Discrete Math with SageMath

Learn math with open-source software

Discrete Math with SageMath

Learn math with open-source software

Zunaid Ahmed, Hellen Colman, Samuel Lubliner
City Colleges of Chicago

February 19, 2025

Website: GitHub Repository[1]

[1] github.com/SageMathOER-CCC/sage-discrete-math
[2] creativecommons.org/licenses/by/4.0

Preface

This book was written by undergraduate students at Wright College who were enrolled in my Math 299 class, Writing in the Sciences.

For many years, I have been teaching Discrete Math using the open source mathematical software SageMath. Despite the fabulous capabilities of this software, students were often frustrated by the lack of specific documentation geared towards beginning undergrad students in Discrete Math.

This book was born out of this frustration and the desire to make our own contribution to the Open Education movement, from which we have benefitted greatly. In the context of Open Pedagogy, my students and I ventured into a challenging learning experience based on the principles of freedom and responsibility. Each week, students wrote a chapter of this book. They found the topics and found their voice. We critically analyzed their writing, and they edited and edited again and again. They wrote code, tested it and polished it. In the process, we all learned so much about Sage, and we found some bugs in the software that are now in the process of being fixed thanks to its very active community of developers.

The result is the book we dreamed of having when we first attempted Discrete Math with Sage.

Our book is intended to provide concise and complete instructions on how to use different Sage functions to solve problems in Discrete Math. Our goal is to streamline the learning process, helping students focus more on mathematics and reducing the friction of learning how to code. Our textbook is interactive and designed for all math students, regardless of programming experience. Rooted in the open education philosophy, our textbook is, and always will be, free for all.

I am very proud of the work of my students and hope that this book will serve as inspiration for other students to take ownership of a commons-based education. Towards a future where higher education is equally accessible to all.

Hellen Colman
Chicago, May 2024

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- Moty Katzman, University of Sheffield
- Ken Levasseur, University of Massachusetts Lowell
- Vartuyi Manoyan, Wright College

We would like to acknowledge the following proof-readers:

- Ted Jankowski, Wright College
- Justin Lowry, Wright College
- Yolanda Nieves, Wright College
- Fabio Re, Rosalind Franklin University
- Tineka Scalzo, Wright College

The making of this book is supported in part by the Wright College Open Educational Resource Expansion grant from the Secretary of State/Illinois State Library.

From the Student Authors

This textbook is a testament to our collaborative spirit and the Open Education movement, aiming to make higher education accessible to all by providing approachable resources for students to learn open-source mathematics software.

The creation of this textbook was a joint effort by a dedicated and inspirational team. The quality of our work reflects our collective contributions and enthusiasm.

We would like to acknowledge Hellen Colman, Professor of Mathematics at Wright College, our co-author, and our guiding star. Her mathematical expertise ensured the accuracy and relevance of our material. Inspired by her teaching and Discrete Math lectures at the City Colleges of Chicago-Wilbur Wright College, this project owes much to her mentorship and support. Her encouragement and trust have been invaluable, shaping our perspectives and approaches from our Discrete Math course to this OER development.

We extend our heartfelt gratitude to Ken Levasseur for his invaluable guidance and expertise in creating open-source textbooks. His contributions have significantly enhanced the quality and accessibility of our work.

A special thanks is due to Tineka Scalzo, Wright College librarian, for her valuable advice on publishing and copyright issues. Her insights have been instrumental in helping us navigate the complexities of academic publishing.

We also express our gratitude to the many talented developers and mathematicians within the open-source communities. The PreTeXt community played an essential role in our authoring process, while the SageMath community provided crucial subject matter expertise. We are very grateful to everyone who has worked to develop Sage and to the creators of PreTeXt.

We would also like to thank our peer reviewers and proofreaders, whose meticulous attention to detail ensured the clarity and quality of this textbook. Your contributions have been instrumental in bringing this project to fruition.

Finally, we express our deepest gratitude to all the contributors who made this project possible. Your dedication and collaborative spirit have made a lasting impact on this work and the field of open education.

Zunaid Ahmed and Samuel Lubliner

Authors and Contributors

Zunaid Ahmed
Computer Engineering
Truman College
zunaid.ahmed@hotmail.com

Hellen Colman
Math Department
Wright College
hcolman@ccc.edu

Samuel Lubliner
Computer Science
Wright College
sage.oer@gmail.com

Allaoua Boughrira (Contrib.)
Mathematics
Wright College
a.boughrira@gmail.com

Contents

Chapter 1

Getting Started

Welcome to our introduction to SageMath (also referred to as Sage). This chapter is designed for learners of all backgrounds—whether you're new to programming or aiming to expand your mathematical toolkit. There are various options for running Sage, including the SageMathCell, CoCalc, and a local installation. The easiest way to get started is to use the SageMathCell embedded directly in this book. We will also cover how to use CoCalc, a cloud-based platform that provides a collaborative environment for running Sage code.

Sage is a free open-source mathematics software system that integrates various open-source mathematics software packages[1]. We will cover the basics, including SageMath's syntax, data types, variables, and debugging techniques. Our goal is to equip you with the foundational knowledge needed to explore mathematical problems and programming concepts in an accessible and straightforward manner.

Join us as we explore the capabilities of SageMath!

1.1 Intro to Sage

You can execute and modify Sage code directly within the SageMathCells embedded on this webpage. Cells on the same page share a common memory space. To ensure accurate results, run the cells in the sequence in which they appear. Running them out of order may cause unexpected outcomes due to dependencies between the cells.

1.1.1 Sage as a Calculator

Before we get started with discrete math, let's see how we can use Sage as a calculator. Here are the basic arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Exponentiation: `**`, or `^`
- Division: `/`

[1] `doc.sagemath.org/html/en/reference/spkg/`

- Integer division: `//`
- Modulo: `%`

There are two ways to run the code within the cells:

- Click the Evaluate (Sage) button located under the cell.
- Use the keyboard shortcut Shift + Enter if your cursor is active in the cell.

```
# Lines that start with a pound sign are comments
# and ignored by Sage
1+1
```

```
100 - 1
```

```
3*4
```

```
# Sage uses two exponentiation operators
# ** is valid in Sage and Python
2**3
```

```
# Sage uses two exponentiation operators
# ^ is valid in Sage
2^3
```

```
# Returns a rational number
5/3
```

```
# Returns a floating point approximation
5/3.0
```

```
# Returns the quotient of the integer division
5//3
```

```
# Returns the remainder of the integer division
5 % 3
```

1.1.2 Variables and Names

We can assign the value of an expression to a variable. A variable is a name that refers to a value in the computer's memory. Use the assignment operator `=` to assign a value to a variable. The variable name is on the left side of the assignment operator, and the value is on the right side. Unlike the expressions above, the assignment statement does not display anything. To view the value of a variable, type the variable name and run the cell.

```
a = 1
b = 2
sum = a + b
sum
```

When choosing variable names, use valid identifiers.

- Identifiers cannot start with a digit.

- Identifiers are case-sensitive.
- Identifiers can include:
  - letters (`a - z`, `A - Z`)
  - digits (`0 - 9`)
  - underscore character `_`
- Do not use spaces, hyphens, punctuation, or special characters when naming identifiers.
- Do not use keywords as identifiers.

Below are some reserved keywords that you cannot use as variable names:

`False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

Use the Python `keyword` module to check if a name is a keyword.

```
import keyword
keyword.iskeyword('if')
```

The output is `True` because `if` is a keyword. Try checking other names.

1.2 Display Values

Sage offers various ways to display values on the screen. The simplest way is to type the value into a cell, and Sage will display it. Sage also has functions that display values in different formats.

- `print()` displays the value of the expression inside the parentheses on the screen.
- `pretty_print()` displays rich text.
- `show()` is an alias for `pretty_print()`.
- `latex()` produces the raw LaTeX code for the expression inside the parentheses. You can paste this code into a LaTeX document to display the expression.
- `%display latex` renders the output of commands as LaTeX automatically.
- While Python string formatting is available, the output is unreliable for rendering rich text and LaTeX due to compatibility issues.

Sage will display the value of the last line of code in a cell.

```
"Hello, World!"
```

`print()` outputs a similar result without the quotes.

```
print("Hello, World!")
```

View mathematical notation with rich text.

```
show(sqrt(2) / log(3))
```

If we want to display values from multiple lines of code, we can use multiple functions to display the values.

```
a = x^2
b = pi
show(a)
show(b)
```

Obtain raw LaTeX code for an expression.

```
latex(sqrt(2) / log(3))
```

If you are working in a Jupyter notebook or SageMathCell, `%display latex` sets the display mode.

```
%display latex
# Notice we don't need the show() function
sqrt(2) / log(3)
```

The expressions will continue to render as LaTeX until you change the display mode. The display mode is still set from the previous cell.

```
ZZ
```

Revert to the default output with `%display plain`.

```
%display plain
sqrt(2) / log(3)
```

```
ZZ
```

1.3 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that models the world as a collection of interacting **objects**. More specifically, an object is an **instance** of a **class**. A class can represent almost anything.

Classes are like blueprints that define the structure and behavior of objects. A class defines the **attributes** and **methods** of an object. An attribute is a variable that stores information about the object. A method is a function that can interact with or modify the object. Although you can create custom classes, the open-source community has already defined classes for us. For example, there are specialized classes for working with integers, lists, strings, graphs, and more.

In Python and Sage, almost everything is an object. When assigning a value to a variable, the variable references an object. In this case, the object is a list of strings.

```
vowels = ['a', 'e', 'i', 'o', 'u']
type(vowels)
```

```
type('a')
```

The `type()` function confirms that `'a'` is an instance of the `string` class and `vowels` is an instance of the `list` class. We create a `list` object named `vowels` by assigning a series of characters within square brackets to a variable. This object, `vowels`, now represents a `list` of `string` elements, and we can interact with it using various methods.

Dot notation is a syntax used to access an object's attributes and call an object's methods. For example, the list class has an `append` method, allowing us to add elements to the list.

```
vowels.append('y')
vowels
```

A **parameter** is a variable passed to a method. In this case, the parameter is the string `'y'`. The `append` method adds the string `'y'` to the end of the list. The list class has many more methods that we can use to interact with the list object. While `list` is a built-in Python class, Sage offers many more classes specialized for mathematical applications. For example, we will learn about the Sage `Set` class in the next chapter. Objects instantiated from the `Set` class have methods and attributes useful for working with sets.

```
v = Set(vowels)
type(v)
```

While OOP might seem abstract at first, it will become clearer as we dive deeper into Sage. We will see how Sage utilizes OOP principles and built-in classes to offer a structured way to represent data and perform powerful mathematical operations.

1.4 Data Types

In computer science, **Data types** represent data based on properties of the data. Python and Sage use data types to implement these classes. Since Sage builds upon Python, it inherits all the built-in Python data types. Sage also provides classes that are well-suited for mathematical computations.

Let's ask Sage what type of object this is.

```
n = 2
print(n)
type(n)
```

The `type()` function reveals that `2` is an instance of the **Integer** class. Sage includes numerous classes for different types of mathematical objects.

In the following example, Sage does not evaluate an approximation of `sqrt(2) * log(3)`. Sage will retain the **symbolic** value.

```
sym = sqrt(2) / log(3)
show(sym)
type(sym)
```

String: a `str` is a sequence of characters used for text. You can use single or double quotes.

```
greeting = "Hello, World!"
print(greeting)
print(type(greeting))
```

Boolean: The type `bool` can be one of two values, `True` or `False`.

```
# Check if 5 is contained in the set of prime numbers
b = 5 in Primes()
```

```
print(f"{b} is {type(b)}")
```

List: A mutable collection of items within a pair of square brackets `[]`. If an object is mutable, you can change its value after creating it.

```
l = [1, 3, 3, 2]
print(l)
print(type(l))
```

Lists are indexed starting at `0`. Here, we access the first element of a list by asking for the value at index zero.

```
l[0]
```

Lists have many helpful methods.

```
# Find the number of elements in the list
len(l)
```

Tuple: An immutable collection within a pair of parenthesis `()`. If an object is immutable, you cannot change the value after creating it.

```
t = (1, 3, 3, 2)
print(t)
type(t)
```

set: A collection of items within a pair of curly braces `{}`. `set()` with lowercase `s` is built into Python. The items in a set are unique and unordered. After printing the set, we see there are no duplicate values.

```
s = {1, 3, 3, 2}
print(s)
type(s)
```

Set is a built-in Sage class. `Set` with a capital `S` has added functionality for mathematical operations.

```
S = Set([1, 3, 3, 2])
type(S)
```

We start by defining a `list` within square brackets `[]`. Then, the `Set()` function creates the Sage set object.

```
S = Set([5, 5, 1, 3, 5, 3, 2, 2, 3])
print(S)
```

Dictionary: A collection of key-value pairs.

```
d = {
    "title": "Discrete Math with SageMath",
    "institution": "City Colleges of Chicago",
    "topics_covered": [
        "Set Theory",
        "Combinations and Permutations",
        "Logic",
        "Quantifiers",
        "Relations",
```

```
        "Functions",
        "Recursion",
        "Graphs",
        "Trees",
        "Lattices",
        "Boolean Algebras",
        "Finite State Machines"
    ],
    "format": ["Web", "PDF"]
}
type(d)
```

Use the `pprint` module to print the dictionary in a more readable format.

```
import pprint
pprint.pprint(d)
```

1.5 Iteration

Iteration is a programming technique that allows us to efficiently write code by repeating instructions with minimal syntax. The `for` loop assigns a value from a sequence to the loop variable and executes the loop body once for each value.

```
# Print the numbers from 0 to 19
# Notice the loop is zero-indexed and excludes 20
for i in range(20):
    print(i)
```

```
# Here, the starting value of the range is included
for i in range(10, 20):
    print(i)
```

```
# We can also specify a step value
for i in range(30, 90, 9):
    print(i)
```

Here is an example of list comprehension, a concise way to create lists. Unlike Python's `range()`, the Sage range syntax for list comprehension includes the ending value.

```
# Create a list of the cubes of the numbers from 9 to 20
# The for loop is written inside the square brackets
[n**3 for n in [9..20]]
```

We can also specify a condition in list comprehension. Let's create a list that only contains even numbers.

```
[n**3 for n in [9..20] if n % 2 == 0]
```

1.6 Debugging

Error messages are an inevitable part of programming. When you make a syntax error and see a message, read it carefully for clues about the cause of the

error. While some messages are helpful and descriptive, others may seem cryptic or confusing. With practice, you will develop valuable skills for debugging your code and resolving errors. Not all errors will produce an error message. Logical errors occur when the syntax is correct, but the program does not produce the expected output. Remember, mistakes are learning opportunities, and everyone makes them. Here are some tips for debugging your code:

- Read the error message carefully for information to help you identify and fix the problem.
- Study the documentation.
- Google the error message. Someone else has likely encountered the same issue.
- Search for previous posts on Sage forums.
- Take a break and return with a fresh perspective.
- If you are still stuck after trying these steps, ask the Sage community.

Let's dive in and make some mistakes together!

```
# Run this cell and see what happens
1message = "Hello, World!"
print(1message)
```

Why didn't this print `Hello, World!` on the screen? The error message informed us of a `SyntaxError`. While the phrase `invalid decimal literal` may seem confusing, the key issue here is the invalid variable name. Valid identifiers must start with a letter or underscore. They cannot begin with a number or use any special characters. Let's correct the variable name by using a valid identifier.

```
message = "Hello, World!"
print(message)
```

Here is another error:

```
print(Hi)
```

In this case, we encounter a `NameError` because `Hi` is not defined. Sage assumes that `Hi` is a variable because there are no quotes. We can make `Hi` a string by enclosing it in quotes.

```
print("Hi")
```

Alternatively, if we intended `Hi` to be a variable, we can assign a value to it before printing.

```
Hi = "Hello, World!"
print(Hi)
```

Reading the documentation is essential to understanding how to use methods correctly. If we incorrectly use a method, we will likely get a `NameError`, `AttributeError`, `TypeError`, or `ValueError`, depending on the mistake.

Here is an example of a `NameError`:

```
l = [6, 1, 5, 2, 4, 3]
sort(l)
```

The `sort()` method must be called on the list object using dot notation.

```
l = [4, 1, 2, 3]
l.sort()
print(l)
```

Here is an example of an `AttributeError`:

```
l = [1, 2, 3]
l.len()
```

Here is the correct way to use the `len()` method:

```
l = [1, 2, 3]
len(l)
```

Here is an example of a `TypeError`:

```
l = [1, 2, 3]
l.append(4, 5)
```

The `append()` method only takes one argument. To add multiple elements to a list, use the `extend()` method.

```
l = [1, 2, 3]
l.extend([4, 5])
print(l)
```

Here is an example of a `ValueError`:

```
factorial(-5)
```

Although the resulting error message is lengthy, the last line informs us that the argument must be a non-negative integer.

```
factorial(5)
```

Finally, we will consider a logical error. If your task is to print the numbers from 1 to 10, you may mistakenly write the following code:

```
for i in range(10):
    print(i)
```

The output will be the numbers from 0 to 9. To include 10, we need to adjust the range because the start is inclusive and the stop is exclusive.

```
for i in range(1, 11):
    print(i)
```

For more information, read the CoCalc article about the top mathematical syntax errors in Sage[1]

[1] `github.com/sagemathinc/cocalc/wiki/MathematicalSyntaxErrors`

1.7 Defining Functions

Sage comes with many built-in functions. Math terminology is not always standard, so be sure to read the documentation to learn what these functions do and how to use them. You can also define custom functions yourself. You are welcome to use the custom functions we define in this book. However, since these custom functions are not part of the Sage source code, you will need to copy and paste the functions into your Sage environment. If you try to use a custom function without defining it, you will get a `NameError`.

To define a custom function in Sage, use the `def` keyword followed by the function name and the function's arguments. The function's body is indented. When you call the function, the `return` keyword returns a value from the function. The function definition is only stored in memory after you run the cell. You will not see any output when you run the cell that defines the function. You will see output only when you call the function. A green box under the cell indicates the successful execution of the cell. If the box is not green, you must run the cell to define the function.

You may have heard of Pascal's Triangle, a triangular array of numbers in which each number is the sum of the two numbers directly above it. Here is an example function that returns the n^{th} (0-indexed) row of Pascal's Triangle:

```
def pascal_row(n):
    return [binomial(n, i) for i in range(n + 1)]
```

Try calling the function for yourself. First, run the Sage cell with the function definition to define the function. If you try to call a function without defining it, you will get a `NameError`. After defining the function, you can use it in other cells. You won't see any output when you run the cell that defines the function. The Sage cells store the function definition memory. You will see output only when you call the function. After running the above cell, you can call the `pascal_row()` function.

```
pascal_row(5)
```

Input validation makes functions more robust. We may get some validation out of the box. For example, if we try to call the function using a string or decimal value as input, we will get a `TypeError`:

```
pascal_row("5")
```

However, if we try to call the function with a negative integer, the function will return an empty list without raising an error.

```
pascal_row(-5)
```

This lack of error handling is risky because it can go undetected and cause unexpected behavior. Let's add a `ValueError` to handle negative input:

```
def pascal_row(n):
    if n < 0:
        raise ValueError("`n` must be a non-negative
            integer")
    return [binomial(n, i) for i in range(n + 1)]
```

Running the above cell redefines the function. Try calling the function with a negative integer to see the input validation.

```
pascal_row(-5)
```

Functions can also include a `docstring` to provide documentation for the function. The `docstring` is a string that appears as the first statement in the function body. It describes what the function does and how to use it.

```
def pascal_row(n):
    r"""
    Return row `n` of Pascal's triangle.

    INPUT:

    - ``n`` -- non-negative integer; the row number of
    Pascal's triangle to return.
      The row index starts from 0, which corresponds to the
    top row.

    OUTPUT: list; row `n` of Pascal's triangle as a list of
    integers.

    EXAMPLES:

    This example illustrates how to get various rows of
    Pascal's triangle (0-indexed) ::

        sage: pascal_row(0)  # the top row
        [1]

        sage: pascal_row(4)
        [1, 4, 6, 4, 1]

    It is an error to provide a negative value for `n` ::

        sage: pascal_row(-1)
        Traceback (most recent call last):
        ...
        ValueError: `n` must be a non-negative integer

    .. NOTE::

        This function uses the `binomial` function to
    compute each
        element of the row.
    """
    if n < 0:
        raise ValueError("`n` must be a non-negative
            integer")

    return [binomial(n, i) for i in range(n + 1)]
```

After redefining the function and running the above cell, view the `docstring` by calling the `help()` function on the function name. You can also access the `docstring` with the `?` operator.

```
help(pascal_row)
# pascal_row? also displays the docstring
# pascal_row?? reveals the function's source code
```

For more information on code style conventions and writing documentation strings, refer to the General Conventions article from the Sage Developer's Guide.

1.8 Documentation

Sage can do many more mathematical operations. If you want an idea of what Sage can do, check out the Quick Reference Card[1] and the Reference Manual[2].

The tutorial[3] is an overview to become familiar with Sage.

The Sage documentation[4] can be found at this link. Right now, reading the documentation is optional. We will do our best to get you up and running with Sage with this text.

You can quickly reference Sage documentation with the `?` operator. You may also view the source code with the `??` operator.

```
Set?
```

```
Set??
```

```
factor?
```

```
factor??
```

1.9 Run Sage in the browser

The easiest way to get started is by running SageMath online. However, if you do not have reliable internet, you can also install the software locally on your own computer. Begin your journey with SageMath by following these steps:

1. Navigate to the SageMath website[1]
2. Click on Sage on CoCalc[2]
3. Create a CoCalc account[3]
4. Go to Your Projects[4] on CoCalc and create a new project.
5. Start your new project and create a new worksheet. Choose the SageMath Worksheet option.
6. Enter SageMath code into the worksheet. Try to evaluate a simple expression and use the worksheet like a calculator. Execute the code by clicking `Run` or using the shortcut `Shift + Enter`. We will learn more ways to run code in the next section.
7. Save your worksheet as a PDF for your records.

[1] wiki.sagemath.org/quickref
[2] doc.sagemath.org/html/en/reference/
[3] doc.sagemath.org/html/en/tutorial/
[4] doc.sagemath.org/html/en/index.html
[1] https://www.sagemath.org/
[2] https://cocalc.com/features/sage
[3] https://cocalc.com/auth/sign-up
[4] https://cocalc.com/projects

8. To learn more about SageMath worksheets, refer to the documentation[5]

9. Alternatively, you can run Sage code in a Jupyter Notebook[6] for some additional features.

10. If you are feeling adventurous, you can install Sage[7] and run it locally on your own computer. Keep in mind that a local install will be the most involved way to run Sage code.

[5] https://doc.cocalc.com/sagews.html
[6] doc.cocalc.com/jupyter-start.html
[7] doc.sagemath.org/html/en/installation/index.html

Chapter 2

Set Theory

This chapter presents the study of set theory with Sage, starting with a description of the `Set()` function, its variations, and how to use it to calculate the basic set operations.

2.1 Creating Sets

2.1.1 Set Definitions

To construct a set, encase the elements within square brackets `[]`. Then, pass this `list` as an argument to the `Set()` function. It's important to note that the `S` in `Set()` should be uppercase to define a Sage set. In a set, each element is unique.

```
M = Set(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
    "Aug", "Sep", "Oct", "Nov", "Dec"])
show(M)
```

Notice that the months in set M do not appear in the same order as when you created the set. Sets are unordered collections of elements.

We can ask Sage to compare two sets to see whether or not they are equal. We can use the `==` operator to compare two values. A single equal sign `=` and double equal sign `==` have different meanings.

The **equality operator** `==` is used to ask Sage if two values are equal. Sage compares the values on each side of the operator and returns the Boolean value. The `==` operator returns `True` if the sets are equal and `False` if they are not equal.

The **assignment operator** `=` assigns the value on the right side to the variable on the left side.

```
M = Set(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
    "Aug", "Sep", "Oct", "Nov", "Dec"])
M_duplicates = Set(["Jan", "Jan", "Jan", "Feb", "Feb",
    "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
    "Oct", "Nov", "Dec"])

# The Set function eliminates duplicates
M == M_duplicates
```

If you have experience with Python, you may have used a Python `set`. Notice how the Python `set` begins with a lowercase `s`. Even though Sage supports Python sets, we will use Sage `Set` for the added features. Be sure to define `Set()` with an upper case `S`.

2.1.2 Set Builder Notation

Instead of explicitly listing the elements of a set, we can use a set builder notation to define a set. The set builder notation is a way to define a set by describing the properties of its elements. Here, we use the Sage `srange` instead of the Python `range` function for increased flexibility and functionality.

```
# Create a set of even numbers between 1 and 10
A = Set([x for x in srange(1, 11) if x % 2 == 0])
A
```

Iteration is a way to repeat a block of code multiple times and can be used to automate repetitive tasks. We could have created the same set by typing `A = Set([2, 4, 6, 8, 10])`. Imagine if we wanted to create a set of even numbers between 1 and 100. It would be much easier to use iteration.

```
B = Set([x for x in srange(1, 101) if x % 2 == 0])
B
```

2.1.3 Subsets

To list all the subsets included in a set, we can use the `Subsets()` function and then use a `for` loop to display each subset.

```
W = Set(["Sun", "Cloud", "Rain", "Snow", "Tornado",
    "Hurricane"])
subsets_of_weather = Subsets(W)

subsets_of_weather.list()
```

2.1.4 Set Membership Check

Sage allows you to check whether an element belongs to a set. You can use the `in` operator to check membership, which returns `True` if the element is in the set and `False` otherwise.

```
"earthquake" in W
```

We can check if $Severe = \{Tornado, Hurricane\}$ is a subset of W by using the `issubset` method.

```
Severe = Set(["Tornado", "Hurricane"])
Severe.issubset(W)
```

When we evaluate `W.issubset(Severe)`, Sage returns `False` because W is not a subset of $Severe$.

```
W.issubset(Severe)
```

2.2 Cardinality

To find the cardinality of a set, we use the `cardinality()` function.

```
A = Set([1, 2, 3, 4, 5])
A.cardinality()
```

Alternatively, we can use the Python `len()` function. Instead of returning a Sage `Integer`, the `len()` function returns a Python `int`.

```
A = Set([1, 2, 3, 4, 5])
len(A)
```

In many cases, using Sage classes and functions will provide more functionality. In the following example, `cardinality()` gives us a valid output while `len()` does not.

```
P = Primes()
P.cardinality()
```

```
# This results in an error because the
# Python len() function is not defined for the primes class
P = Primes()
P.len()
```

2.3 Operations on Sets

2.3.1 Union of Sets

There are two distinct methods available in Sage for calculating unions.

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `union()` function to calculate $A \cup B$.

Notes. The union operation is relevant in real-world scenarios, such as merging two distinct music playlists into one. In this case, any song that appears in both playlists will only be listed once in the merged playlist.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.union(B)
```

Alternatively, we can use the `|` operator to perform the union operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A | B
```

2.3.2 Intersection of Sets

Similar to union, there are two methods of using the intersection function in Sage.

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `intersection()` function to calculate $A \cap B$.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.intersection(B)
```

Alternatively, we can use the `&` operator to perform the intersection operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A & B
```

2.3.3 Difference of Sets

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `difference()` function to calculate the difference between sets.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.difference(B)
```

Alternatively, we can use the `-` operator to perform the difference operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A - B
```

2.3.4 Multiple Sets

When performing operations involving multiple sets, we can repeat the operations to get our results. Here is an example:

Suppose $A = \{1, 2, 3, 4, 5\}$, $B = \{3, 4, 5, 6\}$ and $C = \{5, 6, 7\}$. To find the union of all three sets, we repeat the `union()` function.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
C = Set([5, 6, 7])
A.union(B).union(C)
```

Alternatively, we can repeat the `|` operator to perform the union operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
C = Set([5, 6, 7])
A | B | C
```

The `intersection()` and `difference()` functions can perform similar chained operations on multiple sets.

2.3.5 Complement of Sets

Let $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the universal set. Given the set $A = \{1, 2, 3, 4, 5\}$. We can use the `difference()` function to find the complement of A.

```
U = Set([1, 2, 3, 4, 5, 6, 7, 8, 9])
A = Set([1, 2, 3, 4, 5])
U.difference(A)
```

Alternatively, we can use the `-` operator.

```
U = Set([1, 2, 3, 4, 5, 6, 7, 8, 9])
A = Set([1, 2, 3, 4, 5])
U - A
```

2.3.6 Cartesian Product of Sets

Suppose $A = \{1, 2, 3, 4, 5\}$ and $D = \{x, y\}$. We can use the `cartesian_product()` and `Set()` functions to display the Cartesian product $A \times D$.

```
A = Set([1, 2, 3, 4, 5])
D = Set(['x', 'y'])
Set(cartesian_product([A, D]))
```

Alternatively, we can use the `.` notation to find the Cartesian product.

```
A = Set([1, 2, 3, 4, 5])
D = Set(['x', 'y'])
Set(A.cartesian_product(D))
```

2.3.7 Power Sets

The power set of the set V is the set of all subsets, including the empty set $\{\emptyset\}$ and the set V itself. Sage offers several ways to create a power set, including the `Subsets()` and `powerset()` functions. First, we will explore the `Subsets()` function. The `Subsets()` function is more user-friendly due to the built-in `Set` methods. Next, we will examine some limitations of the `Subsets()` function. We introduce the `powerset()` function as an alternative for working with advanced sets not supported by `Subsets()`.

The `Subsets()` function returns all subsets of a finite set in no particular order. Here, we find the power set of the set of vowels and view the subsets as a `list` where each element is a `Set`.

```
V = Set(["a", "e", "i", "o", "u"])
S = Subsets(V)
list(S)
```

We can confirm that the power set includes the empty set.

```
Set([]) in S
```

We can also confirm that the power set includes the original set.

```
V in S
```

The `cardinality()` method returns the total number of subsets.

```
S.cardinality()
```

There are limitations to the `Subsets()` function. For example, the `Subsets()` function does not support non-hashable objects.

About hashable objects:

- A hashable object has a hash value that never changes during its lifetime.
- A hashable object can be compared to other objects.
- Most of Python's immutable built-in objects are hashable.
- Mutable containers (lists or dictionaries) are not hashable.
- Immutable containers (tuples) are only hashable if their elements are hashable.

You will see an `unhashable type` error message when trying to create `Subsets` of a list containing a list. The `powerset()` function returns an iterator over the `list` of all subsets in no particular order. The `powerset()` function is ideal when working with non-hashable objects.

```
N = [1, [2, 3], 4]
list(powerset(N))
```

The `powerset()` function supports infinite sets. Let's generate the first 7 subsets from the power set of integers.

```
P = powerset(ZZ)

i = 0
for subset in P:
    print(subset)
    i += 1
    if i == 7:
        break
```

While the `Subsets()` function can represent infinite sets symbolically, it is not practical.

```
P = Subsets(ZZ)
P
```

Observe the `TypeError` message when trying to retrieve a random element from `Subsets(ZZ)`

```
P.random_element()
```

Pay close attention to the capitalization of function names. There is a difference between the functions `Subsets()` and `subsets()`. Notice the lowercase `s` in `subsets()`, which is an alias for `powerset()`.

2.3.8 Viewing Power Sets

Power sets can contain many elements. The powerset of the set R contains elements 128 elements.

```
R = Set(["red", "orange", "yellow", "green", "blue",
    "indigo", "violet"])
S = Subsets(R)
S.cardinality()
```

If we only want to view part of the power set, we can specify a range of elements with a technique called slicing. For example, here are the first 5 elements of the power set.

```
S.list()[:5]
```

Now, let's retrieve the following 5 elements of the power set.

```
# Slicing to get elements from index 5 to 9
S.list()[5:10]
```

Chapter 3

Combinatorics

Counting techniques arise naturally in computer algebra as well as in basic applications in daily life. This chapter covers the treatment of the enumeration problem in Sage, including counting combinations, counting permutations, and listing them.

3.1 Combinatorics

3.1.1 Factorial Function

The factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n.

Compute the factorial of 5:

```
factorial(5)
```

3.1.2 Combinations

The combination (n, k) is an unordered selection of k objects from a set of n objects.

Notes. Use combinations when order does not matter, such as determining possible Poker hands. The order in which a player holds cards does not affect the kind of hand. For example, the following hand is a royal flush: 10, J, Q, K, A. The following hand is also a royal flush: A, K, J, 10, Q.

Calculate the number of ways to choose 3 elements from a set of 5:

```
Combinations(5, 3).cardinality()
```

List the combinations:

```
Combinations(5, 3).list()
```

The `binomial()` function provides an alternative method to compute the number of combinations.

```
binomial(5, 3)
```

3.1.3 Permutations

A permutation (n, k) is an ordered selection of k objects from a set of n objects.

Notes. Use permutations in situations where order does matter, such as when creating passwords. Longer passwords have more permutations, making them more challenging to guess by brute force.

To calculate the number of ways to choose 3 elements from a set of 5 when the order matters, use the `Permutations()` method.

```
Permutations(5, 3).cardinality()
```

List the permutations:

```
Permutations(5, 3).list()
```

When $n = k$, we can calculate permutations of n elements.
Calculate the number of permutations of a set with 3 elements:

```
Permutations(3).cardinality()
```

List the permutations:

```
Permutations(3).list()
```

The following is an example of permutations of specified elements:

```
A = Permutations(['a', 'b', 'c'])

A.list()
```

Choose 2:

```
A = Permutations(['a', 'b', 'c'], 2)

A.list()
```

Chapter 4

Logic

In this chapter, we introduce different ways to create Boolean formulas using the logical functions `not`, `and`, `or`, `if then`, and `iff`. Then, we show how to ask Sage to create a truth table from a formula and determine if an expression is a contradiction or a tautology.

4.1 Logical Operators

In Sage, the logical operators are AND `&`, OR `|`, NOT `~`, conditional `->`, and biconditional `<->`.

Name	Sage Operator	Mathematical Notation
AND	`&`	$\wedge$
OR	`|`	$\vee$
NOT	`~`	$\neg$
Conditional	`->`	$\rightarrow$
Biconditional	`<->`	$\leftrightarrow$

4.1.1 Boolean Formula

Sage's `propcalc.formula()` function allows for the creation of Boolean formulas using variables and logical operators. We can then use `show` function to display the mathematical notations.

```
A = propcalc.formula('(p & q) | (~p)')
show(A)
```

4.2 Truth Tables

The `truthtable()` function in Sage generates the truth table for a given logical expression.

Notes. Truth tables aid in the design of digital circuits.

```
A = propcalc.formula('p -> q')
A.truthtable()
```

An alternative way to display the table with better separation and visuals would be to use `SymbolicLogic()`, `statement()`, `truthtable()` and the `print_table()` functions.

```
A = SymbolicLogic()
B = A.statement('p -> q')
C = A.truthtable(B)
A.print_table(C)
```

`SymbolicLogic()` creates an instance for handling symbolic logic operations, while `statement()` defines the given statement. The `truthtable()` method generates a truth table for this statement, and `print_table()` displays it.

Expanding on the concept of truth tables, we can analyze logical expressions involving three variables. This provides a deeper understanding of the interplay between multiple conditions. The `truthtable()` function supports expressions with a number of variables that is practical for computational purposes, if the list of variables becomes too lengthy (such as extending beyond the width of a LaTeX page), the truth table's columns may run off the screen. Additionally, the function's performance may degrade with a very large number of variables, potentially increasing the computation time.

```
B = propcalc.formula('(p & q) -> r')
B.truthtable()
```

4.3 Analyzing Logical Equivalences

4.3.1 Equivalent Statements

When working with Sage symbolic logic, the `==` operator compares semantic equivalence.

```
h = propcalc.formula("x | ~y")
s = propcalc.formula("x & y | x & ~y | ~x & ~y")
h == s
```

Do not attempt to compare equivalence of truth tables.

```
# Warning:
# Even though these truth tables look identical,
# the comparison will return False.
h.truthtable() == s.truthtable()
```

However, we can compare equivalence of truth table `lists`.

```
h_list = h.truthtable().get_table_list()
s_list = s.truthtable().get_table_list()
h_list == s_list
```

4.3.2 Tautologies

A tautology is a logical statement that is always true. The `is_tautology()` function checks whether a given logical expression is a tautology.

Notes. Tautologies are relevant in the field of cybersecurity. Attackers exploit vulnerabilities by injecting SQL code that turns a `WHERE` clause into a tautology, granting unintended access to the system.

```
a = propcalc.formula('p | ~p')
a.is_tautology()
```

4.3.3 Contradictions

In contrast to tautologies, contradictions are statements that are always false.

```
A = propcalc.formula('p & ~p')
A.is_contradiction()
```

Chapter 5

Relations

In this chapter, we will explore the relationships between elements in sets, building upon the concept of the Cartesian product introduced earlier. We will begin by learning how to visualize relations using Sage. Then, we will introduce some new functions that can help us determine whether these relations are equivalence or partial order relations.

5.1 Introduction to Relations

A **relation** R from set A to set B is any subset of the Cartesian product $A \times B$, indicating that $R \subseteq A \times B$. We can ask Sage to decide if R is a relation from A to B. First, construct the Cartesian product $C = A \times B$. Then, build the set S of all subsets of C. Finally, ask if R is a subset of S.

Recall the Cartesian product consists of all possible ordered pairs (a, b), where $a \in A$ and $b \in B$. Each pair combines an element from set A with an element from set B.

In this example, an element in the set A relates to an element in B if the element from A is twice the element from B.

```
A = Set([1, 2, 3, 4, 5, 6])
B = Set([1, 2, 7])

C = Set(cartesian_product([A, B]))
S = Subsets(C)

R = Set([(a, b) for a in A for b in B if a==2*b])

print("R =", R)
print("Is R a relation from set A to set B?", R in S)
```

Let's use relations to explore matching items of clothes. Let's define two sets, jackets and shirts, as examples:

$$\text{jackets} = \{j_1, j_2, j_3\}$$

$$\text{shirts} = \{s_1, s_2, s_3, s_4\}$$

The Cartesian product of jackets and shirts includes all possible combinations of jackets with shirts.

```
# Define the sets of jackets and shirts
jackets = Set(['j1', 'j2', 'j3'])
shirts = Set(['s1', 's2', 's3', 's4'])

# View all the possible combinations of jackets and shirts
C = cartesian_product([jackets, shirts])

Set(C)
```

Since the Cartesian product returns all the possible combinations, some jackets and shirts will clash. Let's create a relation from jackets to shirts based on matching the items of clothing.

```
# Define a matching relation between jackets and shirts
R = Set([('j1', 's1'), ('j2', 's3'), ('j3', 's4'), ('j1',
    's2')])

print("Matching relation R =", R)
```

5.2 Relations on a set

When $A = B$ we refer to the relation as a relation **on** A.

Consider the set $A = \{2, 3, 4, 6, 8\}$. Let's define a relation R on A such that aRb iff $a|b$ (a divides b). The relation R can be represented by the set of ordered pairs where the first element divides the second:

```
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a divides b
R = Set([(a, b) for a in A for b in A if a.divides(b)])

show(R)
```

5.3 Digraphs

A digraph, or directed graph, is a visual representation of a relation R on the set A. Every element in set A is shown as a node (vertex). An arrow from the node a to the node b represents the pair (a, b) on the relation R.

```
# Define the set A
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a divides b
R = [(a, b) for a in A for b in A if a.divides(b)]

DiGraph(R, loops=true)
```

We can add a title to the digraph with the `name` parameter.

Notes. Digraphs come in handy when relationships have a clear direction, like who follows who on social media or how academic papers cite one another.

```
DiGraph(R, loops=true, name="Look at my digraph")
```

If the digraph does not contain a relation from a node to itself, we can omit the `loops=true` parameter. If we happen to forgot to include the parameter when we need to, Sage will give us a descriptive error message.

```
# Define the set A
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a < b
R = [(a, b) for a in A for b in A if a < b]
DiGraph(R)
```

We can also define the digraph using pair notion for relations.

```
DiGraph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])
```

Alternatively, we can define the digraph directly. The element on the left of the `:` is a node. The node relates to the elements in the list on the right of the `:`.

```
# 1 relates to 2, 3, and 4
# 2 relates to 3 and 4
# 3 relates to 4
DiGraph({1: [2, 3, 4], 2: [3, 4], 3: [4]})
```

5.4 Properties

A relation on A may satisfy certain properties:

- **Reflexive**: $aRa \, \forall a \in A$
- **Symmetric**: If aRb then $bRa \, \forall a, b \in A$
- **Antisymmetric**: If aRb and bRa then $a = b \, \forall a, b \in A$
- **Transitive**: If aRb and bRc then $aRc \, \forall a, b, c \in A$

So far, we have learned about some of the built-in Sage methods that come out of the box, ready for us to use. Sometimes, we may need to define custom functions to meet specific requirements or check for particular properties. We define custom functions with the `def` keyword. If you want to reuse the custom functions defined in this book, copy and paste the function definitions into your own Sage worksheet and then call the function to use it.

5.4.1 Reflexive

A relation R is reflexive if a relates to a for all elements a in the set A. This means all the elements relate to themselves.

```
A = Set([1, 2, 3])
R = Set([(1, 1), (2, 2), (3, 3), (1, 2), (2, 3)])
show(R)
```

Let's define a function to check if the relation R on set A is reflexive. We will create a set of (a, a) pairs for each element a in A and check if this set is a subset of R. This will return `True` if the relation is reflexive and `False` otherwise.

```
def is_reflexive_set(A, R):
    reflexive_pairs = Set([(a, a) for a in A])
    return reflexive_pairs.issubset(R)

is_reflexive_set(A, R)
```

If we are working with `DiGraphs`, we can use the method `has_edge` to check if the graph has a loop for each vertex.

```
def is_reflexive_digraph(A, G):
    return all(G.has_edge(a, a) for a in A)

A = [1, 2, 3]
R = [(1, 1), (2, 2), (3, 3), (1, 2), (2, 3)]

G = DiGraph(R, loops=True)

is_reflexive_digraph(A, G)
```

5.4.2 Symmetric

A relation is symmetric if a relates to b, then b relates to a.

```
def is_symmetric_set(relation_R):
    inverse_R = Set([(b, a) for (a, b) in relation_R])
    return relation_R == inverse_R

A = Set([1, 2, 3])

R = Set([(1, 2), (2, 1), (3, 3)])

is_symmetric_set(R)
```

We can check if a `DiGraph` is symmetric by comparing the edges of the graph with the `reverse` edges. In our definition of symmetry, we are only interested in the relation of nodes, so we set edge `labels=False`.

```
def is_symmetric_digraph(digraph):
    return digraph.edges(labels=False) ==
        digraph.reverse().edges(labels=False)

relation_R = [(1, 2), (2, 1), (3, 3)]

G = DiGraph(relation_R, loops=true)

is_symmetric_digraph(G)
```

5.4.3 Antisymmetric

When a relation is antisymmetric, the only case that a relates to b and b relates to a is when a and b are equal.

```
def is_antisymmetric_set(relation):
    for a, b in relation:
        if (b, a) in relation and a != b:
            return False
    return True
```

```
relation = Set([(1, 2), (2, 3), (3, 4), (4, 1)])

is_antisymmetric_set(relation)
```

While Sage offers a built-in `antisymmetric()` method for `Graphs`, it checks for a more restricted property than the standard definition of antisymmetry. Specifically, it checks if the existence of a path from a vertex x to a vertex y implies that there is no path from y to x unless $x = y$. Observe that while the standard antisymmetric property forbids the edges to be bidirectional, the Sage antisymmetric property forbids cycles.

```
# Example with the more restricted
# Sage built-in antisymmetric method
# Warning: returns False

relation = [(1, 2), (2, 3), (3, 4), (4, 1)]

DiGraph(relation).antisymmetric()
```

Let's define a function to check for the standard definition of antisymmetry in a `DiGraph`.

```
def is_antisymmetric_digraph(digraph):
    for edge in digraph.edges(labels=False):
        a, b = edge
        # Check if there is an edge in both directions (a to
            b and b to a) and a is not equal to b
        if digraph.has_edge(b, a) and a != b:
            return False
    return True

relation = DiGraph([(1, 2), (2, 3), (3, 4), (4, 1)])

is_antisymmetric_digraph(relation)
```

5.4.4 Transitive

A relation is transitive if a relates to b and b relates to c, then a relates to c.

Let's define a function to check for the transitive property in a `Set`:

```
def is_transitive_set(A, R):
    for a in A:
        for b in A:
            if (a, b) in R:
                for c in A:
                    if (b, c) in R and not (a, c) in R:
                        return False
    return True

A = Set([1, 2, 3])

R = Set([(1, 2), (2, 3), (1, 3)])

is_transitive_set(A, R)
```

You may be tempted to write a function with a nested loop because the logic

is easy to follow. However, when working with larger sets, the time complexity of the function will not be efficient. This is because we are iterating through the set A three times. We can improve the time complexity by using a dictionary to store the relation R. Alternatively, we can use built-in Sage `DiGraph` methods.

```
D = DiGraph([(1, 2), (2, 3), (1, 3)], loops=True)

D.is_transitive()
```

5.5 Equivalence

A relation on a set is called an **equivalence relation** if it is reflexive, symmetric, and transitive. The **equivalence class** of an element a in a set A is the set of all elements in A that are related to a by this relation, denoted by:

$$[a] = \{x \in A \mid xRa\}$$

Here, $[a]$ represents the equivalence class of a, comprising all elements in A that are related to a through the relation R. This illustrates the grouping of elements into equivalence classes.

Consider a set A defined as:

$$A = \{x \mid x \text{ is a person living in a given building}\}$$

```
# Define the set of people
A = Set(['p_1', 'p_2', 'p_3', 'p_4', 'p_5', 'p_6', 'p_7',
    'p_8', 'p_9', 'p_10'])
A
```

Create sets for the people living on each floor of the building:

```
import pprint

# Define the floors as a dictionary, mapping floor names to
    sets of people
floors = {
    'first_floor': Set(['p_1', 'p_2', 'p_3', 'p_4']),
    'second_floor': Set(['p_5', 'p_6', 'p_7']),
    'third_floor': Set(['p_8', 'p_9', 'p_10'])
}

pprint.pprint(floors)
```

Let R be the relation on A described as follows:

xRy iff x and y live in the same floor of the building.

```
# Define the relation R based on living on the same floor
R = Set([(x, y) for x in A for y in A if any(x in
    floors[floor] and y in floors[floor] for floor in
    floors)])
R
```

This relation demonstrates the properties of an equivalence relation:
Reflexive: A person lives in the same floor as themselves.

```
def is_reflexive_set(A, R):
    reflexive_pairs = Set([(a, a) for a in A])
    return reflexive_pairs.issubset(R)

is_reflexive_set(A, R)
```

Symmetric: If person a lives in the same floor as person b, then person b lives in the same floor as person a.

```
def is_symmetric_set(relation_R):
    inverse_R = Set([(b, a) for (a, b) in relation_R])
    return relation_R == inverse_R

is_symmetric_set(R)
```

Transitive: If person a lives in the same floor as person b and person b lives in the same floor as person c, then person a lives in the same floor as person c.

```
G = DiGraph(list(R), loops=true)
G.is_transitive()
```

5.6 Partial Order

A relation R on a set is a Partial Order (PO) $\prec$ if it satisfies the reflexive, antisymmetric, and transitive properties. A poset is a set with a partial order relation. For example, the following set of numbers with a relation given by divisibility is a poset.

```
A = Set([1, 2, 3, 4, 5, 6, 8])

R = [(a, b) for a in A for b in A if a.divides(b)]

D = DiGraph(R, loops=True)

plot(D)
```

A Hasse diagram is a simplified visual representation of a poset. Unlike a digraph, the relative position of vertices has meaning: if x relates to y, then the vertex x appears lower in the drawing than the vertex y. Self-loops are assumed and not shown. Similarly, the diagram assumes the transitive property and does not explicitly display the edges that are implied by the transitive property.

Notes. Partial orders and Hasse diagrams help analyze task dependencies in scheduling applications.

If R is a partial order relation on A, then the function `Poset((A, R))` computes the Hasse diagram associated to R.

```
A = Set([1, 2, 3, 4, 5, 6, 8])

R = [(a, b) for a in A for b in A if a.divides(b)]

P = Poset((A, R))
```

```
plot(P)
```

Moreover, the `cover_relations()` function shows the pairs depicted in the Hasse diagram after the previous simplifications.

```
P.cover_relations()
```

Chapter 6

Functions

This chapter will briefly discuss the implementation of functions in Sage and will delve deeper into the sequences defined by recursion, including Fibonacci's. We will show how to solve a recurrence relation using Sage.

6.1 Functions

A function from a set A into a set B is a relation from A into B such that each element of A is related to exactly one element of the set B. The set A is called the domain of the function, and the set B is called the co-domain. Functions are fundamental in both mathematics and computer science for describing mathematical relationships and implementing computational logic.

In Sage, functions can be defined using direct definition.

For example, defining a function $f : \mathbb{R} \to \mathbb{R}$ to calculate the cube of a number, such as 3:

```
f(x) = x^3
show(f)
f(3)
```

6.1.1 Graphical Representations

Sage provides powerful tools for visualizing functions, enabling you to explore the graphical representations of mathematical relationships.

For example, to plot the function $f(x) = x^3$ over the interval $[-2, 2]$:

```
f(x) = x^3
plot(f(x), x, -2, 2)
```

6.2 Recursion

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem. This approach is extensively used in mathematics and computer science, especially in the computation of binomial coefficients, the evaluation of polynomials, and the generation of sequences.

6.2.1 Recursion in Sequences

A recursive sequence is defined by one or more base cases and a recursive step that relates each term to its predecessors.

Notes. Use recursion to solve problems by breaking them down into similar steps. In programming, recursively defined functions often improve code readability.

Given a sequence defined by a recursive formula, we can ask Sage to find its closed form. Here, `s` is a function representing the sequence defined by recursion. The equation `eqn` defines the recursive relation $s_n = s_{n-1} + 2 \cdot s_{n-2}$. The `rsolve()` function is then used to find a closed-form solution to this recurrence, given the initial conditions $s_0 = 2$ and $s_1 = 7$. At last, we use the `SR()` function to convert from Python notation to mathematical notation.

```
from sympy import Function, rsolve
from sympy.abc import n
s = Function('s')
eqn = s(n) - s(n-1) - 2*s(n-2)
sol = rsolve(eqn, s(n), {s(0): 2, s(1): 7})
show(SR(sol))
```

We can use the `show()` function to make the output visually more pleasing; you can try removing it and see how the output looks.

Similarly, the Fibonacci sequence is another example of a recursive sequence, defined by the base cases $F_0 = 0$ and $F_1 = 1$, and the recursive relation $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. This sequence is a cornerstone example in the study of recursion.

```
from sympy import Function, rsolve
from sympy.abc import n
F = Function('F')
fib_eqn = F(n) - F(n-1) - F(n-2)
fib_sol = rsolve(fib_eqn, F(n), {F(0): 0, F(1): 1})
show(SR(fib_sol))
```

The `show()` function is again used here to present the solution in a more accessible mathematical notation, illustrating the power of recursive functions to describe complex sequences with simple rules.

We can also write a function `fib()` to compute the nth Fibonacci number by iterating and updating the values of two consecutive Fibonacci numbers in the sequence. Let's calculate the third Fibonacci number.

```
def fib(n):
    if n == 0 or n == 1: return n
    else:
        U = 0; V = 1  # the initial terms F0 and F1
        for k in range(2, n+1):
            W = U + V; U = V; V = W
        return V
fib(3)
```

We go back to the previous method where we calculated the closed form `fib_sol` and evaluate it now at $n = 3$.

```
from sympy import Function, rsolve, Symbol, simplify
n = Symbol('n')
F = Function('F')
fib_eqn = F(n) - F(n-1) - F(n-2)
fib_sol = rsolve(fib_eqn, F(n), {F(0): 0, F(1): 1})
# Evaluate the solution at n=3
fib3 = simplify(fib_sol.subs(n, 3))
show(SR(fib3))
```

As we can see, we obtain the same number either by evaluating the closed form at $n = 3$ or by finding the third Fibonacci number directly by iteration.

6.2.2 Recursion with Binomial Coefficients

Binomial coefficients, denoted as $\binom{n}{k}$, count the number of ways to choose k elements from an n-element set. They can be defined recursively. Sage can compute binomial coefficients using the `binomial(n, k)` function.

```
binomial(5, 3)
```

We can also explore the recursive nature of binomial coefficients by defining a function ourselves recursively.

```
def binomial_recursive(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial_recursive(n-1, k-1) +
            binomial_recursive(n-1, k)

binomial_recursive(5, 3)
```

This function implements the recursive formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, with base cases $\binom{n}{0} = \binom{n}{n} = 1$.

Chapter 7

Graph Theory

Sage is extremely powerful for graph theory. This chapter presents the study of graph theory with Sage, starting with a description of the Graph class through the implementation of optimization algorithms. We also illustrate Sage's graphical capabilities for visualizing graphs.

7.1 Basics

7.1.1 Graph Definition

A **graph** $G = (V, E)$ consists of a set V of vertices and a set E of edges, where

$$E \subset \{\{u, v\} \mid u, v \in V\}$$

The set of edges is a set whose elements are subsets of two vertices.

Terminology:

- Vertices are synonymous with **nodes**.
- Edges are synonymous with **links** or **arcs**.
- In an **undirected graph** edges are **unordered** pairs of vertices.
- In a **directed graph** edges are **ordered** pairs of vertices.

There are several ways to define a graph in Sage. We can define a graph by listing the vertices and edges:

```
V = ['A', 'B', 'C', 'D', 'E']
E = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E',
    'A'), ('A', 'D'), ('C', 'E')]
G = Graph([V, E])
G.plot()
```

We can define a graph with an edge list. Each edge is a pair of vertices:

```
L = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E',
    'A'), ('A', 'C'), ('B', 'E')]
G = Graph(L)
G.plot()
```

We can define a graph with an edge dictionary like so: `{edge: [neighbor, neighbor, etc], edge: [ neighbor, etc], etc: [etc]}` Each dictionary key is a vertex. The dictionary values are the vertex neighbors.

```
E = {1: [2, 3, 4], 2: [1, 3, 4], 3: [1, 2, 4], 4: [1, 2, 3]}
G = Graph(E)
G.plot()
```

You can improve the readability of a dictionary by placing each item of the collection on a new line:

```
E = {
    1: [2, 3, 4],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [1, 2, 3]
}
G = Graph(E)
G.plot()
```

Sage offers a collection of predefined graphs. Here are some examples:

```
graphs.PetersenGraph().show()
graphs.CompleteGraph(5).show()
graphs.TetrahedralGraph().show()
graphs.DodecahedralGraph().show()
graphs.HexahedralGraph().show()
```

Notes. Concepts from graph theory have practical applications related to social networks, computer networks, transportation, biology, chemistry, and more.

7.1.2 Weighted Graphs

A **weighted graph** has a weight, or number, associated with each edge. These weights can model anything including distances, costs, or other relevant quantities.

To create a weighted graph, add a third element to each pair of vertices.

```
E = [('A', 'B', 2), ('B', 'C', 3), ('C', 'D', 4), ('D', 'E',
    5), ('E', 'A', 1)]
G = Graph(E, weighted=True)
G.plot(edge_labels=True)
```

7.1.3 Graph Characteristics

Sage offers many built-in functions for analyzing graphs. Let's examine the following graph:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.show()
```

The `vertices()` method returns a list of the graph's vertices.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
```

```
G.vertices()
```

The `G.edges()` method returns triples representing the graph's vertices and edge labels.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.edges()
```

Return the edges as a tuple without the label by setting `labels=false`.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.edges(labels=false)
```

The **order** of $G = (V, E)$ is the number of vertices $|V|$.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.order()
```

The size of $G = (V, E)$ is the number of edges $|E|$.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.size()
```

The degree of the vertex v, $deg(v)$ is the number of edges incident with v.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.degree('A')
```

The degree sequence of $G = (V, E)$ is the list of degrees of its vertices.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.degree_sequence()
```

7.1.4 Graphs and Matrices

The *adjacency matrix* of a graph is a square matrix used to represent which vertices of the graph are adjacent to which other vertices. Each entry a_{ij} in the matrix is equal to 1 if there is an edge from vertex `i` to vertex `j`, and is equal to 0 otherwise.

```
G = Graph([('A', 'B'), ('A', 'E'), ('B', 'C'), ('C', 'D'),
    ('D', 'E')])
G.adjacency_matrix()
```

We can also define a graph with an adjacency matrix:

```
A = Matrix([
          [0, 1, 0, 0, 1],
          [1, 0, 1, 0, 0],
          [0, 1, 0, 1, 0],
          [0, 0, 1, 0, 1],
```

```
          [1, 0, 0, 1, 0]
])
G = Graph(A)
G.plot()
```

The *incidence matrix* is an alternative matrix representation of a graph, which describes the relationship between vertices and edges. In this matrix, rows correspond to vertices, and columns correspond to edges, with entries indicating whether a vertex is incident to an edge.

```
G = Graph([('A', 'B'), ('A', 'E'), ('B', 'C'), ('C', 'D'),
    ('D', 'E')])
G.incidence_matrix()
```

7.1.5 Manipulating Graphs in Sage

Add a vertex to a graph:

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
G.add_vertex(5)
G.show()
```

Add a list of vertices:

```
G.add_vertices([10, 11, 12])
G.show()
```

Remove a vertex from a graph:

```
G.delete_vertex(12)
G.show()
```

Remove a list of vertices from a graph:

```
G.delete_vertices([5,10,11])
G.show()
```

Add an edge between two vertices:

```
G.add_edge(1, 3)
G.show()
```

Delete an edge from a graph:

```
G.delete_edge(2, 3)
G.show()
```

Deleting a nonexistent vertex returns an error. Deleting a nonexistent edge leaves the graph unchanged. Adding a vertex or edge already in the graph, leaves the graph unchanged.

7.2 Plot Options

The `show()` method displays the graphics object immediately with default settings. The `plot()` method accepts options for customizing the presentation of the graphics object. You can import more features from Matplotlib or LaTeX

for fine-tuned customization options. Let's examine how the plotting options improve the presentation and help us discover insights into the structure and properties of a graph. The presentation of a Sage graphics object may differ depending on your environment.

7.2.1 Size

Here is a graph that models the primary colors of the RGB color wheel:

```
E = [
    ('r', 'g'),
    ('g', 'b'),
    ('b', 'r')
]

Graph(E).show()
```

Let's increase the `vertex_size`:

```
Graph(E).plot(vertex_size=1000).show()
```

Resolve the cropping by increasing the `figsize`. Specify a single number or a `(width, height)` tuple.

```
Graph(E).plot(vertex_size=1000, figsize=10).show()
```

Increasing the `figsize` works well in a notebook environment. However, in a SageCell, a large `figsize` introduces scrolling. Setting `graph_border=True` is an alternate way to resolve the cropping while maintaining the size of the graph.

```
Graph(E).plot(vertex_size=1000, graph_border=True).show()
```

7.2.2 Edge Labels

Let's add some edge labels. Within the `list` of edge `tuples`, the first two values are vertices, and the third value is the edge label.

```
E = [
    ('r', 'g', 'yellow'),
    ('g', 'b', 'cyan'),
    ('b', 'r', 'magenta')
]

G = Graph(E).plot(
    edge_labels=True,
)

G.show()
```

7.2.3 Color

There are various ways to specify `vertex_colors`, including hexadecimal, RGB, and color name. Hexadecimal and RGB offer greater flexibility because Sage does not have a name for every color. The color is the `dictionary` key, and the vertex is the `dictionary` value.

The following example specifies the color with RGB values. The values can range anywhere from `0` to `1`. Color the vertex `r` red by setting the first element in the RGB tuple to full intensity with a value of `1`. Next, ensure vertex `r` contains no green or blue light by setting the remaining tuple elements to `0`. Notice vertex `g` is darker because the green RGB value is `.65` instead of `1`.

```
set_vertex_colors = {
    (1,0,0): ['r'], # Color vertex `r` all red
    (0,.65,0): ['g'], # Color vertex `g` dark green
    (0,0,1): ['b'] # Color vertex `b` all blue
}

G = Graph(E).plot(
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

The following example specifies the color by name instead of RGB value. Sage will return an error if you use an undefined color name.

```
set_vertex_colors = {
    'red': ['r'],
    'green': ['g'],
    'blue': ['b']
}

G = Graph(E).plot(
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

Let's specify the `edge_colors` with RGB values. The edge from vertex `r` to vertex `g` is yellow because the RGB tuple sets red and green light to full intensity with no blue light. For darker shades, use values less than `1`.

```
set_edge_colors = {
    (1,1,0): [('r', 'g')],
    (0,1,1): [('g', 'b')],
    (1,0,1): [('b', 'r')]
}

G = Graph(E).plot(
    edge_colors=set_edge_colors,
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

This alternate method specifies the color by name instead:

```
set_edge_colors = {
    'yellow': [('r', 'g')],
```

```
    'cyan': [('g', 'b')],
    'magenta': [('b', 'r')]
}

G = Graph(E).plot(
    edge_colors=set_edge_colors,
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

Consider accessibility when choosing colors on a graph. For example, the red and green on the above graph look indistinguishable to people with color blindness. Blue and red are usually a safe bet for contrasting two colors.

Here is a Sage Interact to help identify hexadecimal color values.

- First, click Evaluate (Sage) to define and load the interact. You are welcome to modify the interact definition to suit your needs.
- You may define a new edge list, vertex size, and graph border within an input box.
- After entering new values, press Enter on your keyboard to load the new graph.
- Click on the color selector square to change the color. The hexadecimal value appears to the right of the color square.
- After selecting a new color, the graph will update when you click outside the color selector.

```
@interact
def _(
    edges=input_box(default=[(1, 2), (2, 3), (3, 4), (4,
        1)], label="Graph", width=40),
    vertex_size=input_box(default=2000, label="Vertex Size",
        width=40),
    graph_border=input_box(default=True, label="Border",
        width=40),
    color=color_selector(widget='colorpicker', label="Click
        ->")
):
    g = Graph(edges)
    color_str = color.html_color()

    show(
        g.plot(
            vertex_size=vertex_size,
            graph_border=graph_border,
            vertex_colors=color_str
        )
    )
```

7.2.4 Layout

Let's define and examine the following graph. Evaluate this cell multiple times and notice the vertex positions are not consistent.

```
N = [
    ('g', 'b',),
    ('g', 'd',),
    ('g', 'f',),
    ('b', 'd',),
    ('b', 'f',),
    ('d', 'f',)
]

G = Graph(N)

G.show()
```

Layout options include: “acyclic”, “circular”, “ranked”, “graphviz”, “planar”, “spring”, or “tree”.

A planar graph can be drawn without any crossing edges. The default graph layout does not ensure the planar layout of a planar graph. Sage will return an error if you try to plot a non-planar graph with the planar layout.

```
G.plot(layout='planar').show()
```

Sage’s planar algorithm sets the vertex positions. Alternatively, we can specify the positions in a dictionary. Let’s position the G node in the center.

```
positions = {
    'g': (0, 0),
    'd': (-1, 1),
    'b': (1, 1),
    'f': (0, -1)
}

G.plot(pos=positions).show()
```

The following graph modeling the intervals in the C major scale is challenging to read. Let’s think about how we can improve the presentation.

```
I = [
    ("c", "d", "M2"), ("c", "e", "M3"), ("c", "f", "P4"),
        ("c", "g", "P5"), ("c", "a", "M6"), ("c", "b", "M7"),
    ("d", "e", "M2"), ("d", "f", "m3"), ("d", "g", "P4"),
        ("d", "a", "P5"), ("d", "b", "M6"), ("d", "c", "m7"),
    ("e", "f", "m2"), ("e", "g", "m3"), ("e", "a", "P4"),
        ("e", "b", "P5"), ("e", "c", "m6"), ("e", "d", "m7"),
    ("f", "g", "M2"), ("f", "a", "M3"), ("f", "b", "a4"),
        ("f", "c", "P5"), ("f", "d", "M6"), ("f", "e", "M7"),
    ("g", "a", "M2"), ("g", "b", "M3"), ("g", "c", "P4"),
        ("g", "d", "P5"), ("g", "e", "M6"), ("g", "f", "m7"),
    ("a", "b", "M2"), ("a", "c", "m3"), ("a", "d", "P4"),
        ("a", "e", "P5"), ("a", "f", "m6"), ("a", "g", "m7"),
    ("b", "c", "m2"), ("b", "d", "m3"), ("b", "e", "P4"),
        ("b", "f", "d5"), ("b", "g", "m6"), ("b", "a", "m7"),
]

C = DiGraph(I, multiedges=True,)

C.plot(edge_labels=True).show()
```

In this case, the graph is not planar. The circular layout organizes the

vertices for improved readability.

```
C.plot(edge_labels=True, layout='circular')
```

7.2.5 View in a New Tab

Increasing the `figsize` improves the definition of the arrows. For an even better view of the Graph, right-click the image and view it in a new tab.

```
C.plot(
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

7.2.6 Edge Style

The options for `edge_style` include "`solid`", "`dashed`", "`dotted`", or "`dashdot`".

```
C.plot(
    edge_style='dashed',
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

Improve the definition between the edges by using a different color for each edge. The `color_by_label` method automatically maps the colors to edges.

```
C.plot(
    edge_style='dashed',
    color_by_label=True,
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

7.2.7 3-Dimensional

View a 3D representation of graph with `show3d()`. Click and drag the image to change the perspective. Zoom in on the image by pinching your computer's touchpad.

```
G = graphs.CubeGraph(3)
G.show3d()
```

```
G = graphs.TetrahedralGraph()
G.show3d()
```

```
G = graphs.IcosahedralGraph()
G.show3d()
```

```
G = graphs.DodecahedralGraph()
G.show3d()
```

```
G = graphs.CompleteGraph(5)
G.show3d()
```

7.3 Paths

A path between two vertices u and v is a sequence of consecutive edges starting at u and ending at v.

To get all paths between two vertices:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.all_paths(1, 4)
```

The length of a path is defined as the number of edges that make up the path.

Finding the shortest path between two vertices can be achieved using the `shortest_path()` function:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.shortest_path(1, 4)
```

A graph is said to be connected if there is a path between any two vertices in the graph.

To determine if a graph is connected, we can use the `is_connected()` function:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.is_connected()
```

A connected component of a graph G is a maximal connected subgraph of G. If the graph G is connected, then it has only one connected component.

For example, the following graph is not connected:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.is_connected()
```

To identify all connected components of a graph, the `connected_components()` function can be utilized:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.connected_components()
```

We can visualize the graph as a disjoint union of its connected components, by plotting it.

```
G = Graph({1: [2, 3], 2: [4], 5: [6, 7], 6: [7]})
G.show()
```

The diameter of a graph is the length of the longest shortest path between any two vertices.

```
G = Graph({1: [2, 3], 2: [3, 4], 3: [4]})
G.diameter()
```

```
# Calculates the diameter of the graph.
```

A graph is bipartite if its set of vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set:

```
G = Graph({1: [2, 3], 2: [4], 3: [4]})
G.is_bipartite()
```

7.4 Isomorphism

Informally, we can say that an **isomorphism** is a relation of sameness between graphs. Let's say that the graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

This means there is a bijection between the set of vertices such that every time two vertices determine an edge in the first graph, the image of these vertices by the bijection also determines an edge in the second graph, and vice versa. Essentially, the two graphs have the same structure, but the vertices are labeled differently.

Notes. Graph isomorphism identifies structures relevant to chemistry, biology, machine learning, and neural networks.

```
C = Graph(
  {
    'a': ['b', 'c', 'g'],
    'b': ['a', 'd', 'h'],
    'c': ['a', 'd', 'e'],
    'd': ['b', 'c', 'f'],
    'e': ['c', 'f', 'g'],
    'f': ['d', 'e', 'h'],
    'g': ['a', 'e', 'h'],
    'h': ['b', 'f', 'g']
  }
)

D = Graph(
  {
    1: [2, 6, 8],
    2: [1, 3, 5],
    3: [2, 4, 8],
    4: [3, 5, 7],
    5: [2, 4, 6],
    6: [1, 5, 7],
    7: [4, 6, 8],
    8: [1, 3, 7]
  }
)

C.show()
D.show()
```

The sage `is_isomorphic()` method can be used to check if two graphs are isomorphic. The method returns `True` if the graphs are isomorphic and `False` if the graphs are not isomorphic.

```
C.is_isomorphic(D)
```

The **invariants under isomorphism** are conditions that can be checked to determine if two graphs are not isomorphic. If one of these fails then the graphs are not isomorphic. If all of these are true then the graph may or may not be isomorphic. The three conditions for invariants under isomorphism are:

$$G = (V, E) \text{ is connected iff } G' = (V', E') \text{ is connected}$$

$$|V| = |V'| \text{ and } |E| = |E'|$$

$$\text{degree sequence of } G = \text{ degree sequence of } G'$$

To summarize, if one graph is connected and the other is not, then the graphs are not isomorphic. If the number of vertices and edges are different, then the graphs are not isomorphic. If the degree sequences are different, then the graphs are not isomorphic. If all three invariants are satisfied, then the graphs may or may not be isomorphic.

Let's define a function to check if two graphs satisfy the invariants under isomorphism. Make sure you run the next cell to define the function before using the function.

```
def invariant_under_isomorphism(G1, G2):
    print("Are both graphs connected? ", end="")
    are_connected: bool = (
        G1.is_connected() == G2.is_connected()
    )
    print("Yes" if are_connected else "No")

    print(
        "Do both graphs have same number of "
        "vertices and edges? ", end=""
    )
    have_equal_vertex_and_edge_counts: bool = (
        G1.order() == G2.order() and
        G1.size() == G2.size()
    )
    print(
        "Yes" if have_equal_vertex_and_edge_counts else "No"
    )

    # Sort the degree-sequences because
    # the order of vertices doesn't matter.
    print(
        "Do both graphs have the same degree sequence? ",
        end=""
    )
    have_same_degree_sequence: bool = (
        sorted(G1.degree_sequence()) ==
        sorted(G2.degree_sequence())
    )
    print("Yes" if have_same_degree_sequence else "No")

    # All checks
    are_invariant_under_isomorphism =(
```

```
        are_connected and
        have_equal_vertex_and_edge_counts and
        have_same_degree_sequence
    )
    print(
        "\nTherefore, the graphs {0} isomorphic.".format(
            "may be" if are_invariant_under_isomorphism
            else "are not"
        )
    )
```

If we use `invariant_under_isomorphism` on the C and D, the output will let's know that the graphs may or may not be isomorphic. We can use the `is_isomorphic()` method to check if the graphs are definitively isomorphic.

```
invariant_under_isomorphism(C, D)
```

Let's construct a different pair of graphs A and B defined as follow

```
A = Graph(
  [
    ('a', 'b'),
    ('b', 'c'),
    ('c', 'f'),
    ('f', 'd'),
    ('d', 'e'),
    ('e', 'a')
  ]
)

B = Graph(
  [
    (1, 5),
    (1, 9),
    (5, 9),
    (4, 6),
    (4, 7),
    (6, 7)
  ]
)

A.show()
B.show()
```

This time, if we apply `invariant_under_isomorphism` function on A and B, the output will show us that they are not isomorphic.

```
invariant_under_isomorphism(A, B)
```

7.5 Euler and Hamilton

7.5.1 Euler

An **Euler path** is a path that uses every edge of a graph exactly once. An Euler path that is a circuit is called an **Euler circuit**.

The idea of an Euler path emerged from the study of the **Königsberg bridges** problem. Leonhard Euler wanted to know if it was possible to walk through the city of Königsberg, crossing each of its seven bridges exactly once. This problem can be modeled as a graph, with the land masses as vertices and the bridges as edges.

```
konigsberg = [('A', 'B', 'b_1'),
              ('A', 'B', 'b_2'),
              ('A', 'C', 'b_3'),
              ('A', 'C', 'b_4'),
              ('D', 'A', 'b_5'),
              ('D', 'B', 'b_6'),
              ('D', 'C', 'b_7')]
G = Graph(konigsberg, multiedges=True)
G.show(edge_labels=True)
```

Notes. Eulerian circuits and paths have practical applications for reducing travel and costs in logistics, waste management, the airline industry, and postal service.

While exploring this problem, Euler discovered the following:

- A connected graph has an **Euler circuit** iff every vertex has an even degree.
- A connected graph has an **Euler path** iff there are at most two vertices with an odd degree.

We say that a graph is **Eulerian** if contains an Euler circuit.

We can use Sage to determine if a graph is Eulerian.

```
G.is_eulerian()
```

Since this returns `False`, we know that the graph is not Eulerian. Therefore, it is not possible to walk through the city of Königsberg, crossing each of its seven bridges exactly once.

We can use `path=True` to determine if a graph contains an Euler path. Sage will return the beginning and the end of the path.

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1), (2, 4), (1, 3),
    (1, 4)], multiedges=True)
G.show()
G.is_eulerian(path=True)
```

If the graph is Eulerian, we can ask Sage to find an Euler circuit with the `eulerian_circuit` function. Let's take a look at the following graph.

```
G = Graph([(1, 2), (2, 3), (2, 3),(3, 4), (4, 1), (2, 4),
    (1, 3), (1, 4)], multiedges=True)
G.show()
G.eulerian_circuit()
```

If we are not interested in the edge labels, we can set `labels=False`. We can also set `return_vertices=True` to get a list of vertices for the path

```
G = graphs.CycleGraph(6)
G.eulerian_circuit(labels=False, return_vertices=True)
```

7.5.2 Hamilton

A **Hamilton path** is a path that uses every vertex of a graph exactly once. A Hamilton path that is a circuit is called a **Hamilton circuit**. If a graph contains a Hamilton circuit, we say that the graph is **Hamiltonian**.

Hamilton created the "Around the World" puzzle. The object of the puzzle was to start at a city and travel along the edges of the dodecahedron, visiting all of the other cities exactly once, and returning back to the starting city.

We can represent the dodecahedron as a graph and use Sage to determine if it is Hamiltonian. See for yourself if the dodecahedron is Hamiltonian.

```
graphs.DodecahedralGraph().show()
```

We can ask Sage to determine if the dodecahedron is Hamiltonian.

```
graphs.DodecahedralGraph().is_hamiltonian()
```

By running `Graph.is_hamiltonian??` we see that Sage uses the `traveling_salesman_problem()` function to determine if a graph is Hamiltonian.

The traveling salesperson problem is a classic optimization problem. Given a list of cities and the lengths between each pair of cities, what is the shortest possible route that visits each city and returns to the original city? This is one of the most difficult problems in computer science. It is **NP-hard**, meaning that no efficient algorithm is known to solve it. The complexity of the problem increases with the number of nodes. When working with many nodes, the algorithm can take a long time to run.

Let's explore the following graph:

```
G = Graph({1:{3:2, 2:1, 4:3, 5:1}, 2:{3:6, 4:3, 5:1},
    3:{4:5, 5:3}, 4:{5:5}})
G.show(edge_labels=True)
```

We can ask Sage if the graph contains a Hamiltonian cycle.

```
G.hamiltonian_cycle(algorithm='backtrack')
```

The function `hamiltonian_cycle` returns `True` and lists an example of a Hamiltonian cycle as the list of vertices `[1, 2, 3, 4, 5]`. This is just one of the many Hamiltonian cycles that exist in the graph. Now lets find the minimum Hamiltonian cycle.

```
h = G.traveling_salesman_problem(use_edge_labels=True,
    maximize=False)
h.show(edge_labels=True)
```

Now we have the plot of the minimum Hamiltonian cycle. The minimum Hamiltonian cycle is the shortest possible route that visits each city and returns to the original city. The minimum Hamiltonian cycle is the solution to the traveling salesperson problem. We can ask Sage for the sum of the weights of the edges in the minimum Hamiltonian cycle.

```
sumWeights = sum(h.edge_labels())
print(sumWeights)
```

If there is no Hamiltonian cycle, Sage will return `False`. If we use the `backtrack` algorithm, Sage will return a list that represents the longest path found.

```
G = Graph([(1, 2), (1, 3), (2, 3), (1,4), (4, 7), (3, 5),
    (5, 8), (8, 9), (2,6), (6, 9), (7, 9)])
G.show()
G.hamiltonian_cycle(algorithm='backtrack')
```

7.6 Graphs in Action

Imagine you are a bike courier tasked with making deliveries to each City Colleges of Chicago (CCC) campus location. Per your contract, you get paid per delivery, not per hour. Therefore, finding the most efficient delivery route is in your best interest. We assume the bike delivery routes are the same distance in each direction.

7.6.1 Bike Courier Delivery Route Problem

Let's make a plan to solve our delivery route problem.

1. Find the distances in miles between each CCC location.
2. Make a graph of the CCC locations. Each location is a node. Each edge is a bike route. The weight of the edges represents the distance of the bike route between locations.
3. Use the traveling salesperson algorithm to calculate the optimal delivery route.

7.6.2 Locations

Table 7.6.1 CCC Addresses

Name	**Address**
Harold Washington College	30 E. Lake Street, Chicago, IL 60601
Harry Truman College	1145 West Wilson Ave, Chicago, IL 60640
Kennedy-King College	6301 South Halsted St, Chicago, IL 60621
Malcolm X College	1900 W. Jackson, Chicago, IL 60612
Olive-Harvey College	10001 South Woodlawn Ave, Chicago, IL 60628
Richard J. Daley College	7500 South Pulaski Rd, Chicago, IL 60652
Wilbur Wright College	4300 N. Narragansett Ave, Chicago, IL 60634

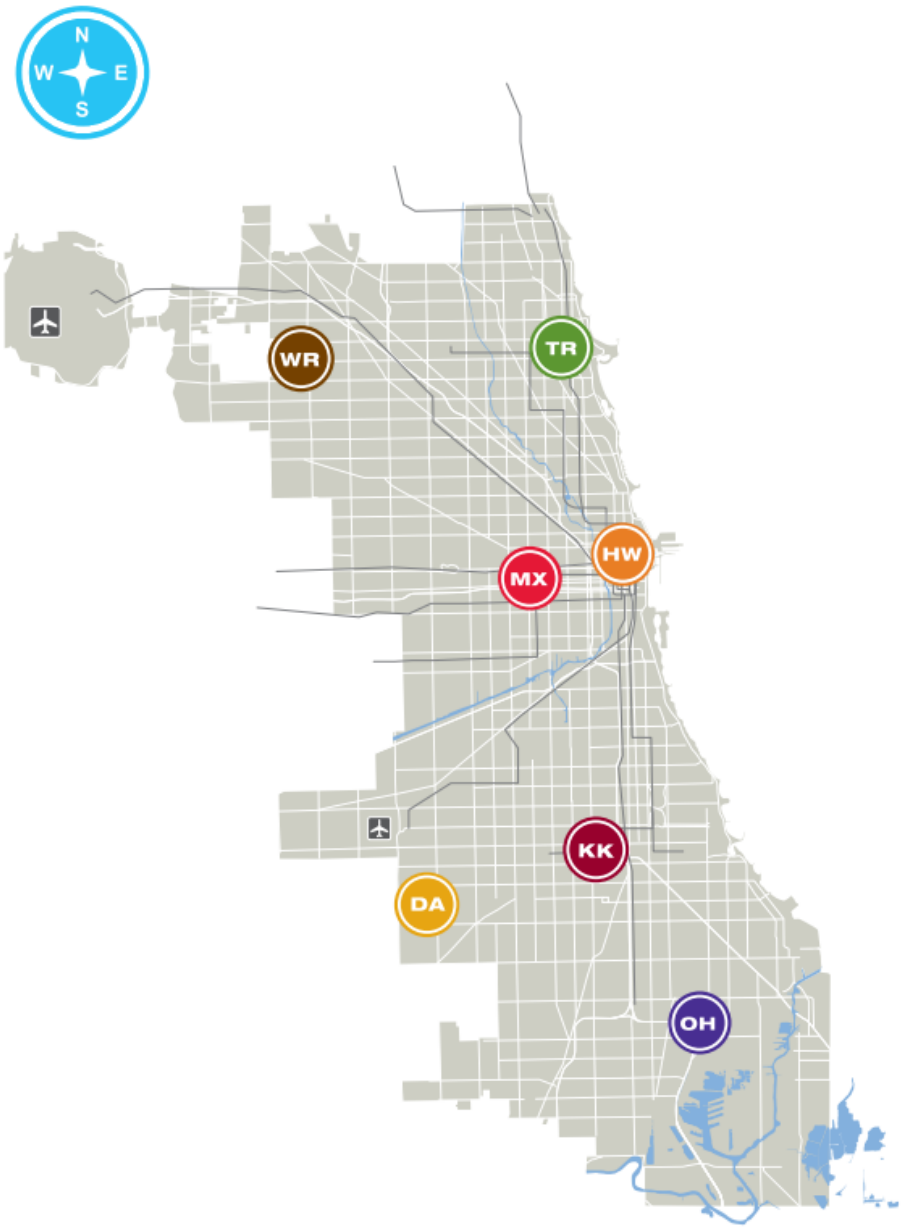

Figure 7.6.2 City Colleges of Chicago

7.6.3 Graph

We will represent each College as a node with the initials of the College name. The weight of the edge will represent the miles in between the locations. Since we are using bike routes, we are assuming each direction between two locations has the same distance. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", 6.5)`.

```
routes = [
    ("HW", "HT", 6.5),
    ("HW", "KK", 8.3),
    ("HW", "MX", 3.2),
```

```
    ("HW", "OH", 15.4),
    ("HW", "RD", 11.9),
    ("HW", "WW", 10.7),

    ("HT", "KK", 13.6),
    ("HT", "MX", 7.1),
    ("HT", "OH", 22.1),
    ("HT", "RD", 17.3),
    ("HT", "WW", 7.9),

    ("KK", "MX", 8.3),
    ("KK", "OH", 8.1),
    ("KK", "RD", 5.7),
    ("KK", "WW", 16.9),

    ("MX", "OH", 16.2),
    ("MX", "RD", 10.2),
    ("MX", "WW", 10.2),

    ("OH", "RD", 10.0),
    ("OH", "WW", 24.9),

    ("RD", "WW", 18.3)
]
routes
```

Create a `Graph` from the edge `list`:

```
G = Graph(routes)
G.show(edge_labels=True)
```

The trailing zeros of the floating point values are hard to read. Let's loop through the edge `list` and display the numbers with 3 points of precision.

```
for u, v, label in G.edge_iterator():
    G.set_edge_label(u, v, n(label, digits=3))

G.show(edge_labels=True)
```

Since this graph is not planar, improve the layout with the `"circular"` parameter. We can also improve the readability by increasing the `vertex_size` and `figsize`.

```
G.show(
    edge_labels=True,
    layout="circular",
    vertex_size=500,
    figsize=10,
)
```

Now that we have a clearer idea of the routes, let's find the most efficient delivery route using the traveling salesperson algorithm.

```
optimal_route =
    G.traveling_salesman_problem(use_edge_labels=True,
    maximize=False)
optimal_route.show(
    edge_labels=True,
```

```
    vertex_size=500,
)
```

We can set the vertex positions to resemble their positions on the map. We can use the latitude and longitude values of the locations and then reverse them when we supply the values to the position `dictionary`.

```
positions = {
    'HW': (-87.62682604591349, 41.88609733324964),
    'HT': (-87.65901943241516, 41.9646769664519),
    'KK': (-87.6435785385309, 41.77847328856264),
    'MX': (-87.67453475017268, 41.87800548491064),
    'OH': (-87.5886722734757, 41.71006715754713),
    'RD': (-87.72315805813204, 41.75677704810169),
    'WW': (-87.78738482318016, 41.95836512405638),
}

Graph(optimal_route).show(
    pos=positions,
    edge_labels=True,
    vertex_size=500,
    figsize=10,
)
```

Chapter 8

Trees

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given graph is a tree and then introduce further searching algorithms for trees.

8.1 Definitions and Theorems

Given a graph, a **cycle** is a circuit with no repeated edges. A **tree** is a connected graph with no cycles. A graph with no cycles and not necessarily connected is called a **forest**.

Let $G = (M, E)$ be a graph. The following are all equivalent:

- G is a tree.
- For each pair of distinct vertices, there exists a unique path between them.
- G is connected, and if $e \in E$ then the graph $(V, E - e)$ is disconnected.
- G contains no cycles, but by adding one edge, you create a cycle.
- G is connected and $|E| = |v| - 1$.

Let's explore the following graph:

```
data = {
  1: [4],
  2: [3, 4, 5],
  3: [2],
  4: [1, 2, 6, 7],
  5: [2, 8],
  6: [4, 9, 11],
  7: [4],
  8: [5, 10],
  9: [6],
  10: [8],
  11: [6]
  }

G = Graph(data)
G.show()
```

Notes. Trees are a common data structure used in file explorers, parsers, and decision making.

Let's ask Sage if this graph is a tree.

```
G.is_tree()
```

If we remove an edge, we can see that the graph is no longer a tree.

```
G_removed_edge = G.copy()
G_removed_edge.delete_edge((1, 4))
G_removed_edge.show()
G_removed_edge.is_tree()
```

However, we can see that the graph is still a forest.

```
G_removed_edge.is_forest()
```

If we add an edge, we can see that the graph contains a cycle and is no longer a tree.

```
G_added_edge = G.copy()
G_added_edge.add_edge((1, 2))
G_added_edge.show()
G_added_edge.is_tree()
```

8.2 Search Algorithms

The graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq \{\{u, v\} \in E \mid u, v \in V'\}$.

The subgraph $G' = (V', E')$ is a **spanning subgraph** of $G = (V, E)$ if $V' = V$.

A **spanning tree** for the graph G is a spanning subgraph of G that is a tree.

Given a graph, various algorithms can calculate a spanning tree, including depth-first search and breadth-first search.

Breadth-first search algorithm

1. Choose a vertex of the graph (root) arbitrarily.
2. Travel all the edges incident with the root vertex.
3. Give an order to this set of new vertices added.
4. Consider each of these vertices as a root, in order, and add all the unvisited incident edges that do not produce a cycle.
5. Repeat the method with the new set of vertices.
6. Follow the same procedure until all the vertices are visited.

The output of this algorithm is a spanning tree.

The `breadth_first_search()` function provides a flexible method for traversing both directed and undirected graphs. Let's consider the following graph:

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
G.show()
print(list(G.breadth_first_search(start="a",
```

```
    report_distance=True)))
```

In the example above, the `start` parameter begins the traversal at vertex `a`. The `report_distance=True`, parameter reports pairs in the format `(vertex, distance)`. Distance is the length of the path from the start vertex. From the output above, we see:

- The distance from vertex `a` to vertex `a` is `0`.
- The distance from vertex `a` to vertex `d` is `1`.
- The distance from vertex `a` to vertex `e` is `1`.
- The distance from vertex `a` to vertex `c` is `1`.
- The distance from vertex `a` to vertex `b` is `2`.

We can also set the parameter `edges=True` to return the edges of the BFS tree. Sage will raise an error if you use the `edges` and `report_distance` parameters simultaneously.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
s = list(G.breadth_first_search("a", edges=True))
print(s)
Graph(s)
```

The above graph is a spanning tree, but not necessarily a minimum spanning tree. Let's check how many spanning trees exist.

```
G.spanning_trees_count()
```

Iterate over all the spanning trees of a graph with `spanning_trees()`.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
spanning_trees = list(G.spanning_trees(labels=True))
for i, tree in enumerate(spanning_trees):
    print(f"Spanning Tree {i + 1}: {tree.edges()}")
    show(tree.plot())
```

Given a weighted graph of all possible spanning trees we can calculate, we may be interested in the minimal one. A **minimal spanning tree** is a spanning tree whose sum of weights is minimal. Prim's Algorithm calculates a minimal spanning tree.

Prim's Algorithm: Keep two disjoint sets of vertices. One (L) contains vertices that are in the growing spanning tree, and the other (R) that are not in the growing spanning tree.

1. Choose a vertex $u \in V$ arbitrarily. At this step, $L = \{u\}$ and $R = V - \{u\}$.
2. In R, select the cheapest vertex connected to the growing spanning tree L and add it to L
3. Follow the same procedure until all the vertices are in L

The output of this algorithm is a minimal spanning tree.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
G.show(edge_labels = True)
```

We can ask Sage for the minimal spanning tree of this graph. By running `Graph.min_spanning_tree??` We can see that `min_spanning_tree()` uses a variation of Prim's Algorithm by default. We can also use other algorithms such as Kruskal, Boruvka, or NetworkX.

Notes. Minimal spanning trees influence the efficient design of networks and roads.

```
G.min_spanning_tree(by_weight=True)
```

From the output of `min_spanning_tree(by_weight=True)`, we see an edge `list` of the minimal spanning tree. Each element of the edge lis is a `tuple` where the first two values are vertices, and the third value is the edge weight or label.

Let's visualize the minimal spanning tree.

```
h = Graph(G.min_spanning_tree(by_weight=True))
h.show(edge_labels = True)
```

Let's define a function to view the minimal spanning tree in the context of the original graph. The function parameters include:

- `graph`: A SageMath Graph object.
- `mst_color`: Color for edges part of the MST (default: `'darkred'`).
- `non_mst_color`: Color for edges not part of the MST (default: `'lightblue'`).
- `figsize`: Dimensions for the graph image.

```
def visualize_mst(input_graph, mst_color='darkred',
    non_mst_color='lightblue', figsize=None):
    try:
        if not input_graph.is_connected():
            print("The graph must be connected")
            return

        mst_edges =
            input_graph.min_spanning_tree(by_weight=True)
        print("MST Edges:", mst_edges)
        Graph(mst_edges).show(edge_labels=True,
            figsize=figsize, edge_color=mst_color)

        edge_colors = {mst_color: [], non_mst_color: []}

        mst_edge_set = set((v1, v2) for v1, v2, _ in
            mst_edges)

        for edge in input_graph.edges():
            v1, v2, _ = edge
            if (v1, v2) in mst_edge_set or (v2, v1) in
                mst_edge_set:
                edge_colors[mst_color].append((v1, v2))
            else:
                edge_colors[non_mst_color].append((v1, v2))
```

```
        print("MST overlaid on the original graph:")
        p = input_graph.plot(edge_labels=True,
            edge_colors=edge_colors, figsize=figsize)
        show(p)

    except Exception as e:
        print("Error:", e)
```

Let's generate a random graph and view the minimal spanning tree.

```
import random

vertices = 5
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G)
```

The following graph contains 9 vertices.

```
import random

vertices = 9
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G, figsize=10)
```

The following graph contains 15 vertices.

```
import random

vertices = 15
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G, figsize=10)
```

8.3 Trees in Action

Imagine your task is to create a railway between all the City Colleges of Chicago (CCC) campus locations. The contract requests that you use minimal track material to save construction costs. For simplicity's sake, assume each railway is a straight line between campuses.

8.3.1 Railway Problem

Let's make a plan to solve our railway construction optimization problem.

1. Find the latitude and longitude of each CCC campus location.
2. Use the Haversine formula to calculate the distances between the locations. The Haversine formula requires latitude and longitude for inputs and computes the shortest path between two points on a sphere.

3. Make a graph of the CCC campuses. Each location is a node. Each railway path is an edge. Each railway path is the shortest path between locations. The weight of the edges represents the distance between locations.

4. Find the minimum spanning tree (MST) of the CCC graph.

8.3.2 Location Distances

Table 8.3.1 City Colleges of Chicago Locations

Name	**(Latitude, Longitude)**
Harold Washington College	(41.88609733324964, -87.62682604591349)
Harry Truman College	(41.9646769664519, -87.65901943241516)
Kennedy-King College	(41.77847328856264, -87.6435785385309)
Malcolm X College	(41.87800548491064, -87.67453475017268)
Olive-Harvey College	(41.71006715754713, -87.5886722734757)
Richard J. Daley College	(41.75677704810169, -87.72315805813204)
Wilbur Wright College	(41.95836512405638, -87.78738482318016)

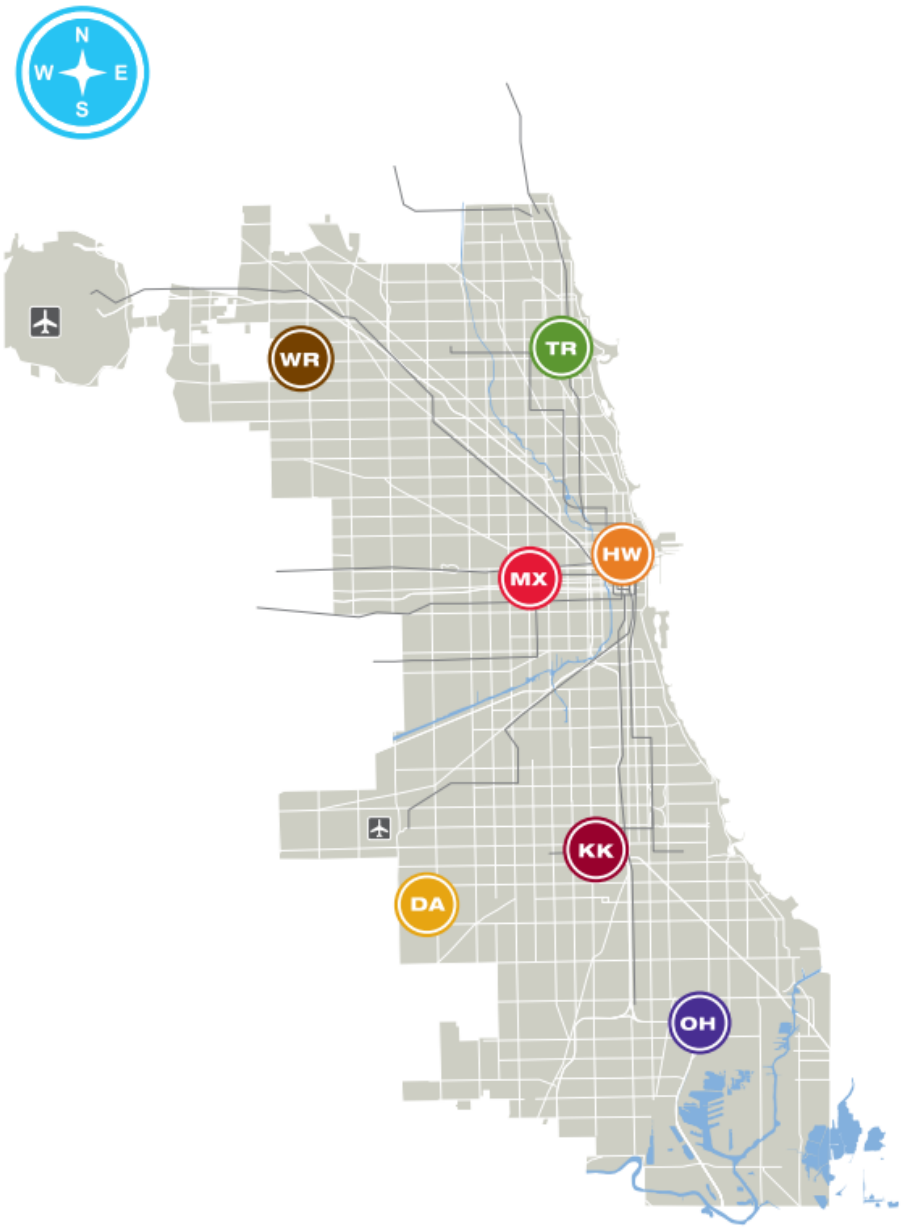

Figure 8.3.2 City Colleges of Chicago

Now, let's calculate the distances between campus locations. We will first create a `dictionary` to store the campus name, latitude, and longitude values.

```
lat_long = {
    "HW": (41.88609733324964, -87.62682604591349),
    "HT": (41.9646769664519, -87.65901943241516),
    "KK": (41.77847328856264, -87.6435785385309),
    "MX": (41.87800548491064, -87.67453475017268),
    "OH": (41.71006715754713, -87.5886722734757),
    "RD": (41.75677704810169, -87.72315805813204),
    "WW": (41.95836512405638, -87.78738482318016)
}
lat_long
```

Since the Earth is curved, we cannot use the Euclidean distance. We will use the Haversine formula instead. Note that the Haversine formula still produces an approximation because the Earth is not a perfect sphere. Here is a function to compute the Haversine formula.

```
def haversine(lat1, lon1, lat2, lon2):
    '''Reference:
        https://cs.nyu.edu/~visual/home/proj/tiger/gisfaq.html'''
    import math

    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1,
        lat2, lon2])

    dlat = lat2 - lat1
    dlon = lon2 - lon1

    a = math.sin(dlat / 2)**2 + \
        math.cos(lat1) * math.cos(lat2) * math.sin(dlon /
            2)**2

    c = 2 * math.asin(min(1.0, math.sqrt(a)))

    # Earth's approximate radius in kilometers
    R = 6367.0

    distance = R * c

    return distance

print("Ready to use `haversine()`")
```

Now we can make an edge `list`. We will represent each campus as a node with the initials of the college name. The weight of the edge will represent the Haversine value between the locations. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", Haversine)`.

```
distances = []
colleges = list(lat_long.items())
for i in range(len(colleges)):
    college1, (lat1, lon1) = colleges[i]
    for j in range(i + 1, len(colleges)):
        college2, (lat2, lon2) = colleges[j]
        dist = haversine(lat1, lon1, lat2, lon2)
        distances.append((college1, college2, dist))

print("\nDistances between colleges (in kilometers):")
for edge in distances:
    college1, college2, dist = edge
    print(f"{college1} - {college2}: {dist:.2f} km")
```

8.3.3 Graph

Swap $(Latitude, Longitude)$ coordinates for plotting with (x, y) coordinates.

```
pos = {college: (lon, lat) for college, (lat, lon) in
    lat_long.items()}
```

```
pos
```

Create a `Graph` from the edge `list`:

```
G = Graph(distances)
G.show(
    pos=pos,  # Positions are (longitude, latitude)
    edge_labels=True,
    vertex_size=500,
    figsize=20,
    title="CCC Distance Graph"
)
```

8.3.4 Railway

So far, we have encountered various concepts for connecting a graph's vertices, including the Hamilton path and the MST. Let's consider what technique is best suited for solving the problem of constructing a railway that optimizes material costs.

The previous chapter used the traveling salesperson algorithm to optimize a delivery route. Since we aim to optimize material costs, you might think of following a similar approach: apply the traveling salesperson algorithm, eliminate the greatest edge from the Hamilton circuit, and design the railway with the minimum Hamilton path. If we take a Hamilton circuit and eliminate one edge, we obtain a spanning tree. While the Hamilton path optimizes graph traversal by visiting each vertex exactly once in a single path, it does not guarantee that all vertices are connected with the minimal total weight.

In a Hamilton path, the requirement to visit each vertex in a single path can force the inclusion of high-weight edges. Alternatively, the MST is not restricted by the requirement of connecting vertices with a path. The MST can avoid high-weight edges by connecting vertices without regard to forming a path as long as the graph remains connected and acyclic. Although the minimum Hamilton path is one of many possible spanning trees, it is not an MST. Prim's Algorithm ensures the weight of the spanning tree is minimal because, at each iteration, it selects the smallest-weight edge.

Let's find the MST edge `list` of the campus locations with the `min_spanning_tree(by_weight=True)` function.

```
mst = G.min_spanning_tree(by_weight=True)
mst
```

Visualize the MST with the vertex positions mapped to the geographical coordinates of each campus location.

```
Graph(mst).show(
    pos=pos,
    edge_labels=True,
    vertex_size=500,
    figsize=15,
    title="CCC Minimum Spanning Tree"
)
```

8.3.5 Conclusion

In this exercise, we only optimized construction material costs. In a real-world

scenario, we may want to create a railway that optimizes both travel time and material costs. In the case of the Chicago L train system, the railway resembles a tree when ignoring the downtown Loop. The L receives criticism for its lack of interconnectivity. For example, finding an efficient route connecting the end of the Blue Line with the end of the Red Line is challenging because a traveler may need to commute all the way downtown from one end of the railway to reach another end. As an interesting challenge, you can optimize both travel time and construction costs.

Chapter 9

Lattices

This chapter builds on the partial order sets introduced earlier and explains how to ask Sage to decide whether a given poset is a lattice. Then, we show how to calculate the meet and join tables using built-in and customized Sage functions.

9.1 Lattices

9.1.1 Definition

A **lattice** is a partially ordered set (**poset**) in which any two elements have a least upper bound (also known as join) and greatest lower bound (also known as meet).

In Sage, a lattice can be represented as a poset using the `Poset()` function. This function takes a tuple as its argument, where the first element is the set of elements in the poset, and the second element is a list of ordered pairs representing the partial order relations between those elements.

First, let's define the lists of elements and relations we will use for the following examples:

```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]
print("Elements: ", elements)
print("Relations: ", relations)
```

Create a poset from a tuple of elements and relations.

```
P0 = Poset((elements, relations))
P0.show()
```

The function `is_lattice()` determines whether the poset is a lattice.

```
P0.is_lattice()
```

Notes. Lattices have practical applications in computer science, such as static program analysis and distributed programming.

We can also use `LatticePoset()` function to plot the lattice. The function `Poset()` can be used with any poset, even when the poset is not a lattice. The `LatticePoset()` function will raise an error if the poset is not a lattice.

```
LP = LatticePoset((elements, relations))
LP.show()
```

9.1.2 Join

The join of two elements in a lattice is the least upper bound of those elements.

To check if a poset is a join semi-lattice (every pair of elements has a least upper bound), we use `is_join_semilattice()` function.

```
PO.is_join_semilattice()
```

We can also find the join for individual pairs using the `join()` function.

```
PO.join('b', 'f')
```

9.1.3 Meet

The meet of two elements in a lattice is their greatest lower bound.

To check if a poset is a meet semi-lattice (every pair of elements has a greatest lower bound), we use `is_meet_semilattice()` function.

```
PO.is_meet_semilattice()
```

We can also find the meet for individual pairs using the `meet()` function.

```
PO.meet('a', 'b')
```

9.1.4 Divisor Lattice

The Sage `DivisorLattice()` function returns the divisor lattice of an integer.

The elements of the lattice are divisors of n and $x < y$ in the lattice if x divides y.

```
Posets.DivisorLattice(12).show()
```

9.2 Tables of Operations

This section examines the representation of meet ( ) and join ( ) operations within lattices using operation tables.

9.2.1 Meet Operation Table

The meet operation table illustrates the greatest lower bound, or meet, for every pair of elements in the lattice.

To output the table as a matrix, we need to specify that the poset is indeed a lattice, thus requiring us to use the function `LatticePoset()`. Then, we can use the function `meet_matrix()` to process the table.

```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]

L = LatticePoset((elements, relations))
M = L.meet_matrix()
show(M)
```

From the output matrix, we can see that each entry a_{ij} is not the actual value of the meet of the elements a_i and a_j but just its position in the lattice. Let's show the values:

```
linear_extension = L.linear_extension()

values_meet_matrix = [
    [
        linear_extension[M[i, j]]
        for j in range(len(elements))
    ]
    for i in range(len(elements))
]

values_meet_matrix
```

Show the output as a table:

```
import pandas as pd

df = pd.DataFrame(
    values_meet_matrix,
    index=elements,
    columns=elements
)

df
```

9.2.2 Join Operation Table

Conversely, the join operation table presents the least upper bound, or join, for each pair of lattice elements.

```
J = L.join_matrix()

show(J)
```

Output the elements of the poset:

```
linear_extension = L.linear_extension()

values_join_matrix = [
    [
        linear_extension[J[i, j]]
        for j in range(len(elements))
```

```
    ]
    for i in range(len(elements))
]

values_join_matrix
```

Show the output as a table instead of a matrix.

```
import pandas as pd

df = pd.DataFrame(
    values_join_matrix,
    index=elements,
    columns=elements
)

df
```

Chapter 10

Boolean Algebra

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given lattice is a Boolean algebra. We also illustrate basic operations with Boolean functions.

10.1 Boolean Algebra

A Boolean algebra is a bounded lattice that is both complemented and distributive. Let's define the `is_boolean_algebra()` function to determine whether a given poset is a Boolean algebra. The function accepts a finite partially ordered set as input and returns a tuple containing a boolean value and a message explaining the result. Run the following cell to define the function and call it in other cells.

```
def is_boolean_algebra(P):
    try:
        L = LatticePoset(P)
    except ValueError as e:
        return False, str(e)
    if not L.is_bounded():
        return False, "The lattice is not bounded."
    if not L.is_distributive():
        return False, "The lattice is not distributive."
    if not L.is_complemented():
        return False, "The lattice is not complemented."
    return True, "The poset is a Boolean algebra."
```

Let's check if the following poset is a Boolean algebra.

```
S = Set([1, 2, 3, 4, 5, 6])

P = Poset((S, attrcall("divides")))

show(P)
```

```
is_boolean_algebra(P)
```

When we pass `P` to the `is_boolean_algebra()` function, `LatticePoset()` raises an error because `P` is not a lattice. The `ValueError` provides more

information about the absence of a top element. Therefore, `P` is not a Boolean algebra.

```
T = Subsets(['a', 'b', 'c'])

Q = Poset((T, lambda x, y: x.issubset(y)))

Q.plot(vertex_size=3500, border=True)
```

```
is_boolean_algebra(Q)
```

Let's examine the divisor lattice of 30:

```
dl30 = Posets.DivisorLattice(30)
show(dl30)
is_boolean_algebra(dl30)
```

Now for the divisor lattice of 20:

```
dl20 = Posets.DivisorLattice(20)
show(dl20)
is_boolean_algebra(dl20)
```

Here is a classic example in the field of computer science:

```
B = posets.BooleanLattice(1)
show(B)
is_boolean_algebra(B)
```

10.2 Boolean functions

A **Boolean function** is a function that takes only values 0 or 1 and whose domain is the Cartesian product $\{0,1\}^n$.

Notes. Boolean algebra influences the design of digital circuits. For example, simplifying a digital circuit can minimize the number of gates used and reduce the manufacturing cost.

A **minterm** of the Boolean variables $x_1, x_2, \ldots, x_n$ is the Boolean product $y_1 \cdot y_2 \cdot \ldots \cdot y_n$ where each $y_i = x_i$ or $y_i = \overline{x_i}$.

A sum of minterms is called a **sum-of-products** expansion. In this section, we will examine various methods for finding the sum-of-products expansion of a Boolean function.

To find the sum-of-products expansion using a truth table, we first convert the `truthtable()` into a form that is iterable with `get_table_list()`. For every row where the output value is `True`, we construct a minterm:

- Include the variable as is if its value is `True`
- Include the negation of the variable if its value is `False`
- The `zip` function pairs each variable with its corresponding value, allowing us to create minterms efficiently.
- We add each minterm to the `sop_expansion list` using the `&` operator.
- Finally, we join all minterms with the `|` operator to form the sum-of-products expansion.

- The function returns the sum-of-products expansion as a `sage.logic.boolformula.BooleanFormula` instance.

```
def truth_table_sop(expression):
    # Check if the input is a string, and if so, convert it
        to a formula object
    if isinstance(expression, str):
        h = propcalc.formula(expression)
    elif isinstance(expression,
        sage.logic.boolformula.BooleanFormula):
        h = expression
    else:
        raise ValueError

    table_list = h.truthtable().get_table_list()
    sop_expansion = []

    for row in table_list[1:]:  # Skip the header row
        if row[-1]:  # If the output value is True
            minterm = []
            for var, value in zip(table_list[0], row[:-1]):
                # Iterate over each variable and its value
                in this row
                if value:
                    minterm.append(var)  # Include variable
                        as is if True
                else:
                    minterm.append(f'~{var}')  # Include the
                        negated variable if False
            sop_expansion.append(' & '.join(minterm))  #
                Join variables in the minterm using the AND
                operator

    sop_result = ' | '.join(f'({m})' for m in sop_expansion)
         # Join minterms using the OR operator
    return propcalc.formula(sop_result)
```

For your convenience, our `truth_table_sop` function converts `String` input with `propcalc.formula`. Therefore, the input accepts `String` representations of Boolean expressions. Alternatively, you may pass an instance of `sage.logic.boolformula.BooleanFormula` directly to the function.

```
truth_table_sop("x & (y | z)")
```

Let's verify that the sum-of-products expansion we found with the truth table is equivalent to the original expression.

```
truth_table_sop("x & (y | z)") == propcalc.formula("x & (y |
    z)")
```

Our `sop_expansion` function mimics the manual process of finding the sum-of-products expansion of a Boolean function. This process does not guarantee the minimal form of the Boolean expression.

If we dig around in the Sage source code, we can find a commented-out `Simplify()` function that relied on the `Boolopt` package and the Quine-McCluskey algorithm. The Quine-McCluskey algorithm guarantees the minimal form of the Boolean expression, but the exponential complexity of the algorithm makes it impractical for large expressions. Moreover, in the Sage documentation, we

see a placeholder function called `Simplify()` that returns a `NotImplementedError` message. The Sage community is waiting for someone to implement this function with the Espresso algorithm. While the Espresso algorithm does not guarantee the minimal form of the Boolean expression, it is more efficient than the Quine-McCluskey algorithm.

Sage integrates well with Python libraries like SymPy, which have built-in functions for Boolean simplification. The SymPy `SOPform` function takes the variables as the first argument and the minterms as the second argument. The function returns the sum-of-products expansion of the Boolean function in the smallest sum-of-products form. To use the SymPy `SOPform` function in Sage, first extract the variables and minterms of an expression.

We extract the variables from the first row of the truth table.

```
expression = propcalc.formula("x & (y | z)")
table_list = expression.truthtable().get_table_list()
variables = table_list[0]
print(variables)
```

We make the variables compatible with the SymPy `SOPform` function by converting them to SymPy symbols.

```
from sympy import symbols
sympy_variables = symbols(' '.join(variables))
print(sympy_variables)
```

We extract the minterms from the rows where the output is `True`.

```
minterms = [row[:-1] for row in table_list[1:] if row[-1]]
print(minterms)
```

Now that we have the variables and minterms, we can use the SymPy `SOPform` function to find the sum-of-products expansion of the Boolean function.

```
from sympy.logic import SOPform
from sympy import symbols

def sympy_sop(expression):
    # Convert input expression to a SageMath formula object
        if necessary
    if isinstance(expression, str):
        formula_object = propcalc.formula(expression)
    elif isinstance(expression,
        sage.logic.boolformula.BooleanFormula):
        formula_object = expression
    else:
        raise ValueError("Invalid input: expression must be
            a string or a BooleanFormula object.")

    # Generate the truth table from the formula object
    truth_table = formula_object.truthtable()
    table_list = truth_table.get_table_list()

    # Extract variables and minterms from the truth table
    variables = table_list[0]
    minterms = [row[:-1] for row in table_list[1:] if
        row[-1]]
```

```
    # Convert the SageMath variables to SymPy symbols
    sympy_variables = symbols(' '.join(variables))

    # Use SymPy to compute the SOP form
    sop_result = SOPform(sympy_variables, minterms)

    return propcalc.formula(str(sop_result))
```

```
sympy_sop("x & (y | z)")
```

Let's verify that the sum-of-products expansion we found with SymPy is equivalent to the original expression.

```
sympy_sop("x & (y | z)") == propcalc.formula("x & (y | z)")
```

Now, we present a manual method for finding the sum of products by applying the Boolean identities. Let's find the sum-of-products expansion of the Boolean function

$$h(x, y) = x + \bar{y}.$$

We can apply the Boolean identities and use Sage to verify our work. Currently, we have a sum of two terms but no products. We can apply the identity law to introduce the product terms. Now, we have the equivalent expression

$$h(x, y) = x \cdot 1 + 1 \cdot \bar{y}.$$

Warning: Do not attempt to apply the identity law or null law within the `formula` function. If you try to directly apply the identity law within the formula function like so, `propcalc.formula("x & 1 | 1 ~y")`, Sage will raise an error because `propcalc.formula` interprets `1` as a variable. Variables cannot start with a number.

The `formula` function only supports variables and the following operators:

- `&` *and*
- `|` *or*
- `~` *not*
- `^` *xor*
- `->` *if then*
- `<->` *if and only if*

```
h = propcalc.formula("x | ~y")
show(h)
```

Apply the complement law and verify that our transformed expression is equivalent to the original expression.

```
h_complement = propcalc.formula("x & (y | ~y) | (x | ~x) &
    ~y")
show(h_complement)
h_complement == h
```

Apply the distributive law and verify that our transformed expression is equivalent to the original expression.

```
h_distributive = propcalc.formula("x & y | x & ~y | x & ~y |
    ~x & ~y")
show(h_distributive)
h_distributive == h
```

Apply the idempotent law and verify that our transformed expression is equivalent to the original expression.

```
h_idempotent = propcalc.formula("x & y | x & ~y | ~x & ~y")
show(h_idempotent)
h_idempotent == h
```

We started with the expression,

$$h(x, y) = x + \bar{y}$$

After applying the identity, complement, and distributive laws, we transformed the Boolean function into the sum-of-products expansion

$$h(x, y) = x \cdot y + x \cdot \bar{y} + x \cdot \bar{y} + \bar{x} \cdot \bar{y}.$$

Chapter 11

Logic Gates

This chapter explains how to process binary inputs in Sage to produce specific outputs based on basic logic gates, such as AND, OR, and NOT. Then, we show how these gates combine to form more complex circuits and integrate into everyday electronics, using built-in and customized Sage functions to simulate and analyze their behavior.

11.1 Logic Gates

Logic gates are the foundation of digital circuits. They process binary inputs to produce specific outputs. The basic logic gates are AND, OR, and NOT. Derived gates include $NAND$, NOR, XOR, and $XNOR$. Each gate has its own symbol and behavior defined by a truth table.

Notes. Logic gates combine to form complex systems such as CPUs and memory circuits.

11.1.1 AND Gate

The `AND` gate produces a `1` only when both inputs are `1`.

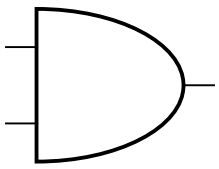

Figure 11.1.1 AND Gate

```
from sympy.logic.boolalg import And
from sympy.abc import A, B
And(A, B)
```

Truth table for the `AND` gate:

```
# Generate truth table for AND gate
print("\nA | B | A AND B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)} | {int(B)} | {int(bool(And(A,
```

```
            B))))}")
```

11.1.2 OR GATE

The `OR` gate produces a `1` if at least one input is `1`.

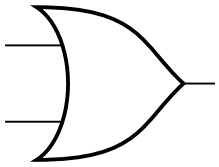

Figure 11.1.2 OR Gate

```
from sympy.logic.boolalg import Or
from sympy.abc import A, B
Or(A, B)
```

Truth table for the `OR` gate:

```
# Generate truth table for OR gate
print("\nA | B | A OR B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)} | {int(B)} | {int(bool(Or(A, B)))}")
```

11.1.3 NOT Gate

The `NOT` gate inverts the input: `1` becomes `0`, and `0` becomes `1`.

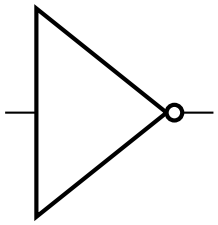

Figure 11.1.3 NOT Gate

```
from sympy.logic.boolalg import Not
from sympy.abc import A
Not(A)
```

Truth table for the `NOT` gate:

```
# Generate truth table for NOT gate
print("\nA | NOT A")
print("--|-------")
for A in [False, True]:
    print(f"{int(A)} | {int(bool(Not(A)))}")
```

11.1.4 NAND Gate

`NAND`: Produces `0` only when both inputs are `1`.

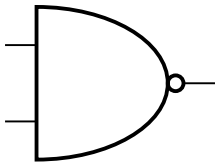

Figure 11.1.4 NAND Gate

11.1.5 NOR Gate

`NOR`: Produces `1` only when both inputs are `0`.

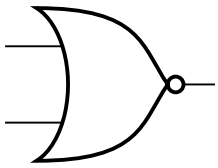

Figure 11.1.5 NOR Gate

11.1.6 XOR Gate

`XOR`: Produces `1` when inputs differ.

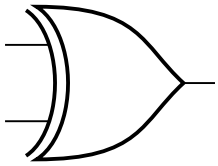

Figure 11.1.6 XOR Gate

11.1.7 XNOR Gate

`XNOR`: Produces `1` when inputs are the same.

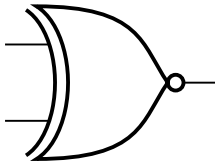

Figure 11.1.7 XNOR Gate

```
from sympy.logic.boolalg import And, Or, Not, Xor

def nand(A, B):
    return Not(And(A, B))

def nor(A, B):
    return Not(Or(A, B))

def xor(A, B):
    return Xor(A, B)

def xnor(A, B):
    return Not(Xor(A, B))

# User-defined inputs
A = 1  # Replace with 0 or 1 for input A
```

```
B = 0  # Replace with 0 or 1 for input B
gate = "xor"  # Replace with "nand", "nor", "xor", or "xnor"


if gate == "nand":
    result = nand(A, B)
elif gate == "nor":
    result = nor(A, B)
elif gate == "xor":
    result = xor(A, B)
elif gate == "xnor":
    result = xnor(A, B)
else:
    result = "Invalid gate type! Please use 'nand', 'nor',
        'xor', or 'xnor'."

result
```

11.2 Combinations of Logic Gates

Logic gates can be combined to create more complex circuits that perform specific tasks. By linking gates together, we can create circuits that process multiple inputs to produce a desired output. For example, combining an AND gate and a NOT gate results in a NAND gate, which inverts the output of the AND gate. More complex circuits, such as half-adders and multiplexers, are built by combining basic gates in strategic ways.

Let's look at a circuit. We evaluate this circuit by setting True for X, Y, and False for Z below using Sage.

```
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import X, Y, Z

# Define the logic circuit
F = Or(And(Not(X), Y, Z), And(X, Not(Y), Z), And(X, Y,
    Not(Z)), And(X, Y, Z))

# Evaluate the logic circuit with values for X, Y, and Z
circuit_output = F.subs({X: True, Y: True, Z: False})
circuit_output
```

Boolean algebra provides a way to simplify complex logic circuits. By using Boolean algebra rules, you can take a complicated circuit and reduce it to a simpler form without changing its functionality.

Here's a practical example. Consider the following Boolean expression, which combines several gates:

```
# Original Boolean expression
from sympy import simplify
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import x, y, z

# Define the expression
D = Or(And(Not(x), y, z), And(x, Not(y), z), And(x, y,
```

```
    Not(z)), And(x, y, z))
D
```

```
# Simplified Boolean expression
simplified_D = simplify(D)
simplified_D
```

Truth tables are a visual way to represent how inputs to a logic circuit map to its outputs. For each possible combination of inputs, the table shows the corresponding outputs, making it easier to analyze and understand the behavior of the circuit.

Let's create a truth table for the simplified circuit.

$F = (x \text{ and } y) \text{ or } (x \text{ and } z) \text{ or } (y \text{ and } z)$

Here, we will show the intermediary steps to find the final output of the function.

```
from sympy.logic.boolalg import And, Or, truth_table
from sympy.abc import x, y, z

# Define the logic function
intermediate1 = And(x, y)  # x AND y
intermediate2 = And(x, z)  # x AND z
intermediate3 = And(y, z)  # y AND z
final_output = Or(intermediate1, intermediate2,
    intermediate3)  # (x AND y) OR (x AND z) OR (y AND z)

# Variables and expressions
variables = [x, y, z]
expressions = [intermediate1, intermediate2, intermediate3,
    final_output]

# Header names and column widths
headers = ["x", "y", "z", "x AND y", "x AND z", "y AND z",
    "F"]
column_widths = [5, 5, 5, 10, 10, 10, 5]  # Adjust widths as
    needed

# Print header row with adjusted spacing
header_row = " | ".join(h.ljust(w) for h, w in zip(headers,
    column_widths))
print(header_row)
print("-" * len(header_row))

# Generate and print the truth table rows
for row in truth_table(final_output, variables):
    inputs = row[0]
    outputs = [int(bool(expr.subs(dict(zip(variables,
        inputs))))) for expr in expressions]
    table_row = " | ".join(str(int(bool(x))).ljust(w) for x,
        w in zip(list(inputs) + outputs, column_widths))
    print(table_row)
```

Chapter 12

Finite State Machines

This chapter delves into a powerful abstract model, namely the *finite-state machines*. Beyond the theoretical framework, the content of this chapter demonstrates the use of Sage to define, model, build, visualize, and execute examples of state machines, showcasing their application in solving real-world problems.

12.1 Definitions and Components

The defining feature of any abstract machine is its memory structure, ranging from a *finite* set of states in the case of finite-state machines to more complex memory systems (e.g., *Turing machines* and *Petri nets*).

A **Finite-State Machine (FSM)** is a computational model that has a finite set of possible states S, a finite set of possible input symbols (the input alphabet) X, and a finite set of possible output symbols (the output alphabet) Z. The machine can exist in one of the states at any time, and based on the machine's input and its current state, it can transition to any other state and produce an output. The functions that take in the machine's current state and its input and map them to the machine's future state and its output are referred to as the *state transition* function and the *output* function, respectively. The default state of an FSM is referred to as the *initial state*.

12.1.1 Mealy State Machine

A Mealy finite-state machine is defined by the tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, a finite set that corresponds to the set of all memory configurations that the machine can have at any time.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \times X \to Z$ is the output function, which specifies which output symbol $w(s, x) \in Z$ is written onto the output device when the machine is in state s and the input symbol x is read.
- $t : S \times X \to S$ is the next-state (or transition) function, which specifies which state $t(s, x) \in S$ the machine should move to when it is currently in state s and it reads the input symbol x.

12.1.2 Other Types of Finite State Machines

12.1.2.1 Moore Machine

In a **Moore Machine**, the output depends *solely* on the current state. Unlike Mealy state machine, this machine must enter a new state for the output to change.

A Moore machine is also represented by the 6-tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, and s_0 is the initial state.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \to Z$ is the output function, which specifies which output symbol $w(s) \in Z$ associated with the machine current state s.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

12.1.2.2 Finite-State Automaton

A *final state* (also known as the accepted state) is defined as a *special* predefined state that indicates whether an input sequence is valid or accepted by the finite-state machine. The set F of all final states is a subset of the states set S.

A **Finite-State Automaton** is a finite-state machine *with no output*, and it is represented by the 5-tuple (S, X, t, s_0, F) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, s_0 is the initial state, and F is the set of final states.
- The state s_0 is called the *initial state*.
- The subset $F \subset S$ is the set of all final states of the machine.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

When the state machine processes a finite input sequence, it transitions through various states based on each input in the sequence and the current state of the machine. If, after processing the entire sequence, the machine lands in any of the *final states*, then the input sequence is considered valid (or recognized according to the machine's rules). Otherwise, the input sequence is rejected as invalid.

12.1.2.3 Deterministic Finite Automaton (DFA)

A **Deterministic Finite Automaton (DFA)** is a simplified automaton in which each state has exactly one transition for each input. DFAs are typically used for lexical analysis, language recognition, and pattern matching.

Note. A text parser or a string-matching application that recognizes a specific language or regular expressions are real-world examples of DFA use.

12.1.2.4 Nondeterministic Finite Automaton (NFA)

Unlike a DFA, an **NFA** allows multiple transitions for the same input or even transitions without consuming input (ϵ-transitions).

12.1.2.5 Turing Machine

A **Turing Machine** is an expansion of an FSM, which includes infinite tape memory representing both the input and output streams (shared stream). Unlike all other FSMs, a Turing machine can alter the input/output stream, and as such, it is capable of simulating any algorithm. Turing machines are the theoretical foundation for modern computation (any general-purpose computer executing any algorithm can be modeled as a Turing Machine).

Finite state machines are a foundational concept in computer science, often associated with tasks related to system designs (circuits and digital computers, algorithms, etc.). However, the vast and rich domain of applications of state machines extends far beyond simple simulations to the full control logic of complex industrial processes and workflows. These tasks can vary in complexity, ranging from a simple parity check to managing traffic patterns, a programming language compiler, or natural language recognition and processing.

State machines offer a structured way to model systems with discrete states and transitions. Different variants, such as the Mealy machine and Moore machine, have distinct characteristics and, as such, can adapt to various applications.

12.2 Finite State Machines in Sage

Although Sage includes a dedicated built-in rich module to handle various types of state machines, it may not always be sufficient to address certain use cases or implement specific custom behaviors of the machine. Additionally, the built-in module allows state machines to be defined and constructed in different ways, providing greater flexibility and making it more suitable from a programmer's perspective. However, it may not fully conform to the precise definition given earlier. This highlights that it is still possible to model, construct, display, and run relatively simple state machines by utilizing general-purpose tools, such as graphs and transition matrices, to represent and operate on state machines.

Notes. While Sage provides basic tools to represent and simulate state machines, it may not natively support more complex state machine features such as parallel states or hierarchical transitions.

12.2.1 The Elevator State Machine

Let's design a basic controller to an elevator to show the process of defining states, creating a state transition graph, visualizing the state machine, and simulating its execution in Sage.

Consider a 3-level elevator (floors 1 through 3). The elevator has 3 buttons for users to select the destination floor (only one can be selected at a time). Depending on the current position and the selected floor, the elevator can go up, go down, or remain on the same floor.

12.2.2 Description of the Elevator FSM

This elevator system can be modeled and simulated using a finite-state machine with states $S = \{f_1, f_2, f_3\}$ representing each floor, the user inputs set $X = \{b_1, b_2, b_3\}$ (where b_i represents the button for i^{th} floor), and the outputs set $Z = \{U, D, N\}$ for 'going up', 'going down', or 'going nowhere'.

The components of this FSM are transcribed in the following table.

Table 12.2.1 The Elevator State Machine Definition

	next			**output**		
current	b_1	b_2	b_3	b_1	b_2	b_3
f_1	f_1	f_2	f_3	N	U	U
f_2	f_1	f_2	f_3	D	N	U
f_3	f_1	f_2	f_3	D	D	N

The following steps outline the approach to build and test the elevator controller system:

1. Define the elements of the Finite State Machine: States, Inputs, Transitions, and Outputs.
2. Construct the State Machine.
3. Run the machine using a sample input set.

12.2.3 Elements of the Elevator FSM

The first step is to define the states and transitions in the state machine, which can be represented using lists and dictionaries.

```
# Define state, input and output sets
states = ['f1', 'f2', 'f3']
inputs = ['b1', 'b2', 'b3']
outputs = ['U', 'D', 'N']

# Transitions as a dictionary {(current_state, input):
    next_state}
transitions = {
    ('f1', 'b1'): 'f1',
    ('f1', 'b2'): 'f2',
    ('f1', 'b3'): 'f3',

    ('f2', 'b1'): 'f1',
    ('f2', 'b2'): 'f2',
    ('f2', 'b3'): 'f3',

    ('f3', 'b1'): 'f1',
    ('f3', 'b2'): 'f2',
    ('f3', 'b3'): 'f3',
}

# The machine outputs control how the elevator would move
outputs = {
    ('f1', 'b1'): 'N',
    ('f1', 'b2'): 'U',
    ('f1', 'b3'): 'U',

    ('f2', 'b1'): 'D',
    ('f2', 'b2'): 'N',
```

```
    ('f2', 'b3'): 'U',

    ('f3', 'b1'): 'D',
    ('f3', 'b2'): 'D',
    ('f3', 'b3'): 'N',
}

# Display the machine configuration
print('States: ', states)
print('Transitions: ', transitions)
print('Outputs: ', outputs)
```

12.2.4 Graph Model of the Elevator FSM

An FSM can be modeled as a graph where vertices represent the states, and the directed edge between vertices is the relationship between two states (the transition from one state to the other). The weight of a directed edge between two vertices represents the pair of input and output associated with the transition between the two states.

In Sage, the `DiGraph` class can be used to represent the states, transitions, and outputs of the state machine as a directed graph, leveraging the graph structure to visualize the state machine representation.

```
# 'DiGraph' is imported by default. If not, it can be
    imported as follow
# from sage.graphs.digraph import DiGraph

# Initialize a directed graph
elevator_fsm = DiGraph(loops=True)

# Add states as vertices
elevator_fsm.add_vertices(states)

# Add transitions and outputs as edges
for (_state, _input), next_state in transitions.items():
    _output = outputs[(_state, _input)]
    edge_label = f"{_input}, {_output}"
    elevator_fsm.add_edge(_state, next_state,
        label=edge_label)

# Display the graph (state machine)
elevator_fsm.show(
        figsize=[5.6, 5.6],
        layout='circular',
        vertex_size=250,
        edge_labels=True,
        vertex_labels=True,
        edge_color=(.2,.4,1),
        edge_thickness=1.0,
)
```

The `show()` method renders a graphical representation of the state machine. Each vertex in the graph represents a state, and each directed edge represents a transition, labeled as (input, output).

12.2.5 Run the Elevator State Machine

Next, the state machine's behavior can be simulated by defining a function that processes a list of inputs and transitions through the states accordingly.

```
# Function to run the state machine
def run_state_machine(start_state, inputs):
    current_state = start_state
    for _input in inputs:
        print(f"Current state: {current_state}, Input:
            {_input}")

        if (current_state, _input) in transitions:
            current_output = outputs[(current_state, _input)]
            current_state = transitions[(current_state,
                _input)]
            print(
                f"Transitioned to: {current_state}\n"
                f"Output: {current_output}\n"
            )
        else:
            print(
                f"No transition/output available for input
                    {_input} in state {current_state}"
            )
            break

    print(f"Last state: {current_state}")

# Example of running the state machine
start_state = 'f2'
inputs = ['b1', 'b1', 'b3', 'b2']

run_state_machine(start_state, inputs)
```

The `run_state_machine()` function simulates the state machine by processing a list of inputs starting from an initial state.

12.2.6 The Traffic Light State Machine

Let's design a simple traffic light controller to illustrate alternative methods for defining, visualizing, and executing finite state machines in Sage.

Consider a simplified traffic light system controlled by preset timers. This system operates through three phases that represent the flow of road traffic: Free-flowing, Slowing-down, and Halted. These phases correspond to the traffic light signals: green, yellow, and red, controlling the flow of traffic. The system uses three timer settings: 30 seconds, 20 seconds, and 5 seconds. When a timer expires, it triggers the transition to the next phase. Initially, the light is green, the traffic is flowing, and:

- When the 30-second timer expires, the traffic light changes from green to yellow, and traffic begins to slow down.
- When the 5-second timer expires, the traffic light changes from yellow to red, bringing traffic to a complete stop.
- When the 20-second timer expires, the traffic light changes from red to green, allowing traffic to start moving again.

12.2.7 Description of the Traffic Light FSM

In this traffic light system, the three phases representing the flow of road traffic: *Free-flowing (F)*, *Slowing-down (S)*, and *Halted (H)* are the states $S = \{F, S, H\}$ of the FSM. These phases correspond to the traffic light signals: green (G), yellow (Y), and red (R), which are the outputs set $Z = \{G, Y, R\}$ of the system. The timers driving the transitions are the inputs set $X = \{t_{5s}, t_{20s}, t_{30s}\}$ of this traffic light system.

The following table summarize the elements of the traffic light FSM.

Table 12.2.2 The Traffic Light State Machine Definition

	next			**output**		
current	t_{5s}	t_{20s}	t_{30s}	t_{5s}	t_{20s}	t_{30s}
F	F	F	S	G	G	Y
S	H	S	S	R	Y	Y
H	H	F	H	R	G	R

By applying the same steps and approach as in the previous section, the traffic light controller system will be built and tested, this time utilizing the Sage built-in module and functions.

12.2.8 Using `FiniteStateMachine' Module

Sage `FiniteStateMachine` built-in library provides a powerful tool to model, construct as well as simulate state machines of various systems. This module will be leveraged to showcase its capabilities on the given example, and demonstrating how it can be used to construct and display the FSM, manage its state transitions and outputs.

The command `FiniteStateMachine()` constructs an *empty* state machine (no states, no transitions).

```
from sage.combinat.finite_state_machine import FSMState

# FSM states, inputs and outputs
states = ['F', 'S', 'H']         # Free-flowing,
    Slowing-down, Halted
inputs = ['t30s', 't5s', 't20s'] # timer durations before
    state transitions
outputs = ['G', 'Y', 'R']        # traffic light: Green,
    Yellow, Red

# Create an empty state machine object
traffic_light_fsm = FiniteStateMachine()
traffic_light_fsm
```

The function `FSMState()` defines a state for a given label. The `is_initial` flag can be set to true to set the current state as the *initial state* of the finite state machine. The method `add_state()` appends the created state to an existing state machine.

```
# Define a new state then adding it
free_flowing = FSMState('F', is_initial=True)
traffic_light_fsm.add_state(free_flowing)

# Adding more states by their labels (saving state handlers,
```

```
    to use them in state transitions)
slowing_down = traffic_light_fsm.add_state('S')
halted = traffic_light_fsm.add_state('H')

# the FiniteStateMachine instance
traffic_light_fsm
```

To check whether or not a finite state machine has a state defined, `has_state()` method can be used by passing in the state label (case-sensitive).

```
traffic_light_fsm.has_state('F')
```

The function `states()` enumerates the list of all defined states of the state machine.

```
traffic_light_fsm.states()
```

The method `initial_states()` lists the defined initial state(s) of the state machine.

```
traffic_light_fsm.initial_states()
```

To define a new transition between two states (as well as the input triggering the transition, and the output associated with the state transition), the method `FSMTransition()` can be used. The method `add_transition()` attaches the defined transition to the state machine, and the function `transitions()` enumerates the list of all defined transitions of the state machine.

```
from sage.combinat.finite_state_machine import FSMTransition

# defining 3 transitions, and associating them the state
    machine
# After 30sec, transition from free-flowing to slowing-down,
    and set traffic light to yellow
traffic_light_fsm.add_transition(
    FSMTransition(
        from_state=free_flowing,
        to_state=slowing_down,
        word_in='t30s',
        word_out='Y'
    )
)

# After 5sec, transition from slowing-down to halted, and
    set traffic light to red
traffic_light_fsm.add_transition(FSMTransition(slowing_down,
    halted, 't5s', 'R'))

# After 30sec, transition from halted back to free-flowing,
    and set traffic light to green
traffic_light_fsm.add_transition(FSMTransition(halted,
    free_flowing, 't20s', 'G'))

traffic_light_fsm.transitions()
```

An alternative method for defining state transitions in an FSM is by using the `add_transitions_from_function()` method. This approach accepts a callable function that takes two states as arguments: the source state and the target state. The following code demonstrates how this can be implemented.

```
from sage.combinat.finite_state_machine import FSMTransition

# define state transitions, inputs and outputs
def transit_function(state1, state2):
    if state1=='F':
        if state2 =='S':
            return ('t30s', 'Y')

    elif state1=='S':
        if state2 =='H':
            return ('t5s', 'R')

    elif state1=='H':
        if state2 =='F':
            return ('t20s', 'G')

    # all other 'no-transition' combinations
    return None

traffic_light_fsm.add_transitions_from_function(transit_function)
traffic_light_fsm.transitions()
```

Once the states and transitions are defined, the state machine can be run using `process()` method, which then returns the intermediary outputs during the state machine run.

```
# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_fsm.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

# print out the outputs of the state machine run
outputs_history
```

The `graph()` command displays the graph representation of the state machine.

```
traffic_light_fsm.graph().show(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```

The `FiniteStateMachine` class also offers LaTeX representation of the state machine using the `latex_options()` method.

```
# define printout options
traffic_light_fsm.latex_options(
    format_state_label=lambda x: x.label(),
)

# display commands
print(latex(traffic_light_fsm))
```

Note that the LaTeX printout may not have all elements displayed. However, it can still be customized further. The following figure shows a rendering of the above LaTeX commands.

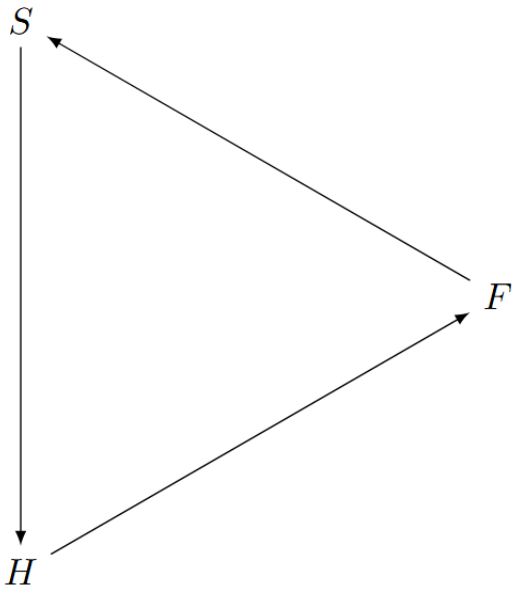

Figure 12.2.3 FSM graph output.

12.2.9 Using `Transducer' Module

Sage `Transducer` is a specialization of the generic `FiniteStateMachine` class. The `Transducer` class creates a finite state machine that support optional final states, and whose transitions have input and output labels.

Let's see how to create another state machine using `Transducer` and for the same traffic light example.

```
# the module allows the instantiation of a state machine by
    passing
# the entire state machine definition to the constructor
state_machine_definition = {
    # from-state: [
    #   a list of tuples
    #   (to-state, input, output)
    # ]
    'F': [
        ('F', 't5s',  'G'),
        ('F', 't20s', 'G'),
        ('S', 't30s', 'Y'),
    ],
```

```
    'S': [
        ('H', 't5s',  'R'),
        ('S', 't20s', 'Y'),
        ('S', 't30s', 'Y'),
    ],
    'H': [
        ('H', 't5s',  'R'),
        ('F', 't20s', 'G'),
        ('H', 't30s', 'R'),
    ],
}

traffic_light_transducer = Transducer(
    state_machine_definition,
    initial_states=['F']
)
traffic_light_transducer
```

The member variable `input_alphabet` lists the set of the transducer inputs set.

```
traffic_light_transducer.input_alphabet
```

The member variable `output_alphabet` lists the set of the transducer outputs set.

```
traffic_light_transducer.output_alphabet
```

Since a `Transducer` is also a `FiniteStateMachine` , the method `has_state()` can still be used to check whether or not a given state exists in the defined transducer (by passing in the case-sensitive state label).

```
traffic_light_transducer.has_state('F')
```

The function `states()` enumerates the list of all defined states of the state machine.

```
traffic_light_transducer.states()
```

The method `initial_states()` lists the defined initial state(s) of the state machine.

```
traffic_light_transducer.initial_states()
```

After defining the states and transitions, the transducer can be executed using the `process()` method from the parent `FiniteStateMachine` class. This method returns the intermediate outputs generated during the execution of the state machine.

```
# fetching the initial state by its label
free_flowing = traffic_light_transducer.state('F')

```

```
# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_transducer.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

outputs_history
```

The `graph()` command displays the graph representation of the transducer-based state machine.

Notes. The `latex_options()` method of the base class `FiniteStateMachine` also is inherited and can also be used with `Transducer` state machine to output LaTeX representation.

```
traffic_light_transducer.graph().show(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```

The above are basic commands with a typical workflow of defining and running of simple finite state machines. The general structure of the state machine can be adapted to fit different use cases. The examples shown can be customized and fine-tuned to reflect more complex scenarios (more states, different input sequences, etc.)

12.3 State Machine in Action

Traffic light systems are crucial for regulating traffic. These systems use carefully coordinated signals to ensure safety for both vehicles and pedestrians. In the previous section, the traffic light system was modeled in an overly simplistic way. This section adds complexity to account for pedestrian presence, ensuring safe crossings while maintaining smooth traffic flow.

12.3.1 Traffic Light Controller: Problem Overview

Let's design a traffic light system for a two-way road with pedestrian crossings. This system coordinates the movement of vehicles and pedestrians using lights to indicate when vehicles can proceed, slow down, or stop, and when pedestrians can cross safely. Vehicle traffic lights include three signals: Red, Yellow, and Green. For simplicity, the pedestrian lights also use three signals: red, yellow, and green. Signal transitions are governed by timers, as described in the previous section, with each timer triggering a transition event after a predefined duration.