unbounded subset of the real positive numbers, and $g(x)$ be strictly positive for all large enough values of x. We write

$$f(x) = \mathcal{O}(g(x))$$

as $x \to \infty$.

If the absolute value of $f(x)$ is at most a positive constant multiple of $g(x)$ for all sufficiently large value of x. That is, $f(x) = \mathcal{O}(g(x))$ if there exists a positive real number M and a real number x_0 such that

$$|f(x)| \leq Mg(x)$$

for all $x \geq x_0$.

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that

$$f(x) = \mathcal{O}(g(x))$$

.

Big Omega notation: Another asymptotic notation is Ω, read "big Omega", defined as

$$f(x) = \Omega g(x)(x \to a)$$

where a is some real number, ∞, or - ∞, where f and g are real functions defined in a neighbourhood of a, and where g is positive in this neighbourhood.

Big Theta notation: Another asymptotic noation is Θ read "Big Theta", defined as:

$$f(n) = \Theta(g(n))$$

f is bounded both above and below by g asymptotically.

P and NP-hardness: We now say few word about hardness of an algorithm. *class P* or just P represents a general class of questions for which some algorithm can provide an answer in polynomial time. We do not always have an answer to some problems or there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it is possible to verify the answer quickly. The class of questions for which an answer can be verified in polynomial time is called NP, which stands for *non-deterministic polynomial time.*

For more reading on these topics, refer to Kleinberg and Tardos [15].

2.2.1 Time complexity

An imporant property of every algorithm is its time complexity, which refers to the amount of time needed to perform all of the steps of the algorithm and successfully execute it. In general, estimates for time complexity are made by counting the number of elementary operations performed by the algorithm, under the assumption that they all take a fixed amount of time. Hence, this estimate differs by at most a constant from actual time complexity.

However, it is important to note that the running time of the algorithm may depend on the size of the input, since it can directly influence the number of operations that need to be performed. Due to this state of affairs, it is common to consider the worst case time complexity.

Another, less common, approach is to consider the average time complexity for inputs of a fixed size, which there are, of course, only finitely many of.

Either way, time complexity can be understood as a function of the size of the input. Because it is difficult to compute exactly and one usually only cares about larger inputs, it often suffices to consider the behaviour of the complexity as the size of the input increases. This behaviour is called asymptotic behaviour of the complexity.

Therefore, the time complexity is commonly expressed using big $\mathcal{O}$ notation, and is typically $\mathcal{O}(n)$, $\mathcal{O}(nlog(n))$, $\mathcal{O}(n^{\alpha})$, $\mathcal{O}(2^{n})$ etc., where n is the input size in units of bits needed to represent the input. Algorithmic complexities are classified according to the type of function appearing in the big $\mathcal{O}$ notation. For example, an algorithm with time complexity $\mathcal{O}(n)$ is a linear-time algorithm, an algorithm with time complexity $\mathcal{O}(n^{\alpha})$ for some constant $\alpha > 1$ is a polynomial-time algorithm and an algorithm with time complexity $\mathcal{O}(2^{n})$ is an exponential-time algorithm. For more regarding this topic, refer to Sipser [24].

2.2.2 Space complexity

The space complexity of an algorithm is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm to execute a program and produce output.

Similar to time complexity, space complexity is often expressed asymptotically in big $\mathcal{O}$ notation, such as $\mathcal{O}(n)$, $\mathcal{O}(nlog(n))$, $\mathcal{O}(n^{\alpha})$, $\mathcal{O}(2^{n})$ etc., where n is a characteristic of the input influencing space complexity.

While executing, the algorithm uses memory space for three reasons:

1. **Instruction Space:** representing the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack:** at the times, it may happen that inside an algorithm another algorithm is called. In such a situation, the second algorithm must terminate before the original one's execution can be continued. This however requires for the variables of the first algorithm to be temporarily suspended. These variables are pushed onto the system stack, where they remain until after the second algorithm is executed.

While calculating the space complexity of any algorithm, we usually consider only data space while neglecting the instruction space and environmental stack.

Calculating the Space Complexity: In order to calculate space complexity, we need to know the value of memory used by a different type of datatype variables, which depends on different operating systems, but the calculating method remains the same. In Table 1 we present value of memory different type of datatype variables are using. For more reading on this topic, refer to [32] and to Sanjeev and Boaz [30].

Table 1: Value of memory used by different type of datatype variables.

Type	**Size**
bool, char, unsigned char, signed char, _int8	1 byte
_int16, short, unsigned short, wchar_t, _wchar_t	2 bytes
float, _int32, int, unsigned int, long, unsigned long	4 bytes
double, _int64, long double, long long	8 bytes

2.2.3 Programming environment

We chose SageMath as the programming environment because it is open-source, free software that has many features covering many aspects of mathematics, including graph theory, for which we are particularly interested. SageMath uses a syntax matching Python's, supporting procedural, functional and object-oriented constructs. All algorithms we created are written in SageMath. For more regarding SageMath, refer Stein [40].

2.3 Representation of graphs

2.3.1 Combinatorial embedding

An embedded graph uniquely defines cyclic orders[1] of edges incident to the same vertex. The rotation system is set of all cyclic orders. Embedding with the same rotation system are considered to be equivalent and combinatorial embedding represents corresponding equivalence class of embedding. Sometimes, the rotation system is called a combinatorial embedding. For further reading on this topic, refer to Mutzel and Weiskircher [28], Didjev [26] and to Duncan, Goodrich and Kobourov [8].

Idea of combinatorial embedding: Suppose we start with an embedding of a graph in the plane. For each vertex of the graph, we walk around vertex counterclockwise and we encounter edges in some order, ending up in the same place we started.

[1]a cyclic order is a way to arrange a set of objects in a circle

The embedding thus defines a cyclic permutation (called a rotation) of the edges incident to the vertex. More about combinatorial embedding and representation, refer to Klen and Mozes [20].

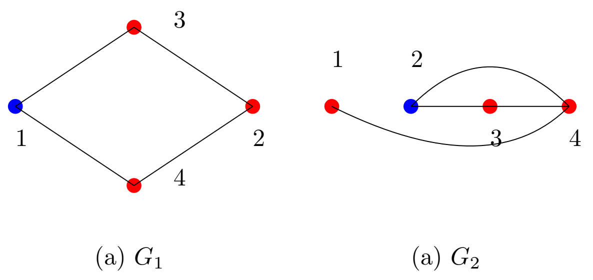
2.3.2 Canonical form

For computing canonical forms of graphs we use Bliss algorithm by Junttila and Kaski [17]. We briefly explain what does it do.
A colored graph is a triple $G = (V, E, c)$ where $V = 1, 2, ..., N$ is a finite set of vertices, E is a set of 2- element subsets of V and $c : V \to N$ is a function that associates to each vertex a nonnegative integer (a color). For example, identifying blue with 0 and red with 1, the graphs in Figure 3 are

$$G_1 = (\ \{1,2,3,4\}, \{\{1,3\},\{1,4\},\{2,3\},\{2,4\}\}, \langle 1 \to 0, 2 \to 1, 3 \to 1, 4 \to 1\rangle)$$

and

$$G_2 = (\ \{1,2,3,4\}, \{\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}, \langle 1 \to 1, 2 \to 0, 3 \to 1, 4 \to 1\rangle)$$



(a) G_1 (a) G_2

Figure 3: Two colored graphs.

Let $\gamma : V \to V$ be a permutation of V. Denote by v^γ the image of $v \in V$ under γ. For a colored graph $G = (V, E, c)$, define the colored graph

$$G^\gamma = (\ V, \{\{v^\gamma, w^\gamma\} | \{v, w\} \in E\}, c^\gamma)\ ,$$

where $c^\gamma : V \to N$ is defined for all $v \in V$ by $c^\gamma(\ v^\gamma) \ = c(v)$.
A canonical representative map p is defined to be a function between colored graphs satisfying the following:

- $p(G) \cong G$, and its representative are isomorphic

- if G_1 and G_2 are colored graphs, then $p(G_1) = p(G_2)$ iff $G_1 \cong G_2$. This in particular means that representatives can be shared only by isomorphic graphs.

A canonical labeling of G (under p) is an isomorphism of G onto its canonical representative $p(G)$. Given a colored graph G as input, the software tool bliss computes the canonical representative $p = (G)$ and an associated canonical labeling for it.

2.3.3 Data interchange formats

The representations are still mathematical and abstract, so to manage/manipulate graphs via computers more easily, we need data interchange format - JSON. JSON is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and array data types. It has a minimal number of value types: strings, numbers, booleans, lists, objects, and null. Although the notation is a subset of JavaScript, these types are represented in all common programming languages, making JSON a suitable candidate to transmit data across language gaps. For more reading on this topic, refer to [16].

3 Embedding non-planar graphs

In this section, we present our algorithms and their applications. We first explain why do we choose SageMath as our working environment. In the subsection 3.1 we explain first how we do manually procedure and later in the subsection 3.2 we explain how we automate it and how we get a compact file with necessary information. In the sub-sub sections "Analysis", we analyze our algorithms, thus their time and space complexities. In the end, in subsection 3.3 we present applications of our algorithms.

Environment: We use SageMath that enables us to use graph operations without implementing them on our own. SageMath has a nice implementation of the object graph and related operations that we use as well, including graph embedding, deletion and addition of edges, a genus of the graph, edge subdivision etc.

3.1 Initial approach

We first investigate small graphs on up to 6 vertices. Since we are interested in non-planar graphs, we just look at these. By looking at them, we try to resolve crossing. We try to redraw a graph such that we get a minimum number of crossings. Then we add another vertex or vertices (depends on the crossing number) at the point of intersection s. t. we get a planar graph. After getting such a graph, we get its embedding and we iterate.

We explain manually procedure at concrete example presented in the Listing 3.1.

```
gen=graphs.nauty_geng("-c 6")
test=True
while test:
    G=gen.next()
    if G.crossing_number()!=0:
        break
    G = gen.next()
G.crossing_number()
G.canonical_label()
G.show()
G.subdivide_edge(1,3,1)
G.delete_edge(2,4)
```

```
13 G.add_edge(2,5)
14 G.add_edge(4,5)
15 G.genus()
16 G.get_embedding()
```

Listing 3.1: Testing for crossing number

At the very beginning we do the whole procedure, that we will later automate, manually to better understand the flow of the algorithm. To have a clearer perception, the explanation follows the lines from the Listing 3.1.

Line 1: We generate all graphs on n number of vertices, in our example we set $n = 5$.

Line 2-7: We set the flag *test* to be True. We generate graphs from the family of connected graphs on five vertices, then we use the flag in while loop until the condition is met. Since our goal is to work with non-planar graphs, we generate such a graph with the condition of crossing number. If the crossing number of a given graph is not equal to 0 (in other words, if the graph has crossing number ≥ 1), we stop and set the flag to be false, we get that graph, and get out of the loop.

Line 8-10: We check the crossing number of a given graph, canonically relabel vertices and we look at the picture of a graph to see where is the crossing point. We get the picture presented in Figure 4 on the left.

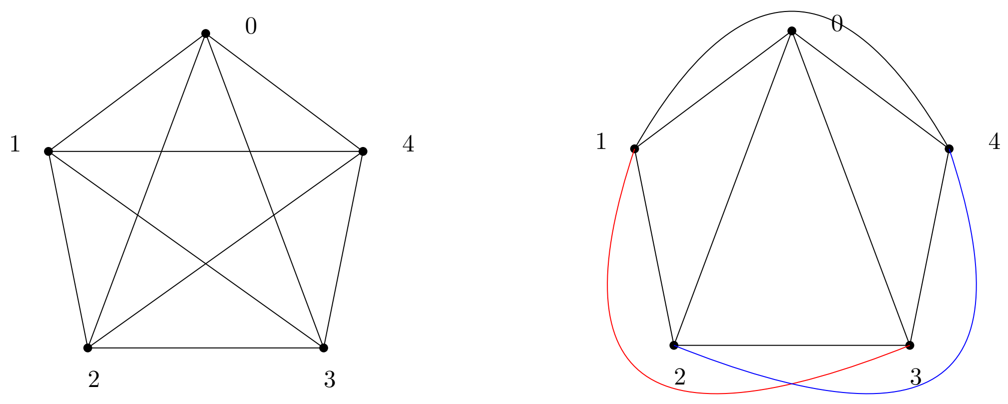


Figure 4: Original graph G on the left and redrawn graph G on the right with crossing edges colored in red and blue.

Line 11-14: First we redraw a graph manually until we get one crossing as in Figure 4 on the right. We see the point of intersection of the two edges $\{1, 3\}$ and $\{2, 4\}$ that needs to be solved. We introduce another point as follows: we subdivide edge $\{1, 3\}$, then we delete the second edge $\{2, 4\}$ and we connect the newly added point (vertex 5 obtained from the subdivision, since we canonically relabel vertices we know that 5 is our next vertex) to both endpoints of the second edge. In this way, we get a new graph with planar embedding presented in Figure 5.

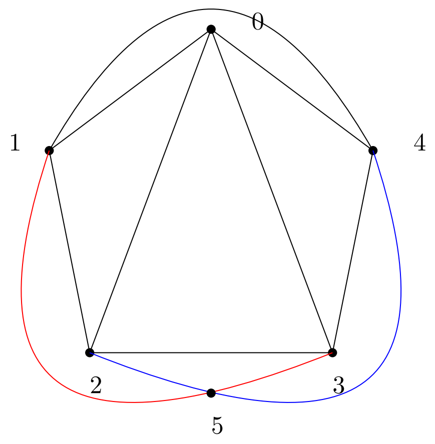


Figure 5: Planar embedding of the graph G.

Line 15-16: We check genus of the graph (if we succeeded in getting planar graph), and in the end, we get the embedding of such a graph G if its genus is 0.

3.2 Automating the procedure

We proceed with explaining the Algorithm 1.

We take an algorithm for computing crossing number in SageMath and we modify it to get what we want. We explain the algorithm on a concrete example. We have a look at the Petersen graph presented in Figure 6.

The first step of our algorithm is constructing all pairs of non-incident edges of G, meaning that the two different edges cannot share the same vertex. In our example from Figure 6 those pairs of edges are $\{\{0,1\},\{2,3\}\}$, $\{\{4,9\},\{5,8\}\}$ etc. In the beginning, we set crossing number k to be 0, to check if the graph is already planar, if it is we return k if it is not, we increment k to be 1 and we are going through the set of pairs of non-incident edges and for each k we modify graph until we get planar embedding, in the following way: we take the first pair, in our example pair $\{\{0,1\},\{2,3\}\}$ and we delete edges $\{0,1\}$ and $\{2,3\}$. Then we add new vertex v to which we connect vertices of the deleted edges: $\{0\},\{1\},\{2\},\{3\}$.

We test for planarity, if the graph is planar, return its crossing number and we are finished. If the result is not planar in any of these pairs, try $k+1$ pairs.

We can clearly see that our graph is planar after three iterations since the crossing number of the Peterson graph is 2, so we can reorient vertices s.t. we get a planar embedding that is presented in Figure 6.

Algorithm 1: Determining the crossing number.

```
Input: Non-planar graph $G$
Output: Planar embedding, crossing number and added vertices of $G$
1  $V \leftarrow V(G)$
2  $edgeParis \leftarrow \{ab, cd$, where $a, b, c, d \in V(G)$ and $ab, cd \in E(G)\}$
3  $k \leftarrow 0$
4  while $G$ is not planar do
5      $S \leftarrow$ all $k$-subsets from $edgePairs$
6      for $\{a_1, a_2, \ldots, a_k\}$ in $S$ do
7          $E(G) \leftarrow E(G) \setminus \bigcup_{i=1}^{k} a_i$
8          for $a_i$ element in $\{a_1, a_2, \ldots, a_k\}$ do
9              $\{\{a_i^1, a_i^2\}, \{a_i^3, a_i^4\}\} \leftarrow a_i$
10             $V(G) \leftarrow V(G) \cup \{v_i\}$ where $v_i \neq u \ \forall u \in V(G)$
11             $E(G) \leftarrow E(G) \cup \{a_i^1 v_i,\ v_i a_i^2,\ a_i^3 v_i,\ v_i a_i^4\}$
12         if $G$ is planar then
13             return $G, k, V(G) \setminus V$
14     $k \leftarrow k + 1$
```

Speeding up Algorithm 1: In order to speed up our Algorithm 1, in the beginning, we delete all the vertices with degree 2 and we connect their neighbours, to simplify the path. Topologically speaking, it is the same path. In the example presented in Figure 7 we have an example of speeding up our Algorithm. You can find the code of the Algorithm 1 in Appendix A.

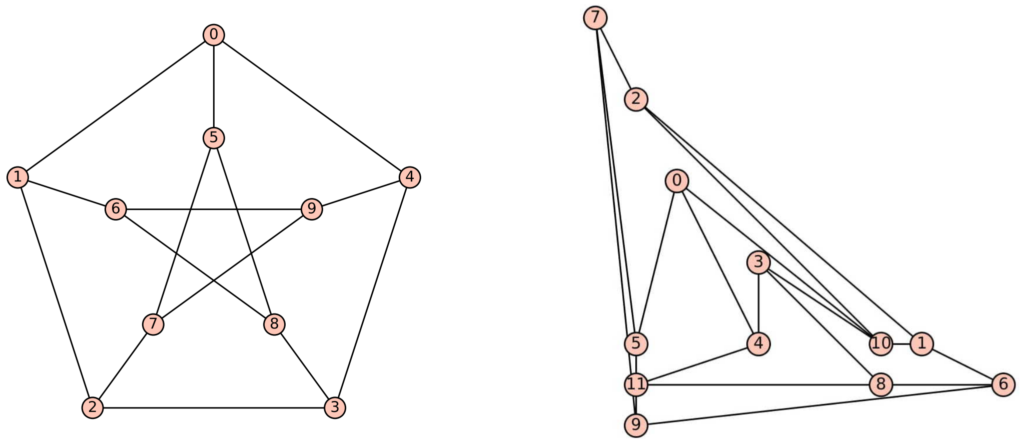


Figure 6: Left image is our original Petersen graph and the right image represents its planar embedding after using our Algorithm 1.

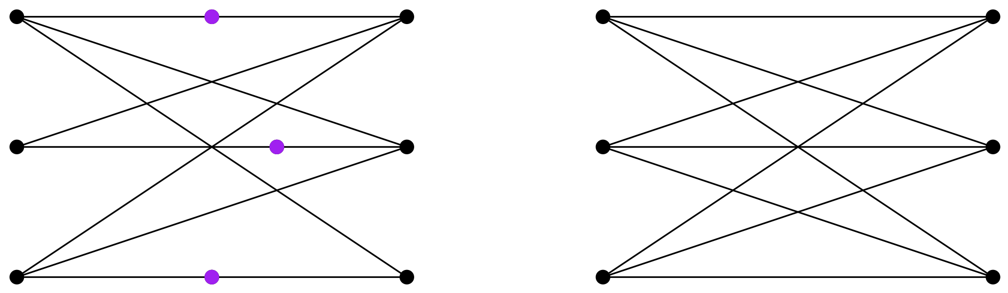


Figure 7: On the left is original graph and on the right is the same graph with deleted vertices of degree 2 that are colored in purple - we get $K_{3,3}$.

Analysis

We start by explaining the space complexity of the Algorithm 1. Amount of memory used by the Algorithm 1 to execute and produce the result is linear, because most of the work on the graph is done in-place, by modifying graph locally and not taking more space even after many graph manipulations. We can conclude that the Algorithm 1 does not take too much memory.

We proceed by explaining the time complexity of the Algorithm 1. In order to determine the time complexity, we need to consider all of the SageMath integrated functions we called in our main function. Function `is_planar` is implemented in linear time complexity, in the terms of graph meaning $\mathcal{O}(n+m)$ where n is number of vertices and m is number of edges, or just $\mathcal{O}(m)$. For more reading on the time complexity of the planarity algorithm, refer to Boyer and Myrvold [6]. To remove a vertex in a graph, we first need to find the vertex in the data structure and the time complexity depends on the structure we use; if we use a HashMap it will be $\mathcal{O}(1)$. Then we remove it in $\mathcal{O}(V)$ time. Adding and removing an edge operation is $\mathcal{O}(1)$. Adding vertex in a graph is $\mathcal{O}(n)$. Checking if there is an edge between vertices is $\mathcal{O}(V)$ since a node can have at most $\mathcal{O}(V)$ neighbours. The time complexity of getting an embedding of the graph and of finding the neighbours is linear since we need to perform the Breadth First Search algorithm. In the end, running time of the function *copy* is $\mathcal{O}(n)$, for more look at [33].

We start by analyzing the lines from Algorithm 1, to achieve overall time complexity. Line 1 takes as $\mathcal{O}(n)$ time. Line 2 $edgePairs$ is of size m^2. Now we discussing the size of the executions inside while loop. We ask ourselves what is the size of the execution of line 6. The size is $\binom{edges}{k}$. Since the size of edges is m^2 we get $\binom{m^2}{k}$ that is m^{2k}. On line 7, we increase set by k elements. Line 8 repeats k times, for each element. Lines 9:11 take us constant time $\mathcal{O}(1)$. And on line 12, we have a linear-time algorithm, so $\mathcal{O}(m)$. When we sum up altogether, we get overall time complexity: $\mathcal{O}(m^{2k}(k+m))$.

Writing in a file: We now explain Algorithm 2 presented in the Listing 3.2.

```
1  def embedding_(G,certificate=False):
2      with open("embedding-file.txt", "a") as f:
3          s = G.graph6_string()
4          k,G1,added = _crossing_number(G, certificate = True)
5          G2 = G1.canonical_label()
6          if G2.is_planar(set_embedding = True):
7              d = G2.get_embedding()
8              l = sorted(d)
9              f.write(s)
10             f.write(";" + str(added))
11             f.write(";" + "{")
12             i, n=0, len(l)
13             for elm in l:
14                 d[elm].sort()
15                 if i != n-1:
16                     f.write(str(elm) + ':' + str(d[elm]) + ',')
17                 else:
18                     f.write(str(elm) + ':' + str(d[elm]) + '}'+ '\n')
19                 i+=1
```

Listing 3.2: Writing in a file

Since we want to store graphs in the database, we create a function that stores data about an individual Graph in a single text file, understandable to a computer (for the database). Besides, we add certificate flag so that we have an output for people, with more explanation when set to be *True*).

Line 1-2 We define our function and we open a new text file where we put all the further details, we set 'a' as *append*, to append all graphs into one single file. Then we get the Graph6 representation of the graph as an ASCII string and we write it to a file.

Line 4-5 From the previous Algorithm 1 we get crossing number (k), planar embedding ($G1$) and the added vertices (*added*). Then we again canonically relabel vertices.

Line 6-8: Since we need to write the planar embedding of the non-planar graph G we get the embedding from the Graph that was previously rearranged and modified in the Algorithm 1 and we sort the elements.

Line 9-19: If user set flag *certificate* to True we write graph6 string, added vertices and embedding of G in more details where we explain what is the meaning of the output. Else, if the flag *certificate* is set to False, we output all information in a single line separated with the column to facilitate storage in the database. In Appendix B you can find the other function, with more details.

Analysis

We start by explaining the space complexity of the Algorithm presented in the Listing 3.2. We can observe that our algorithm does not take too much space. Space complexity is linear, the same reason holds: most of the work on the graph is done in-place, by modifying graph locally and not taking more space even after many graph manipulations.

Now we discuss time complexity. All of the functions used in Algorithm presented in Listing 3.2 are super fast except the following four:

`is_planar()`: implemented in linear time complexity, in the terms of graph meaning $\mathcal{O}(n+m)$ where n is number of vertices and m is number of edges, refer to Boyer and Myrvold [6].

`_crossing_number()`: takes us $\mathcal{O}(m^{2k}(k+m))$ time complexity as we concluded earlier in our discussion of time complexity of the Algorithm 1.

`graph6_string()`: linear time, because we have encoding of edge lists.

`canonical_label()`: exponential time, so $\mathcal{O}(2^n)$, refer to [34].

3.3 Applications of our approach

Drawing: One of our applications is coloring graphs. In the Algorithm 1, we labeled our newly added edges, then we use method `plot()` within the SageMath and with the property color by label we get different colors for our newly added edges. In Figure 8 we can see an example of the transformed Petersen graph from Figure 6. As you can see, the original embedding of a graph is colored in red while green and blue color represent the newly added edges.

We calculated embeddings, saved them together with added vertices and Graph6 strings in files and plotted images (by now) of vertex-transitive graphs on less than 20 edges. In Figure 9 you can see them separately with the Graph6 string of each of them written in the caption.

Storing graphs: Another application of our approach is related to storing of graphs with their combinatorial embedding, added vertices (if they are non-planar) and with the Graph6 string. We defined two algorithms for writing graph details into the file. By now, we processed cubic graphs up to 21 edges, vertex-transitive graphs up to 20 edges and all graphs up to 13 edges. Our files can be used to store into any database

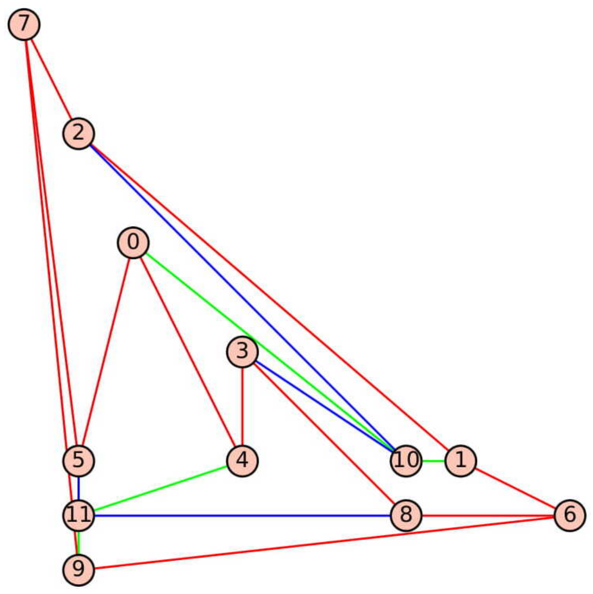


Figure 8: Coloring of planar embedding of Petersen graph.

since we created a non-verbose mode of writing into them. In Table 2 we presented overall of our work so far.

Table 2: Processed families of graphs and their files

family of graphs	**graphs generated**	**up to edges**	**size of a file**
cubic	752	21	188.6 KB
vertex-transitive	16	20	1.5 KB
general	376899	13	42.2 MB

Now in the Listing 3.3 present an example of a file format in non-verbose mode, suitable for the database.

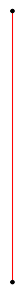
```
:Ea@_Q_QM@Gs;{6: {(0, 1), (2, 3)}, 7: {(1, 5), (3, 4)}, 8: {(2, 5),
   (0, 4)}};{0:[5, 6, 7, 8],1:[3, 4, 7, 8],2:[3, 4, 5, 6],3:[1, 2, 4,
    6, 8],4:[1, 2, 3, 5, 7],5:[0, 2, 4, 6, 7],6:[0, 2, 3, 5, 8],7:[0,
    1, 4, 5, 8],8:[0, 1, 3, 6, 7]}
```

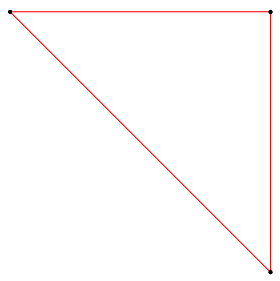
Listing 3.3: Non-verbose file format of a non-planar graph that consists of Graph6 string as well as of added vertices and graph combinatorial embedding.

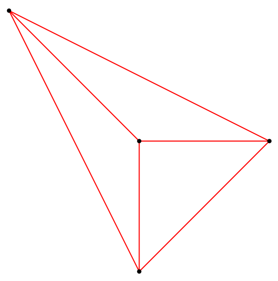
We now in the Listing 3.4 present an example of a file format in verbose mode, more suitable for human-reading. Here we can see that we have an example of a graph from the Listing 3.3. Here we explain more what is what and we write everything nicer, with more spaces and in a more intuitive way.

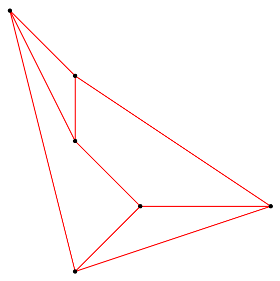
```
The Graph6 string: ':Ea@_Q_QM@Gs'

The added vertices are: {6: {(0, 1), (2, 3)}, 7: {(1, 5), (3, 4)}, 8:
    {(2, 5), (0, 4)}}

The planar embedding of G: {
0:[5, 6, 7, 8],
1:[3, 4, 7, 8],
2:[3, 4, 5, 6],
3:[1, 2, 4, 6, 8],
4:[1, 2, 3, 5, 7],
5:[0, 2, 4, 6, 7],
6:[0, 2, 3, 5, 8],
7:[0, 1, 4, 5, 8],
8:[0, 1, 3, 6, 7]}
```

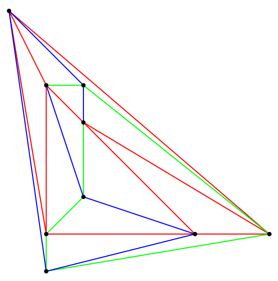
Listing 3.4: Verbose file format of a non-planar graph from the Listing 3.3

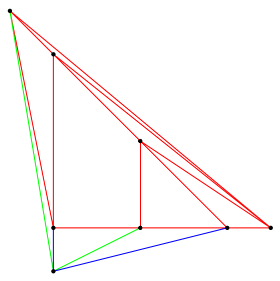

(a) `:An`

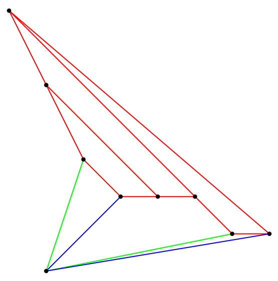

(b) `:DaHg~`

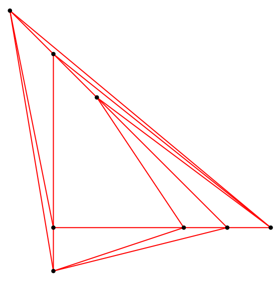

(c) `:CcKI`

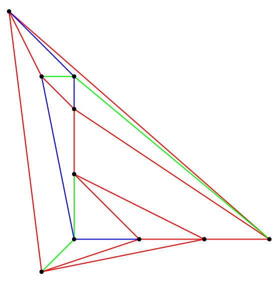

(d) `:Ea@aRgs`


(e) `:Ea@_Q_QM@Gs`


(f) `:Fa@_WIRQbP^`


(g) `:GaGecctgs`


(h) `:Ga@_WIRhDlDZ`


(i) `:Ga@_QaShDlDZ`

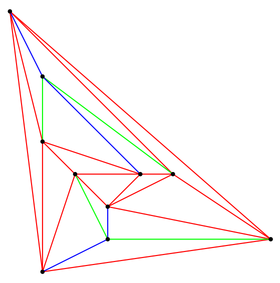

(j) `:Ga@_WGwChLDgsTn`

Figure 9: Colored images of some vertex-transitive graphs on up to 20 edges.

4 Conclusion

In this final project paper, we studied embedding of non-planar graphs, their storage and representation. We proposed algorithms for computing embedding of non-planar graphs and storing them in a file suitable for human-reading and the databases. Since there was no standard way of representing such embeddings, this contributes to the field of representation theory. In a collaboration with both mentors Matjaž Krnc, PhD and Katja Berčič, PhD we are in process of writing a research paper about this, and we plan to disseminate these ideas in the corresponding conferences.

We presented applications of our approach, that is drawing of graphs and writing graph information in file formats. Our algorithms can be used to enrich more or less any graph database and we plan to do that. So far we generated vertex-transitive graphs till 20 edges, all graphs till 13 edges and all cubic graphs till 21 edges. We will publish our files in collaboration with Katja Berčič, PhD to MathDataHub database.

We are in the process of contributing our code to the SageMath project.

5 Povzetek naloge v slovenskem jeziku

Matematiki vse pogosteje uporabljajo računalnike za podporo svojih raziskav. Matematiki tudi ustvarjajo in uporabljajo podatke, na nekaterih področjih pa sta proizvodnja in manipulacija z velikimi nabori podatkov vse bolj pomembna. Glavne uporabe teh matematičnih nizov in baz podatkov so raziskovalne narave. Raziskovalci jih uporabljajo za preizkušanje hipotez ali za iskanje vzorcev in kontrakseksov ter za shranjevanje nekaterih matematičnih rezultatov, da se izognejo ponovnemu izračunu. Ker gre za matematične predstavitve, potrebujemo obliko izmenjave podatkov - JSON, da lahko v računalnik pošljemo matematične predmete.

Teorija grafov je zelo pomembna veja matematike, ki ima tudi visoko uporabo v računalništvu. V tej tezi nas zanima vložitev neplanarnih grafov ter njihova reprezentacija in shranjevanje. Ravni grafi so tisti, ki jih je mogoče vgraditi v ravnino, z drugimi besedami, ki jih je mogoče risati brez robov. Problem testiranja planarnosti je algoritemski problem preizkušanja, ali je določen graf ravninski graf. To je dobro raziskana težava v računalništvu, za katero se je pojavilo veliko praktičnih algoritmov, mnogi izkoriščajo nove strukture podatkov. Edinstveno predstavitev (ravninskih) vdelav (ravninskih) grafov je mogoče dobiti s kombiniranjem kanoničnega označevanja s kombinatorno vdelavo. Ni standardnega načina računanja vdelave neplanarnih grafov, zato predlagamo obliko, kako to učinkovito storiti.

V 1. poglavju (Uvod) opisujemo motivacijo matematičnih baz podatkov in shranjevanje grafov v bazo podatkov. Pojasnimo tudi kanonično označevanje in vdelavo grafof ter razpravljamo o njegovi prehodni številki.

V poglavju 2 (Predhodni opisi) najprej v razdelku 2.1 predstavimo nekaj osnovnih pojmov iz teorije grafov: kaj je graf, kako narišemo graf, kaj je izomorfizem grafa in nato govorimo o prečkanju števila in o ravninskih grafih. V razdelku 2.2 predstavimo algoritme, uvajamo asimptotične zapise $\mathcal{O}$, Ω, Θ in opišemo časovno in prostorsko zapletenost algoritma ter postopek izračuna. Na koncu tega razdelka podamo nekaj besed o našem programskem okolju, ki ga uporabljamo. V zadnjem delu tega poglavja (razdelek 2.3) opišemo predstavitev grafov, kombinatorialno vdelavo grafa, njegovo kanonsko obliko in oblike izmenjave podatkov.

V tretjem poglavju se osredotočamo na cilj naše teze: opredelitev načina za predstavitev ravninske vdelave neplanarnih grafov. Najprej v razdelku 3.1 opišemo začetni pristop (kako ročno opravimo postopek): Vzeli bi graf, nato bi ga preoblikovali, da bi videli, kje je prehod, nato pa bi na mestu križanja uvedli še eno točko. Potem bi kanonično postavili verbel in dobili vgrajevanje grafov. V razdelku 3.2 postopek avtomatiziramo. Predstavimo algoritem, ki celoten postopek, opisan v poglavju 3.1, učinkovito opravi. Izračunamo časovno kompleksnost algoritma, ki je eksponentna (torej zelo visoka). Nato predstavimo algoritem za zapisovanje podrobnosti o grafu v datoteko (vložitev, dodajanje umetnih točk in niz graph6). Dodamo potrdilo o zastavi za branje s človekom (z več razlagami) in za bazo podatkov. V razdelku 3.3 predstavljamo aplikacije našega pristopa. Najprej predstavimo barvanje grafov in s tem algoritem za barvanje robov, ki so po našem postopku na novo dodani. Nato navajamo primere besedilne in neverbalne datoteke.

Izračunavanje ravninske vdelave neplanarnih grafov je zelo težaven in dolgotrajen postopek. Ker ni bilo standardnega načina, naše delo prispeva k teoriji zastopanja. Naše algoritme lahko uporabimo za obogatitev bolj ali manj katere koli baze grafov, kar tudi načrtujemo. Do sedaj smo ustvarili vertikalno tranzitivne grafe do 20 robov, vse grafe do 13 robov in vse kubične grafe do 21 robov. Naše datoteke bomo objavili v sodelovanju s Katjo Berčič, doktorico MathDataHub baze podatkov. V sodelovanju z obema mentorjema dr. Matjažem Krncom in dr. Katjo Berčič trenutno pišemo raziskovalni prispevek o tem in načrtujemo širjenje teh idej na ustreznih konferencah.

6 References

[1] Persinger. C. A. Subsets of n-books, pacific journal of mathematics. volume 18, pages 169–173, 1966. *(Cited on page 4.)*

[2] I. Anderson. *A First Course in Discrete Mathematics.* Oxford University Press, 2 edition, 1989. *(Cited on page 5.)*

[3] M. Barthelemy. *Morphogenesis of Spatial Networks.* New York: Springer, 2017. *(Cited on page 3.)*

[4] Sara C. Billey and Bridget E. Tenner. Fingerprint databases for theorems. *Notices of the AMS*, 60(8):1034, 2013. *(Cited on page 1.)*

[5] Nikolaos G. Bourbakis. *Artificial Intelligence and Automation.* World Scientific, 1 edition, 1998. *(Cited on page 2.)*

[6] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. *(Cited on pages 18 and 20.)*

[7] Purchase Helen C., Cohen Robert F., and James Murray I. Validating graph drawing aesthetics. in brandenburg, symposium on graph drawing, gd '95. volume 1027, pages 435–446, Passau,Germany, 1995. *(Cited on page 4.)*

[8] Duncan Christian, Goodrich Michael T., and Kobourov Stephen. Planar drawings of higher-genus graphs, graph drawing, 17th international symposium, gd 2009. volume 5849, pages 45–56, Chicago, IL, USA, 2010. *(Cited on page 11.)*

[9] Cormen Thomas H.; Leiserson Charles E.; Rivest Ronald L.; Stein Clifford. *Introduction to Algorithms, section 22.1: Representations of graphs.* MIT Press and McGraw-Hill, second edition, 2001. *(Cited on page 2.)*

[10] Reinhard Diestel. *Graph Theory.* Springer-Verlag New York, 2000. *(Cited on page 6.)*

[11] Knuth Donald E. *Art of computer programming, volume 2: Seminumerical algorithms.* Addison-Wesley Professional, 2014. *(Cited on page 8.)*

[12] C. Godsil and G. Royle. *Algebraic Graph Theory.* Springer, New York, NY, 2001. *(Cited on page 6.)*

[13] P. Goyal. and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. 2018. *(Cited on page 3.)*

[14] Hopcroft John and Tarjan Robert E. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974. *(Cited on page 8.)*

[15] Kleinberg Jon and Eva Tardos. *Algorithm Design.* Boston: Pearson Education, 2006. *(Cited on page 9.)*

[16] A modern reintroduction to ajax. `https://www.javascript-coder.com/tutorials/re-introduction-to-ajax/`. Accessed: 2020-07-10. *(Cited on page 13.)*

[17] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007. *(Cited on pages 2 and 12.)*

[18] Kuratowski Kazimierz. Sur le problème des courbes gauches en topologie. 15:271–285, 1930. *(Cited on page 7.)*

[19] Yoon. Byoung-Ha Kim. Seon-Kyu Kim and Seon-Young. Use of graph database for the integration of heterogeneous biological data. *Genomics and Informatics*, 15(1):19–27, 2017. *(Cited on page 2.)*

[20] P. Klein and S. Mozes. *Optimization Algorithms for Planar Graphs.* *(Cited on page 12.)*

[21] Oscar Levin. *Discrete Mathematics: An Open Introduction.* School of Mathematical Science, University of Northern Colorado, 3rd edition, 2013-2019. *(Cited on page 8.)*

[22] Babai László and Luks Eugene. Canonical labeling of graphs, proc. 15th acm symposium on theory of computing. pages 171–183. *(Cited on page 2.)*

[23] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014. *(Cited on page 2.)*

[24] Sipser Michael. *Introduction to the Theory of Computation. Course Technology Inc.* 2006. *(Cited on page 10.)*

[25] B. Mohar and C. Thomassen. *Graphs on Surfaces.* 2001. *(Cited on page 4.)*

[26] Didjev Hristo N. On drawing a graph convexly in the plane, graph drawing, dimacs international workshop, gd '94, princeton. volume 894, pages 76–83, New Jersey, USA, 1995. *(Cited on page 11.)*

[27] Turán. P. A note of welcome. pages 7–9. Journal of Graph Theory, 1977. *(Cited on page 4.)*

[28] Mutzel Petra and Weiskircher René. Computing optimal embeddings for planar graphs, computing and combinatorics, 6th annual international conference, cocoon 2000. volume 1858, pages 95–104, Sydney, Australia, 2000. *(Cited on page 11.)*

[29] Angles Renzo and Gutierrez Claudio. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008. *(Cited on page 2.)*

[30] Arora Sanjeev and Barak Boaz. *Computational Complexity: A Modern Approach.* 2007. *(Cited on page 11.)*

[31] A. Marcus Schaefer. *The Graph Crossing Number and its Variants: A Survey.* Chicago, Illinois, fourth edition, 2020. *(Cited on page 6.)*

[32] Space complexity of algorithms. `https://www.studytonight.com/data-structures/space-complexity-of-algorithms`, 2020. *(Cited on page 11.)*

[33] Python - time complexity. `https://wiki.python.org/moin/TimeComplexity`. Accessed: 2020-08-07. *(Cited on page 18.)*

[34] Time complexity of bliss algorithm. `https://stackoverflow.com/questions/63340982/time-complexity-of-bliss-algorithm`, 2020. *(Cited on pages IV and 20.)*

[35] Trudeau and Richard J. *Introduction to Graph Theory.* New York: Dover Pub, 1993. *(Cited on page 3.)*

[36] Arvind V., Das. Bireswar, and Köbler. Johannes. The space complexity of k-tree isomorphism", algorithms and computation: 18th international symposium. volume 4835, pages 822–833, Sendai, Japan, 2007. *(Cited on page 2.)*

[37] Math Vault. The definitive glossary of higher mathematical jargon - canonical. 2019-08-01. *(Cited on page 2.)*

[38] Arvind Vikraman, Das. Bireswar, and Köbler Johannes. A logspace algorithm for partial 2-tree canonization, computer science - theory and applications: Third international computer science symposium in russia. volume 5010, pages 40–51, Moscow, Russia, 2008. *(Cited on page 2.)*

[39] Ernst W. and Mayr Gunther Schmidt Gottfried Tinhofer. Graph-theoretic concepts in computer science. volume 903, 1995. *(Cited on page 4.)*

[40] Stein William. *SAGE: A Computer System for Algebra and Geometry Experimentation. (Cited on page 11.)*

A Appendix Testing for crossing number

```python
def _crossing_number(G, certificate = True): #main function
    from sage.combinat.subset import Subsets
    h = copy(G)
    two = [v for v in h if h.degree(v) == 2]
    for v in two:
        u, w = h.neighbors(v)
        if not h.has_edge(u, w):
            h.add_edge(u, w)
            h.delete_vertex(v)
    edgepairs = Subsets(h.edge_iterator(labels = False), 2)
    nonincident = [x for x in edgepairs if x[0][0] not in [x[1][0], x[1][1]] and x[0][1] not in [x[1][0],x[1][1]]]
    k = 0
    while True:
        for edges in Subsets(nonincident, k):
            g = copy(h)
            add = {}
            for pair in edges:
                g.delete_edges(pair)
            for edge in edges:
                v = g.add_vertex()
                add[v] = edge
                g.add_edge(edge[0][0], v, 0)
                g.add_edge(v, edge[0][1], 0)
                g.add_edge(edge[1][0], v, 1)
                g.add_edge(v, edge[1][1], 1)
            if g.is_planar(set_pos = certificate):
                if certificate:
                    return k, g, add
                return k
        k += 1
```

B Appendix Writing in a file

```python
def embedding_(G,certificate=True):
    with open("embedding-file.txt", "a") as f:
        s = G.graph6_string()
        k,G1,added = _crossing_number(G, certificate = True)
        G2 = G1.canonical_label()
        if G2.is_planar(set_embedding = True):
            d = G2.get_embedding()
            l = sorted(d)
            f.write("The Graph's six string: " +s + '\n' + '\n')
            f.write("The added vertices are:" + str(added) + '\n' + '\
n')
            f.write("The planar embedding of G:" + '\n' + "{")
            i, n=0, len(l)
            for elm in l:
                d[elm].sort()
                if i != n-1:
                    f.write(str(elm) + ':' + str(d[elm]) + ',' + '\n')
                else:
                    f.write(str(elm) + ':' + str(d[elm]) + '}' + '\n'
+ '\n')
                i+=1
```