From knots to braids.

Abstract

Note (Jun 2025): The text below is an English summary of the results of my Bachelor's and Master's theses. The document is a work in progress; some sections have not yet been fully edited. Updates leading to the final version will be introduced shortly.

This work introduces a novel approach for obtaining braid representations of knots through a new data structure—the enriched graph. Inspired by foundational ideas from Vogel's algorithm, the enriched graph provides a lightweight, efficient, and self-sufficient representation of knot diagrams. It encapsulates all the necessary topological information to perform braiding directly, without relying on auxiliary structures or external dependencies. We present a complete implementation of the braiding process using this representation. The implementation incorporates several algorithmic innovations that improve the manipulation and analysis of the proposed structure. In particular, the complexity of identifying admissible pairs is reduced to $O(1)$, and elementary operations are performed in linear time with respect to the number of crossings on the boundary of the joining Seifert face. We also provide an upper bound for the overall algorithm. These enhancements significantly outperform existing methods for solving the braiding problem and establish a new computational benchmark in the field.

The paper is organized as follows: the first section offers a concise introduction to key concepts in knot theory — such as knot definitions, Reidemeister moves, braids, and braid representations — along with a brief overview of Vogel's algorithm. This section is designed to be accessible to readers new to the topic, while more experienced readers may choose to skip it. The second section introduces the enriched graph data structure and details the braiding procedure built upon it. The third section presents the implementation, including the architecture of the core classes, an analysis of the complexity of elementary operations and the overall braiding algorithm, and a description of how the final braid word is extracted.

Contents

1 Introduction **2**

1.1 Knots 2

1.2 Braids 6

1.3 Braid Representations of Knots 8
1.4 Vogel's Algorithm 9

2 Algorithm **10**
2.1 Enriched Graph 11
2.2 Braiding on Enriched Graph 13
2.3 Getting braid word from the final form of the graph 18

3 Implementation **18**
3.1 Enriched Graph Architecture 18
3.2 Complexity Analysis 22
3.3 Reading Braid Word 24

4 Experiments **26**

5 Conclusions **27**

1 Introduction

In this section, we present fundamental concepts from knot theory that are essential for understanding the ideas discussed in this paper. These include: knot definition, knot planar representations and their transformations, as well as braid representations of knots, which constitute a central focus of this work. We also introduce key results from Vogel that form the foundation for the methods developed in this paper.

1.1 Knots

Knot

A **knot** is any closed curve embedded in R^3, considered up to isotopy (i.e., smooth deformation without cutting or gluing). Intuitively, a knot can be imagined as a piece of string with its ends joined together; as long as we do not cut or glue any part of the string, we can deform it freely and it will still represent the same knot. A **link** is a generalization of a knot, consisting of multiple such closed curves (called components) that may be interlinked. In this paper, we focus primarily on knots, although many results naturally extend to links.

Fundamental Problems of Knot Theory

Despite its seemingly simple definition, the study of knots gives rise to a wide range of non-trivial theoretical questions and reveals a rich and complex mathematical landscape. Two fundamental questions in knot theory are:

- **Knot equivalence**: How can we tell whether two representations describe the same knot?

- **Knot classification**: How many distinct knots exist at a given level of complexity (i.e., the minimal number of crossings in a planar projection)?

While the second question is primarily of theoretical interest studied by topologists, the first is of central importance across many scientific disciplines that use knot theory to describe real-world structures. Applications include biology (e.g., DNA and protein topology), chemistry (e.g., molecular knots), and physics (e.g., quantum field theory), where a key task can be determining to which known knot a given structure is equivalent, or whether two structures represent the same or different knots.

There are a number of ways to represent and study knots. In the following sections, we present a subset of those that are relevant to the topics explored in this paper.

Planar Representations of Knots

Since visualizing knots in 3D can be challenging, they are typically represented by two-dimensional diagrams. A knot diagram is a planar representation of a knot, obtained by projecting the knot onto a two-dimensional plane such that no more than two strands cross at any given point. To retain the essential topological information lost in the straightforward projection, each crossing is annotated to indicate which strand passes over and which passes under — typically by introducing a break in the underpassing arc. For oriented knot diagrams, each crossing can also be assigned a sign — either positive or negative — depending on how the strands interact with the orientation (see: [fig.]). Notably, this sign does not depend on the choice of orientation of a diagram.

It's also clear that knot diagrams are not unique: a single knot can be represented by infinitely many diagrams, depending on the projection chosen and the way the knot is deformed.

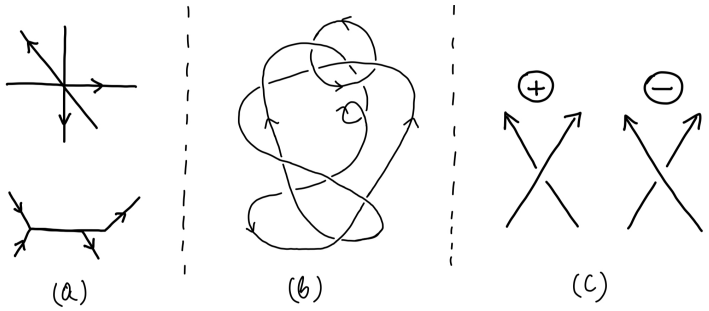


Figure 1: (b) Knot diagram.

Reidemeister Moves

Let's introduce three planar operations on a knot diagram, known as first, second and third Reidemeister moves (see: 1.1).

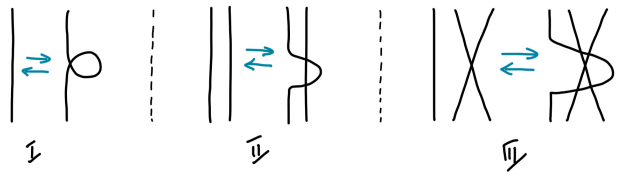


Figure 2: Reidemeister Moves

From the works of Alexander and Briggs (1926) [ref] and Reidemeister (1927) [ref], we have a foundational result in knot theory: two planar diagrams represent the same knot if and only if there exists a finite sequence of Reidemeister moves that transforms one into the other (up to planar isotopy).

This is a powerful theorem, as it guarantees that for any pair of diagrams derived from the same knot, we can find a finite sequence of moves to transform one of them to the other.

However, despite the importance of this discovery, finding an explicit sequence of moves between two diagrams remains highly non-trivial. There exist algorithms to solve this problem, like Haken's algorithm, but the space of possible move sequences grows rapidly with the complexity of the diagrams. This makes the problem computationally intensive and generally infeasible to solve by hand, especially for large knots.

Knot Invariants

To facilitate the task of determining knot equivalence, a variety of tools known as knot invariants have been developed. Knot invariants are mathematical objects – most often numbers or algebraic structures – assigned to knot representations that remain unchanged under different representations of the same knot (in particular, they are invariant under the Reidemeister moves for knot diagrams). This means that if two representations have different invariants, they must represent different knots.

Examples of commonly used knot invariants include tricolorability, the knot group, knot polynomials, signature, or bridge number. Different invariants capture different aspects of a knot's structure and can help distinguish between non-equivalent knots.

The key advantage of using invariants is that they are generally much easier to compute than attempting to find a sequence of transformations to show equivalence. However, there is often a trade-off: stronger invariants tend to be more computationally intensive. A complete invariant—one that can distinguish every pair of non-equivalent knots—has not been found so far. Nevertheless,

combinations of existing invariants are very powerful in the study of knot theory and its wide-ranging applications.

A number of important invariants, such as the Alexander polynomial, Jones polynomial, and HOMFLY-PT polynomial, are calculated using braid representations of knots – see [braid section].

Seifert Picture of a Knot Diagram

This section is slightly more technical and may be omitted during an initial reading. It introduces the Seifert picture construction derived from a knot diagram, along with several concepts that are important for the subsequent discussion. These form the foundation for both Vogel's method and the data structure and algorithm developed in this work.

The Seifert picture of a knot diagram is a collection of disjoint simple closed curves associated with a given knot diagram. Originally, this construction was introduced as the first step in Seifert's algorithm – a key tool in 3-manifold theory – whose aim is to build an orientable surface whose boundary is the original knot or link. This construction also provides insight into some knot invariants, like genus, signature, or the Alexander polynomial.

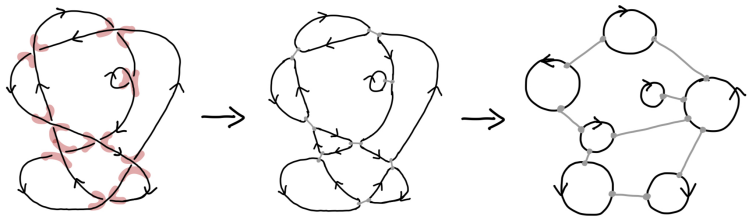


Figure 3: Seifert picture of oriented knot diagram. Diagram crossings highlighted in red. Gray segments represent smoothed crossings.

Given an oriented knot diagram, the Seifert picture is obtained through the following steps (see: [fig]):

1. Resolve every crossing in the diagram in a consistent way according to the orientation.
2. At each crossing, replace it with a "smoothing" that respects the orientation;
3. Mark each "smoothed" crossing by adding a non-oriented segment to track its location (this step is not part of the classical construction but is important in the approach presented in the next sections)
4. The result of each smoothing is that the strands at the crossing are connected without intersecting, preserving the local orientation.

After performing this at all crossings, we are left with a collection of disjoint, oriented circles in the plane. These are called the Seifert circles. In our context, we will call the full set of Seifert circles, together with the segments marking crossings, the Seifert picture. The regions of the plane divided by Seifert circles are called **Seifert faces**. In this paper, we will refer to the regions formed by the Seifert circles together with the segments marking crossings as **Seifert areas**.

The Seifert picture construction plays a central role in Vogel's algorithm (see [ref section]) and in the enriched graph construction introduced in this work.

1.2 Braids

A powerful approach to understanding and classifying knots is through their braid representations.

Braid

Intuitively, a braid consists of n strands arranged from left to right, where adjacent strands may cross each other a finite number of times in one of two possible ways (see [fig]).

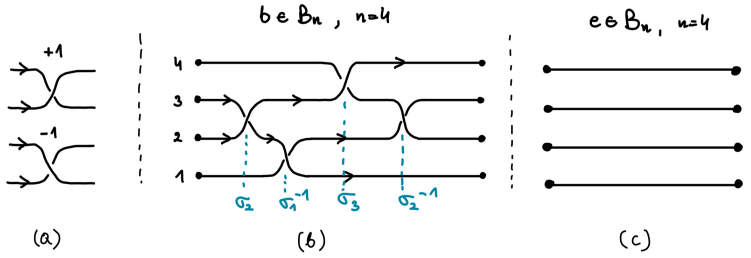


Figure 4: Braid. (a) Positive and negative braid crossing. (b) Braid from B_4 and its representation by generators. (c) Neutral element in B_4.

Formally, a braid with n strands can be described by a word over the alphabet: $\{\sigma_1^{\pm 1}, \sigma_2^{\pm 1}, ..., \sigma_{n-1}^{\pm 1}\}$ where $\sigma_i^{\pm 1}$ represents a crossing between the i-th and $(i+1)$-th strand, and the exponent indicates the crossing's orientation (see [fig] (a)).

For each n, the set of all braids with n strands, equipped with the concatenation operation (connecting the endpoints of one braid to the start points of another), forms a group known as the braid group B_n. This group is infinite, finitely generated, and non-abelian, with defining relations:

$$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}, \text{ for } i \in \{1, .., n\},$$
$$\sigma_i\sigma_j = \sigma_j\sigma_i, \text{ for } |i-j| > 1,$$

The identity element is the braid with n parallel strands and no crossings (the empty braid word).

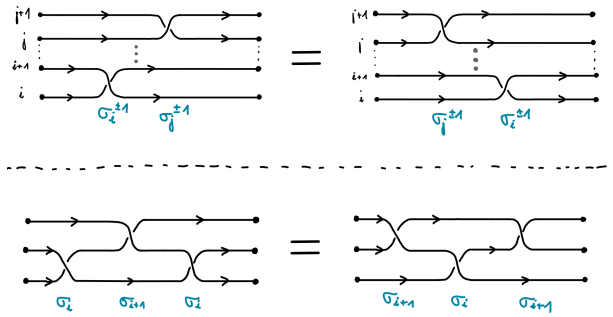


Figure 5: Braid equivalence relations.

Closed Braid

The closure of a braid is formed by connecting the corresponding endpoints of its strands in a standard way (see [fig]).

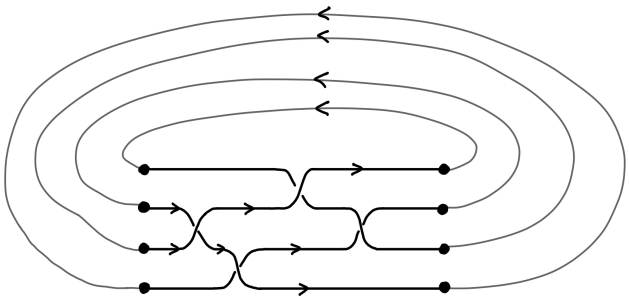


Figure 6: Closure of a braid.

Let's note that this closure naturally yields a knot or link diagram. A fundamental question arises: is every knot representable as the closure of some braid? The answer is affirmative. By Alexander's Theorem (1923), every knot or link can be realized as the closure of a braid.

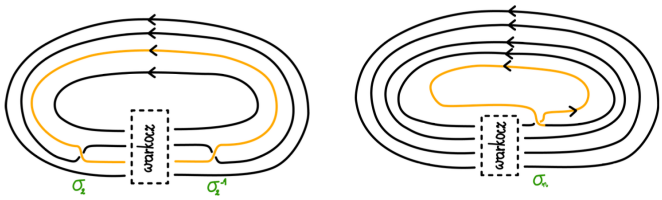
This theorem is significant for several reasons:

- It creates a bridge between topological knot theory and algebraic braid theory, enabling the application of algebraic tools to study knots.
- Braids provide highly convenient and efficient representations of knots, especially useful for computational approaches.
- Knot equivalence can be studied algorithmically via braid equivalence, using Markov moves (see [box]). Two braids represent the same knot if

and only if they are connected by a finite sequence of braid relations and Markov moves.

- Many important knot invariants - such as the Alexander, Jones, and HOMFLY-PT polynomials — can be computed directly from braid representations.

Two closed braids represent the same knot if and only if they can be transformed into one another by a sequence of braid equivalence operations [ref], together with two additional operations known as Markov moves [**?**].



(a) koniugacja (b) stabilizacja

Figure 7: Markov Moves.

In summary, braid representations not only provide a rich algebraic framework for studying knots but also play a central role in computational knot theory.

1.3 Braid Representations of Knots

Alexander's Theorem

Alexander's theorem (1923) provides a constructive proof by showing how to transform any 3D knot representation into a braid form. However, the original proof is expressed in geometric and intuitive terms rather than as a precise, step-by-step algorithm. This makes it difficult to translate directly into a rigid, deterministic procedure suitable for computer implementation. Another drawback of Alexander's method is that it often, even for simple knots, produces braids with a large number of strands and crossings, which is not a desirable quality.

Yamada

Yamada (1987) and Vogel (1990) proposed significant improvements to Alexander's method. Yamada's algorithm uses two types of operations, known as "bunching operations," to produce a braid whose number of strands matches the number of Seifert circles in the original knot diagram. Compared to Alexander's original method, this approach also results in a more modest increase in the number of crossings.

Vogel

Vogel further refined the method by reducing the number of operations to a single, more easily characterized one (a type II Reidemeister move applied to specific parts of the diagram). His algorithm also preserves the number of strands, keeping it equal to the number of Seifert circles in the original diagram. Each elementary operation adds only two crossings to the diagram. Additionally, Vogel established an upper bound on the number of operations required, which is quadratic with respect to the number of Seifert circles in the diagram.

What's important is that the method is suitable for implementation for knot diagrams on a computer.

Among these algorithms, Vogel's stands out as the most efficient in terms of operations and diagram size. Still, like Alexander's and Yamada's methods, although it is described in the original paper as "easy to implement on computer," it was primarily developed as a conceptual pen-and-paper technique. Although some implementations exist ([Sage] [KontTheory Mathematica]) they tend to be slow [[maybe add complexity estimate]] and sometimes fail to complete even on moderately sized diagrams. This inefficiency largely stems from operating on naive planar diagram encodings, with key tasks like searching for admissible pairs being particularly slow (see [ref]).

In this paper, we propose an efficient method for representing information about knot diagrams, along with an optimized algorithm for obtaining the braid representation of a knot, which builds upon ideas introduced by Vogel. Notably, we reduce the major bottleneck — the complexity of finding admissible pairs to constant time, which is a crucial improvement that significantly accelerates the overall algorithm and enables the processing of larger and more complex knot diagrams.

1.4 Vogel's Algorithm

Here, we provide a brief overview of Vogel's algorithm, which serves as the foundation for the algorithm presented in this work.

Braiding procedure by Vogel

The algorithm proposed by Vogel consists of repeatedly performing Reidemeister move II on so-called admissible pairs of diagram segments, as long as such pairs exist within the diagram. If no further admissible pairs remain, the diagram is in closed braid form.

We define an admissible pair of segments as follows. Given a directed knot diagram, consider its Seifert image ([ref – section]). Two diagram segments form an admissible pair if and only if they satisfy all of the following conditions:

i They lie on the boundary of the same Seifert face,

ii They belong to different Seifert circles,

iii They have the same orientation with respect to any orientation of boundary from (i).

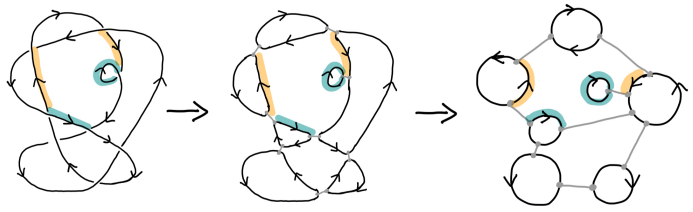


Figure 8: Example of admissible pairs of segments in oriented knot diagram.

To transform the diagram into closed braid form, we select any admissible pair of segments and perform a Reidemeister move of type II on them. We refer to this move performed on an admissible pair as the *elementary operation*. Note that this operation may or may not create new admissible pairs. We then repeat the procedure —- constructing the Seifert image of the new diagram and searching for admissible pairs —- until no admissible pairs can be found.

Vogel does not provide an efficient method for locating admissible pairs of segments. The most we can infer from his paper is that, by examining the graph construction used in the proof of the main theorem, one can identify the Seifert faces that contain admissible pairs. Specifically, these are the faces corresponding to nodes with at least two incoming or outgoing edges.

Vogel's graph

To prove the main theorem of his work, Vogel constructs a directed graph corresponding to the Seifert image of a diagram (see [section]). The graph is defined as follows: each node represents a face in the Seifert diagram, and for each pair of faces v_1 and v_2, a directed edge e_x is added from v_1 to v_2 if and only if it is possible to traverse from v_1 to v_2 by moving from the left to the right side of a Seifert circle s_x. As a result, the graph contains m nodes corresponding to the m Seifert faces and n edges corresponding to the n Seifert circles in the Seifert image. Note that in this construction, some information is lost —- specifically, the exact correspondence between edges and Seifert circles when a node has more than one incoming or outgoing edge.

We adopt the core idea of this graph as the basis for our proposed data structure and algorithm (see [next section]).

2 Algorithm

In this chapter, we introduce the enriched graph, a compact data structure that encodes knot planar diagrams and supports efficient transformation into braid form. We describe the structure, show how it enables the identification of

admissible pairs and execution of elementary operations, and present a braiding algorithm based on these operations. The enriched graph also allows direct reconstruction of the knot diagram, making it both computationally efficient and interpretable.

The enriched graph offers several key advantages:

- **Self-sufficiency:** it encodes all necessary information to represent a knot diagram and perform braiding operations without the need for auxiliary structures.
- **Efficiency:** the most computationally intensive steps—finding admissible pairs and applying elementary operations—are highly optimized. Admissible pairs can be identified in constant time O(1), and elementary operations are performed in linear time with respect to the number of crossings on the boundary of the merged Seifert area.
- **Transparency and reconstructibility:** the original knot diagram can be reconstructed (up to stereographic projection on the Riemann sphere) from the graph at any stage, enabling visual inspection, debugging, and deeper theoretical analysis.

2.1 Enriched Graph

In this section, we outline how the enriched graph is constructed. The base structure follows Vogel's graph construction, with two key enhancements: we introduce crossings and color information to support further steps in the braiding algorithm. Each of these additions is detailed below.

Graph Base

At first, for a given knot diagram D, we create Seifert picture S and the Vogel's-like graph (see [prev chapter]). This will be the basis for the enriched graph structure.

Recall that, what was shown by Vogel, such graph is in a form of a chain if and only if the knot diagram is isotopic in the Riemann sphere to the closed braid. We will use this fact in the later steps of our algorithm.

Crossings

Let's index the crossings of D. This will result in each crossing index appearing exactly twice in the Seifert picture corresponding to D, with each occurrence lying on a different Seifert circle.

Now, while constructing the graph associated with the diagram, for each circle s from the Seifert picture S, when adding the edge e_s, corresponding to s, to the graph, we annotate it with all crossing indices and their associated signs, in the order they appear along s, starting from an arbitrarily chosen one (see img [xxx]). Crossing signs won't be used for the graph transformations,

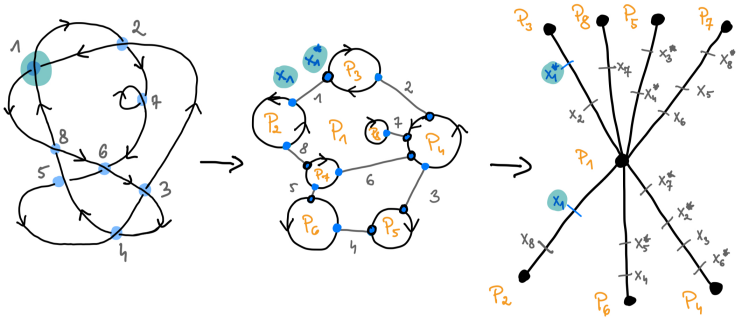


Figure 9: Construction of enriched graph – adding crossings.

but they are necessary to read the resulting braid word at the final stage of the algorithm.

As a result, we obtain a *partly-enriched graph*, that encodes the crossing sequences along the Seifert circles. Each crossing index appears exactly twice. Let's note that these repeated indices occur on consecutive edges of the directed graph. Note that, unlike in Vogel's graph, the edges of the *partly-enriched graph* correspond unambiguously to the Seifert circles of D.

We refer to the portions of graph edges between crossing indices as **graph segments** (including the one divided by the graph node -— one such per edge). Naturally, these graph segments correspond one-to-one with segments in the Seifert diagram of D; thus, they represent all potential candidates for admissible pairs.

We make the following observations:

REMARK [XXXX]

- Edges incident to the node v represent all Seifert circles that bound the Seifert face corresponding to v.
- Edges outgoing from v have the face represented by v on their left, while edges incoming to v have it on their right.
- If the segments (d1,d2) form an admissible pair, they lie on edges sharing a common initial or terminal node.

The above remark leads to an important conclusion: the search for admissible pairs can be restricted to segments located on edges that share a common initial or terminal node. The only challenge that remains for specifying whether a pair of segments forms an admissible pair is determining whether these segments lie on the boundary of the same Seifert region — this is addressed by introducing colors into the graph structure.

Colors

Let's assign a unique color to each Seifert region in the Seifert diagram S associated with D. Each segment is then annotated with the color of the region

to its left and the color of the region to its right (these two colors are always distinct). These color labels are then transferred from the Seifert picture to the corresponding segments in the *partly-enriched graph*.

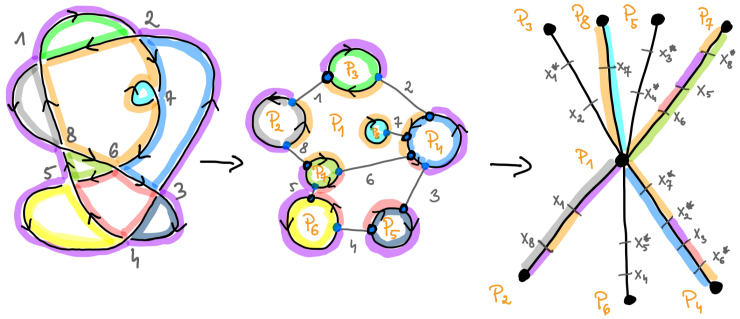


Figure 10: Construction of enriched graph – crossings and colors.

Observations on Colored Graph Segments – For each color c, there is a unique node v such that any segment colored c lies on an edge e where v is either the initial or terminal node.
– If an edge e is incident to v and contains a segment with color c, then c appears on the right side of the segment if e is incoming, and on the left side if e is outgoing.

Corollary: Two graph segments $(d1, d2)$ form an admissible pair if and only if they lie on distinct edges, both incoming to or outgoing from the same node v, and share the same color on the right (for incoming edges) or on the left (for outgoing edges).

Efficient recognition of admissible segments on the enriched graph From the above, we now know that this construction provides one of the three essential components for performing braiding directly on the enriched graph: the ability to efficiently recognize admissible pairs. The next challenge lies in determining whether, and how, the extended graph can be modified to reflect the diagram after applying an elementary operation to a selected admissible pair of segments. Finally, we must ensure that the braid word can be extracted from the final form of the graph. Taken together, these three capabilities make the enriched graph a fully self-contained and algorithmically actionable structure.

2.2 Braiding on Enriched Graph

The core action for transforming a knot projection into the closed-braid form is the iterative application of Reidemeister move II on admissible pairs of segments

(**elementary operation**). As the recognition of admissible pairs was already discussed in the previous section, we now focus on the transformations of the graph that reflect the changes made in the knot diagram under the elementary operation

Elementary Operation on Enriched Graph

It turns out that, using only the information encoded in the graph, we can update the structure so that it corresponds to the updated knot diagram after performing the elementary operation on a given admissible pair of segments. Below, we outline the general rules for how it is done. To maintain clarity and avoid technical overloading, we do not include the full formal treatment here (this is addressed in detail in [1] and [2]). Instead, we focus on the essential conceptual steps of the process.

Changes in Seifert Picture Under Elementary Operation

We begin by examining how the elementary operation is reflected in the color-annotated Seifert picture. Then we transfer this understanding to the graph representation.

Suppose we have selected an admissible pair of segments $(tau1, tau2)$ on the enriched graph. Let $S1$, $S2$ denote the Seifert circles that contain the segments represented by $tau1$, $tau2$ respectively.

The application of the elementary operation to this pair results in the following modifications to the color-annotated Seifert picture (see: fig [xxx.a]):

1. $S1$ and $S2$ are merged into a single circle $S12$.
2. A new circle $S3$ and is created (either nested within or outside S_12, depending on the relative positioning of $S1$ and $S2$, see: [fig]).
3. Two new pairs of conjugated crossings are introduced: $**x** = (x, x*)$, $**y** = (y, y*)$, where $sign(x) = -sign(y)$ ¡!–note: revisit language fluency –¿ we denote by x,y the new crossings on s12 and x*, y* new crossings that appear on s3.
4. The ordering of crossings on S_12 is as follows: beginning at the endpoint of $tau1$, we first traverse the remaining crossings on $S1$, followed by the new crossing x; this is continued with the crossings on $S2$, starting from the endpoint of $tau2$, and concluding with the crossing y. The labels x and y are assigned arbitrarily, so their order in the description may be interchanged without loss of generality.
5. A new region appears in the interior of $S3$, marked with a new color cA
6. Part of the original 'joining' region becomes separated, forming a separate Seifert area with a new color.

7. Colors of not shared areas of S1 and S2 are propagated to S3 segments on circle-exterior side

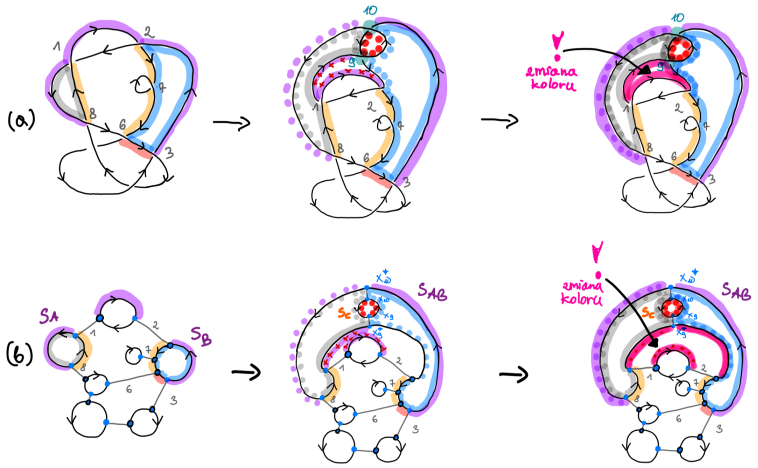


Figure 11: Elementary operation. Changes in diagram and Seifert picture with color and crossing annotations.

Changes on Enriched Graph Under Elementary Operation

This is reflected in the following way on the enriched graph (see: Fig. [xxx.b]). Let $e1$ and $e2$ denote the edges corresponding to the Seifert circles $S1$ and $S2$, respectively:

1. $e1$ and $e2$ are merged into a single edge $e12$
2. A new edge $e3$ is introduced. It starts from the terminal node of $e12$ or finish in the initial node of $e12$, depending on case (see [comment])
3. Two pairs of conjugated crossings $**x** = (x, x*)$ and $**y** = (y, y*)$, where sign(x) = -sign(y) are added to the graph. Lets denote by x,y crossings added on e12, and by x*,y* these added on e3.
4. Crossings order on e12 is the same as described in (iv) above for the Seifert picture.
5. The entire 'interior' side of (for recognition of 'interior' side – see below) edge $e3$ is colored in new color cA
6. A portion of the original "joining" color is reassigned to a new color c'_j on $e3$ and some segments previously colored with c_j on other edges are updated accordingly (mechanism – see below).

7. Segments created by adding x,y on e12 and x*, y* on e3 are colored with non-shared colors of tau1 and tau2 appropriately on appropriate sides (not-joining side on e12, 'outer' side of e3)

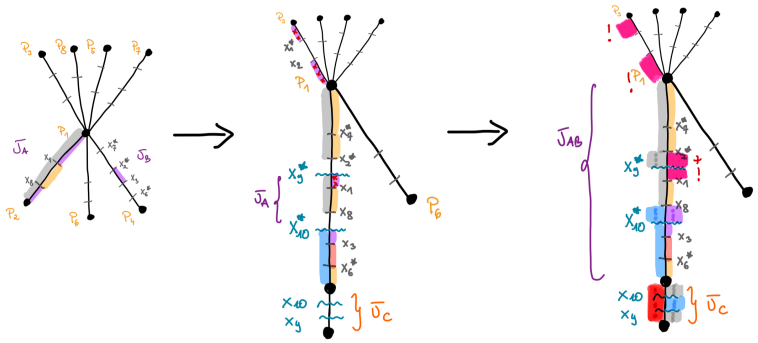


Figure 12: Elementary operation. Changes in annotated Seifert picture [add] and enriched graph.

Remarks on enriched graph transformations

Comment 1, ad (i), (ii) By REMARK [XXX], we know that admissible segments appear between pairs of edges that share either an initial or terminal node. In each case, the non-shared nodes n1,n2 are merged to a single node n12 as a result of the elementary operation (this reflects the fact that the plane regions bounded by Seifert circles S1, S2 are unified into a single face). As a result of this, all edges incident to n1,n2 will now be incident to n12 in the same way as they were to n1 or n2 (see [fig]).

Determining the 'interior' side of the newly created edge Unlike the Seifert picture, the graph representation of a knot projection lacks explicit spatial information. Specifically, it does not indicate which side of an edge representing Seifert circle corresponds to the 'interior' or 'exterior' of the circle. For example, consider a node v with several incoming edges. One of these edges may represent the circle that actually encloses the face associated with v, meaning the face lies inside that particular circle. However, the graph does not encode this distinction: from construction, all such edges are assigned the color from v on their right side, so there is no way to distinguish them based on the graph structure alone.

Consequently, when creating a new edge during the elementary operation, we do not immediately know which side of it should be treated as the 'interior' — which is necessary for proper coloring in step (v). This ambiguity is resolved by a key geometric observation (see [1]/[2]): in every configuration, the 'interior' of the new Seifert circle lies on the opposite side from the side that carries the

joining color on the admissible segments. Thus, when edges with a shared initial node are joined, the color cA will appear on the ... side of e3; and when joining edges with a shared terminal node, it will appear on the ... side. This rule allows us to determine the 'interior' side of the new created edge and complete step (v).

how do we know how to traverse the boundary of a particular Seifert area using the graph representation? Another key challenge, caused by the fact that the graph lacks explicit spatial data, is that it does not directly indicate the boundaries of a Seifert area. In other words, it's not immediately clear how to traverse the graph in a way that corresponds to traversing along the boundary of a color area in the Seifert picture. The first simple observation we can make is: when we traverse a color boundary, we follow along a circle (edge) on one of its sides, either with or against the edge's orientation, and sometimes we switch to another circle via a conjugate crossing. However, it's not always obvious when such a switch should happen (in some cases, we switch even if the same color continues on the current edge — so color continuity on the edge alone doesn't give us the full picture). We also need to know the correct side and direction (left/right side, with/against orientation) to continue on the next edge. Fortunately, there is a general rule, derived from observations detailed in [1] and [2], that helps resolve this ambiguity. When we reach the terminal crossing of a segment while moving with a given (side, orientation), we check whether the same color continues in the direction (-side, -orientation) from the conjugate crossing. If it does, we switch to the conjugate crossing, update our direction to (-side, -orientation), then continue the traversal using the described rule. The process stops when we reach the starting point for the first or second time (depending on the color configuration around the starting point — see [2, Appendix B] for details). This approach allows us to simulate traversal along the boundary of a color area using the graph structure. With slight adaptations, the same logic enables recoloring of the c'_j region, as required in step (vi).

Note: This c'_j recoloring is the most complex and complexity-determining step of the elementary operation. The rest of the changes are done in constant time; this one depends on the size of the area border to be recolored, i.e., the number of segments (equivalent to crossings) it contains.

Summary

Thanks to color annotations and crossing conjugations, we're able to perform all updates related to the elementary operation directly from the graph's structure — no external geometry or data is needed. This makes the graph representation fully self-contained for the entire braiding process. The final step is to read the braid word from the final chain graph.

2.3 Getting braid word from the final form of the graph

As already mentioned, we know that we are ready to read the final braid word when the graph is in chain form.

Colors are no longer needed at this stage. We now use information about crossing conjugates and the relative order of crossings on adjacent edges to establish their sequence in the resulting closed braid. Then, we use the crossing signs to generate the final braid word.

The resulting braid will have n strands, where n is the number of edges in the graph. Therefore, the braid word will use generators $\sigma_1^{\pm 1}, ..., \sigma_{n-1}^{\pm 1}$.

The procedure is described in detail, along with its implementation, in Section [section].

3 Implementation

In this section, we present the key concepts behind an efficient implementation of the directed graph structure that enables fast and scalable braiding operations. This implementation introduces additional techniques for representing enriched graphs that significantly improve the efficiency of generating a knot's braid form.

A central innovation is the dynamic encoding of colors: instead of being passive labels, colors are implemented as active data structures that track the edges and segments they occupy. This enables rapid updates, efficient lookups, and structural operations essential for high-performance braid construction.

This section is organized as follows: the first part introduces the classes used in the implementation, highlighting key ideas and architectural decisions. The second part outlines how the elementary operation is performed within this framework. The final part focuses on extracting the braid word from the enriched graph once its terminal form is reached.

3.1 Enriched Graph Architecture

Let us recall: we aim to represent a directed graph enriched with structural data — signed and pairwise-conjugated crossings dividing edges into segments, along with segment-level color annotations. The core procedures that must be supported are:

- Efficient identification of admissible pairs (one at a time).
- Performing the elementary operation on a selected admissible pair (including edge merging, insertion of crossings, addition of a new color, and modification of colors of Seifert areas).
- Assessing whether the stop condition is reached (i.e., whether the graph has reached its terminal "chain form," meaning no admissible pairs remain).
- Extracting the final braid representation (braid word) from the enriched graph structure.

The enriched graph is encoded using the following classes: core elements: ‘Crossing‘, ‘Color‘, ‘Edge‘; container classes: ‘CrossingsCollection‘, ‘ColorsCollection‘, ‘EdgesCollection‘; and a top-level orchestrator class: ‘Graph‘, which serves as the central interface for managing and operating on the entire structure.

Core Elements

Class Crossing The Crossing class serves as the primary unit for representing the structural features of the enriched graph. Each instance corresponds to a single crossing ‘cx‘ (with each member of a conjugate pair represented as a separate object). As such, every ’Crossing’ object also defines the starting point of a graph segment. Each ‘Crossing‘ object includes: - The sign of the crossing (positive or negative) - A unique crossing identifier (integer) - Identifier of an edge that crossing lays on (integer) - A pointer to its conjugate ‘Crossing‘ object - Pointers to the previous and next crossings along the same edge - Identifiers (as integers) of the left and right colors associated with the segment beginning at this crossing.

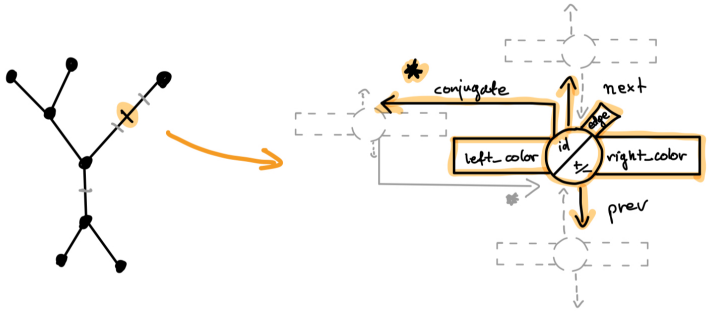


Figure 13: Schematic representation of a Crossing object.

Class Edge The Edge class functions as an organizational entity facilitating interactions between crossings and colors. Crossings themselves form doubly linked lists that represent their cycles along graph edges, while the Edge object encapsulates these cycles at a higher level. Each Edge instance is identified by an integer and holds a pointer, $anchor_c x$, to one crossing within its cycle. The selection of the anchor crossing is arbitrary and does not impact the overall structure.

Class Color The Color class introduces foundational innovations for enriched graph representation, serving as the backbone of the presented algorithm’s efficiency. These design choices result in performance that significantly surpasses all existing implementations of the Vogel algorithm. By organizing colors within

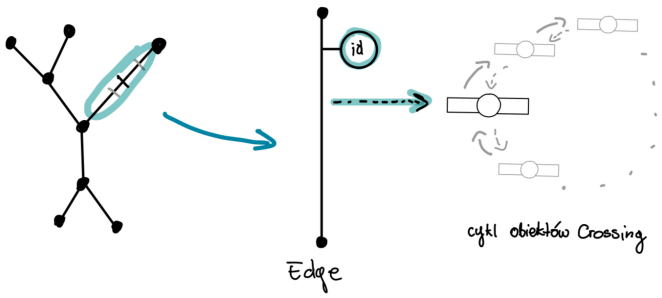


Figure 14: Schematic representation of an Edge object.

the ColorsCollection and coordinating with graph crossings, the class achieves major complexity reductions — enabling constant-time identification of admissible segment pairs and verification of the graph's chain form. Colors serve as a compact representation that allows the full reconstruction of the graph's shape at any point, although the algorithm operates using only partial shape information at a time. Each instance of the Color class contains the following attributes: - An integer identifier - Two dictionaries, $left_edges$ and $right_edges$, which map edge identifiers to sets of crossing identifiers. The edge identifiers denote edges with this color on their left or right side respectively, while the crossing identifiers represent crossings that feature this color on the corresponding side. From the Seifert picture perspective: each Color object corresponds to a single Seifert area The $left_edges$ dictionary gathers all Seifert circles that border this area on the right, along with all associated segments where the color appears on the left. The $right_edges$ dictionary performs the analogous function for the opposite side.

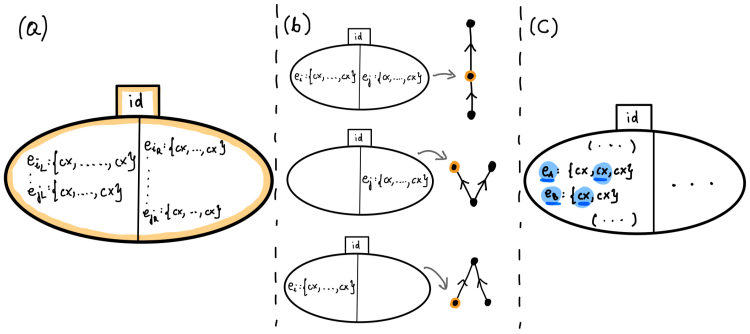


Figure 15: .....

Getting admissible pairs in O(1) time complexity Let's highlight a very important observation: in this setup, the Color object keeps, in an easily ac-

cessible form, information about **all** admissible segments associated with this color. Indeed, each pair of crossings from one of the side dictionaries, taken from sets under different edge keys, represents an admissible pair. This way, we gain access to admissible pairs in O(1) time.

Detecting the Final Graph State in Constant Time [O(1)] Another key observation is that the absence of admissible pairs within a given Seifert area is equivalent to having at most one edge present in each side dictionary ($left_edges$ and $right_edges$). This insight allows dynamic classification of Color objects into todo and done groups based on whether admissible pairs are still available. During the algorithm's execution, colors may transition between these groups. When the 'todo' group becomes empty, it signifies that the graph has reached its final chain form, and the final braid word can be extracted.

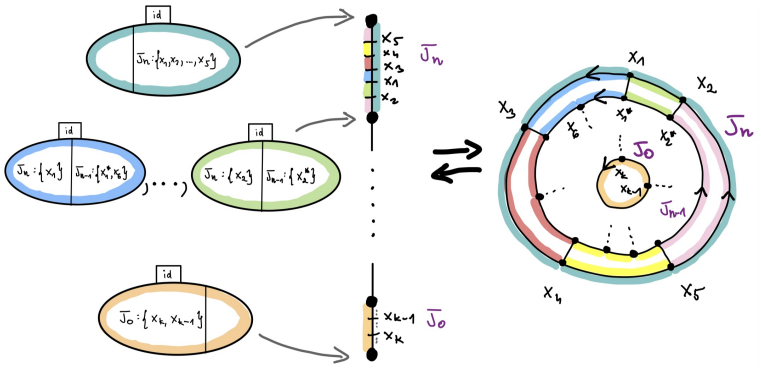


Figure 16: .....

Container Classes

These classes serve as graph–level managers for Crossing, Edge, and Color objects. Each collection class is built around a primary attribute: an $unordered_map < int, Type* >, where Type \in Edge, Crossing, Color$. Both CrossingsCollection and EdgesCollection rely solely on this map, while ColorsCollection extends the structure with two additional set attributes that track which colors are in the todo and done states (see: [ref]). Every Type object stores the key under which it is registered in its respective collection. This design facilitates efficient representation of inter-object relationships and allows for seamless information flow during graph transformations. By using pointers as values in the containers, the collections can be resized (e.g., during dynamic expansion) without costly object relocation—a significant performance benefit over storing objects directly.

CrossingsCollection This class manages all crossing-level data for the graph. Each elementary operation adds four new crossings to the collection. No cross-

ings are ever deleted throughout the algorithm's execution.

EdgesCollection This is the only container that maintains a constant number of elements between each elementary operation. Indeed, the number of Seifert circles (graph edges) remains constant throughout Vogel's algorithm. In each elementary operation, one edge is removed (merged into another), and one new edge is added.

ColorsCollection ColorsCollection contains three core attributes: - an '$unordered_map < int, Color* >$' mapping color IDs to Color objects - a set of colors marked as todo - a set of colors marked as done

Each elementary operation introduces two new color entries: the first corresponds to the exclusive color of the newly created Seifert circle. Initialized with a single edge and two crossings, it is immediately placed in the done set. The second replaces the joining color in one of the newly separated Seifert areas. This Color object inherits a subset of crossings from the joining color, which are reassigned under the appropriate edge key during the recoloring phase. The todo and done sets are continuously updated to track the current status of all colors.

ColorsCollection maintains a strong link with CrossingsCollection; any modifications to a crossing's color or edge assignment are reflected in the associated Color objects.

Class Graph

The Graph class encapsulates the entire enriched graph structure. Its primary attributes are instances of the three collection classes: CrossingsCollection, EdgesCollection, and ColorsCollection. By monitoring the todo set in ColorsCollection, the Graph can determine in constant time (O(1)) whether it has reached its final chain form.

3.2 Complexity Analysis

As described above, finding an admissible pair of segments is performed in O(1) time using Color objects and their grouping within the ColorsCollection. Below is a brief description of the changes each class undergoes during an elementary operation.

Let's summarize the steps of the elementary operation once again (compare [section], [section]), this time with a focus on class-level changes: - Merge the two edges that contain the admissible segments - Add a new edge to the graph - Introduce four new crossings (two pairs of conjugate crossings), forming four new segments on two edges - Create a new color assigned to the newly added edge - Reassign a portion of the original (joining) color to a second, newly created color

The main changes for each step are outlined below in terms of their impact on the relevant classes, along with notes on time complexity.

Joining edges *(Crossing, Color, CrossingsCollection(??), ColorsCollection(****), EdgesCollection, Edge(-))* In this step, edge e2 is merged into edge e1 at the appropriate position (see [descr]). The identifier of e2 is removed from the EdgesCollection and Edge object representing e2 is removed. Completing the merge requires updating the edge identifiers in all crossings originally belonging to e2, and moving these crossings under the e1 key within their respective Color objects. The relevant Color objects are efficiently retrieved in constant time using the $left_color$ and $right_color$ attributes of crossings via the ColorsCollection. Time complexity: $O(|e2|)$, where —e2— is the number of crossings in e2 Space complexity: O(1)

Adding a new edge *(EdgesCollection, Edge(+))* A new, initially empty Edge object is created and represented in the EdgesCollection. Time complexity: O(1). Space complexity: O(1).

Adding new crossings (without color assignment) *(Crossing(+), CrossingsCollection, Edge)* Four new Crossing objects are created and represented in the CrossingsCollection. These crossings are inserted into the appropriate positions on edges e12 and e3. Each crossing is assigned an edge identifier, neighborhood pointers are updated, conjugate pairs are linked, and the anchor crossing is assigned for the new edge created in the previous step.

Time complexity: O(1). Space complexity: O(1).

Adding new color and filling color annotations for new crossings *(ColorsCollection, $Color(+)$, Crossing)* A new empty Color object is created and represented in ColorsCollection. Crossings from e3 are assigned this new color on their internal side, and appropriate existing colors on their external sides. Crossings added to e12 receive their color assignments based on neighboring colors. The new Color is updated with e3 and its crossings on the appropriate side. Existing colors are also updated: new crossings are added under the e12 (existing) and e3 (new) keys on the appropriate sides. ¡!– (?Optional todo-done modifications are done on the level on ColorsCollection (are they possible here?)) –¿

Time complexity: O(1). Space complexity: O(1).

Recoloring part of joining color *(ColorsCollection, Crossing, Color)* To perform the color change, we traverse the border of one of the Seifert areas that emerged from dividing the joining area. The implementation of this process is slightly more complex and technical—it is thoroughly described in [2], so we won't go into details here. Generally, it involves following edge segments crossing by crossing, making certain decisions along the way, and occasionally switching the current edge via a conjugate crossing—continuing this until the start crossing is reached. Decisions are made in O(1) time and space, so the traversal has time complexity O(—area—), where —area— is the number of crossings in the Seifert area being recolored. Note that this is upper-bounded

by —Cj— + 2, where —Cj— is the total number of crossings originally assigned to the joining color.

To summarize, the changes made during this process are: - Changing the value of the color attribute on the appropriate side from the joining color to the new color for each crossing along the path (or just adding in two of the newly added crossings), - Moving crossings from the corresponding edge and side in the joining color to the appropriate side in the new color, - Updating the 'todo' and 'done' status of the joining color and the new color when the process finishes.

Time complexity:$O(|joining_area_1/2|) <= O(|Cj|)$ Space complexity: $O(1)$

Final step: reading the braid word

.....

Overall Complexity

To summarize: finding an admissible pair is done in O(1). Performing the elementary operation on this pair takes $O(|e_2| + |joining_area1/2|)$, where $|e_2|$ is the number of crossings on the second (arbitrarily selected during the joining phase) of joined edges, and $|joining_area1/2|)$ refers to the number of crossings along the boundary of the recolored part of the joining Seifert area.

No redundant searching or auxiliary restructuring is required — each step is efficient and localized. The algorithm maintains a lightweight structure throughout.

A loose upper bound for the full braiding process is: $[n_moves_vogel*((|max_edge|+ 2 * n_moves_vogel) + (|max_color| + 2 * n_moves_vogel))] =$

$[n_moves_vogel * (4 * n_moves_vogel + |max_edge| + |max_color|)] <=< br > C * n^2 * (n^2 + |max_edge| + |max_color|)$

What gives $O(n^2 * (n^2 + |max_edge| + |max_color|))$;

The final step of the algorithm is reading a braid word from the chain form of the graph, described in the next section.

[**PLACEHOLDER**: Box for new findings]

3.3 Reading Braid Word

To construct the final braid word, we traverse the graph from bottom to top, processing each edge in order and dynamically building a sequence of crossing pointers. Once this sequence is established, it is transcribed into a word composed of sigma-generators with exponents of +1 or -1 — representing the final braid output.

The procedure goes as follows:

For each edge, we iterate through its crossings one by one and insert them into the developing sequence at the appropriate position. A dynamic pointer, $'braid_prev_cx'$, is maintained throughout to indicate the current position in the

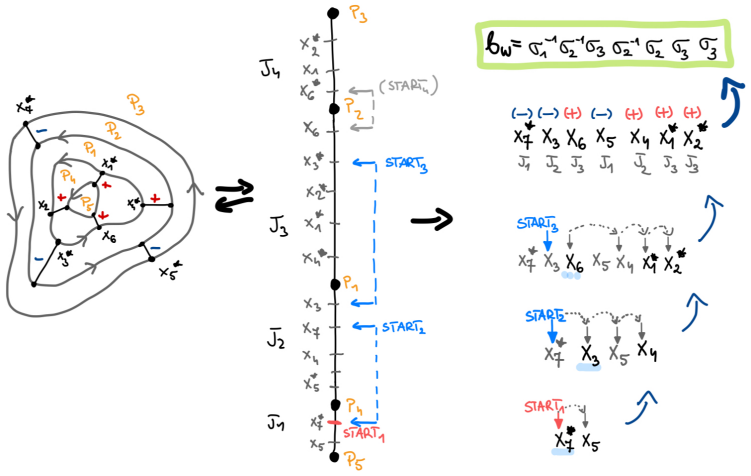


Figure 17: Obtaining crossing sequence from enriched graph and generating final braid word.

sequence — the crossing after which the next one should be inserted. Technically, the sequence is encoded as a map of the form Crossing*: Crossing*, where each key is a crossing and its corresponding value is the subsequent crossing in the current braid sequence.

The traversal begins at the bottom edge of the graph. This edge can be identified as the one whose index appears in the $left_edges$ dictionary of a Color object with an empty $right_edges$. Symmetrically, the top edge is the one listed in the $right_edges$ of a Color object with an empty $left_edges$.

We initialize the sequence with a crossing from the bottom edge (chosen arbitrarily) and set $braid_prev_cx$ to point to it. As the traversal proceeds, we handle each crossing as follows: - If the crossing is the first of its conjugate pair (i.e., its conjugate lies on the edge above): - Insert the crossing immediately after $braid_prev_cx$ in the sequence. - If this is the first such crossing with a conjugate on the upper edge, record its conjugate as the starting crossing for that next edge. - Update $braid_prev_cx$ to the newly inserted crossing. - If its conjugate has already been processed (i.e., it lies on a lower edge and is already part of the sequence): - Simply update $braid_prev_cx$ to that conjugate's position in the sequence. After all crossings on the current edge are handled, we move to the starting crossing of the next edge and repeat the process.

The traversal ends once the penultimate $n-1$-th edge is fully processed — since all crossings on the final $(n-th)$ edge have their conjugates on the previous edge, they will already be present in the sequence.

Once the sequence of crossings is established, we generate the final braid word as a string. For each crossing in the sequence, we put a generator of the form: $sigma_k^(d)$), where k is the index of the edge on which the crossing lies (this is always index of the lower edge from conjugate pair, from construction) and d

is the sign of the crossing: $+1$ or -1.

The resulting string — a sequence of such $signa_k^+ - 1$ terms — represents the final braid word.

Note: [[braid is not unique] [braid and markov moves] [+- signs are assigned arbitrary] [possible impact on training machine learning models]]

4 Experiments

TBA

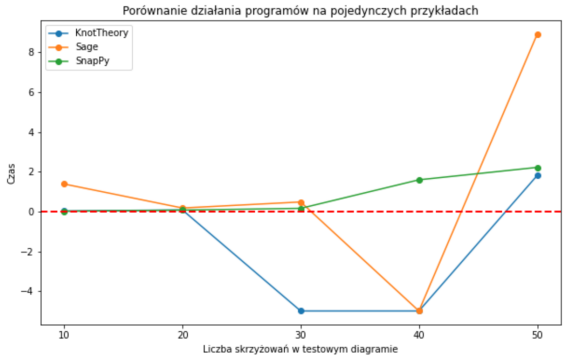


Figure 18: Results of a test comparison of the execution times of existing programs on individual PD-code files containing between 10 and 50 crossings. Values of -5 indicate that the program failed to return a braid word due to an error or exceeding the time limit of twenty minutes.

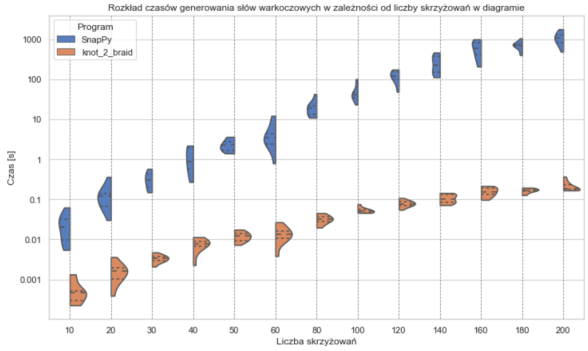


Figure 19: Braid word generation times for the Snappy and knot-2-braid programs. For each specified number of crossings, generation times were measured using ten files, each containing the PD-code encoding of a knot diagram with that number of crossings.

5 Conclusions

TBA