LONDON'S GLOBAL UNIVERSITY

UCL

A Compiler for String Diagrams

Akkaraphonphan Tai[1]

BSc Computing Studies

Supervisors: Fabio Zanasi, Paul Wilson

Submission date: 3 May 2021

[1] **Disclaimer:** This report is submitted as part requirement for the BSc Degree in Computing Studies at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged

Abstract

We introduce a tool that allows users to create and edit string diagrams of symmetric monoidal categories, and translate them to their textual representation or vice versa. Before this project, there has been no tool that supports drawing a string diagram and then translating it to a textual representation and vice versa, as there is a challenge of how a string diagram can be represented with multiple different textual representations. Some of the additional features the tool provides includes a GUI and a way to export translated string diagrams. Development was carried out on an incremental model, starting from core features and building up more features from the previous stage.

Contents

Chapter 1

Introduction

1.1 Problem Outline

String Diagrams are used in a branch of mathematics called Category Theory, and they can be thought of as a language to describe or model various concepts or structures such as circuits. String diagrams are represented graphically with boxes and wires, where basic "atomic" building blocks called "generators" can be combined together to create more complex string diagrams. Below is an example of a string diagram that goes from left-to-right. (It is acceptable to draw string diagrams in other directions, such as from top-to-bottom.)

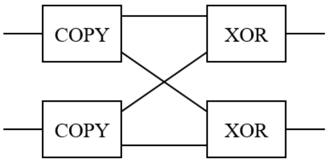


Figure 1.1: A string diagram representing a circuit, specifically a half adder, formed from four generators.

On the other hand, string diagrams can be represented textually with a string of letters and symbols. The example above may be written as $(COPY \otimes COPY); (id \otimes twist \otimes id); (XOR \otimes XOR)$. There will be more details on how a textual representation is created in the next chapter. Currently there exists no tool to convert a string diagram represented graphically to a textual representation, or vice versa. The goal of this project is to create a tool that can provide functionalities that can address this issue.

1.2 Why is it difficult?

There are many different types of categories, and the properties of a string diagram can change depending on the category the string diagram is representing. For example, string diagrams of a "symmetric monoidal category" allows wires to "twist" by crossing over each other, such as Figure 1.1, whereas in a regular "monoidal category", this property is not allowed. As there are many types of categories, a tool that can adapt depending on the category it is representing is very difficult to create, and therefore the scope of the project had to be lowered to cover only string diagrams in symmetric monoidal categories. Formal definitions and further information of these terms will be covered in the next chapter.

String diagrams can also be represented by many different but equivalent textual representations. As a result of this property, as long as a string representing a diagram is valid, they can be written in many different ways without following a general syntax or structure. Therefore, the tool will require an ability to adapt to different kinds of inputs fed by a user, as it cannot expect an input to follow a specific form. A more detailed explanation of this property will be explained in later chapters.

1.3 Why is it worthwhile?

As string diagrams are a graphical representation, they allow humans to grasp a very intuitive understanding of the meaning of the diagram. Having a tool to convert the different representations of a string diagram allows for a more convenient and easier approach to string diagrams without the user having to fully understand the rules required for a conversion.

On the other hand, it may be more appropriate for some people to use the textual representation of a string diagram. Having a feature to convert the graphical representation to a textual representation provides a way for any string diagram to conform to the format required in a specific situation.

Having a textual representation of a string diagram provides a syntax of sorts for string diagrams, however it can be difficult to visualise the structure of the string diagram, as a textual representation is one-dimensional, whereas a graphical representation is a two-dimensional diagram that is much easier to visualise. Therefore, it would be worthwhile to create this tool.

1.4 Why are String Diagrams useful?

String diagrams are becoming more and more important in Computer Science. They are useful due to their versatile properties to represent almost any system or structure. For example, a box in a string diagram can be considered as a "process" and wires representing the inputs and outputs of each process, where the number of wires describes the number of inputs and outputs. This therefore creates a "system". The fact that string diagrams have compositionality means that a system can be built by combining smaller, simpler systems together, and analysis of the system can be performed by looking at each part of the system. Therefore, the ability for each box to be named, joined together or formed in parallel also allows the ability to model and describe many different concepts or processes, from cooking recipes, chemical reactions to circuits.[4] Recently, string diagrams has been used to model concepts described below:

1. **Signal flow graphs** - Signal flow graphs are used to represent equations or circuits in Control Theory. There are many ways to represent these equations, and string diagrams of symmetric monoidal categories has recently been used to relate them against traditional signal flow graphs. A simple example is a signal amplification operation of $f \to cf$, where a signal f is amplified by a constant c. This can be represented by the string diagram below.[2]

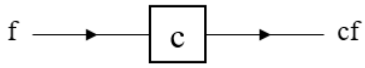


Figure 1.2: A string diagram representing signal amplification.

2. **Concurrency** - String diagrams can be used to model concurrent systems. In concurrency, race conditions can occur, where two or more threads running at the same time can unexpectedly change the behaviour of a system. Part of why this is a problem is that certain resources in a system are sensitive to when their values are changed. For example, with traditional algebraic theories, resources are allowed to be copied (duplicated) or discarded (deleted) implicitly without any consequences. String diagrams on the other hand are a "resource-sensitive" syntax that considers the resource aspect of concurrency more carefully, because if resources need to be "copied" or "discarded", it is required to explicitly create generators that perform these functions as shown in the example below.

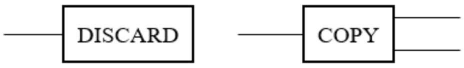
   

   Figure 1.3: Generators that explicitly perform the discard and copy operations.

   Having these resource sensitive operations be explicitly declared allows for a clearer description of a system when modelled by string diagrams, and therefore can be used in this field in the form of "Petri Nets".[3]

3. **Quantum circuits** - Quantum circuits are used to model quantum computation, a way to perform calculations using phenomena such as superposition and entanglement. As a result of these properties, quantum circuits are naturally very complicated. String diagrams of compact closed categories (where wires can turn back to form "cups" and "caps") are used to model quantum circuits. For example, the "ZX-calculus" is a string diagrammatic representation of quantum circuits with an associated rewriting system which can be used to simplify them. In turn, this simplifies the quantum circuit represented by the string diagram. This enables the circuit to be easier to read and be built by humans.[9]

1.5 Project Aims and Goals

A list of "Aims" and "Goals" were created before the project was undertaken. For this report, an "Aim" is considered something that we hope to achieve, and a "Goal" is considered something we hope to deliver.

1.5.1 Aims

1. To understand the various motivations and contexts behind the development of the tool.
2. To research any existing tools relevant to the project such that any useful or missing features can be addressed.
3. To learn about any relevant libraries that could potentially be involved in the creation and development of the tool. For example, NumPy (data structures) and tkinter (GUI development)
4. To understand the various properties and behaviours of the operations involved for string diagrams in symmetric monoidal categories.
5. To develop algorithms for converting string diagrams from one representation to another, to be used in the development of the tool.

1.5.2 Goals

1. To develop a functioning tool that allows the user to write their own string diagram and convert it to the other representation. The tool should work for string diagrams in symmetric monoidal categories.
2. To develop a GUI for the tool. When converting graphical representations to a textual representation, it is much easier for users to work in a GUI environment.
3. To develop any quality of life improvements to the tool once the above goals has been achieved. For example, this may include documentation or a way to export the converted string diagram.

1.6 Project Development

Development of the tool was divided into small steps:

1. Designing a GUI based on requirements.
2. Implementing placeable generators and wires.
3. Implementing an algorithm for converting a graphical representation to a textual representation.
4. Implementing an algorithm for converting a textual representation to a graphical representation.
5. Implementing any additional features.

Development of this tool was based on an incremental model. This is where a tool with basic functionality was developed first, then slowly more and more features are added until a final product was achieved. Each step was tested and the tool needed to function properly before moving on to the next step. This is to ensure that any flaws in the tool could be quickly identified and fixed, as it is much harder to fix problems in the tool if discovered at a later stage.

1.7 Overview of Chapters

In the following chapters, this report will provide more details about the different parts of the project. Chapter 2 provides background information on the concepts related to string diagrams to give an deeper understanding of the theory behind them, and also provides information about related works to this project. Chapter 3 elaborates more about the requirements that this project must, should or could cover. Chapter 4 gives details on how the tool was developed and implemented. Chapter 5 provides details on how the tool was tested and gives an evaluation based on the results. Finally, chapter 6 gives a summary of what was achieved and any future work that could be done on the tool.

Chapter 2

Context

In this chapter, we will first present basic definitions in Category Theory to provide background information for the project.

2.1 Introduction to Category Theory

Category Theory is an abstract branch of mathematics that deals with the description of mathematical structures such as groups and rings.[13] One of the main concepts introduced in this field is called a category. A category has a collection of "objects", and for each pair of objects, there exists a collection of arrows called "morphisms" between them. Every object and morphism needs to be labelled, and each morphism needs to have a direction. Morphisms are able to be composed, for example if we have $f : X \to Y$ and $g : Y \to Z$, then a morphism of $g \circ f : X \to Z$ is produced.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\
 & Z
\end{tikzcd}$$

Figure 2.1: (From [5]) A graphical representation of a composition of f and g.

Every category must have these two properties:

1. **Associative** - Rearranging brackets do not affect the meaning of the expression. For example $(f \circ g) \circ h = f \circ (g \circ h)$
2. **Contain identities** - Each object must have an identity morphism tied to it. For example, an object X in a category has an identity morphism $id_x : X \to X$. Diagrammatically, this can be represented as an arrow going from X to itself.

DEFINITION 1 *A category* ***C*** *consists of[17]:*

- *a set of objects,* $\boldsymbol{C} = \{A, B, C, ...\}$*:*

- *each pair of objects must have a set* $hom_C(A, B)$ *of morphisms,* $f : A \rightarrow B$*;*

- *every object must have an identity morphism* $id_A : A \rightarrow A$

- *every trio of objects must have morphisms that are able to be composed: if* $f : A \rightarrow B$ *and* $g : B \rightarrow C$*, then* $g \circ f : A \rightarrow C$*,*

such that for all $f : A \rightarrow B$*,* $g : B \rightarrow C$*, and* $h : C \rightarrow D$ *the following are satisfied:*

- $id_B \circ f = f$

- $f \circ id_A = f$

- $(h \circ g) \circ f = h \circ (g \circ f)$

A category also allows for a "functor" - a morphism from one category to another. A functor preserves composition and identity morphisms.[14]

DEFINITION 2 *A functor* F *from a category* C *to a category* D *is a morphism such that any object* $x \in C$ *is mapped to an object* $F(x) \in D$*, and each morphism* $f : x \rightarrow y$ *in* C *is mapped to a morphism* $F(f) : F(x) \rightarrow F(y)$ *in* D*, and the following conditions must hold[14]:*

- $F(g \circ f) = F(g) \circ F(f)$ *for every morphism* $f : X \rightarrow Y$ *and* $g : Y \rightarrow Z$ *in* C*, preserving compositions.*

- *Every object* $x \in C$*,* $F(id_x) = id_{F(x)}$*, preserving identities.*

2.2 Monoidal Categories

A Monoidal Category shares the same properties as a Category, but with an additional "Tensor Product" which is a functor.

An example of a Tensor Product being used is in a bilinear morphism.[16] If $f : W \rightarrow Y$ and $g : X \rightarrow Z$, then $f \otimes g : W \otimes X \rightarrow Y \otimes Z$. It's also worth noting that the Tensor Product is not commutative, so the position of the object is important.[17]

DEFINITION 3 *A monoidal category* $\boldsymbol{C}$ *is a category that also consists of[15]:*

- *a functor called the Tensor Product,* $\otimes : C \times C \rightarrow C$

- *an unit object* $I \in C$*;*

- *a natural isomorphism* $\alpha_{A,B,C} : (A \otimes B) \otimes C \overset{\approx}{\Rightarrow} A \otimes (B \otimes C)$

- *a natural isomorphism* $\lambda_A : I \otimes A \overset{\approx}{\Rightarrow} A$

- *a natural isomorphism* $\rho_A : A \otimes I \overset{\approx}{\Rightarrow} A$

such that:

- *for all $A, B \in \boldsymbol{C}$, a commutative diagram called the Triangle Identity is created:*

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "1_A \otimes \lambda_B"'] & & (A \otimes I) \otimes B \arrow[dl, "\rho_A \otimes 1_B"] \\
& A \otimes B &
\end{tikzcd}$$

Figure 2.2: (From [6]) The Triangle Identity (or equation)

- *for all $A, B, C, D \in \boldsymbol{C}$, a commutative diagram called the Pentagon Identity is created:*

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "\alpha_{A,B,C\otimes D}"] \arrow[d, "1_A \otimes \alpha_{B,C,D}"'] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "\alpha_{A\otimes B,C,D}"] & ((A \otimes B) \otimes C) \otimes D \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "\alpha_{A,B\otimes C,D}"'] & & (A \otimes (B \otimes C)) \otimes D \arrow[u, "\alpha_{A,B,C} \otimes 1_D"']
\end{tikzcd}$$

Figure 2.3: (From [8]) The Pentagon Identity (or equation)

A commutative diagram shows you can go from one point to another in different ways, and each path taken is considered equal as long as the starting point and end point are the same.

An isomorphism is when a morphism has an inverse. For example, $f : X \to Y$ is isomorphic if there exists a morphism $g : Y \to X$ such that $g \circ f = id_X$ and $f \circ g = id_Y$. [1]

An natural isomorphism is a special type of natural transformation where the morphism between two functors also contains an inverse.

2.3 String diagrams

A String Diagram is a way to represent morphisms of a category. They provide a visual intuition with wires representing objects, and boxes representing morphisms. Composition is represented by "joining" morphisms together from left to right.[17]

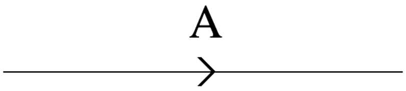


Figure 2.4: A string diagram representing object A.

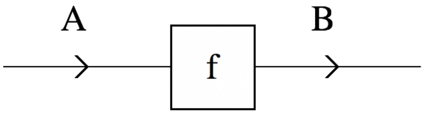


Figure 2.5: A string diagram representing morphism $f : A \to B$.

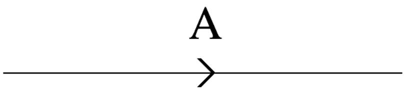


Figure 2.6: A string diagram representing identity $id_A : A \to A$.

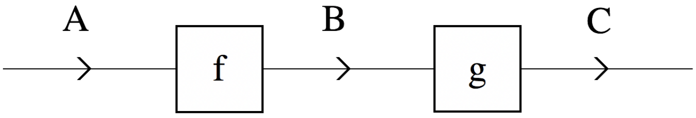


Figure 2.7: A string diagram representing composition $g \circ f : A \to C$.

Definitions of string diagrams may vary depending on which category they represent. A monoidal category for example introduces a functor called a “Tensor Product” which cannot be used in a normal (plain) category as this functor is not defined. Graphically, tensor products are represented by drawing the objects and morphisms in parallel.

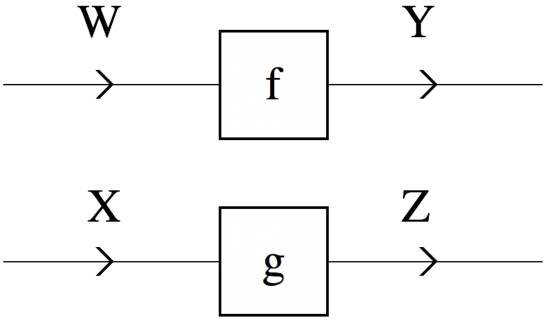


Figure 2.8: Tensor product $f \otimes g : W \otimes X \to Y \otimes Z$.

2.4 Braided Monoidal Categories

Information on a braided monoidal category is provided because a braided monoidal category is a more general structure of a symmetric monoidal category (this category will be explained in the next section), and therefore it is worth defining it first to help provide an easier transition to a symmetric monoidal category.

A Braided Monoidal Category is a Monoidal Category with an additional feature called the “braiding”. This allows objects to be swapped, such as from $X \otimes Y$ to $Y \otimes X$, as for every pair $X \otimes Y$, the braiding is an isomorphism to a pair $Y \otimes X$[12]

DEFINITION 4 *A braided monoidal category* $\boldsymbol{C}$ *is a monoidal category that also consists of a natural isomorphism* $b_{X,Y} : X \otimes Y \to Y \otimes X$ *called the braiding, such that for all objects* $X, Y, Z \in \boldsymbol{C}$*, two commutative diagrams called the Hexagon Identities are created[1]:*

$$\begin{tikzcd}
X \otimes (Y \otimes Z) \arrow[r, "a^{-1}_{X,Y,Z}"] \arrow[d, "b_{X,Y\otimes Z}"'] & (X \otimes Y) \otimes Z \arrow[r, "b_{X,Y}\otimes 1_Z"] & (Y \otimes X) \otimes Z \arrow[d, "a_{Y,X,Z}"] \\
(Y \otimes Z) \otimes X & Y \otimes (Z \otimes X) \arrow[l, "a^{-1}_{Y,Z,X}"] & Y \otimes (X \otimes Z) \arrow[l, "1_Y\otimes b_{X,Z}"]
\end{tikzcd}$$

Figure 2.9: (From [1]) First Hexagon Identity (or equation)

$$\begin{tikzcd}
(X \otimes Y) \otimes Z \arrow[r, "a_{X,Y,Z}"] \arrow[d, "b_{X\otimes Y,Z}"'] & X \otimes (Y \otimes Z) \arrow[r, "1_X\otimes b_{Y,Z}"] & X \otimes (Z \otimes Y) \arrow[d, "a^{-1}_{X,Z,Y}"] \\
Z \otimes (X \otimes Y) & (Z \otimes X) \otimes Y \arrow[l, "a_{Z,X,Y}"] & (X \otimes Z) \otimes Y \arrow[l, "b_{X,Z}\otimes 1_Y"]
\end{tikzcd}$$

Figure 2.10: (From [1]) Second Hexagon Identity (or equation)

The first identity shows that swapping the positions of X and $Y \otimes Z$ by moving the brackets and swapping X to Y and then to Z is equivalent to swapping X and $Y \otimes Z$ together directly. The second identity demonstrates the same property except it involves swapping $X \otimes Y$ and Z.[1]

The braiding term can be visualised as a pair of crossed wires shown below:

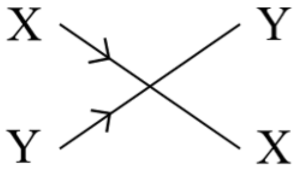


Figure 2.11: A visual representation of braiding $b_{X,Y} : X \otimes Y \to Y \otimes X$

Another property of a braided monoidal category can be shown in this diagram, for morphisms $f : X \to X'$ and $g : Y \to Y'$:

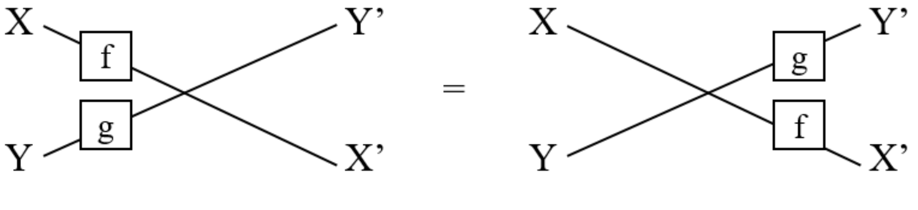


Figure 2.12: A property as a result of the braiding.

This property is a result of the naturality of the braiding, meaning that you can ”slide” boxes along the wires. This is because the braiding is a natural isomorphism, and from the naturality condition:

$$\begin{tikzcd}
X \arrow[d, "f"'] & F(X) \arrow[r, "\eta_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\
Y & F(Y) \arrow[r, "\eta_Y"'] & G(Y)
\end{tikzcd}$$

Figure 2.13: (From [7]) The naturality condition described by a commutative diagram.

If the braiding is a natural transformation, it needs to go between two functors F and G. If F is an identity and G is a swap, where $G(A \otimes B) = B \otimes A$, then $G(f \otimes g) = g \otimes f$ and it therefore meets the naturality condition.

2.5 Symmetric Monoidal Categories

A Symmetric Monoidal Category is a special case of a Braided Monoidal Category where swapping objects twice is equivalent to not swapping the objects in the first place.[1] Unlike a Braided Monoidal Category, it sometimes might not be the case where swapping objects twice gives the same result. A morphism in a Symmetric Monoidal Category can be depicted visually in the following diagram:

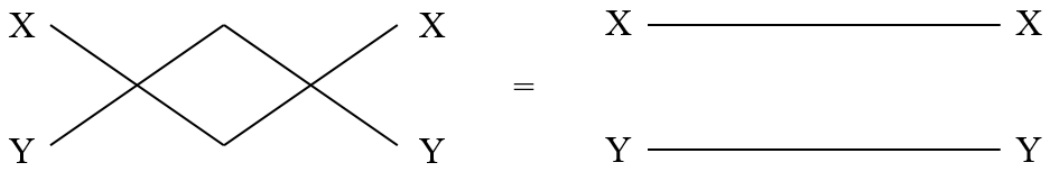


Figure 2.14: A visual intuition of a morphism in a Symmetric Monoidal Category.

DEFINITION 5 *A symmetric monoidal category* $\boldsymbol{C}$ *is a braided monoidal category where for all objects* $X, Y \in \boldsymbol{C}$*, the braiding satisfies the property[1]:* $b_{X,Y} = b^{-1}_{Y,X}$

2.6 String Diagrams of Symmetric Monoidal Categories

This chapter started from a basic definition of a category, and worked its way up to a definition of symmetric monoidal category, which is what string diagrams in this report intends to cover. For this report, the composition of morphisms f from $A \rightarrow B$ with g from $B \rightarrow C$ is in diagrammatic order (reading a diagram from left to right) as $f; g$ to make reading the diagrams slightly more intuitive.

2.6.1 Rules of String Diagrams in symmetric monoidal categories

1. **String diagrams can be constructed with two operations.** - The two operations are the Tensor Product and composition, with symbols $\otimes$ and ; respectively. The Tensor Product

"stacks" objects on top of each other and the composition operation "links" objects together.

2. **Wires cannot "turn back".** - Although having a wire that acts as a "feedback loop" is allowed in traced monoidal categories, or having wires that act as "cups" and "caps" are allowed in a compact closed categories, having these kind of wires are not allowed in symmetric monoidal categories.

3. **Wires can cross** - As a result of the property of the symmetric monoidal category, having the twist morphism to represent overlapping wires to help construct more advanced string diagrams is allowed.

4. **Boxes can slide along wires** - This property is explained in more detail in section 2.4, and as a result diagrams such as the one below are considered equal up to the laws of symmetric monoidal categories.

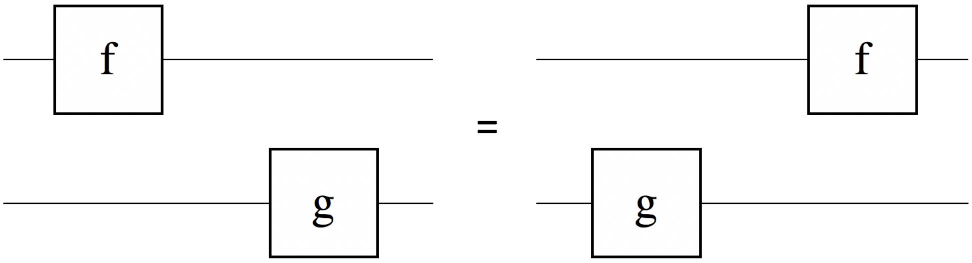


Figure 2.15: An example of the slide property.

2.6.2 More complicated string diagrams

As string diagrams are created from smaller building blocks, it is natural for some of them to become very complex if they are formed from many building blocks. Consider the diagram below. How could we get a textual equivalent of this diagram?

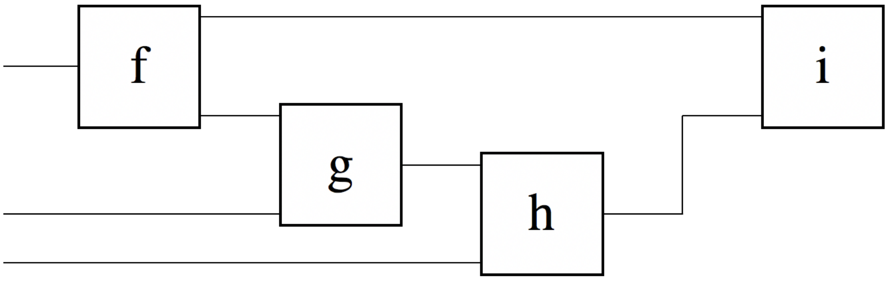


Figure 2.16: A more complex diagram

In this case, the diagram can be split vertically into smaller and more simpler sections:

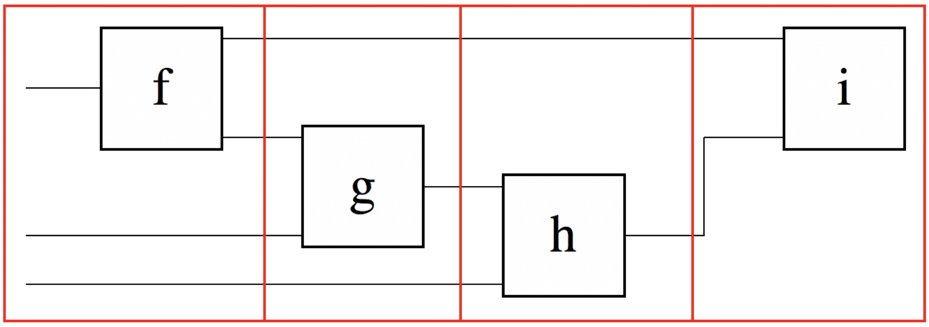


Figure 2.17: The diagram split into smaller vertical sections

Then, it is much easier to give a textual representation for each section and then combining them back into the whole diagram. In this case from left to right:

Section 1 - $f \otimes id \otimes id$

Section 2 - $id \otimes g \otimes id$

Section 3 - $id \otimes h$

Section 4 - i

Combining each section with compositions will result in $(f \otimes id \otimes id);(id \otimes g \otimes id);(id \otimes h);i$.

2.6.3 String Diagrams and other concepts

String Diagrams can be related to many of the concepts discussed in this chapter. For example:

- A functor on a String Diagram can be thought of as mapping one syntax to another. If a textual representation is a type of syntax for morphisms of a category, and we have the representation $f;g$, then it is able to be mapped to the diagram in Figure 2.14, by mapping all the objects and then composing them together, as this is a property of a functor.

- In a monoidal category, the tensor product can be used as a way to "stack" morphisms together, and composition is used to "link" morphisms together.

- In a symmetric monoidal category, a new syntactic "building block" is introduced as a result of the braiding property of the category. In this case, the braiding is called a "twist".

2.7 Why are String Diagrams useful?

String Diagrams when represented graphically are a very useful tool to help provide a visual intuition of a morphism in a category, especially if the term is very complex. Consider the following diagram:

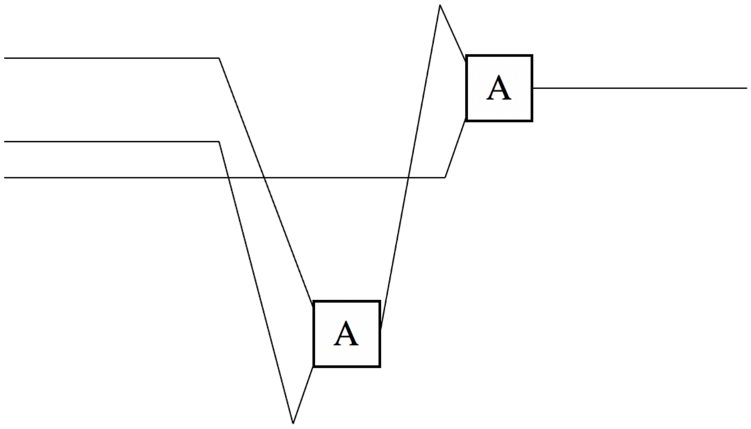


Figure 2.18: Example of a string diagram with overlapping wires.

This String Diagram represents the following term in textual form: $(id * twist); (twist * id); (id * A); twist; A$. However, by looking at this diagram, it is very intuitive to "slide" the bottom morphism up until there are no crossing wires:

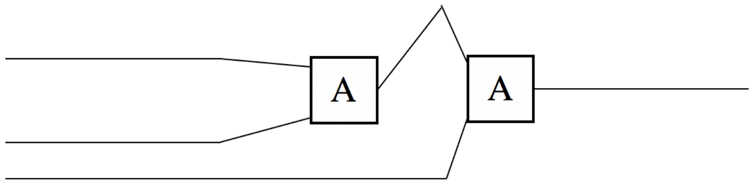


Figure 2.19: The same string diagram from Figure 2.15 with no overlapping wires.

Now, the String Diagram represents $(A * id); A$. In fact, these two terms are equivalent to each other. Without the String Diagram, it would be necessary to apply the axioms of symmetric monoidal categories to prove these two terms are equivalent, but with the help of a String Diagram, it allows the proof to be completed almost instantly from looking at the diagram. Therefore, String Diagrams are a very useful tool for mathematicians.

2.8 Similar Projects

- **Cartographer** - An open source tool used for manipulating graphical string diagrams of symmetric monoidal categories. It features proof assistance using a "rewrite" system that helps simplify or equate string diagrams. Users can use the tool to help develop proofs of their own theorems without directly using the axioms of symmetric monoidal categories.

  Users are able to create their own generators and wires and placing them on a grid. Generators and wires can also be removed in case the user makes a mistake. Additionally, the tool also features a "Rules" section where users can create their own theories and rewrite rules, and then demonstrating a proof based on the theories created by clicking on parts of a string diagram to "simplify" the diagram by having the tool rewrite the part selected.[18]

  - Open Source

  - Proof Assistant
  - Manipulate diagrams
  - Web-based
  - Specialized to Symmetric Monoidal Categories
  - Generators may be moved

- **DisCoPy** - An open source tool able to generate string diagrams from user input. Users can name generators with any arbitrary numbers of inputs and outputs, producing string diagrams ranging from recipes to performing a "Bell Test". Unlike Cartographer, string diagrams in DisCoPy are top to bottom instead of left to right. Another difference is that diagrams produced have a "fixed" format, where users are unable to "move" generators or wires according to their preferences. DisCoPy also has the ability to simplify diagrams visually by providing a .gif image where the string diagram can be rewritten step by step..[10]
  - Open Source
  - Diagram rewriting
  - No GUI
  - Desktop-based
  - Covers Monoidal Categories
  - Generate string diagrams
  - Diagrams are exported to a file location specified
- **Catlab.jl** - An open source library that can be used for applications of category theory. Like DisCoPy, it is able to draw string diagrams from user input. However, it does not provide proof assistance or a GUI. It is currently in development at the time of writing.[11]
  - Open Source
  - Programming library
  - Desktop-based
  - Generate string diagrams
  - Covers Monoidal categories

From these tools listed above, it is apparent that they all provide a means for the user to draw a string diagram. In Cartographer, diagram are visually created by placing objects on a grid. In DisCoPy and Catlab.jl, a string diagram has to be typed in text and an output image of a string diagram is created. The tool we have created will be similar to the method described in Cartographer, where a GUI will be provided and users can place generators and wires on a grid, allowing them to have freedom on where exactly they want to place or arrange objects.

Tools such as DisCoPy are able to turn a textual input to a string diagram visually, although there is no option to perform this process the other way round. This is one of the features that our tool has implemented and hopes to fill in this missing gap.

Chapter 3

Requirements and Analysis

3.1 Detailed Problem Statement

As previously stated in the Introduction Chapter, the goal of this project is to design a tool where users can create a string diagram, and be able to get the tool to convert it to the other representation. The tool should work for string diagrams in symmetric monoidal categories. However, one of the main reasons why creating such a tool is difficult is that the correspondence between a graphical representation to a textual representation is not 1:1. Consider this following diagram:

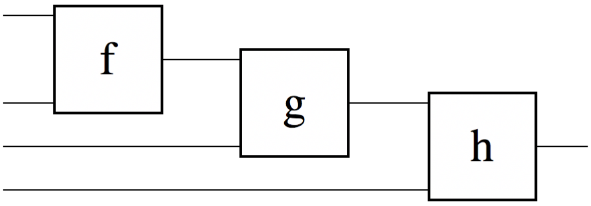


Figure 3.1: An example string diagram

In order to turn this into a textual representation, this diagram can be split into smaller parts such as the one below:

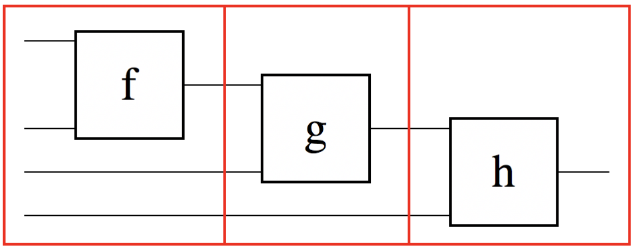


Figure 3.2: The diagram split into three vertical sections

By applying the rules and operations of the tensor product and composition, this diagram can be read as $(f \otimes id \otimes id); (g \otimes id); h$. However, this diagram could have been split in a different way:

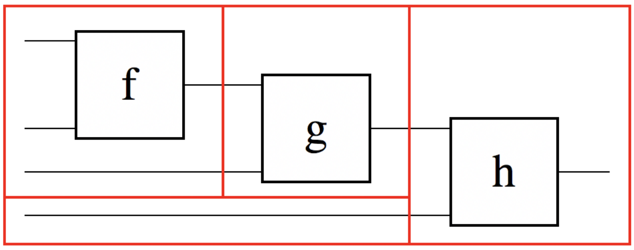


Figure 3.3: The diagram split into four different sections

Now, by considering each section, the diagram is now read as $(((f \otimes id); g) \otimes id); h$. Therefore, the same diagram can be written as many different but equivalent textual representations.

3.2 Requirements

Most of the requirements gathered were taken from the problem statement and the research undertaken for this project. The MoSCoW method (a method to structure a list of requirements) was used to help create and classify each requirement with respect to priority.

3.2.1 Must have

The tool must have this requirement or the whole project will be considered a failure.

- Developing a functioning tool that can take in one representation (it can be either graphical or textual) and is able to translate the string diagram to the other representation.

3.2.2 Should have

The tool should have these requirements but it is not necessary.

- Developing a GUI for the tool.
- Developing a functioning tool that can take in both representations and is able to translate the string diagram to the other representation.
- Including documentation for the tool.
- Developing the tool to be able to take in a textual representation of any format. (e.g. The tool should be able to understand and translate $(((f \otimes id); g) \otimes id); h$ instead of having to force the format of $(f \otimes id \otimes id); (g \otimes id); h)$

3.2.3 Could have

The tool could have these requirements as it can improve quality of life, and it will only be implemented if there is enough time.

- Developing a way for the user to export the translated string diagram. (Either textual, or as a picture if the output is a graphical representation).
- Developing a way for the user to "save" their current progress and be able to import their string diagram at a later date.

- Developing a way for the user to ”factorise” the graphical representation of the string diagram to obtain a textual representation in the format of their own preferences (as more than one term corresponds to the same diagram).
- Developing the tool to recognize and give error messages to string diagrams that are not drawn properly in the graphical representation.

3.2.4 Won’t have

The tool will not have this requirement given the time frame of the project.

- Accommodating different kinds of string diagrams such as traced monoidal categories or compact closed monoidal categories.

3.3 Analysis

From the list of requirements, the tool heavily involves the usage of data structures. Therefore, the tool will be programmed in Python with NumPy, a linear algebraic Python library that helps with graphs.

Many of the requirements require the user to perform complicated actions, so developing a GUI for the tool will be very beneficial for the user experience. The tkinter library will be used in the creation and development of the GUI for the tool.

Chapter 4

Design and Implementation

4.1 Graphical representation to a textual representation

Due to the nature of the project and from the requirements created, the tool will be designed with a GUI in mind, as it provides the user with much more freedom and intuitiveness compared to a command line tool.

As this tool is designed to be used on a computer, typing out the symbol for the tensor product ($\otimes$) will be difficult. Therefore, the symbol of the tensor product will be represented in this tool using the asterisk symbol $*$.

A general scenario was devised, describing what a user operating this tool would do from start to finish:

1. User creates a generator
2. User places said generator in a specific position in the grid.
3. Repeat until all generators are placed.
4. If relevant, twist morphisms are created on the grid by intersecting wires with each other.
5. Generators are connected with wires
6. The tool reads the graph and provides a textual output to the user.

From this scenario, the GUI was designed with the following features:

4.1.1 A grid to place generators

Each square of the grid represents a space to hold either a generator or a wire. The grid provided is 20x20, a reasonable size to support most string diagrams. This is created by an array filled with zeros using the NumPy library. Information of the diagram is partially kept in this array, such that wires are stored with value -1, generators are stored in numerical order $1, 2, .., n$ depending on the order of creation, and twists are stored with value -2.

4.1.2 A cursor

Instead of using the default arrow cursor provided by default from the operating system, the tool's cursor is a red circle. This was created because the process of drawing a string diagram can be very precise, especially when dealing with wires and it can sometimes be unclear for the user to know what position the cursor is pointing at with an arrow cursor. Having a circular cursor eliminates this ambiguity for the user.

4.1.3 A way to create and place generators

When creating an generator, the user needs to give it a name, and specify the number of inputs and outputs it has. Although a generator can have any finite number of inputs and outputs, given that the tool only provides a 20x20 grid, any number of inputs or outputs greater than 20 will not be allowed and will give the user an error message. The user can also assign a colour of either black or white for the generator to account for situations where the string diagram drawn contains many different generators to better distinguish them from one another. Inputs are drawn on the left side of the generator, and outputs are drawn on the right side.

When the user clicks on a square in the grid, a dialog box will appear, allowing the user to fill in the information required. Once done, the generator will be created based on the position the user clicked.

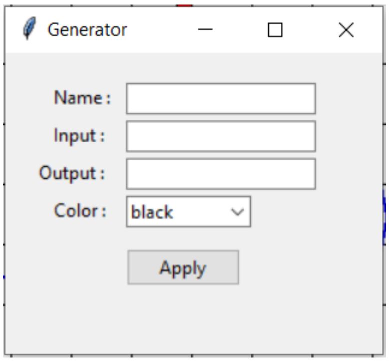


Figure 4.1: Dialog box created whenever a user clicks on a position on the grid.

The tool also contains a check of the user inputs in the dialog box. Inputting abnormal data such as text in the number of inputs or outputs will be rejected. Leaving any of the boxes blank will also be rejected by the tool.

The appearance of the generator depends on the number of inputs and outputs it has. As any square of the grid can only occupy one wire, and therefore the amount of space the generator will occupy vertically will depend on the highest number of inputs or outputs it has.

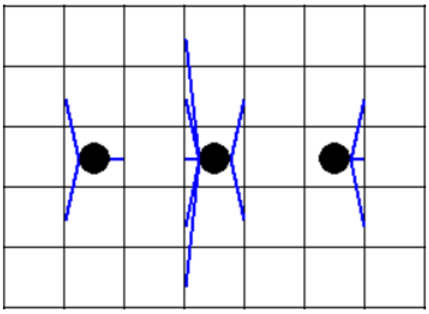


Figure 4.2: Generators occupy different heights depending on the highest number of inputs.

Therefore, if a generator's highest number of inputs or outputs is n, if n is an even number, it will occupy $n/2$ spaces above and below the square containing the circle. If n is an odd number, it will occupy $(n-1)/2$ spaces above and below.

Each generator's information is stored in a text file called *create_gt.txt*. Each line stores information about a generator in the following format: [column number],[row number],[x-coordinate],[y-coordinate],[number of inputs],[number of outputs],[colour],[name]. The reason for storing the x and y coordinates stem from a need to store a more accurate position of each generator and it's inputs and outputs. When an generator is created, its information will be added to the text file and will be drawn on the grid.

4.1.4 A drop down menu

At this point, it was apparent that there are many features that needs to be implemented for the tool. Therefore, a drop down menu that stores most of the functionality in the tool was added to the GUI. Users can access this drop down menu by clicking on the "Action" button on the top left of the GUI.

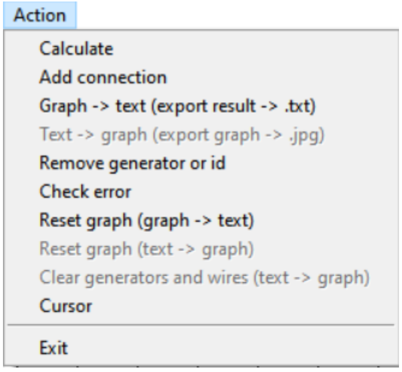

Action
Calculate
Add connection
Graph -> text (export result -> .txt)
Text -> graph (export graph -> .jpg)
Remove generator or id
Check error
Reset graph (graph -> text)
Reset graph (text -> graph)
Clear generators and wires (text -> graph)
Cursor
Exit

Figure 4.3: The action menu in Graph → Text mode.

However, some of the options in the menu are exclusive depending on the mode the user is on. To prevent any errors or confusion, options that the user is not currently allowed to select are greyed out.

4.1.5 A way to create wires

Once all the generators are in place, the next step for the user is to connect them with wires. Wires in this tool are always straight lines, and are created by clicking any position on the grid to start the wire and clicking another position to join the wire from the previous point to the current point clicked. This behaviour of clicking a new position to connect the wire allows the user to click more than twice to "shape" the wire, but is only used practically for more complicated diagrams to make the diagram look nicer. To start a new wire, the user would have to press a button to notify the tool to start a different wire from a new point without connecting it to the previous point.

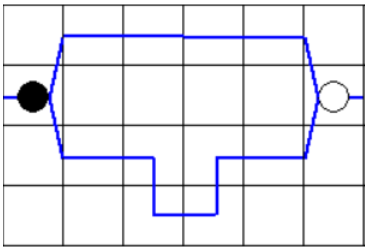


Figure 4.4: Two different but equivalent ways to connect wires.

As creating wires involves left clicking with the mouse, in the perspective of the tool, there is currently no way for it to know whether a left click of the mouse would create a wire or a generator. Therefore, the tool needs to be aware of when the user wants to place a generator, or when to create wires. This was part of the reason a drop down menu was added so the user can "switch" between the two "states" by going to the menu and selecting if the user wants to create a wire or an generator. These two modes are called *Add connection* and *Cursor* respectively. It was named *Cursor* because this state is considered the default state of the tool and it is usually a good idea to create and place generators before adding wires.

It was also considered to bind this feature to a right click instead of going to the menu every time to create a new wire. However, the problem with this method is that users can accidentally right click without knowing, and there is no visual feedback for the user to know which "state" the tool is in. Requiring the user to go to the menu and selecting the option on the other hand involves a deliberate action, making the user aware of the action without the tool requiring to provide any visual feedback, and is very difficult to perform accidentally.

4.1.6 A way to remove generators and wires

There are some situations where the user can make a mistake or would like to make a change in their diagram. To make it easier for the user, instead of having to start from scratch the user can press the "Remove generator or id" button from the action menu to select any generator or wire to remove from the grid. If a wire is 10 squares long, selecting this wire for removal would only erase the one square selected from the grid, not the entire wire. This feature is done to account for scenarios where the user may want to make very slight adjustments to their diagram, and removing the wire entirely might be more of an inconvenience for the user.

Removing a generator from the grid will erase its entry in the text file. The tool will be notified to re-read the file and redraw all generators written in the text file in order to provide the user with the current state of the grid.

A way to reset the grid entirely was also added in the Action menu. This option can be done at any time, even after conversion from a graphical representation to a textual representation.

4.1.7 A way to create the "twist" generator

As a result of the properties of a symmetric monoidal category, it is possible to create and use a twist generator to construct more advanced string diagrams. An alternative way to recognize a twist is by looking for intersecting wires, which is the method used in the tool.

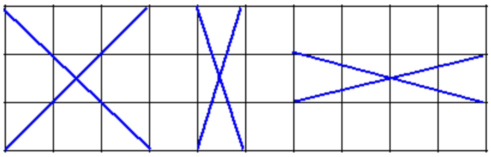


Figure 4.5: Possible ways to draw the twist morphism.

However, a problem arises generally when the twist morphism is drawn narrow vertically, such as twist in the middle of Figure 4.5. Due to the way the tool stores values in the array, the middle twist will have wire values of -1 stored one square above and below the intersection point. As a result, converting this representation to a textual representation will give $(id * twist * id)$. This is an incorrect result. The implemented way to fix this bug will be to reset every square above and below the intersection square to a value of 0. This will result in the correct output of $(twist)$ as the column will only contain the value of the twist (intersection point). This means that if a diagram is drawn that contains a twist, the twist must be drawn first before any generators or identity wires.

4.1.8 A way to convert the graphical representation to text

A "Calculate" button was added to represent this feature in the tool. This feature is implemented under an algorithm similar to the one outlined in Section 2.6.2. It involves splitting the diagram into single columns, reading off each section and then combining each section with compositions. Note that this is a simplified version of the real algorithm implemented in the tool, and does not account for every edge case in a real situation.

Algorithm 1: Algorithm to convert a graphical representation to a textual representation

```
$bracket \longleftarrow 0$;
for each column on the grid do
    $result \longleftarrow result +$ "(";
    for each row on the grid do
        if grid position (row, column) contains wire then
            $result \longleftarrow result +$ "$id *$ ";
        end
        else if grid position (row, column) contains intersection then
            $result \longleftarrow result +$ "$twist *$ ";
        end
        else if grid position (row, column) contains generator then
            $result \longleftarrow result + name +$ " $*$ ";
        end
    end
    remove last " $*$ " of $result$;
    $result \longleftarrow result +$ ");";
end
remove last ; of $result$;
```

As a result of the algorithm, every textual representation will be in a normal form of only tensor products $(*)$ and generators in brackets, and all compositions (;) outside brackets. Below is an example of the tool demonstrating this feature.

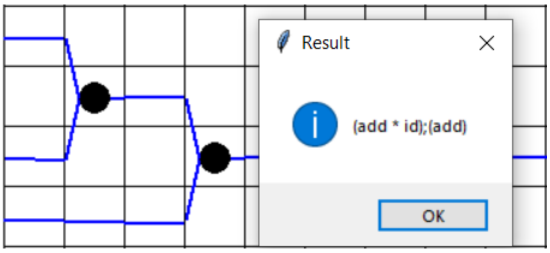


Figure 4.6: Example of the conversion feature in the tool.

4.2 Textual representation to a graphical representation

Similar to the conversion from a graphical representation to a textual representation, a general scenario was devised, describing what a user operating this tool would do in this aspect from start to finish:

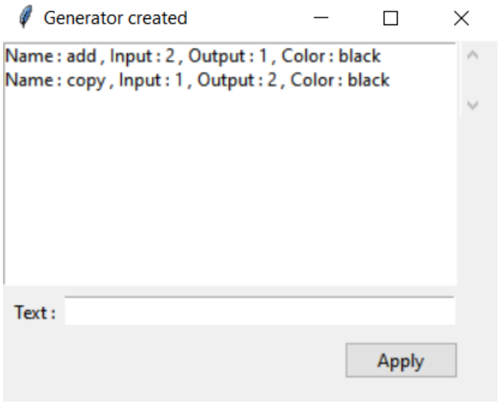
1. User creates generator
2. Repeat until all relevant generators has been created
3. User types in a textual representation of a string diagram
4. The tool reads the user's input and provides a graphical output by drawing on the grid.

4.2.1 Uses some previously implemented features

As the process is somewhat similar to how a user would operate the Graph → Text side of the tool, Text → Graph utilises some of the features already implemented in the previous part. Namely, the process of creating generators remains the same with the same error checks previously described in Section 4.1.3. A small difference however is that the tool is the one drawing the string diagram, so it does not record the position of the grid clicked by the user, as this is considered irrelevant information. Some features such as exporting text or drawing wires in the Action menu are also greyed out as they are not used in this part of the tool.

4.2.2 A way to store generators created by the user

Every time the user creates a generator, the information of the generators will be stored in a text file called *generator_info.txt*. Each line contains information about a generator in the following format: [name]||[order of creation]||[number of inputs]||[number of outputs]||[colour]. Then, to view the generators created or to type in a textual representation for the tool to draw, the user would right click to open a menu. The reason for binding this option to a right click instead of the Action menu was to allow a very fast and easy way to access the menu, due to a possible situation where a user would want to constantly check the generators created and not have to keep accessing the Action menu which could be tedious.



Name : add , Input : 2 , Output : 1 , Color : black
Name : copy , Input : 1 , Output : 2 , Color : black

Figure 4.7: The menu to view generators created or to type an input.

4.2.3 Layout algorithm

As this part of the tool involves the tool itself placing generators on the grid instead of a human, there needs to be a set of rules the tool must follow in order to produce an output that is consistent and predictable. Although the concept of aesthetics was a subjective factor, it was also considered while designing this algorithm.

1. Generators will be drawn starting from the leftmost column and the highest possible row.
2. If there is a tensor product, the generator will be drawn in the same column at the highest possible row, directly below the previous generator.
3. If there is a composition, the generator will be drawn at the next possible column to the right at the highest possible row.
4. If there is an identity, it is treated as an generator with one input and one output with length 1.
5. Twists are treated as an generator with two inputs and two outputs.
6. Any generator's output that is not linked will have a wire from the generator's output to the rightmost column of the grid. (There also needs to be a check for generators with no output).

These rules will ensure consistency in the tool drawing the diagram and simultaneously makes the diagram as compact as possible in algorithmic standards.

4.2.4 Converting a textual representation to a graphical representation

Due to additional factors such as the layout algorithm, implementation of this feature was much more complex compared to the graphical → textual representation feature. Despite the layout algorithm providing some structure to keep in mind, certain types of string diagrams make it very hard to implement an algorithm that the tool can follow without error. Consider the following diagram:

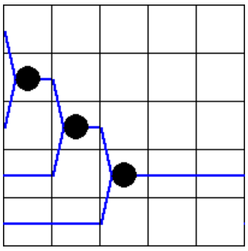


Figure 4.8: A string diagram representing $(A * id * id); (A * id); A$.

This string diagram was drawn by a human. However, this diagram is very difficult for an algorithm to draw.

The main problem with the tool trying to recreate this diagram involves the generators in column two and three having to shift down a specific number of rows. Although their original positions at the top row are positions that are not occupied by any generator or wire, the shifting occurs in order for the generators to be connected properly with other objects on their left. This additional factor that needs to be considered makes the algorithm extremely difficult to implement, and an alternative will have to be considered while preserving the principles of the layout algorithm.

An alternative method will not require any generators or wires to "shift", but it makes the diagram less compact. Consider the input string of $(A * id * id); (A * id); A$. The method is similar to the algorithm outlined above but this time, each bracket will be drawn on every other column without shifting, leaving a gap of one column between each generator. Then, wires are drawn between each of the columns to connect the generators and wires together.

This algorithm only works for inputs typed in the a normal form, where tensor products and generators are always inside brackets, and compositions are always outside brackets. It is possible for this diagram to be represented in a different way such as $(((A * id); A) * id); A$. Inputting this instead may cause an incorrect result outputted by the tool, so the algorithm will need to be refined in the future to account for other textual forms of string diagrams. Below is a simplified version of the algorithm implemented in the tool.

Algorithm 2: Algorithm to convert a textual representation in a normal form to a graphical representation

```
$column \longleftarrow 1$;
$row \longleftarrow 0$;
$max\_input \longleftarrow 1$;
$height \longleftarrow 1$;
$unconnected\_outputs \longleftarrow []$;
$unconnected\_inputs \longleftarrow []$;
for each bracket in the input string do
    for each generator in the bracket do
        $max\_input \longleftarrow max(input, output)$;
        if max_input is odd then
            $height \longleftarrow height + (max\_input - 1)/2$;
        end
        if max_input is even then
            $height \longleftarrow height + max\_input/2$;
        end
        $row \longleftarrow row + (height)$;
        place generator in (row, column);
        $row \longleftarrow row + (height - 1)$;
        $height \longleftarrow 1$;
    end
    $row \longleftarrow 0$;
    $column \longleftarrow column + 2$;
end
for each column on the grid do
    for each row on the grid do
        if grid position (row, column) contains unconnected output then
            add (x,y) to unconnected_outputs;
        end
        if grid position (row, column) contains unconnected input then
            add (x,y) to unconnected_inputs;
        end
    end
end
for each entry in unconnected_inputs do
    connect $unconnected\_outputs[entry]$ to $unconnected\_inputs[entry]$;
end
connect any remaining unconnected outputs to the right of the grid;
```

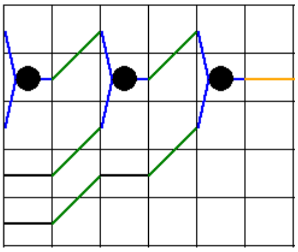


Figure 4.9: An output conversion from an input of $(A * id * id); (A * id); A$.

Therefore, this alternative algorithm is able to convert the input accurately to an equivalent graphical representation with the cost of the diagram not being as compact. Wires are coloured depending on their "role" in constructing the string diagram. In this example, the blue coloured wires represents wires from the generator, black coloured wires represents identity wires, green coloured wires represents the connection between the generators, and orange wires represents wires connected from the rightmost outputs to the right side of the grid.

4.2.5 A way to remove generators and wires

Unlike the Graph → Text version, this option had to be added in two parts in the Action menu. "Reset graph (text → graph)" functions exactly the same as the version in Graph → Text by removing all generators and wires on the grid. "Clear generators and wires (text → graph)" wipes all entries in the text file *generator_info.txt*, and therefore deletes all generators created by the user.

4.3 Additional features

This section includes all additional quality of life features that could be potentially helpful for the user experience.

4.3.1 A way to export the translated string diagram

If the translation's output was in the form of text, its output will be saved in a text file named *Graph_to_text.txt* in the same directory the tool is currently located in. This feature was implemented by creating a text file and writing the output created from the tool into the file.

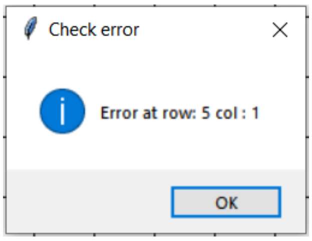
If the translation's output was in the form of a graph, its output will be saved as an image named *Graph_to_text.txt*. This feature was implemented by utilizing an imaging library (Pillow) and taking a screenshot of the screen, and cropping it based on the current dimensions and position of the GUI.

4.3.2 A way to check and give error messages in string diagrams that are incorrectly drawn

This feature is a work in progress and is not complete. Any perceived mistakes will cause the tool to give an error message, indicating the position of the error. It currently works by checking two things:

1. If the graph is connected from start to finish (left to right). If there is any broken connections, an error message will be given.

2. If each generator is connected to the correct amount of inputs and outputs. There are some edge cases where a generator can possibly have zero outputs, where the tool could potentially give a false positive error due to a “broken” connection. This edge case needs to be accounted for by checking the inputs and outputs of each generator.


Check error

Error at row: 5 col : 1

Figure 4.10: An example error message.

This feature is added to the Action menu, but as it is currently in development, it is not a required part of the process for the user. This is because the feature is quite buggy and can stop the entire tool from working if it was a required part of the process. Currently, users can test the feature by selecting the option at any time when drawing their graph. In the future, this would be a required check as the tool should always only accept sensible inputs from the user.

Chapter 5

Testing

5.1 Testing strategy

The tool was tested in multiple ways depending on the type of feature being tested. Each test was also repeated in case the result was anomalous.

Core functionality of the tool such as creating and drawing generators was covered and a key priority to test before any other feature. This is because if any of these features was buggy or unreliable, the whole tool would be rendered useless.

During testing of the main features of the tool (Graph → Text and Text → Graph), the tool was tested by inputting as many different kinds of string diagrams as possible to try to account for every case. If the tool produced an incorrect result, an attempt to refine the algorithm will be made to account for the new case. This method is only effective based on the amount and types of string diagrams provided to the tool. There could be a string diagram that could cause an incorrect result that was not thought of at the time of testing.

Other features such as exporting the output and clearing the generators on the grid was much simpler, as there are relatively fewer cases to test until a satisfactory result was achieved. These cases involved inputting edge cases such as a blank grid and normal cases from typical usage of the tool.

5.2 Testing of key features

5.2.1 Creating and placing generators

This feature worked well. Generators were created exactly in the position clicked by the user with the correct amount of inputs shown. The tool allows naming the input anything the user wants, including numbers as names. The tool will also not accept inputs greater than 20, or abnormal inputs such as letters.

However, the tool accepted negative number inputs and empty strings as names, and these issues were quickly fixed. Another issue was that depending on the position of the grid, the generator may not have enough room to fit in the grid.

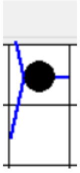


Figure 5.1: A generator with two inputs exceeding the boundaries of the grid.

This issue was not fixed and simply required users to avoid making this mistake, as wires will not be able to be placed beyond the grid.

5.2.2 Drawing wires

As previously stated, to switch from placing generators to placing wires, the user needed to select an option from the Action menu. This meant that if a user needed to draw many different wires, this process can be a bit tedious. However, this design seemed to fare better during testing as it tended to reduce the number of accidental wire placements had this option been binded to a right click, as there is no visual feedback provided for the user to know whether the tool would create a generator or a wire.

Testing was done by drawing different shapes of wire connected to generators. While performing and repeating this task, some instances of switching to the cursor mode and back to drawing wires was done to fully test the scope of the feature. There seemed to be no issues regarding this test

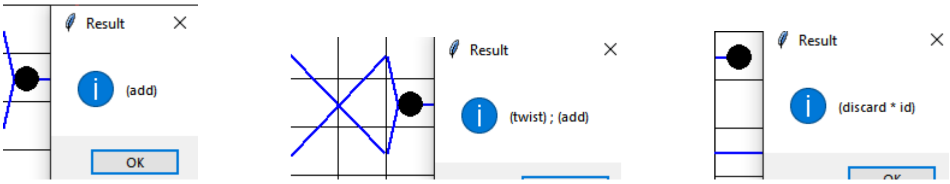
Another test performed was drawing wires over generators. This could potentially cause an incorrect result when the tool reads the graph, so this was fixed by having wires that overlap generators not affect the matrix by having them not being able to override an entry of a generator in the matrix.

There is also an issue of human error, where sometimes the user would forget to re-select the option to start a new wire. Failing to re-select would simply cause the wire to be joined from the previous position clicked to the current position clicked.

5.3 Testing the Graph → Text feature

This feature was tested with many different kinds of diagrams. Simple to complex diagrams were used to ensure the tool was outputting a correct textual representation.

A disadvantage of the tool was due to how it handles the "twist" generator. More details are provided in the previous chapter, however it results in having the user to draw any twist morphisms first before drawing any other generators or wires. Below are some successful test cases that outputted correct results. Images are cropped to save space in this report.



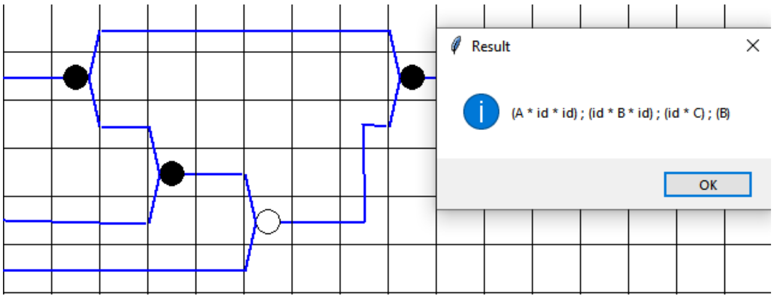
(add)

(twist) ; (add)

(discard * id)

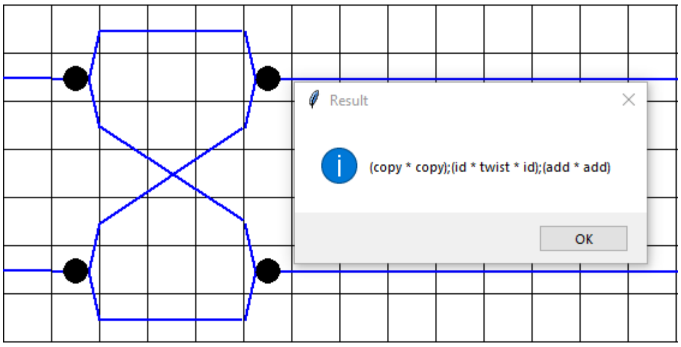
Figure 5.2: Simple test cases with results

Some diagrams tested are more complicated, but the program is able to handle these cases generally.



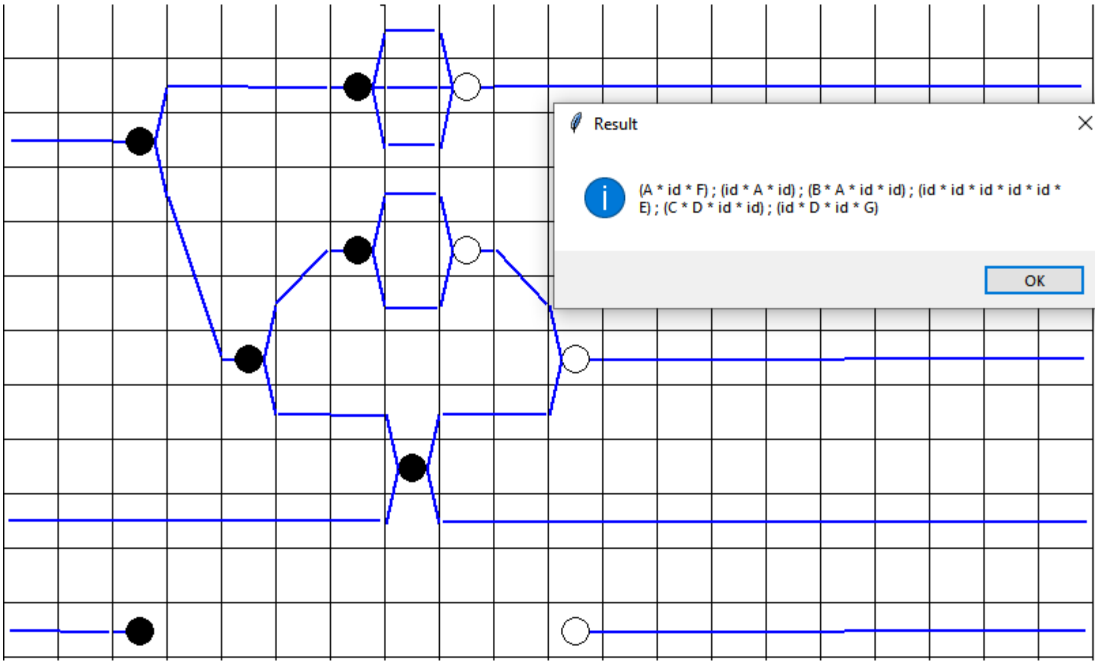
```
(A * id * id) ; (id * B * id) ; (id * C) ; (B)
```

Figure 5.3: A more complicated test case.



```
(copy * copy);(id * twist * id);(add * add)
```

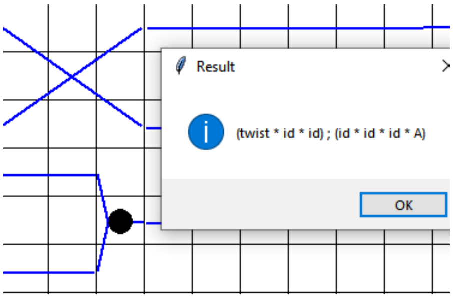
Figure 5.4: A more complicated string diagram with a twist.



```
(A * id * F) ; (id * A * id) ; (B * A * id * id) ; (id * id * id * id * id *
E) ; (C * D * id * id) ; (id * D * id * G)
```

Figure 5.5: Extremely complicated string diagram.

The tool works well generally, but there are few cases where it outputs an incorrect result. Most of these cases are related to the wires of a twist generator occupying multiple squares and the tool treats them as separate identity wires. This suggests a more rigorous algorithm to handle the twist more reliably, or a different way to detect wires in a square.


Result

(twist * id * id) ; (id * id * id * A)

Figure 5.6: Correct result: $(twist * id * id); (id * id * A)$

Overall, this feature seems to work well in most cases. Every output of the tool is typed in a fixed normal form of tensor products and generators always inside brackets and compositions outside brackets. This is similar to how other tools such as Catlab also outputs in normal form. In fact, this can be useful in some cases if the user wants to find a normal form, but in general, we might consider this a limitation of the tool. In the future, a feature to allow a format preference can be added.

5.4 Testing the Text → Graph feature

As previously explained, there an a limitation of the format that the tool can accept. Namely, all inputs in this feature must be typed in the normal form to generally expect correct results. Spaces are ignored in the input. Images are cropped to save space in this report.

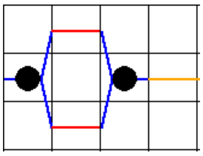


Figure 5.7: Input: $A; B$

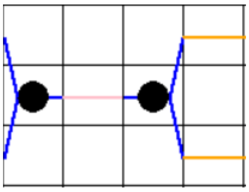


Figure 5.8: Input: $A; B.$

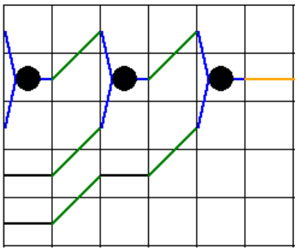


Figure 5.9: Input: $(A * id * id); (A * id); A$

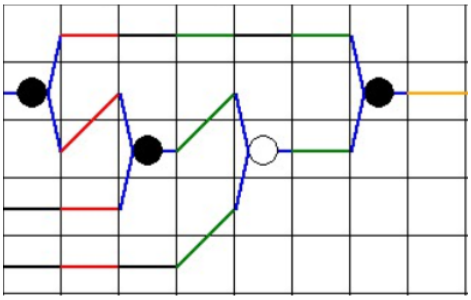


Figure 5.10: Input: $(A * id * id); (id * B * id); (id * C); B$

An issue occurs with some edge cases where generators have zero inputs or outputs. The tool sometimes draws a correct result but sometimes it does not. Therefore, the algorithm was refined to also account for these cases. As a result, string diagrams with zero inputs or outputs seem to be working well.

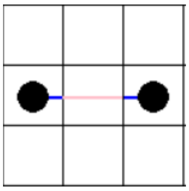


Figure 5.11: Input: $A; B$.

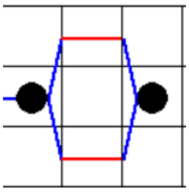


Figure 5.12: Input: $A; B$. Notice that B contains zero outputs.

5.5 Other features

5.5.1 Exporting images and text

This feature works as expected. The user just has to ensure that the tool has done a conversion before using this feature, or the output will be a blank grid or a empty text file. However, if a user is using display (resolution) scaling where their setting is not at 100%, the image output may be cropped incorrectly. So far, there is no way to fix this issue unless the user changes their display scaling to the default 100% setting.

5.5.2 Removing generators and wires

Removing generators seems to work well, however an issue that arises is if a long wire that covers more than one score is drawn at an angle, removing a part of the wire causes the rest of the wire to become flat, horizontal wires. This is an issue involving the method of storing information, as currently, there is no information that stores the exact "aesthetic" of a wire. As a result, this is a disadvantage that needs to be addressed in the future as it is a bug that is very easy to reproduce.

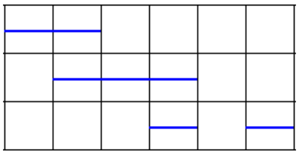


Figure 5.13: Result of deleting part of a diagonal wire.

Features to reset the grid and generators in the Text → Graph part of the tool seems to be working well. However sometimes resetting the grid in the Graph → Text part of the tool seems to affect future textual outputs after doing multiple conversions. So far, the cause of this is unknown and a safe way to avoid this issue is to close and reopen the tool after each calculation.

5.5.3 Checking the string diagram for errors

As previously described, this feature is currently a work in progress. As a result, it is very buggy and would be unfair to test this feature at its current stage. In the future, this feature will be developed and utilised in the tool.

Chapter 6

Conclusions and evaluation

6.1 Achievements

In the third Chapter, a list of requirements were produced to set some goals the tool needed to achieve.

- A tool was developed that could take both representations and in general was able to translate the string diagram to the other representation quite reliably. However, there exists some cases where the tool was not able to produce a correct output.
- A GUI was developed for the tool.
- Documentation is provided in this report, located in the Appendix section.
- The tool was not versatile enough to take in a textual representation of any format. A textual input must follow a specific structure for the tool to be able to understand.
- A feature for the user to export the translated string diagram to either a text file or an image was developed.
- There was no feature developed for a user to "save" their current progress to work on a later date.
- There was no feature developed to account for a preference in format for the output of the translated string diagram.
- A feature that includes error checking was created, but still in very early development.

6.2 Evaluation

In general, most of the important goals were attempted and implemented to a suitable standard, although there is room for improvement for all of the features implemented. Barring a minority of edge cases, the tool can be used and should be able to produce a correct and reliable result. Every design decision for the tool was justified and explained in previous chapters, although some of the features implemented such as the way to store values of generators and wires could have been considered in finer detail.The tool was able to create and implement several quality of life features such as exporting the translated string diagram and removing generators or wires if the user made a mistake.

Had the project been given more time, features that are still under development such as error checking would most likely be completed and would have the opportunity to be tested for further development. A majority of the issues addressed in the previous chapter would also most likely be fixed, as there would have been more time to refine the algorithms and design features in the tool.

In conclusion, results from testing has shown the project still has potential and currently has met its most important goal and most of its other high priority goals, with a few quality of life features implemented.

6.3 Future Work

The tool can be continued in many different ways. Any bugs or issues that are described can be fixed. Any missing features or features described above that are currently in development can be continued and further developed. After these goals are achieved, the tool can extend its scope from covering only symmetric monoidal categories to accommodate other different kinds of string diagrams such as traced monoidal categories or compact closed monoidal categories.

Appendix A

Documentation

A.1 Layout

All of the features implemented in the tool has been previously discussed in Chapter 4. However for users interested to try out the tool could benefit from having a documentation written for the tool. The GUI is structured in the following way:

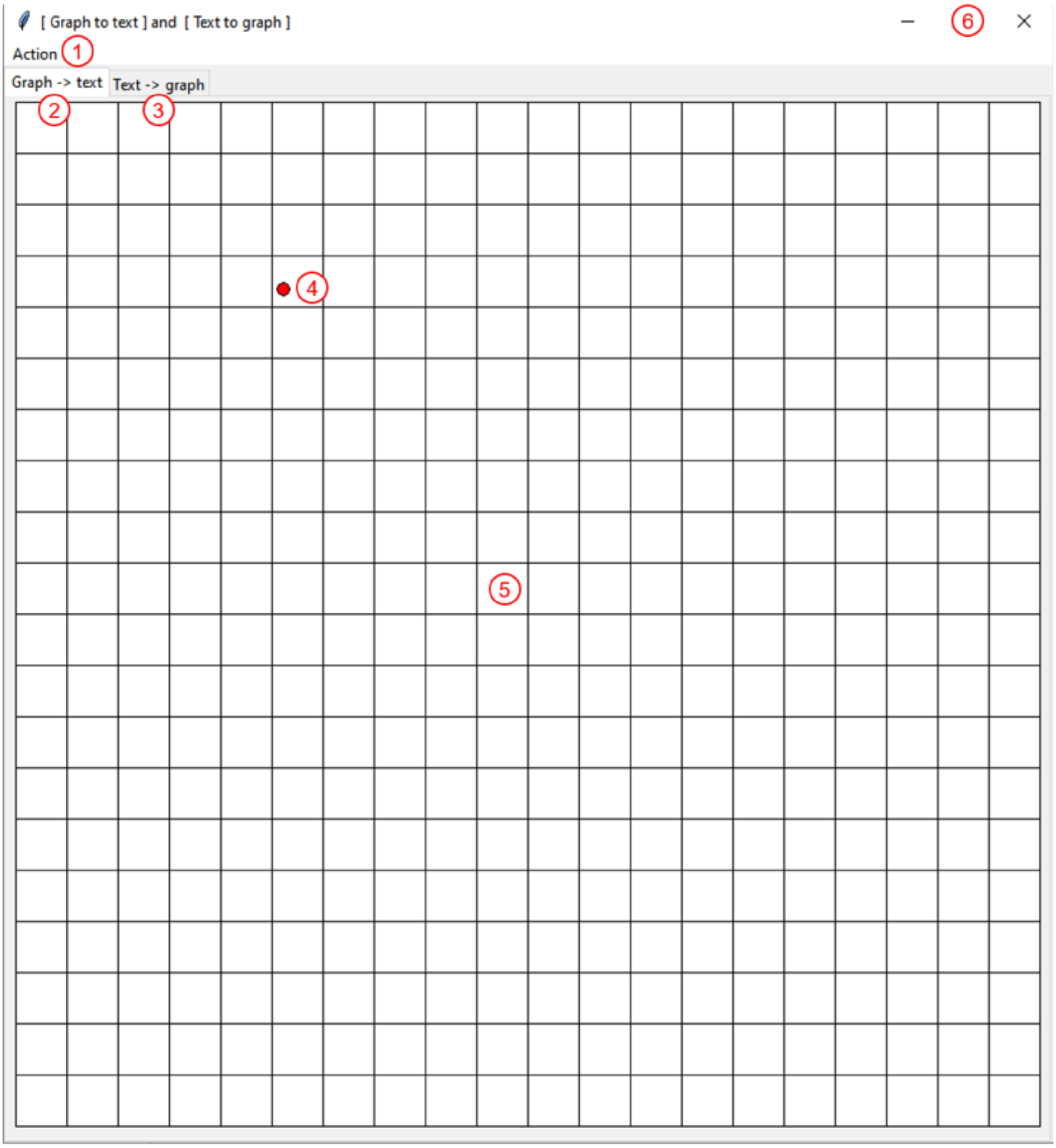


Figure A.1: GUI of the tool.

1. **1** - Action menu containing the following:

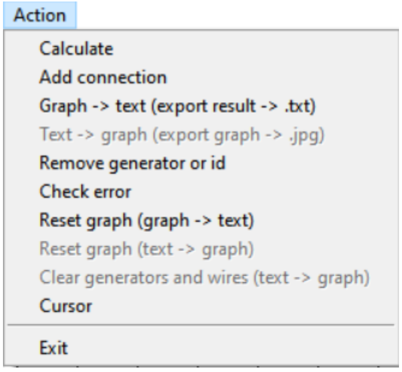

Action
Calculate
Add connection
Graph -> text (export result -> .txt)
Text -> graph (export graph -> .jpg)
Remove generator or id
Check error
Reset graph (graph -> text)
Reset graph (text -> graph)
Clear generators and wires (text -> graph)
Cursor
Exit

Figure A.2: Action menu

- **Calculate** - Only used in Graph → Text mode, after drawing a string siagram on the grid the tool translates the graph to a textual representation.
- **Add connection** - If selected, any subsequent left clicks of the mouse will draw wires on the grid. This option needs to be re-selected if you want to start a new wire at a different position. If this option is not re-selected, a wire will connect from the previous point selected to the current point clicked with the mouse.
- **Graph → Text (export result → .txt)** - After receiving a translated textual representation, users can export this result to a text file called *Graph_to_text.txt*. This file will be overwritten if there are any subsequent uses of this feature.
- **Text → Graph (export graph → .jpg)** - After receiving a translated graphical representation, users can export this result to an image called *Text_to_graph.jpg*. This file will be overwritten if there are any subsequent uses of this feature. Ensure that you are on a 100% display scaling or the image output may result in a misaligned cropping.
- **Remove generator or id** - Selecting this option and clicking any generator or wire will remove it from the grid. This option has to be re-selected if you want to delete another object.
- **Check error** - Extremely buggy and currently in development. Not recommended to use at the moment. After drawing a string diagram in Graph → Text mode, the tool will attempt to analyze the string diagram to check for any errors in the diagram created.
- **Reset graph (graph → text)** - Clears the grid of all generators and wires. If used this feature sometimes has trouble with subsequent calculation outputs from the tool. Currently, it is safer to close and reopen the program after each string diagram calculation to avoid the risk of this bug.
- **Reset graph (text → graph)** - Clears the grid of all generators and wires. Only used in Text → Graph mode.
- **Clear generators and wires (text → graph)** - Removes all stored generators created in the Text → Graph mode.
- **Cursor** - If selected, any left clicks on the grid will create a dialog box asking for details to create a generator. This option is chosen by default.
- **Exit** - Closes the program. Equivalent to clicking the X button at **6**.

2. **2** - Switch to Graph → Text mode. By default the tool starts at this mode.

3. **3** - Switch to Text → Graph mode.

4. **4** - Cursor

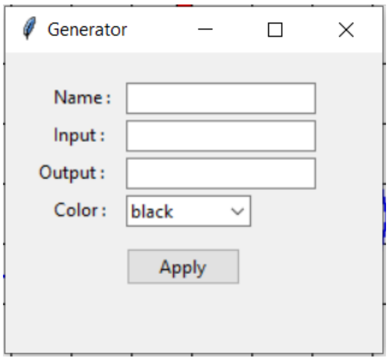
5. **5** - A 20x20 grid to place generators and wires on.

6. **6** - Title bar buttons, including a way to minimize or close the tool.

A.2 How to draw a generator

1. Choose a position to place the generator by left clicking on any square of the grid.

2. A dialog box will appear as follows:



Name :

Input :

Output :

Color : black

Figure A.3: GUI of the tool.

   (a) **Name** - Generator's name

   (b) **Input** - Number of inputs. Note that inputs cannot exceed 20 or be a negative number.

   (c) **Output** - Number of outputs. Note that outputs cannot exceed 20 or be a negative number.

   (d) **Color** - Colour of generator can be chosen as black or white to differentiate itself from other similar generators.

3. After clicking the Apply button, the generator is created.

A.3 How to draw wires

1. Select Action → Add connection.

2. Left click on any point to mark the starting point of the wire.

3. Left click another point to mark the end point of the wire. The wire will be drawn as a straight line from the first point to the second point selected.

4. If you want to continue the wire, any subsequent left clicks will create a wire from the second point selected to the third point selected and so on.

5. If you want to start a new wire, re-select the Add connection button.

A.4 How to draw a twist

1. Select Action → Add connection.

2. Left click on any point to mark the starting point of the wire.

3. Left click another point to mark the end point of the wire. The wire will be drawn as a straight line from the first point to the second point selected.

4. Re-select Add connection.

5. Repeat steps 2 and 3 to create a wire that overlaps the first wire. Twist generators are formed from any intersections of wires. Currently, the twist generator is slightly buggy, but it works for most cases.

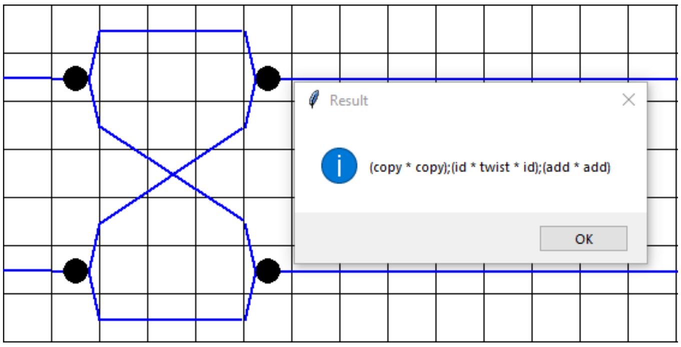
A.5 Translate a string diagram to text

1. Once the diagram has been drawn, select Action → Calculate

2. A dialog box will appear containing the interpreted string diagram in text.

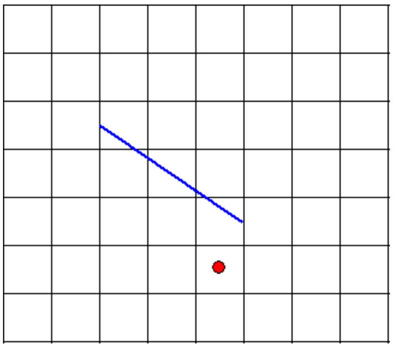
A.6 Example of creating a string diagram

In this example we will draw the string diagram below:

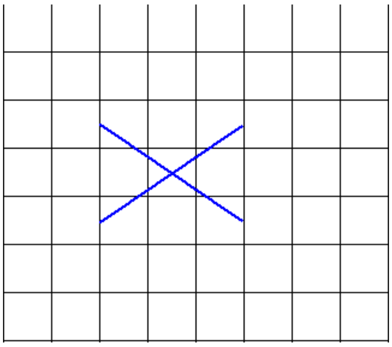


```
(copy * copy);(id * twist * id);(add * add)
```

1. As previously explained in Chapter 4, we will start by drawing the twist generator. Start by selecting the Add connection option in the Action menu and drawing this wire:

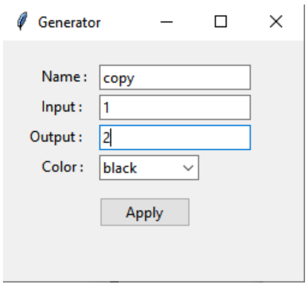


2. Re-select the Add connection option and draw this wire that intersects the first wire. This creates the twist generator.

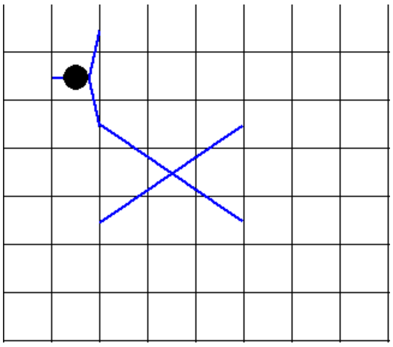


3. Now, we will draw all the generators in the diagram. Select the Cursor option in the Action menu.

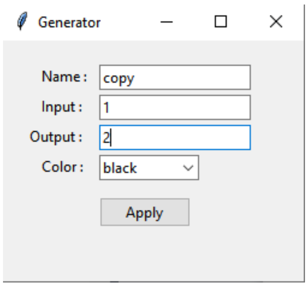
4. Left click the square at row 2 column 2 of the grid, and fill in the details below. If done correctly, the diagram will look like this.



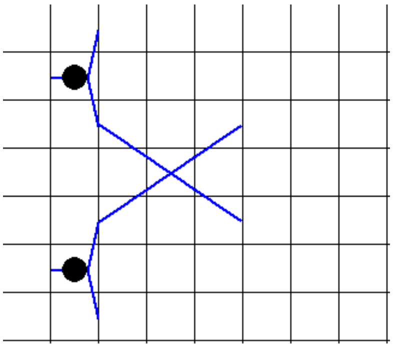
Name : copy
Input : 1
Output : 2
Color : black



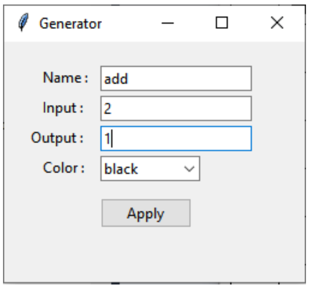
5. Repeat the same process for the square at row 6 column 2 of the grid.



Name : copy
Input : 1
Output : 2
Color : black



6. Repeat the same process for the square at row 2 column 6 of the grid. This time, the generator created will be different.



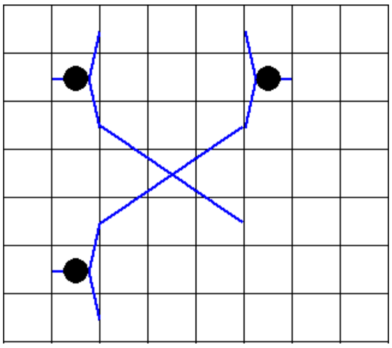
Generator

Name : add

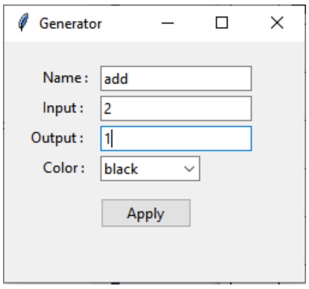
Input : 2

Output : 1

Color : black



7. Repeat for row 6 column 6 of the grid.



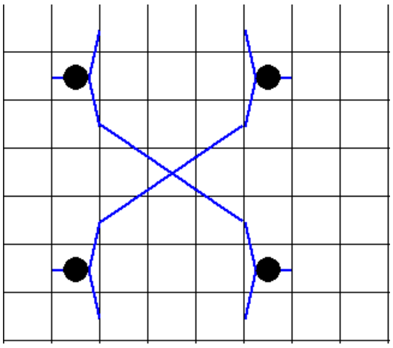
Generator

Name : add

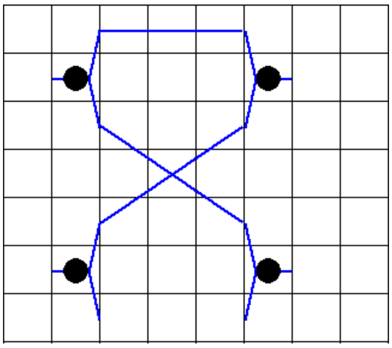
Input : 2

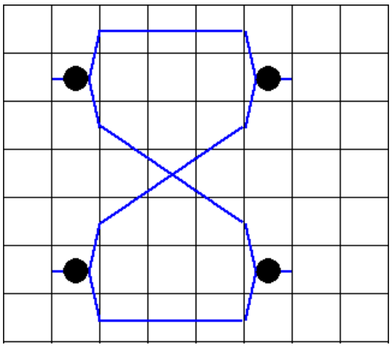
Output : 1

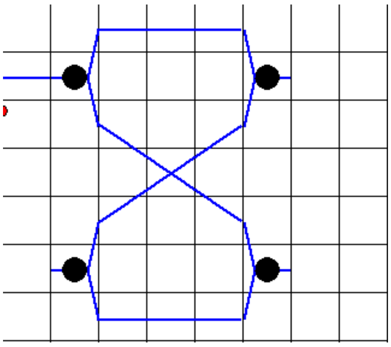
Color : black

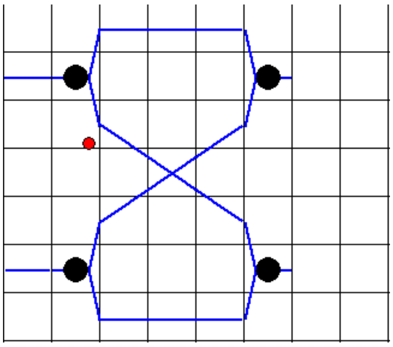


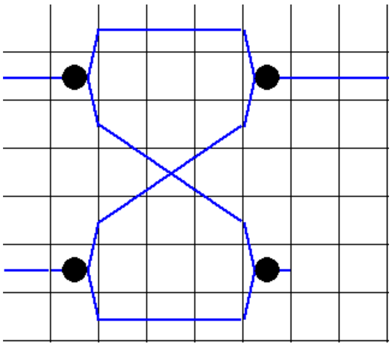
8. Connect the rest of the graph by using the Add connection option to draw wires. Remember to re-select Add connection whenever starting a new wire.

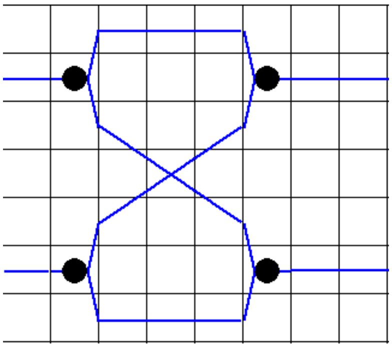




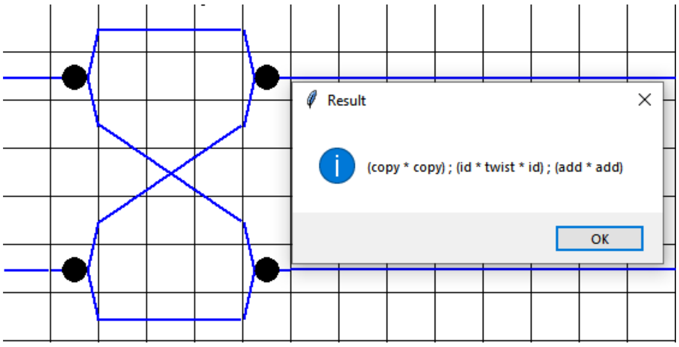








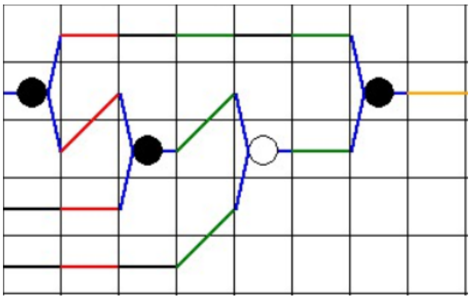
9. Finally, select the Calculate option in the Action menu. If drawn correctly, the output of $(copy * copy); (id * twist * id); (add * add)$ should be produced.



(copy * copy) ; (id * twist * id) ; (add * add)

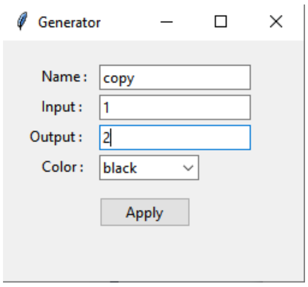
A.7 Example in Text → Graph

In this example, we will make the tool draw the string diagram represented textually as $(copy * id * id); (id * add * id); (id * multiply); add$

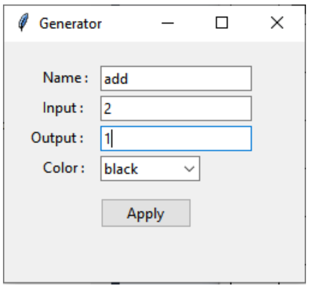


1. If you haven't already, switch to the Text → Graph feature by clicking on the Text → Graph button.

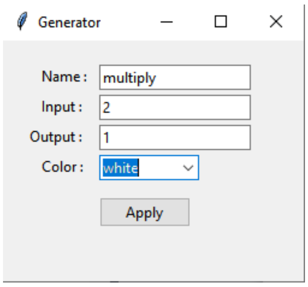
2. We will start to create generators. In this case, the position on the grid clicked doesn't matter. Left click anywhere on the grid to make the dialog box appear. Repeat this three times as we need three generators.



Name: copy
Input: 1
Output: 2
Color: black



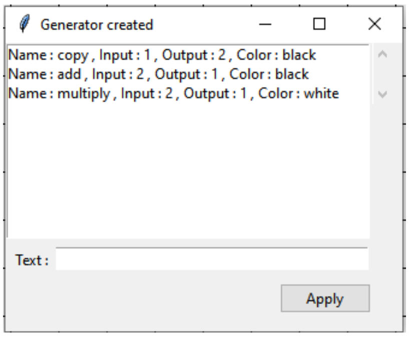
Name: add
Input: 2
Output: 1
Color: black



Name: multiply
Input: 2
Output: 1
Color: white

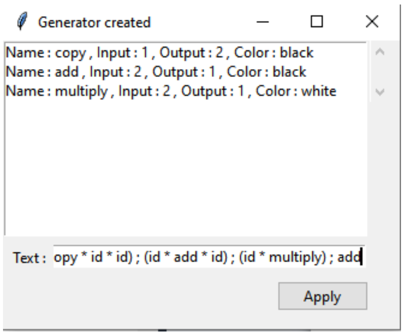
3. After creating these three generators, right click anywhere on the grid to make this dialog box appear.

4. At the bottom of the dialog box, type in the input of $(copy * id * id); (id * add * id); (id * multiply); add$. Whether you choose to type this with spaces or without spaces is up to you.



```
Name : copy , Input : 1 , Output : 2 , Color : black
Name : add , Input : 2 , Output : 1 , Color : black
Name : multiply , Input : 2 , Output : 1 , Color : white
```

Text :



```
Name : copy , Input : 1 , Output : 2 , Color : black
Name : add , Input : 2 , Output : 1 , Color : black
Name : multiply , Input : 2 , Output : 1 , Color : white
```

Text : opy * id * id) ; (id * add * id) ; (id * multiply) ; add

5. Click on the Apply button. If done correctly, the diagram drawn by the tool should match the example image at the beginning of this section.

Appendix B

Code listing

```
import cv2
import numpy as np
import tkinter as tk
from tkinter import ttk
#from PIL import ImageTk, Image
from tkinter import filedialog
import os
from tkinter import messagebox
from tkinter import *

import PIL.Image
import PIL.ImageTk
from PIL import ImageGrab

from os import path
import math
import time

import shapely
from shapely.geometry import LineString, Point

class GridWindow:
    def __init__(self, parent, status):
        self.status = status
        self.myParent = parent
        self.myContainer1 = tk.Frame(parent)
        self.myContainer1.pack()
        self.cellwidth = 40
        self.cellheight = 40
        self.rect = {}
        self.GRID = 3
        self.mode = ''
self.str_GT = ''
self.str_gen_Detial = ''
self.line_intersection = []
self.sub_line_intersection = []
self.status_remove = False
self.Matrix_grid = np.zeros((20, 20), dtype=int)
self.Matrix_id_line = np.zeros((20, 20), dtype=int)
self.Matrix_order_node = np.zeros((20, 20), dtype=int)
self.Matrix_x = np.zeros((20, 20), dtype=float)
self.Matrix_y = np.zeros((20, 20), dtype=float)
self.Clicked = False
self.num_gen = 0
self.line_position = []
self.mode = ''
self.generator_dict = {}
self.counter = 0
self.counter_text = 0
self.gen_column = []
self.twist_status = False

self.menubar = Menu(mainWindow)
self.filemenu = Menu(self.menubar, tearoff=0)
self.filemenu.add_command(label=”Calculate”, command=lambda:
    self.calculate())
self.filemenu.add_command(label=”Add connection”, command=
    lambda: self.connecting())
self.filemenu.add_command(label=”Graph -> text (export result
    -> .txt)”, command=lambda: self.Export())
self.filemenu.add_command(label=”Text -> graph (export graph ->
     .jpg)”, command=lambda: self.Export())
self.filemenu.add_command(label=”Remove generator or id”,
    command=lambda: self.Remove())
self.filemenu.add_command(label=”Check error”, command=lambda:
    self.Check_error())
#self.filemenu.add_command(label=”Twist”, command=lambda: self.
    twist())
self.filemenu.add_command(label=”Reset graph (graph -> text)”,
    command=lambda: self.reset_graph_GT_clear_table())
self.filemenu.add_command(label=”Reset graph (text -> graph)”,
    command=lambda: self.reset_graph_TG_clear_table())
self.filemenu.add_command(label=”Clear objects (text -> graph)
    ”, command=lambda: self.reset_graph_TG_remove_file())
self.filemenu.add_command(label=”Cursor”, command=self.cursor)
self.filemenu.add_separator()

self.filemenu.add_command(label=”Exit”, command=self.myParent.
            quit)

        #self.filemenu.bind('<<MenuSelect>>', self.menucallback)

        self.menubar.add_cascade(label="Action", menu=self.filemenu)
        mainWindow.config(menu=self.menubar)

        self.menucallback()

    def reset_graph_TG_remove_file(self):
        MsgBox = tk.messagebox.askquestion('delete', 'Delete old
            generator ?', icon='warning')
        if MsgBox == 'yes':
            os.remove('generator_info.txt')

    def reset_graph_TG_clear_table(self):
        self.myContainer1.destroy()
        self.myContainer1 = tk.Frame(self.myParent)
        self.myContainer1.pack()
        self.draw_grid(20, 20)
        self.myCanvas.update()

        self.Matrix_grid = np.zeros((20, 20), dtype=int)
        self.Matrix_id_line = np.zeros((20, 20), dtype=int)
        self.Matrix_order_node = np.zeros((20, 20), dtype=int)
        self.Clicked = False
        self.num_gen = 0
        self.line_position = []
        self.mode = ''
        self.generator_dict = {}
        self.counter = 0
        self.counter_text = 0
        self.gen_column = []

    def reset_graph_GT_clear_table(self):
        self.myContainer1.destroy()
        self.myContainer1 = tk.Frame(self.myParent)
        self.myContainer1.pack()
        self.draw_grid(20, 20)
        self.myCanvas.update()

        self.Matrix_grid = np.zeros((20, 20), dtype=int)
        self.Matrix_id_line = np.zeros((20, 20), dtype=int)
        self.Matrix_order_node = np.zeros((20, 20), dtype=int)
        self.Clicked = False
        self.num_gen = 0
        self.line_position = []
        self.mode = ''
        self.generator_dict = {}
        self.counter = 0
        self.counter_text = 0
        self.gen_column = []

    def menucallback(self):
        global draw_tab_1, draw_tab_2

        if draw_tab_1 == True and draw_tab_2 == False:
            self.filemenu.entryconfig(2, state=tk.DISABLED)
            self.filemenu.entryconfig(3, state=tk.NORMAL)

            self.filemenu.entryconfig(6, state=tk.DISABLED)
            self.filemenu.entryconfig(7, state=tk.NORMAL)
            self.filemenu.entryconfig(8, state=tk.NORMAL)

        elif draw_tab_2 == True and draw_tab_1 == False:
            self.filemenu.entryconfig(2, state=tk.NORMAL)
            self.filemenu.entryconfig(3, state=tk.DISABLED)

            self.filemenu.entryconfig(6, state=tk.NORMAL)
            self.filemenu.entryconfig(7, state=tk.DISABLED)
            self.filemenu.entryconfig(8, state=tk.DISABLED)
        else:
            self.filemenu.entryconfig(2, state=tk.NORMAL)
            self.filemenu.entryconfig(3, state=tk.DISABLED)

            self.filemenu.entryconfig(6, state=tk.NORMAL)
            self.filemenu.entryconfig(7, state=tk.DISABLED)
            self.filemenu.entryconfig(8, state=tk.DISABLED)


    def check_intersection_line(self, point_line1_1, point_line1_2,
       point_line2_1, point_line2_2):
        point_of_intersection = []
        try:

            line1 = LineString([point_line1_1, point_line1_2])
            line2 = LineString([point_line2_1, point_line2_2])

            int_pt = line1.intersection(line2)
            point_of_intersection = int_pt.x, int_pt.y

        except Exception as ex:
```

```
        point_of_intersection = []

    return point_of_intersection

def Check_error(self):
    [r, c] = self.Matrix_grid.shape
    for j in range(r - 1):
        for i in range(c - 1):
            if self.Matrix_grid[j, i] == -1:
                count1 = 0
                if self.Matrix_grid[j - 1, i] == -1:
                    count1 += 1
                if self.Matrix_grid[j + 1, i] == -1:
                    count1 += 1
                if self.Matrix_grid[j, i - 1] == -1:
                    count1 += 1
                if self.Matrix_grid[j, i + 1] == -1:
                    count1 += 1

                if count1 <= 1:
                    messagebox.showinfo("Check error", "Have error
                        row : " + str(j + 1) + "col : " + str(i +
                        1))

            elif self.Matrix_grid[j, i] > -1:
                count2 = 0
                if self.Matrix_grid[j, i + 1] > 0 and self.
                    Matrix_grid[j, i - 1] > 0 and self.Matrix_grid[
                    j - 1, i] == 0 and self.Matrix_grid[j + 1, i]
                    == 0:
                    messagebox.showinfo("Check error", "Have error
                        row : " + str(j + 1) + "col : " + str(i +
                        1))

def Remove(self):
    self.status_remove = True
    self.mode = ''

def Export(self):
    global draw_tab_1, draw_tab_2

    '''
    #print('draw_tab_1 : {}'.format(draw_tab_1))
    #print('draw_tab_2 : {}'.format(draw_tab_2))
    if draw_tab_1 == True and draw_tab_2 == False:
        self.status_export = 'GT'
        self.str_GT = '('
        [r, c] = self.Matrix_grid.shape
        for i in range(r):
            if sum(self.Matrix_grid[:, i]) > 0:
                duplicated = -100
                for j in range(c):
                    if self.Matrix_grid[j, i] > 0 and self.
                        Matrix_grid[j, i] != duplicated:
                        with open("create_gt.txt", 'r') as file_in:
                            for line in file_in:
                                str = line.split(',')
                                column = int(str[0])
                                num_gen = int(str[1])
                                x = int(str[2])
                                y = int(str[3])
                                input = int(str[4])
                                output = int(str[5])
                                gen_color = str[6]
                                name = str[7].replace('\n', '')

                                duplicated = num_gen

                                if self.Matrix_grid[j, i] ==
                                    num_gen:
                                    self.str_GT += name + ';'
                                    break
                    elif self.Matrix_grid[j, i] == -1:
                            self.str_GT += 'id;'
                self.str_GT += ');('

        '''

    if draw_tab_1 == True and draw_tab_2 == False:

        self.status_export = 'GT'
        M = self.get_Matrix_grid()

        c = 0
        tmp = ''
        log = 0
        [row, col] = M.shape
        self.gen_column = list(dict.fromkeys(self.gen_column))
        self.gen_column = sorted(self.gen_column)
        for i in range(len(self.gen_column)):
```

```
        tmp += '('
        log = ''
        for j in range(row):
            if M[j, self.gen_column[i] - 1] == -1:
                tmp += 'id * '
            elif M[j, self.gen_column[i] - 1] == -2 and M[j,
                self.gen_column[i] - 1] != log:
                log = M[j, self.gen_column[i] - 1]
                tmp += 'twist * '
            elif M[j, self.gen_column[i] - 1] != 0 and M[j,
                self.gen_column[i] - 1] != log:
                log = M[j, self.gen_column[i] - 1]
                tmp += str(self.generator_dict.get(M[j, self.
                    gen_column[i] - 1])) + ' * '
        tmp = tmp[:-3]
        tmp += '); '

        '''
        # check count
        count = tmp.count(';')
        if count <= 1:
            tmp = tmp.replace(';', '')
        '''

        tmp = tmp.strip()[:-1]
        if len(tmp) < 4:
            with open('graph_to_text_export.txt', 'w') as f:
                f.write(tmp)

    messagebox.showinfo("Export", "Graph to text export
        completed !")

elif draw_tab_2 == True and draw_tab_1 == False:

    self.status_export = 'TG'

    time.sleep(1)
    x = self.myParent.winfo_rootx()
    y = self.myParent.winfo_rooty()
    height = self.myParent.winfo_height() + y
    width = self.myParent.winfo_width() + x
    ImageGrab.grab().crop((x, y, width, height)).save('
        Text_to_graph.jpg')

    messagebox.showinfo("Export", "Text to graph export
        completed !")
    def set_mode(self, m):
        self.mode = m

    def set_num_gen(self, g):
        self.num_gen = g

    def get_Matrix_grid(self):
        return self.Matrix_grid

    def draw_grid(self, rows, columns):

        self.myCanvas = tk.Canvas(self.myContainer1)
        self.myCanvas.configure(width=self.cellheight*rows+4, height=
            self.cellwidth*columns+4)
        self.myCanvas.pack(side=tk.RIGHT)

        for column in range(rows):
            for row in range(columns):
                x1 = column * self.cellwidth+4
                y1 = row * self.cellheight+4
                x2 = x1 + self.cellwidth
                y2 = y1 + self.cellheight
                self.rect[row, column] = self.myCanvas.create_rectangle
                    (x1, y1, x2, y2, fill="white")
                self.Matrix_x[row, column] = x2
                self.Matrix_y[row, column] = y2

        self.myCanvas.bind('<Motion>', self.mot)
        self.myCanvas.bind("<Button 1>", self.getorigin)
        self.myCanvas.bind("<Button 3>", self.click_right_button)

        ov = self.myCanvas.create_oval(-3, -3, 3, 3, tags='point', fill
            ='red')

        #self.myCanvas.mainloop()

    def __del__(self):
        print('Destructor !')

    def On_clicked(self, e):
        if self.mode == 'connecting' and self.Clicked == False:
            self.Clicked = True
        elif self.mode == 'connecting' and self.Clicked == True:
            self.Clicked = False
        print('Click canvas !')
    def mot(self, e):
        #print(self.Clicked)
        xg = (e.x + self.GRID / 2) // self.GRID
        yg = (e.y + self.GRID / 2) // self.GRID
        t = self.myCanvas.find_withtag('point')
        #print('in Class : {}'.format(self.mode))
        if self.mode == 'connecting' and self.Clicked:
            column = e.x // (40)
            row = e.y // (40)

            if self.Matrix_grid[row, column] == 0:
                self.Matrix_grid[row, column] = -1
            print(self.Matrix_grid)

            row += 1
            column += 1
            print("Move ", (xg, yg), "Grid coordinates: ", row, column)

        self.set_oval_coords(t, (xg * self.GRID, yg * self.GRID))
        #print("self.Clicked : {}".format(self.Clicked))

    def set_oval_coords(self, t, xy):
        x, y = xy
        self.myCanvas.coords(t, (x - 5, y - 5, x + 5, y + 5))

    def draw_line_id(self, txt):

        # Mark id
        r = 0
        c = 0
        [rows, cols] = self.Matrix_grid.shape

        #txt = self.T.get().strip() # read input text
        Semi_colon = txt.split(';') # split input text

        # Mark id
        for i in range(len(Semi_colon)):
            Sub_semi_colon = Semi_colon[i]
            Star = Sub_semi_colon.split('*')
            for j in range(len(Star)):
                Sub_star = Star[j].replace('(', '').replace(')', '').
                    strip()
                if Sub_star == 'id':
                    self.Matrix_grid[r, c] = -9
                    #print(self.Matrix_grid)
                    r += 1
                else: # is generator
                    r = 0
                    for m in range(rows):
                        if self.Matrix_grid[m, c] > 0:
                            break
                        else:
                            r += 1
                    r += 3
            c += 2
            r = 0

        # Order node
        for i in range(cols):
            ns = 0
            count = 0
            replace_count = 0
            for j in range(rows):
                v = self.Matrix_grid[j, i]
                if v != 0 and v != ns:
                    ns = v
                    count += 1
                    self.Matrix_id_line[j, i] = count
                    ns = self.Matrix_grid[j, i]
                    #print(self.Matrix_id_line)
                elif v != 0 and v == ns and v != -9:
                    ns = v
                    replace_count += 1
                    if replace_count >= 2:
                        replace_count = 0
                        count += 1
                        self.Matrix_id_line[j, i] = count
                        #print(self.Matrix_id_line)
                    else:
                        self.Matrix_id_line[j, i] = -7
                elif v != 0 and v == ns and v == -9:
                    ns = v
                    replace_count += 1
                    if replace_count > 0:
                        replace_count = 0
                        count += 1
                        self.Matrix_id_line[j, i] = count

        print(self.Matrix_id_line)

    def draw_generator_of_remove(self, x, y, input, output, gen_color,
       Name):

        self.create_circle(x + 6, y - 11, 10, gen_color)

        column = x // (40)
        row = y // (40)

        #self.str_GT = str(column) + ',' + str(self.num_gen) + ',' +
            str(x) + ',' + str(y) + ',' + str(
         #    input) + ',' + str(output) + ',' + gen_color + ',' + Name
             + '\n'
        #with open('create_gt.txt', 'a+') as f:
         #    f.write(self.str_GT)

        '''
        self.Matrix_grid[row, column] = self.num_gen
        self.Matrix_grid[row + 1, column] = self.num_gen
        self.Matrix_grid[row - 1, column] = self.num_gen
        print(self.Matrix_grid)
        '''

        row += 1
        column += 1

        # print('Create gen {} , {}'.format(row, column))

        # output
        str_tmp = ''
        # p_output_x = str(x + 17)
        # p_output_y = str(y - 10)
        if output % 2 == 0:
            for i in range(int(output / 2)):
                self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
                    10)
                self.draw_line(x + 26, y + (30 + (i * 40)), x + 17, y -
                    10)
                str_tmp += 'output,' + str(x + 26) + ',' + str(y - (50
                   + (i * 40))) + '\n'
                str_tmp += 'output,' + str(x + 26) + ',' + str(y + (30
                   + (i * 40))) + '\n'
        else:
            self.draw_line(x + 26, y - 10, x + 17, y - 10)
            str_tmp += 'output,' + str(x + 26) + ',' + str(y - 10) + '\
               n'
            for i in range(int(output / 2)):
                self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
                        10)
                   self.draw_line(x + 26, y + (35 + (i * 40)), x + 17, y -
                        10)
                   str_tmp += 'output,' + str(x + 26) + ',' + str(y - (50
                      + (i * 40))) + '\n'
                   str_tmp += 'output,' + str(x + 26) + ',' + str(y + (35
                      + (i * 40))) + '\n'

           # input
           # p_input_x = str(x - 4)
           # p_input_y = str(y - 10)
           if input % 2 == 0:
               for i in range(int(input / 2)):
                   self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
                       10)
                   self.draw_line(x - 13, y + (30 + (i * 40)), x - 4, y -
                       10)
                   str_tmp += 'input,' + str(x - 13) + ',' + str(y - (50 +
                        (i * 40))) + '\n'
                   str_tmp += 'input,' + str(x - 13) + ',' + str(y + (30 +
                        (i * 40))) + '\n'
           else:
               self.draw_line(x - 13, y - 10, x - 4, y - 10)
               str_tmp += 'input,' + str(x - 13) + ',' + str(y - 10) + '\n
                   '
               for i in range(int(input / 2)):
                   self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
                       10)
                   self.draw_line(x - 13, y + (35 + (i * 40)), x - 4, y -
                       10)
                   str_tmp += 'input,' + str(x - 13) + ',' + str(y - (50 +
                        (i * 40))) + '\n'
                   str_tmp += 'input,' + str(x - 13) + ',' + str(y + (35 +
                        (i * 40))) + '\n'

           #with open('gen.txt', 'a+') as f:
               #f.write(str_tmp)

       def draw_generator_TG(self, x, y, input, output, gen_color, Name):

           self.create_circle(x + 6, y - 11, 10, gen_color)

           column = x // (40)
           row = y // (40)

           self.Matrix_grid[row, column] = self.num_gen
       self.Matrix_grid[row + 1, column] = self.num_gen
       self.Matrix_grid[row - 1, column] = self.num_gen
       print(self.Matrix_grid)

       row += 1
       column += 1

       # print('Create gen {} , {}'.format(row, column))

       # output
       str_tmp = Name + '|'
       # p_output_x = str(x + 17)
       # p_output_y = str(y - 10)
       if output % 2 == 0:
           for i in range(int(output / 2)):
               self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
                    10)
               self.draw_line(x + 26, y + (30 + (i * 40)), x + 17, y -
                    10)
               str_tmp += 'output=' + str(x + 26) + ',' + str(y - (50
                  + (i * 40))) + '|'
               str_tmp += 'output=' + str(x + 26) + ',' + str(y + (30
                  + (i * 40))) + '|'
       else:
           self.draw_line(x + 26, y - 10, x + 17, y - 10)
           str_tmp += 'output=' + str(x + 26) + ',' + str(y - 10) +
               '|'
           for i in range(int(output / 2)):
               self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
                    10)
               self.draw_line(x + 26, y + (35 + (i * 40)), x + 17, y -
                    10)
               str_tmp += 'output=' + str(x + 26) + ',' + str(y - (50
                  + (i * 40))) + '|'
               str_tmp += 'output=' + str(x + 26) + ',' + str(y + (35
                  + (i * 40))) + '|'

       # input
       # p_input_x = str(x - 4)
       # p_input_y = str(y - 10)
       if input % 2 == 0:
           for i in range(int(input / 2)):
               self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
                   10)
               self.draw_line(x - 13, y + (30 + (i * 40)), x - 4, y -
                   10)
```

```
                str_tmp += 'input=' + str(x - 13) + ',' + str(y - (50 +
                    (i * 40))) + '|'
                str_tmp += 'input=' + str(x - 13) + ',' + str(y + (30 +
                    (i * 40))) + '|'
        else:
            self.draw_line(x - 13, y - 10, x - 4, y - 10)
            str_tmp += 'input=' + str(x - 13) + ',' + str(y - 10) + '|'
            for i in range(int(input / 2)):
                self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
                   10)
                self.draw_line(x - 13, y + (35 + (i * 40)), x - 4, y -
                   10)
                str_tmp += 'input=' + str(x - 13) + ',' + str(y - (50 +
                    (i * 40))) + '|'
                str_tmp += 'input=' + str(x - 13) + ',' + str(y + (35 +
                    (i * 40))) + '|'

        str_tmp += "\n"

        with open('gen_input_output_detial.txt', 'a+') as f:
            f.write(str_tmp)

    def draw_generator(self, x, y, input, output, gen_color, Name):

        self.create_circle(x + 6, y - 11, 10, gen_color)

        column = x // (40)
        row = y // (40)

        # Generator
        self.str_GT = str(column) + ',' + str(self.num_gen) + ',' + str
           (x) + ',' + str(y) + ',' + str(input) + ',' + str(output) +
             ',' + gen_color + ',' + Name + '\n'
        with open('create_gt.txt', 'a+') as f:
           f.write(self.str_GT)

        self.Matrix_grid[row, column] = self.num_gen
        self.Matrix_grid[row + 1, column] = self.num_gen
        self.Matrix_grid[row - 1, column] = self.num_gen
        print(self.Matrix_grid)

        row += 1
        column += 1

        #print('Create gen {} , {}'.format(row, column))
```

```
# output
str_tmp = ''
#p_output_x = str(x + 17)
#p_output_y = str(y - 10)
if output % 2 == 0:
    for i in range(int(output / 2)):
        self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
             10)
        self.draw_line(x + 26, y + (30 + (i * 40)), x + 17, y -
             10)
        str_tmp += 'output,' + str(x + 26) + ',' + str(y - (50
           + (i * 40))) + '\n'
        str_tmp += 'output,' + str(x + 26) + ',' + str(y + (30
           + (i * 40))) + '\n'
else:
    self.draw_line(x + 26, y - 10, x + 17, y - 10)
    str_tmp += 'output,' + str(x + 26) + ',' + str(y - 10) + '\
       n'
    for i in range(int(output / 2)):
        self.draw_line(x + 26, y - (50 + (i * 40)), x + 17, y -
             10)
        self.draw_line(x + 26, y + (35 + (i * 40)), x + 17, y -
             10)
        str_tmp += 'output,' + str(x + 26) + ',' + str(y - (50
           + (i * 40))) + '\n'
        str_tmp += 'output,' + str(x + 26) + ',' + str(y + (35
           + (i * 40))) + '\n'


# input
#p_input_x = str(x - 4)
#p_input_y = str(y - 10)
if input % 2 == 0:
    for i in range(int(input / 2)):
        self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
            10)
        self.draw_line(x - 13, y + (30 + (i * 40)), x - 4, y -
            10)
        str_tmp += 'input,' + str(x - 13) + ',' + str(y - (50 +
             (i * 40))) + '\n'
        str_tmp += 'input,' + str(x - 13) + ',' + str(y + (30 +
             (i * 40))) + '\n'
else:
    self.draw_line(x - 13, y - 10, x - 4, y - 10)
    str_tmp += 'input,' + str(x - 13) + ',' + str(y - 10) + '\n
       '
```

```
        for i in range(int(input / 2)):
            self.draw_line(x - 13, y - (50 + (i * 40)), x - 4, y -
                10)
            self.draw_line(x - 13, y + (35 + (i * 40)), x - 4, y -
                10)
            str_tmp += 'input,' + str(x - 13) + ',' + str(y - (50 +
                (i * 40))) + '\n'
            str_tmp += 'input,' + str(x - 13) + ',' + str(y + (35 +
                (i * 40))) + '\n'

    #with open('gen_input_output_detial.txt', 'a+') as f:
        #f.write(str_tmp)

def draw_line(self, x1, y1, x2, y2):
    self.myCanvas.create_line(x1, y1, x2, y2, fill='blue', width=2)

def create_circle(self, x, y, r, gen_color): # center coordinates,
    radius
    x0 = x - r
    y0 = y - r
    x1 = x + r
    y1 = y + r
    if gen_color == 'black':
        return self.myCanvas.create_oval(x0, y0, x1, y1, fill
            ="#000000", outline="#000000")
    else:
        return self.myCanvas.create_oval(x0, y0, x1, y1, fill="#
            ffffff", outline="#000000")

def draw_twist(self, x, y):
    column = x // (40)
    row = y // (40)

    self.Matrix_grid[row, column] = -2
    self.Matrix_grid[row + 1, column] = -2
    self.Matrix_grid[row - 1, column] = -2
    print(self.Matrix_grid)

    row += 1
    column += 1

    print('Create twist {} , {}'.format(row, column))

    xt = 40 * column
    yt = 40 * row
```

```
        self.draw_line(xt + 4, yt - 55, xt - 36, yt + 30)
        self.draw_line(xt + 5, yt + 30, xt - 36, yt - 56)

    def get_input(self, x, y, column):

        Name = self.e4.get()
        print("Name", Name)

        Input = self.e5.get()
        print("Input", Input)

        Output = self.e6.get()
        print("Output", Output)

        input = self.e5.get()
        output = self.e6.get()

        if int(input) > 20 or int(output) > 20:
            tk.messagebox.showerror('error', 'Input or output should
                not exceed 20!', icon='error')
            return

        if Name.find('id') > -1:
            tk.messagebox.showerror('error', 'Not allowed to name
                object id!', icon='error')
            return

        if int(Input) < 0:
            tk.messagebox.showerror('error', 'Negative inputs not
                allowed!', icon='error')
            return

        if int(Output) < 0:
            tk.messagebox.showerror('error', 'Negative outputs not
                allowed!', icon='error')
            return

        if Name == '':
            tk.messagebox.showerror('error', 'Please enter a name for
                ths generator', icon='error')
            return

        generator_color = self.monthchoosen.get()

        self.counter += 1
        self.generator_dict[self.counter] = Name
        print(self.generator_dict)

        if input != '' and output != '' and self.status == 'Graph ->
            text':
             self.set_num_gen(self.counter)
             self.draw_generator(x, y, int(input), int(output),
                 generator_color, Name)
             self.gen_column.append(column)
        else:
            log = Name + '|' + str(self.counter) + '|' + input + '|' +
                output + '|' + generator_color + '\n'
            with open('generator_info.txt', 'a+') as writer:
                 writer.write(log)

        # Save string for export
        #self.str_GT += '(' + Name + ');'
        #with open('Graph_to_text.txt', 'w') as f:
         #    f.write(self.str_GT)

        self.window2.destroy()

    def get_coordinate(self, row, column):
        x = column * self.cellwidth + 4
        y = row * self.cellheight + 4
        return x, y

    def order_node(self, txt):

        # Mark id
        r = 0
        c = 0
        [rows, cols] = self.Matrix_grid.shape

        # txt = self.T.get().strip() # read input text
        Semi_colon = txt.split(';')  # split input text

        # Mark id
        for i in range(len(Semi_colon)):
            Sub_semi_colon = Semi_colon[i]
            Star = Sub_semi_colon.split('*')
            for j in range(len(Star)):
                Sub_star = Star[j].replace('(', '').replace(')', '').
                    strip()
                if Sub_star == 'id':
                     self.Matrix_grid[r, c] = -9
                    # print(self.Matrix_grid)
            r += 1
        else:  # is generator
            # r = 0
            for m in range(rows):
                if self.Matrix_grid[m, c] > 0:
                    r += 3
                    break
                    # else:
                    # r += 1
                    # r += 3
    c += 2
    r = 0

#print(self.Matrix_grid)

# Order node
for i in range(cols):
    ns = 0
    count = 0
    replace_count = 0
    for j in range(rows):
        v = self.Matrix_grid[j, i]
        if v != 0 and v != ns:
            ns = v
            count += 1
            self.Matrix_order_node[j, i] = count
            ns = self.Matrix_grid[j, i]
            # print(self.Matrix_id_line)
        elif v != 0 and v == ns and v != -9:
            ns = v
            replace_count += 1
            if replace_count >= 2:
                replace_count = 0
                count += 1
                self.Matrix_order_node[j, i] = count
                # print(self.Matrix_id_line)
            else:
                self.Matrix_order_node[j, i] = -7
        elif v != 0 and v == ns and v == -9:
            ns = v
            replace_count += 1
            if replace_count > 0:
                replace_count = 0
                count += 1
                self.Matrix_order_node[j, i] = count
        #print(self.Matrix_order_node)

        return self.Matrix_order_node, self.Matrix_grid
        # print(self.Matrix_id_line)

    def draw_line_connecting(self, M_order_node, M_grid,
        num_first_output, pair):

        #print("Start M_order_node : {}".format(M_order_node))
        #print("Start M_grid : {}".format(M_grid))

        M_order_node_draw_line_end = M_order_node.copy()

        x_coordinate = self.Matrix_x
        y_coordinate = self.Matrix_y
        [rows, cols] = M_order_node.shape
        M_order_node_copy = M_order_node

        # Draw id line
        for i in range(cols):
            for j in range(rows):
                x1 = x_coordinate[j, i]
                y1 = y_coordinate[j, i]
                if M_grid[j, i] == -9:
                    self.myCanvas.create_line(x1, y1 - 18, x1 - 40, y1
                       - 18, fill='black', width=2)

        count = 0
        inputs = 0
        outputs = 0

        # Draw connecting
        for i in range(len(pair)):

            P = pair[i]

            outputs = P[0]
            inputs = P[1]

            count += 1

            if outputs % 2 == 0 and inputs % 2 == 0 and i == 0:

                #if i == 0:

                    for j in range(rows):
```

```
            v1 = M_order_node[j, i]
            x1 = x_coordinate[j, i]
            y1 = y_coordinate[j, i]

            if v1 != 0 and v1 != -7:  # Fist couple

                for j2 in range(rows):
                    v2 = M_order_node[j2, i + 2]
                    x2 = x_coordinate[j2, i + 2]
                    y2 = y_coordinate[j2, i + 2]

                    # draw line
                    if v1 == v2:

                        if i >= 0:
                            if y2 < y1:
                                # self.draw_line(x1, y1 -
                                    18, x2 - 40, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 - 40,
                                    y2 - 18, fill='red',
                                                          width
                                                             =2)

                            elif y2 > y1:
                                # self.draw_line(x1, y1 -
                                    18, x2 + 40, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 + 40,
                                    y2 - 18, fill='red',
                                                          width
                                                             =2)

                            else:
                                # self.draw_line(x1, y1 -
                                    18, x2, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 - 40,
                                    y2 - 18, fill='red',
                                                          width
                                                             =2)


elif outputs % 2 == 0  and inputs % 2 == 0 and i > 0:
```

```
#elif i > 0:

    # adjust output
    for s in range(rows):
        a = M_order_node_copy[s, i]
        b = M_grid[s, i]
        if b != -9 and a != -7:
            M_order_node_copy[s, i] = 0

    cs = 0
    for s in range(rows):
        c = M_order_node_copy[s, i]
        if c != 0:
            cs += 1
            M_order_node_copy[s, i] = cs

    # print('M_order_node_copy : \n{}'.format(
        M_order_node_copy))

    for j in range(rows):

        v1 = M_order_node_copy[j, i]
        x1 = x_coordinate[j, i]
        y1 = y_coordinate[j, i]

        if v1 != 0 and v1 != -7:  # Next couple

            # draw connect
            for j2 in range(rows):
                v2 = M_order_node[j2, i + 2]
                x2 = x_coordinate[j2, i + 2]
                y2 = y_coordinate[j2, i + 2]

                # draw line
                if v1 == v2:

                    if i >= 0:
                        if y2 < y1:
                            # self.draw_line(x1, y1 -
                                18, x2 - 40, y2 - 18)
                            self.myCanvas.create_line(
                                x1, y1 - 18, x2 - 40,
                                y2 - 18, fill='green',
                                                      width
                                                          =2)
                            elif y2 > y1:
                                # self.draw_line(x1, y1 -
                                    18, x2 + 40, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 + 40,
                                    y2 - 18, fill='green',
                                                           width
                                                              =2)

                            else:
                                # self.draw_line(x1, y1 -
                                    18, x2, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 - 40,
                                    y2 - 18, fill='green',
                                                           width
                                                              =2)


elif outputs % 2 != 0 and inputs % 2 != 0 and i == 0:

    #if i >= 0:

        print("01 - outputs % 2 != 0 and inputs % 2 != 0
            and i == 0")

        # adjust output
        for s in range(rows):
            a = M_order_node_copy[s, i]
            b = M_grid[s, i]
            if b != -9 and a != -7:
                M_order_node_copy[s, i] = 0

        cs = 0
        for s in range(rows):
            c = M_order_node_copy[s, i]
            if c != 0:
                cs += 1
                M_order_node_copy[s, i] = cs
        if i == 0:
            a = 1

        print('Modulus > 0 , M_order_node_copy : \n{}'.
            format(M_order_node_copy))

        for j in range(rows):
            v1 = M_order_node_copy[j, i]
            x1 = x_coordinate[j, i]
            y1 = y_coordinate[j, i]

            if v1 != 0 and v1 != -7:  # Next couple

                # draw connect
                for j2 in range(rows):
                    v2 = M_order_node[j2, i + 2]
                    x2 = x_coordinate[j2, i + 2]
                    y2 = y_coordinate[j2, i + 2]

                    # draw line
                    if v2 == -7: #v1 == v2:

                        if i >= 0:
                            if y2 < y1:
                                # self.draw_line(x1, y1 -
                                    18, x2 - 40, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 - 40,
                                    y2 - 18, fill='yellow',
                                                          width
                                                             =2)

                            elif y2 > y1:
                                # self.draw_line(x1, y1 -
                                    18, x2 + 40, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 + 40,
                                    y2 - 18, fill='yellow',
                                                          width
                                                             =2)

                            else:
                                # self.draw_line(x1, y1 -
                                    18, x2, y2 - 18)
                                self.myCanvas.create_line(
                                    x1, y1 - 18, x2 - 40,
                                    y2 - 18, fill='pink',
                                                          width
                                                             =2)


elif outputs % 2 != 0 and inputs % 2 != 0 and i > 0:
```

```
print('02 - outputs % 2 != 0 and inputs % 2 != 0 and i
   ')

# adjust output
for s in range(rows):
    a = M_order_node_copy[s, i]
    b = M_grid[s, i]
    if b != -9 and a != -7:
        M_order_node_copy[s, i] = 0

cs = 0
for s in range(rows):
    c = M_order_node_copy[s, i]
    if c != 0:
        cs += 1
        M_order_node_copy[s, i] = cs

# print('M_order_node_copy : \n{}'.format(
   M_order_node_copy))

for j in range(rows):

    v1 = M_order_node_copy[j, i]
    x1 = x_coordinate[j, i]
    y1 = y_coordinate[j, i]

    if v1 != 0 and v1 != -7:  # Next couple

        # draw connect
        for j2 in range(rows):
            v2 = M_order_node[j2, i + 2]
            x2 = x_coordinate[j2, i + 2]
            y2 = y_coordinate[j2, i + 2]

            # draw line
            if v2 == -7: #v1 == v2:

                if i >= 0:
                    if y2 < y1:
                        # self.draw_line(x1, y1 - 18,
                            x2 - 40, y2 - 18)
                        self.myCanvas.create_line(x1,
                            y1 - 18, x2 - 40, y2 - 18,
                            fill='green',
                                                   width
```

```
                                                          =2)

                        elif y2 > y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 + 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 + 40, y2 - 18,
                                fill='green',
                                                       width
                                                          =2)

                        else:
                            # self.draw_line(x1, y1 - 18,
                                x2, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='green',
                                                       width
                                                          =2)


elif outputs % 2 == 0 and inputs % 2 != 0 and i == 0:
    print("1 - outputs % 2 == 0 and inputs % 2 != 0 and i
       == 0")

    i *= 2

    # adjust output
    for s in range(rows):
        a = M_order_node_copy[s, i]
        b = M_grid[s, i]
        if b != -9 and a != -7:
            M_order_node_copy[s, i] = 0

    cs = 0
    for s in range(rows):
        c = M_order_node_copy[s, i]
        if c != 0:
            cs += 1
            M_order_node_copy[s, i] = cs

    # print('M_order_node_copy : \n{}'.format(
       M_order_node_copy))

    for j in range(rows):
        v1 = M_order_node_copy[j, i]
        x1 = x_coordinate[j, i]
        y1 = y_coordinate[j, i]

        if v1 != 0 and v1 != -7:  # Next couple

            # draw connect
            for j2 in range(rows):
                v2 = M_order_node[j2, i + 2]
                x2 = x_coordinate[j2, i + 2]
                y2 = y_coordinate[j2, i + 2]

                # draw line
                if v1 == v2:

                    if i >= 0:
                        if y2 < y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 - 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)

                        elif y2 > y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 + 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 + 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)

                        else:
                            # self.draw_line(x1, y1 - 18,
                                x2, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)


elif outputs % 2 == 0 and inputs % 2 != 0 and i > 0:
    print("2 - outputs % 2 == 0 and inputs % 2 != 0 and i >
        0")

    i *= 2

    # adjust output
    for s in range(rows):
        a = M_order_node_copy[s, i]
        b = M_grid[s, i]
        if b != -9 and a != -7:
            M_order_node_copy[s, i] = 0

    cs = 0
    for s in range(rows):
        c = M_order_node_copy[s, i]
        if c != 0:
            cs += 1
            M_order_node_copy[s, i] = cs

    # print('M_order_node_copy : \n{}'.format(
       M_order_node_copy))

    for j in range(rows):

        v1 = M_order_node_copy[j, i]
        x1 = x_coordinate[j, i]
        y1 = y_coordinate[j, i]

        if v1 != 0 and v1 != -7:  # Next couple

            # draw connect
            for j2 in range(rows):
                v2 = M_order_node[j2, i + 2]
                x2 = x_coordinate[j2, i + 2]
                y2 = y_coordinate[j2, i + 2]

                # draw line
                if v1 == v2:

                    if i >= 0:
                        if y2 < y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 - 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
```

```
                                                          =2)

                        elif y2 > y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 + 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 + 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)

                        else:
                            # self.draw_line(x1, y1 - 18,
                                x2, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)


elif outputs % 2 != 0 and inputs % 2 == 0 and i == 0:
    print("3 - outputs % 2 != 0 and inputs % 2 == 0 and i
       == 0")

    i *= 2

    # adjust output
    for s in range(rows):
        a = M_order_node_copy[s, i]
        b = M_grid[s, i]
        if b != -9 and a != -7:
            M_order_node_copy[s, i] = 0

    cs = 0
    for s in range(rows):
        c = M_order_node_copy[s, i]
        if c != 0:
            cs += 1
            M_order_node_copy[s, i] = cs

    # print('M_order_node_copy : \n{}'.format(
       M_order_node_copy))

    for j in range(rows):
        v1 = M_order_node_copy[j, i]
        x1 = x_coordinate[j, i]
        y1 = y_coordinate[j, i]

        if v1 != 0 and v1 != -7:  # Next couple

            # draw connect
            for j2 in range(rows):
                v2 = M_order_node[j2, i + 2]
                x2 = x_coordinate[j2, i + 2]
                y2 = y_coordinate[j2, i + 2]

                # draw line
                if v1 == v2:

                    if i >= 0:
                        if y2 < y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 - 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)

                        elif y2 > y1:
                            # self.draw_line(x1, y1 - 18,
                                x2 + 40, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 + 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)

                        else:
                            # self.draw_line(x1, y1 - 18,
                                x2, y2 - 18)
                            self.myCanvas.create_line(x1,
                                y1 - 18, x2 - 40, y2 - 18,
                                fill='pink',
                                                       width
                                                          =2)


elif outputs % 2 != 0 and inputs % 2 == 0 and i > 0:
    print("4 - outputs % 2 != 0 and inputs % 2 == 0 and i >
    0”)

i *= 2

# adjust output
for s in range(rows):
    a = M_order_node_copy[s, i]
    b = M_grid[s, i]
    if b != -9 and a != -7:
        M_order_node_copy[s, i] = 0

cs = 0
for s in range(rows):
    c = M_order_node_copy[s, i]
    if c != 0:
        cs += 1
        M_order_node_copy[s, i] = cs

# print('M_order_node_copy : \n{}'.format(
   M_order_node_copy))

for j in range(rows):

    v1 = M_order_node_copy[j, i]
    x1 = x_coordinate[j, i]
    y1 = y_coordinate[j, i]

    if v1 != 0 and v1 != -7:  # Next couple

        # draw connect
        for j2 in range(rows):
            v2 = M_order_node[j2, i + 2]
            x2 = x_coordinate[j2, i + 2]
            y2 = y_coordinate[j2, i + 2]

            # draw line
            if v1 == v2:

                if i >= 0:
                    if y2 < y1:
                        # self.draw_line(x1, y1 - 18,
                            x2 - 40, y2 - 18)
                        self.myCanvas.create_line(x1,
                            y1 - 18, x2 - 40, y2 - 18,
                            fill='yellow',
                                                   width
```

```
                                                                  =2)

                              elif y2 > y1:
                                  # self.draw_line(x1, y1 - 18,
                                      x2 + 40, y2 - 18)
                                  self.myCanvas.create_line(x1,
                                      y1 - 18, x2 + 40, y2 - 18,
                                      fill='yellow',
                                                               width
                                                                  =2)

                              else:
                                  # self.draw_line(x1, y1 - 18,
                                      x2, y2 - 18)
                                  self.myCanvas.create_line(x1,
                                      y1 - 18, x2 - 40, y2 - 18,
                                      fill='yellow',
                                                               width
                                                                  =2)



print('Last output : {}'.format(outputs))

if outputs == 1:

    # Draw line end
    M_order_node_copy = M_order_node_draw_line_end
    print('M_order_node_copy : \n{}'.format(M_order_node_copy))
    print('Draw line end')
    [r, c] = M_order_node_copy.shape
    col_ind = 0
    for i in range(r - 1, -1, -1):
        #print(M_order_node_copy[:, i])
        #print(sum(M_order_node_copy[:, i]))
        if sum(M_order_node_copy[:, i]) != 0:
            col_ind = i
            for j in range(r):
                #print(M_order_node_copy[:, col_ind])
                if M_order_node_copy[j, col_ind] < 0:
                    x1 = x_coordinate[j, col_ind]
                    y1 = y_coordinate[j, col_ind] - 18
                    x2 = x_coordinate[j, c - 1]
                    y2 = y1
                    self.myCanvas.create_line(x1, y1, x2, y2,
                        fill='orange', width=2)
                    break

        elif outputs > 1:

            # Draw line end
            M_order_node_copy = M_order_node_draw_line_end
            print('M_order_node_copy : \n{}'.format(M_order_node_copy))
            print('Draw line end')
            [r, c] = M_order_node_copy.shape
            col_ind = 0
            for i in range(r - 1, -1, -1):
                #print(M_order_node_copy[:, i])
                #print(sum(M_order_node_copy[:, i]))
                if sum(M_order_node_copy[:, i]) != 0:
                    col_ind = i
                    for j in range(r - 1):
                        #print(M_order_node_copy[:, col_ind])
                        if M_order_node_copy[j + 1, col_ind] >
                            M_order_node_copy[j, col_ind] and
                            M_order_node_copy[j, col_ind] > 0:
                            x1 = x_coordinate[j, col_ind]
                            y1 = y_coordinate[j, col_ind] - 18
                            x2 = x_coordinate[j, c - 1]
                            y2 = y1
                            self.myCanvas.create_line(x1, y1, x2, y2,
                                fill='orange', width=2)
                        if M_order_node_copy[j + 1, col_ind] <
                            M_order_node_copy[j, col_ind] and
                            M_order_node_copy[j, col_ind] > 0:
                            x1 = x_coordinate[j, col_ind]
                            y1 = y_coordinate[j, col_ind] - 18
                            x2 = x_coordinate[j, c - 1]
                            y2 = y1
                            self.myCanvas.create_line(x1, y1, x2, y2,
                                fill='orange', width=2)
                    break

    def create_graph_of_text(self):

        x_coordinate = self.Matrix_x
        y_coordinate = self.Matrix_y

        #if path.exists('pair.txt'):
            #os.remove('pair.txt')

        #print(self.Matrix_grid)
```

```
pair = []
sub_pair = [0, 0]
str_pair = []
sub_str_pair = [[], []]
count_pair = 0

# read string diagram
with open('generator_info.txt') as reader:
    content = reader.readlines()

row = 2
column = 1
x = 34
y = 75
do_first = False
have_gen = False
have_id = False
output_prev = 0
detect_first_output = False
num_first_output = 0

txt = self.T.get().strip() # read input text
Semi_colon = txt.split(';') # split input text

# Split semi colon
for i in range(len(Semi_colon)):

    row = 2

    Sub_semi_colon = Semi_colon[i]
    #print('Sub_semi_colon : {}'.format(Sub_semi_colon))
    Star = Sub_semi_colon.split('*')

    find_id = Sub_semi_colon.find('id') # find have id

    # Split star
    for j in range(len(Star)):
        Sub_star = Star[j].replace('(', '').replace(')', '').
           strip()
        if Sub_star == 'id': # id

            have_id = True
            row += 1

        else:
```

```
for k in range(len(content)): # genenrator name

    E = content[k].split('|')
    Name = E[0]
    input = int(E[2])
    output = int(E[3])
    generator_color = E[4].rstrip()
    if Name == Sub_star:

        #column += 2

        if not detect_first_output:
            num_first_output = output
            detect_first_output = True

        # draw generator
        x1 = column * 40
        y1 = row * 40
        dx = x - x1
        dy = y - y1

        x = x - dx - 22
        y = y - dy - 4

        self.counter_text += 1
        self.generator_dict[self.counter_text] =
            Name
        self.set_num_gen(self.counter_text)
        #self.draw_generator_TG(x, y, input, output
            , generator_color, Name)
        self.draw_generator(x, y, input, output,
            generator_color, Name)
        self.gen_column.append(column)

        row += 3

        print("self.Matrix_grid : \n{}".format(self
            .Matrix_grid))

        #column += 2

        #print(self.Matrix_grid)
        #print(self.counter_text)
        #print("column : {}".format(column))

        # Map pair output - input
```

```
                    count_pair += 1

                    if count_pair == 1:
                        #sub_pair.append([output])
                        #sub_str_pair.append([Name, 'output'])
                        sub_str_pair[0] = [Name, 'output']
                        sub_pair[0]= output
                    elif count_pair == 2:

                        #sub_pair.append([input])
                        sub_pair[1] = input
                        pair.append(sub_pair)
                        sub_pair = [0, 0]
                        sub_pair[0] = output

                        #sub_pair.append([output])

                        #sub_str_pair.append([Name, 'input'])

                        sub_str_pair[1] = [Name, 'input']
                        str_pair.append(sub_str_pair)
                        sub_str_pair = [[], []]
                        sub_str_pair[0] = [Name, 'output']

                        #sub_str_pair.append([Name, 'output'])
                        #str_pair.append([Name, 'output'])

                        count_pair = 1

                    break

    column += 2

print("self.Matrix_grid : \n{}".format(self.Matrix_grid))

str_pair.append(sub_str_pair)
pair.append(sub_pair)

# Draw line connecting
M_order_node, M_grid = self.order_node(txt)
#self.draw_line_connecting(M_order_node, M_grid,
    num_first_output, str_pair, pair)
self.draw_line_connecting(M_order_node, M_grid,
    num_first_output, pair)

print("M_order_node : \n{}".format(M_order_node))
```

```
        print("M_grid : \n{}".format(M_grid))

        self.window3.destroy()

        #self.myCanvas.create_text(400, 705, fill="red", font="Arial 20
            bold",
                                   #text=txt)

        #print(self.Matrix_grid)
        #print(pair)
        #print(str_pair)

    def click_right_button(self, eventorigin):

        if self.status == 'Graph -> text':
            if self.twist_status:
                self.create_twist(eventorigin)
        else:

            # Sub figure
            self.window3 = tk.Tk()
            self.window3.title("Generator created")
            self.window_height = 240
            self.window_width = 330

            self.screen_width = self.window3.winfo_screenwidth()
            self.screen_height = self.window3.winfo_screenheight()

            self.x_coordinate = int((self.screen_width / 2) - (self.
               window_width / 2))
            self.y_coordinate = int((self.screen_height / 2) - (self.
               window_height / 2))

            self.window3.geometry(
                "{}x{}+{}+{}".format(self.window_width, self.
                   window_height, self.x_coordinate, self.y_coordinate
                   ))

            TitleLabel = ttk.Label(self.window3)

            listNodes = Listbox(self.window3, width=50, height=10)
            listNodes.place(x=0, y=0)

            scrollbar = Scrollbar(self.window3, orient="vertical")
            scrollbar.config(command=listNodes.yview)
            scrollbar.place(x=305, y=0)
            listNodes.config(yscrollcommand=scrollbar.set)

            with open('generator_info.txt') as reader:
                content = reader.readlines()
            for i in range(len(content)):
                A = content[i].split('|')
                tmp = 'Name : ' + A[0] + ' , Input : ' + A[2] + ' ,
                    Output : '+ A[3] + ' , Color : ' + A[4]
                listNodes.insert(END, tmp)

            self.T = Entry(self.window3, width=43)
            self.T.place(x=40, y=170, bordermode="outside")

            l = Label(self.window3, text="Text : ")
            l.place(x=5, y=170, bordermode="outside")

            btn = ttk.Button(self.window3, text="Apply", command=self.
                create_graph_of_text)
            btn.place(x=228, y=200, bordermode="outside")

    def create_twist(self, eventorigin):

        x = eventorigin.x
        y = eventorigin.y

        column = x // (40)
        row = y // (40)

        row += 1
        column += 1

        self.draw_twist(x, y)

        print("Click ", (x, y), "coordinates: ", row, column)

        self.gen_column.append(column)

    def getorigin(self, eventorigin):

        if self.mode == 'connecting' and not self.Clicked:
            self.Clicked = True
            #print('Click true!')
            print('Click start line !')
            self.sub_line_intersection.append([eventorigin.x,
                eventorigin.y])
```

```
elif self.mode == 'connecting' and self.Clicked:
    self.Clicked = False
    #print('Click false !')
    print('Click stop line !')
    self.sub_line_intersection.append([eventorigin.x,
        eventorigin.y])
    self.line_intersection.append(self.sub_line_intersection)
    self.sub_line_intersection = []

    # Check have twist
    if len(self.line_intersection) == 2:
        line_1 = self.line_intersection[0]
        line_2 = self.line_intersection[1]
        point_line1_1 = line_1[0]
        point_line1_2 = line_1[1]
        point_line2_1 = line_2[0]
        point_line2_2 = line_2[1]
        self.line_intersection = []
        point_of_intersection = self.check_intersection_line(
            point_line1_1, point_line1_2, point_line2_1,
            point_line2_2)
        if len(point_of_intersection) == 2:
            x_intersection = point_of_intersection[0]
            y_intersection = point_of_intersection[1]
            column_intersection = int(x_intersection // (40))
            row_intersection = int(y_intersection // (40))
            print("column_intersection : {} , row_intersection
                : {}".format(column_intersection,
                row_intersection))
            self.Matrix_grid[:, column_intersection] = 0
            self.Matrix_grid[row_intersection,
                column_intersection] = -2
            self.gen_column.append(column_intersection + 1)
            print(self.Matrix_grid)


    # Twist detection
    #A = self.line_intersection[0]
    #print(A[0])
    #print(A[1])

x = eventorigin.x
y = eventorigin.y

column = x // (40)
row = y // (40)
```

```
# Remove generator or id
if self.status_remove == True:

    # assign matrix
    self.status_remove = False
    self.Matrix_grid[row, column] = 0
    self.Matrix_grid[row + 1, column] = 0
    self.Matrix_grid[row - 1, column] = 0
    print('Remove :{}\n'.format(self.Matrix_grid))

    # clear canvas
    self.myContainer1.destroy()
    self.myContainer1 = tk.Frame(self.myParent)
    self.myContainer1.pack()
    self.draw_grid(20, 20)
    self.myCanvas.update()

    # delete text gen
    tmp = ''
    with open("create_gt.txt", 'r') as file_in:
        for line in file_in:
            str = line.split(',')
            s_column = int(str[0])
            if column != s_column:
                tmp += line

    # save of delete
    with open('create_gt.txt', 'w') as f:
        f.write(tmp)

    # redraw generator
    with open("create_gt.txt", 'r') as file_in:
        for line in file_in:
            str = line.split(',')
            column = int(str[0])
            num_gen = int(str[1])
            x = int(str[2])
            y = int(str[3])
            input = int(str[4])
            output = int(str[5])
            gen_color = str[6]
            name = str[7].replace('\n', '')
            self.draw_generator_of_remove(x, y, input, output,
                gen_color, name)
    # redraw id
    x_coordinate = self.Matrix_x
    y_coordinate = self.Matrix_y

    [r, c] = self.Matrix_grid.shape
    for j in range(r):
        for i in range(c):
            if self.Matrix_grid[j, i] == -1:
                x1 = x_coordinate[j, i]
                y1 = y_coordinate[j, i]
                self.myCanvas.create_line(x1, y1 - 18, x1 - 40,
                    y1 - 18, fill='blue', width=2)

    return

# End remove or id


row += 1
column += 1

print("Click ", (x, y), "coordinates: ", row, column)
print(self.mode)

if  self.mode == 'connecting':
    self.line_position.append(x)
    self.line_position.append(y)
    if len(self.line_position) == 4:
        self.draw_line(self.line_position[0], self.
            line_position[1], self.line_position[2], self.
            line_position[3])
        self.line_position = [self.line_position[2], self.
            line_position[3]]
    else:
        print('Points to points : {}'.format(self.line_position
            ))
else:

    W = 40
    H = 40

    x1 = column * 40
    y1 = row * 40
    dx = x - x1
    dy = y - y1
```

```
x = x - dx - 22
y = y - dy - 4
# print('x : {} , y : {}'.format(x, y))

#panel.place(x=x, y=y)

if self.status_remove == False:

    # Sub figure
    self.window2 = tk.Tk() # Toplevel(self.myParent)
    self.window2.title(”Generator creator”)
    self.window_height = 200
    self.window_width = 260

    self.screen_width = self.window2.winfo_screenwidth()
    self.screen_height = self.window2.winfo_screenheight()

    self.x_coordinate = int((self.screen_width / 2) - (self
        .window_width / 2))
    self.y_coordinate = int((self.screen_height / 2) - (
        self.window_height / 2))

    self.window2.geometry(”{}x{}+{}+{}”.format(self.
        window_width, self.window_height, self.x_coordinate
        , self.y_coordinate))

    TitleLabel = ttk.Label(self.window2)

    L4 = ttk.Label(self.window2, text = ”Name : ”, justify=
        RIGHT)
    L4.place(x=30, y=20)

    self.e4 = ttk.Entry(self.window2, width = 20)
    self.e4.place(x=80, y=20, bordermode=”outside”)

    L5 = ttk.Label(self.window2, text = ”Input : ”, justify
        =RIGHT)
    L5.place(x=30, y=45)

    self.e5 = ttk.Entry(self.window2, width = 20)
    self.e5.place(x=80, y=45, bordermode=”outside”)

    L6 = ttk.Label(self.window2, text = ”Output : ”,
        justify=RIGHT)
    L6.place(x=20, y=70, bordermode=”outside”)
```

```
            self.e6 = ttk.Entry(self.window2, width = 20)
            self.e6.place(x=80, y=70, bordermode="outside")

            L7 = ttk.Label(self.window2, text = "Color : ", justify
                =RIGHT)
            L7.place(x=30, y=95)

            #self.variable = StringVar(self.window2)
            #self.variable.set("one")  # default value

            #gen_color = tk.StringVar()
            self.monthchoosen = ttk.Combobox(self.window2, width
                =10, value = ['black', 'white'])
            self.monthchoosen.current(0)
            self.monthchoosen.place(x=80, y=95, bordermode="outside
                ")

            btn = ttk.Button(self.window2, text = "Apply", command
                = lambda:self.get_input(x, y, column))
            btn.place(x=80, y=130, bordermode="outside")

def calculate(self):
    #global var, gen_column
    M = self.get_Matrix_grid()

    c = 0
    tmp = ''
    log = 0
    [row, col] = M.shape
    self.gen_column = list(dict.fromkeys(self.gen_column))
    self.gen_column = sorted(self.gen_column)
    for i in range(len(self.gen_column)):
        tmp += '('
        log = ''
        for j in range(row):
            if M[j, self.gen_column[i] - 1] == -1:
                tmp += 'id * '
            elif M[j, self.gen_column[i] - 1] == -2 and M[j, self.
                gen_column[i] - 1] != log:
                log = M[j, self.gen_column[i] - 1]
                tmp += 'twist * '
            elif M[j, self.gen_column[i] - 1] != 0 and M[j, self.
                gen_column[i] - 1] != log:
                log = M[j, self.gen_column[i] - 1]
                tmp += str(self.generator_dict.get(M[j, self.
                    gen_column[i] - 1])) + ' * '
            tmp = tmp[:-3]
            tmp += ') ; '

        print(tmp)


        '''
        # check count
        count = tmp.count(';')
        if count <= 1:
            tmp = tmp.replace(';', '')
        '''

        tmp = tmp.strip()[:-1]
        if len(tmp) < 4:
            tmp = ''
            messagebox.showinfo("Result", "Null")
        else:
            messagebox.showinfo("Result", tmp)

        #var.set(tmp)
        #print(len(tmp))

    def connecting(self):

        self.mode = 'connecting'
        self.Clicked = False
        self.line_position = []

    def cursor(self):
        self.mode = ''

    def twist(self):
        self.twist_status = True

def on_tab_selected(event, TAB1, TAB2):

    global draw_tab_1, draw_tab_2, grid_tab_1, grid_tab_2

    selected_tab = event.widget.select()
    tab_text = event.widget.tab(selected_tab, "text")
    if tab_text == 'Graph -> text' and not draw_tab_1:
        myapp = GridWindow(TAB1, 'Graph -> text')
        if grid_tab_1 == False:
            myapp.draw_grid(20, 20)
            grid_tab_1 = True
        draw_tab_1 = True
        draw_tab_2 = False
    elif tab_text == 'Text -> graph' and not draw_tab_2:
        myapp = GridWindow(TAB2, 'Text -> graph')
        if grid_tab_2 == False:
            myapp.draw_grid(20, 20)
            grid_tab_2 = True
        draw_tab_2 = True
        draw_tab_1 = False

        if path.exists('gen_input_output_detial.txt'):
            os.remove('gen_input_output_detial.txt')

        #MsgBox = tk.messagebox.askquestion('delete', 'Delete old
            generator ?', icon='warning')
        #if MsgBox == 'yes':
         #    os.remove('generator_info.txt')

if __name__ == '__main__':

    if path.exists('create_gt.txt'):
        os.remove('create_gt.txt')

    if path.exists('graph_to_text_export.txt'):
        os.remove('graph_to_text_export.txt')

    draw_tab_1 = False
    draw_tab_2 = False
    grid_tab_1 = False
    grid_tab_2 = False


    # GUI
    mainWindow = tk.Tk()
    mainWindow.title('[ Graph to text ] and  [ Text to graph ]')
    mainWindow.resizable(width=False, height=False)

    window_height = 840
    window_width = 820

    screen_width = mainWindow.winfo_screenwidth()
    screen_height = mainWindow.winfo_screenheight()

    x_coordinate = int((screen_width / 2) - (window_width / 2))
    y_coordinate = int((screen_height / 2) - (window_height / 2))
```

```
mainWindow.geometry(”{}x{}+{}+{}”.format(window_width,
    window_height, x_coordinate, y_coordinate))

TAB_CONTROL = ttk.Notebook(mainWindow)

TAB1 = ttk.Frame(TAB_CONTROL)
TAB2 = ttk.Frame(TAB_CONTROL)

TAB_CONTROL.add(TAB1, text='Graph -> text')
TAB_CONTROL.add(TAB2, text='Text -> graph')

TAB_CONTROL.pack(expand=1, fill=”both”)

#var = StringVar()
#T = Entry(mainWindow, textvariable = var, width = 90)
#T.place(x=265, y=835, bordermode=”outside”)

#l = Label(mainWindow, text=”Result : ”)
#l.config(font=(”Courier”, 10))
#l.place(x=210, y=835, bordermode=”outside”)

TAB_CONTROL.bind(”<<NotebookTabChanged>>”, lambda event:
    on_tab_selected(event, TAB1, TAB2))

mainWindow.mainloop()
```

Appendix C

Project Plan - 26/10/2020

C.1 Aim

- To write a program that can represent a String diagram in the form of a graph or text, that can be converted to the other form.

C.2 Objectives

- Research the relevant data structures that can be used to represent a String diagram. As the program will be written in Python, NumPy is a Python library that can help with data structures.
- Work on converting a textual representation of a String diagram to a graph. This involves recusively parsing the textual representation and breaking it down into smaller and smaller parts to get the structure of the representation. This structure can then be represented to a graph.
- Work on converting a graphical representation of a String diagram to text. As there are different ways to represent the same String diagram into different kinds of textual forms, this will involve implementing rules to prioritize the form of text used.
- Add small improvements to the program, for example, adding a GUI or documentation for users. If there is time, a setting to adjust a preference to which kind of textual representation the user prefers can also be implemented.

C.3 Deliverables

- A program to convert the different types of String diagram.
- User documentation on how to use the program.
- Example inputs and outputs of String diagrams in both graphical and textual form used to test the program.

C.4 Work plan

- October - November: Establish a plan on the project
- November – December: Writing the main features of the program
- December – January: Depending on the progress, I can still be working on the main features or starting to implement improvements or extra features.
- January – February: Improvements + Testing
- February – March: Work on Final Report

Bibliography

[1] J. Baez and M. Stay. “Physics, Topology, Logic and Computation: A Rosetta Stone”. In: *Lecture Notes in Physics* (2010), pp. 95–172. ISSN: 1616-6361. DOI: 10.1007/978-3-642-12821-9_2. URL: http://dx.doi.org/10.1007/978-3-642-12821-9_2.

[2] John C. Baez and Jason Erbele. *Categories in Control*. 2015. arXiv: 1405.6881 [math.CT].

[3] Filippo Bonchi et al. “Diagrammatic Algebra: From Linear to Concurrent Systems”. In: (2019). URL: https://www.ioc.ee/~pawel/papers/popl19.pdf.

[4] Bob Coecke. *An alternative Gospel of structure: order, composition, processes*. 2013. arXiv: 1307.4038 [math.CT].

[5] Wikimedia Commons. *Commutative diagram for morphism*. Dec. 2006. URL: https://commons.wikimedia.org/wiki/File:Commutative_diagram_for_morphism.svg.

[6] Wikimedia Commons. *Monoidal2*. Mar. 2020. URL: https://commons.wikimedia.org/wiki/File:Monoidal2.svg.

[7] Wikimedia Commons. *Natural Transformation between two functors*. Mar. 2020. URL: https://commons.wikimedia.org/wiki/File:Natural_Transformation_between_two_functors.svg.

[8] Wikimedia Commons. *Pentagonal diagram for monoidal categories*. Mar. 2020. URL: https://commons.wikimedia.org/wiki/File:Pentagonal_diagram_for_monoidal_categories.svg.

[9] Andrew Fagan and Ross Duncan. “Optimising Clifford Circuits with Quantomatic”. In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), pp. 85–105. ISSN: 2075-2180. DOI: 10.4204/eptcs.287.5. URL: http://dx.doi.org/10.4204/EPTCS.287.5.

[10] Giovanni de Felice, Alexis Toumi, and Bob Coecke. “DisCoPy: Monoidal Categories in Python”. In: *Electronic Proceedings in Theoretical Computer Science* 333 (Feb. 2021), pp. 183–197. ISSN: 2075-2180. DOI: 10.4204/eptcs.333.13. URL: http://dx.doi.org/10.4204/EPTCS.333.13.

[11] Micah Halter et al. *Compositional Scientific Computing with Catlab and SemanticModels*. 2020. arXiv: 2005.04831 [math.CT].

[12] nLab authors. *braided monoidal category*. http://ncatlab.org/nlab/show/braided\%20monoidal\%20category. Mar. 2021.

[13] nLab authors. *category theory*. http://ncatlab.org/nlab/show/category\%20theory. Mar. 2021.

[14] nLab authors. *functor*. http://ncatlab.org/nlab/show/functor. Mar. 2021.

[15] nLab authors. *monoidal category.* `http://ncatlab.org/nlab/show/monoidal\%20category`. Mar. 2021.

[16] nLab authors. *tensor product.* `http://ncatlab.org/nlab/show/tensor\%20product`. Mar. 2021.

[17] Peter Selinger. "A Survey of Graphical Languages for Monoidal Categories". In: *Lecture Notes in Physics* 813 (Aug. 2009). DOI: `10.1007/978-3-642-12821-9_4`.

[18] Paweł Sobociński, Paul Wilson, and Fabio Zanasi. "CARTOGRAPHER: a tool for string diagrammatic reasoning". In: (2019).