Enumerating Hypergraphs

Yeshwanth Zagabathuni

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

Greechie diagrams are a category of hypergraphs representing Quantum Structures. Each of these diagrams represents an algebra followed by Quantum Structures called an Orthoalgebra. Although the rules for generating a Greechie diagram appear relatively simple, it is still tricky as, given the number of vertices, there can numerous ways to connect them to form valid Greechie diagrams.

This thesis gives 5 Algorithms that generate a portion of those Greechie diagrams by taking the Number of Vertices as input. Furthermore, a validation function takes the Edges (or blocks) as input and returns whether a given Greechie diagram is valid. Additionally, for intuitional purposes, it provides a function for validating Orthoalgebras. It finally provides the function that generates the conventional diagram used to represent algebras, the Hasse diagram. The Thesis wraps up with the limitations encountered and gives direction for future work.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics Committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yeshwanth Zagabathuni*)

Acknowledgements

The Author thanks Professor Chris Heunen for his constant support in producing the project proposal and the thesis.

Table of Contents

Chapter 1

Introduction

In Mathematical theory and, specifically, in Discrete Mathematics, we might know of Graphs as a combination of vertices and edges (V, E). They have various applications in Electronics, Mathematics, Computer Networks, Web Technologies, Biology, etc. But the one common property you see with Graphs, particularly about Edges, is that each edge has exactly 2 vertices and will not be able to accommodate any further vertices. This is where the generalization of Graphs called Hypergraphs comes into play.

There is no limit on the number of Vertices a Hyperedge can contain as it may contain any non-empty subset of Vertices and has a much more generalized representation than a Graph. This study focuses on a category of hypergraphs called the Greechie diagram. They are also called ”Condensed Hasse diagrams” as in [12] and make one of the essential concepts in Quantum theory.

The Chapter discusses the fundamentals required to understand Greechie diagrams. The subsequent sections discuss the motivation for Quantum theory, the algebras that Greechie diagrams represent called Orthoalgebras, partially ordered sets and Hypergraphs, and a few other definitions, before finally discussing Greechie diagrams. The thesis will further explain the different components of this Project, including relevant algorithms and the task they accomplish in Chapter 2.

Chapter 3 answers the Research Questions below, Big-oh time complexity calculation for the Greechie diagram generation algorithms and the limitations encountered during the project. Finally, it concludes the thesis with critical findings and directs future research.

1. What factors influence the time complexity of Greechie diagrams?

2. Does existing research focus on generating every possible Greechie Diagram?

1.1 Orthoalgebras and Partial Ordered Sets

1.1.1 Heisenberg's Uncertainty Principle

The motivation behind quantum logic is Hesisenberg's Uncertainty Principle. Hesisenberg's Uncertainty Principle states that it is impossible to simultaneously determine a quantum particle's position and momentum.

$$\Delta x \Delta p \geq h/4\pi$$

Here 'x' is the position, and 'p' is the momentum. This principle can be further used to deduce that the distributive property does not hold for quantum particles. Further detail on quantum logic can be looked up in [15, 9, 6].

1.1.2 Orthoalgebra

There is a unique algebra that generalises quantum logic called orthoalgebra. An orthoalgebra can be defined as a set L with two particular elements 0 and 1 that are provided with a relation called orthogonality[18]. This is denoted by $\perp$. For every pair (suppose a,b) in L ($a, b \in L$) with $a \perp b$, there exists an orthogonal sum $a \oplus b$ that satisfies the following axioms:

- Commutative Law: If $a \perp b$ then:
  1. $b \perp a$
  2. $a \oplus b = b \oplus a$
- Associative Law: If $a \perp b$ and $(a \oplus b) \perp c$ then:
  1. $b \perp c$
  2. $a \perp (b \oplus c)$
  3. $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Orthocomplementation Law: For each $a \in L$, there is a unique $a' \in L$ such that:
  1. $a \perp a'$
  2. $a \oplus a' = 1$
- Consistency: If $a \perp a$ then:
  1. a = 0

1.1.3 Partial Ordered sets or posets

A partially ordered set is a set Q $(Q, \preceq)$ where $\preceq$ is a binary relation and $\forall x, y, z \in Q$, the following properties hold:

1. Reflexivity: $x \preceq x$
2. Transitivity: If $x \preceq y$ and $y \preceq z$ then, $x \preceq z$
3. Anti-symmetric: If $x \preceq y$ and $y \preceq x$ then x = y

A common way to represent partially ordered sets is by using Hasse diagrams. These diagrams represent the relationships between elements of a poset graphically.

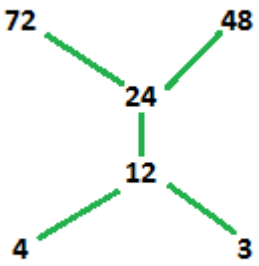

Figure 1.1: Hasse Diagram example for (3,4,12,24,48,72, /)[3]

Figure 1.1 is an example of a Hasse diagram for the set {3,4,12,24,48,72} on the operation divisibility (/). Initially, 3 and 4 would be at the bottom as they are the smallest numbers. At the same time, they are unrelated (4 is not divisible by 3), and thus there is no edge between them.

However, 12 is divisible by both 4 and 3 ($4 \preceq 12$ and $3 \preceq 12$). Thus, both 4 and 3 are branched to 12. Next, 24 is divisible by 12; thus, we have an edge from 12 to 24 ($12 \preceq 24$). Finally, {72, 48} are divisible by 24 ($24 \preceq 48$ and $24 \preceq 72$), but 72 is not divisible by 48. Thus both 72 and 48 are connected to 24 but not to each other. This wraps up the Hasse diagram, which is valid as it holds all three properties. This can make the illustration of a poset much easier.

1.1.4 Orthocomplemented Poset

An orthocomplemented poset is a poset $(P, \preceq)$ with greatest element 1 and least element 0 with unary operation " ' " and it satisfies the following properties $\forall p, q \in L$[17]:

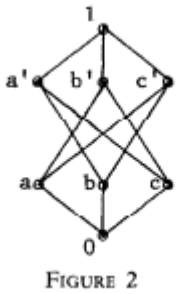

FIGURE 2

Figure 1.2: Hasse diagram for an orthocomplemented poset[8]

1. $(p')' = p$ (Involution Law)
2. $p \vee p' = 1$ (Complementation law)
3. If $p \preceq q$ then $q' \preceq p'$ (Order-reverse)

Consider the Hasse diagram for an orthocomplemented poset in Figure 1.2:

- Since the orthocomplement of a vertex is positioned right above it, clearly:
  1. $(a')' = a$
  2. $(b')' = b$
  3. $(c')' = c$

  Thus, the Involution law holds.
- Since involution law holds,
  1. $(a \vee a') = 1$
  2. $(b \vee b') = 1$
  3. $(c \vee c') = 1$

  Thus the Complementation law holds.
- From the diagram,
  1. $b \preceq a'$ and $a \preceq b'$
  2. $b \preceq c'$ and $c \preceq b'$
  3. $c \preceq a'$ and $a \preceq c'$

  Thus, Order-reverse holds.
- Since all properties hold, it is a **valid** Orthocomplemented poset.

1.2 Hypergraphs

1.2.1 Introduction

A hypergraph H = (V, E) is a graph that contains vertices ($v_i \in V$) and generalized edges known as hyperedges ($e_i \in E$) that connect a non-empty subset of vertices and not necessarily 2 vertices. For example, in Figure 1.3, the hyperedge e1 connects 3 vertices $\{v_1, v_2, v_4\}$.

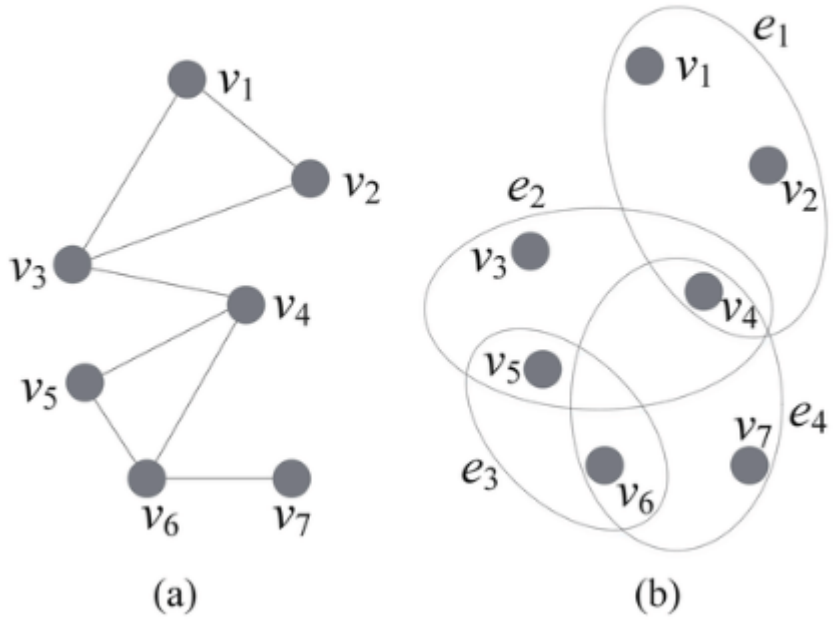

Figure 1.3: Graph v Hypergraph[7]

The most critical difference in hypergraphs is how it differs from a typical graph, as shown in Figure 1.3. Figure 1.3 (a) is a standard graph with edges and vertices, with each edge connecting precisely 2 vertices. This results in a Graph G = (V, E) with 7 vertices and 7 edges.

In contrast, we generalise Figure 1.3 (a), and the result is Figure 1.3 (b). Now this figure has edges connecting not necessarily 2 vertices but as many as possible. Although e3 connects 2 vertices, v5 and v6 like a Graph, Edge e1 connects 3 vertices: v1, v2 and v4. We see this with e2 and e4 as well.

The edge and vertex connections can be summarised in Table 1.1. This table can be better understood if viewed column-wise (by edges).

Vertex	**e1**	**e2**	**e3**	**e4**
v1	1	0	0	0
v2	1	0	0	0
v3	0	1	0	0
v4	1	1	0	1
v5	0	1	1	0
v6	0	0	1	1
v7	0	0	0	1

Table 1.1: Hypergraph table[7]

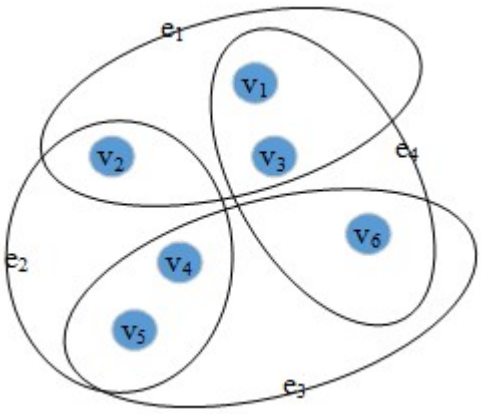

Figure 1.4: Hypergraph example[11]

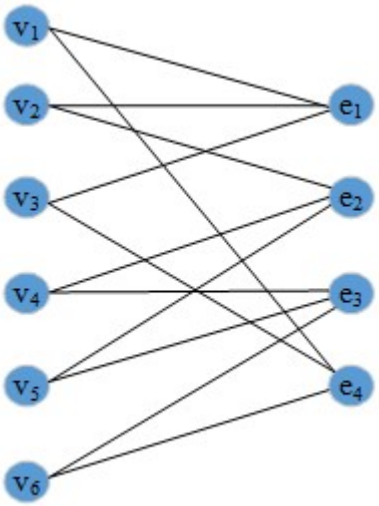

Figure 1.5: Hypergraph converted to bi-partite graph[11]

The generalization of Graphs to Hypergraphs appears as easy as imagining the 1-dimensional edges, connecting 2 vertices as 2-dimensional planes (hyperedges) being able to have as many vertices as possible. But understanding these figures the other way around is a challenge. The simplest hypergraphs with very few atoms (vertices) and edges, like those in Figures 1.3 and 1.4, are easy to convert, interpret and understand as regular Graphs.

For example, Figure 1.4 can be easily converted to a bi-partite graph, as in Figure 1.5. But as the complexity of a hypergraph increases, that interpretation becomes a challenge.

There are various kinds of hypergraphs depending on the application. Undirected hypergraphs do not have directed hyperedges. For example, the hyperedge $(A,B) \in E$ would mean the same as $(B,A) \in E$. On the other hand, directed hypergraphs (like the one in Figure 1.6) have directed hyperedges and, in this case, $(A,B) \neq (B,A)$ where $(A,B) \in E$ and $(B,A) \in E$.

The most common variant in real-time applications is the directed hypergraph with hyperedges carrying weights. But in this study, we are only interested in Undirected Hypergraphs like the ones in Figure 1.3 and Figure 1.4.

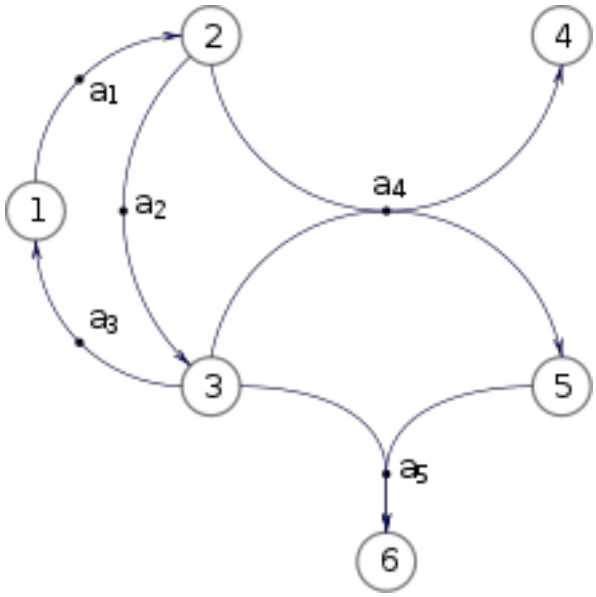

Figure 1.6: A directed hypergraph with vertices, $V = \{1,2,3,4,5,6\}$ and directed hyperedges, $E = \{a1,a2,a3,a4,a5\} : a1 = (\{1\},\{2\}), a2 = (\{2\},\{3\}), a3 = (\{3\},\{1\}), a4 = (\{2,3\},\{4,5\}), a5 = (\{3,5\},\{6\})$[5]

1.2.2 Useful Definitions

A few common definitions in hypergraphs that are useful are as follows[13]:

1. The **order** of a hypergraph $o_H := |V|$
2. The **rank** of a hypergraph is the maximum of cardinalities of hyperedges $r_H := \max_{e \in E} |e|$
3. The **degree** of a vertex ($v_i \in V$) in a hypergraph H, is the number of hyperedges it is a part of: $deg(H) := |H(v_i)|$
4. The length of the shortest cycle in a hypergraph is its **girth**[17].
5. Two hypergraphs, H1 and H2, are **isomorphic**, if they have the same number of vertices and hyperedges and the edge connectivity, is retained. It is a bijection between vertices of 2 different hypergraphs: $f : V(H1) \Rightarrow V(H2)$.

   This means that if a_1 and a_2 are adjacent in H1, then $f(a_1)$ and $f(a_2)$ are adjacent in H2. A graph or hypergraph may have numerous isomorphisms, and [12] suggests an approach to generate Greechie diagrams free of isomorphisms. Figure 1.7 is one of the simplest examples of isomorphism:

   - Both Graphs have 7 vertices and 9 edges.
   - There are exactly 3 vertices of degree 3.
   - We can also map a bijective function as follows: $f(A) = 7, f(B) = 4, f(C) = 3, f(D) = 6, f(E) = 5, f(F) = 2, f(G) = 1$

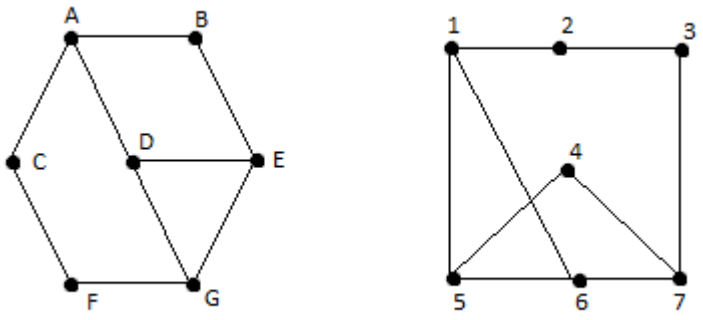

Figure 1.7: Example of isomorphic graphs[2]

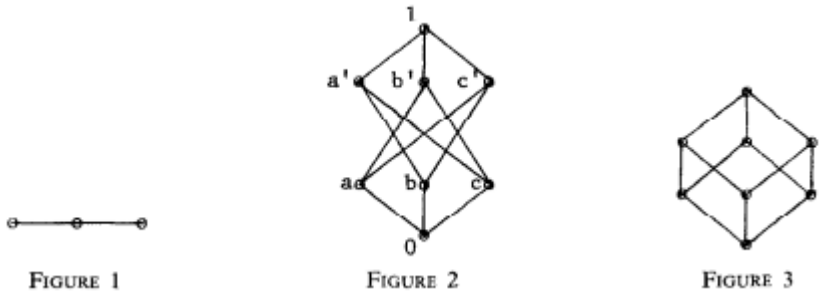

Figure 1.8: Greechie diagram (Figure 1), Hasse Diagram (Figure 2) and the Lattice (Figure 3)[8]

1.3 Greechie Diagrams

1.3.1 Definition

In 1971, R. J. Greechie introduced a set of diagrams that simplified Hasse Diagrams even more[8]. For example, the Hasse diagram of the lattice 2^3 in Figure 1.8(3) is in Figure 1.8(2). This Hasse diagram represents an orthocomplemented poset with the orthocomplement of a vertex appearing directly above it. Then we have the Greechie diagram in Figure 1.8(1), simplifying the Hasse diagram of 2^n vertices with just n vertices ($2^3 \rightarrow 3$).

All Greechie diagrams are Hypergraphs but not all Hypergraphs are Greechie diagrams (like Figure 1.9), as these diagrams follow a set of rules. The following

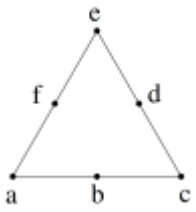

Figure 1.9: Hypergraph with valid orthoalgebra (*Wright's Triangle*)[17] but **not a valid Greechie diagram.**

theory from [16, 10] will help us understand Greechie diagrams better.

1. **Diagram**: A *diagram* is a pair (V, E) such that $V \neq \emptyset$ is a set of atoms (or vertices that are represented as points) and $E \subset expV - \{\phi\}$ is a set of blocks (or hyperedges represented as line segments connecting *points*).

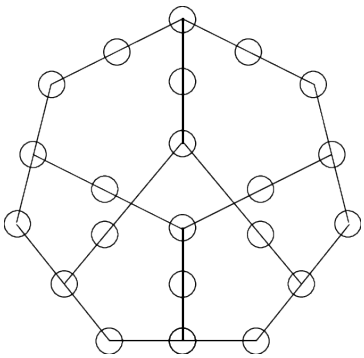

Figure 1.10: A valid Greechie diagram made only out of hyperedges or blocks of only 3 atoms[12]

2. **Loop**: A loop (order $n \geq 2$ where 'n' is a natural number) in a *diagram* is a sequence of mutually different blocks $(e_1, e_2, e_3, ..., e_n) \in E^n$ such that there are mutually distinct atoms $v_1, v_2, ..., v_n$ with $v_i \in e_i \cap e_{i+1} (i = 1, ...n, e_{n+1} = e_1)$.

3. The terms $Vertices \Leftrightarrow Atoms$, $Hyperedge \Leftrightarrow Block$, $GreechieDiagram \Leftrightarrow Diagram$ are used interchangeably in literature[12].

Greechie Diagram: A Greechie Diagram is a diagram satisfying the following conditions:

1. Every atom is part of at least one block.
2. Given the number of atoms ≥ 2, every block has ≥ 2 atoms.
3. Every block intersecting with another block has ≥ 3 atoms.
4. Every pair of blocks intersect with at most one common atom.
5. No loop of order 3 is allowed.

1.3.2 Validation

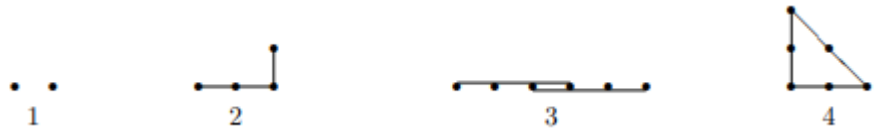

Figure 1.11: Examples of invalid Greechie Diagrams[16]

Figure 1.11 is a collection of examples that represent **Invalid Greechie diagrams** as they violate at least one of the above-defined rules:

- Figure 1.11(a) is invalid by **Rule 2** as one-atom blocks are not allowed. At a minimum, a block must be a 2-piece.
- Figure 1.11(b) is invalid by **Rule 3** as one of the connecting blocks is a 2-piece.
- Figure 1.11(c) is invalid by **Rule 4** as the blocks have more than 1 common atom.
- Figure 1.11(d) is invalid by **Rule 5** as there is a loop of order 3.

And although the 2nd condition makes it clear that a block in a Greechie diagram can have any number of atoms (≥ 2), practically all examples are a combination of multiple 3-atom blocks[12]. Some publications even omit the use of 2-atom blocks. Figure 1.10 is an example of this. Figure 1.9 also represents a valid orthoalgebra called the *Wrights's Triangle* but cannot be considered a Greechie diagram by **Rule 5**.

Generally, a Greechie diagram is considered one the simplest ways to represent an Orthoalgebra. But there also exist Orthoalgebras for which Greechie diagrams do not exist[17].

Chapter 2

Methodology

The methodology comprises several parts, and each part will be discussed in a separate section. The significant tasks involve generating and validating Greechie diagrams and then Orthoalgebras. This section will then discuss the outputs, a comparison with previous works and the limitations of this project. The implementation of the project was carried out in Python on a Google co-lab notebook. It heavily utilizes the **graphviz**[4] and **numpy**[1] modules. **Graphviz** facilitates a very easy and quick generation of the diagrams. **numpy** allows for smoother array operations which will be helpful in the validation part.

2.1 Greechie diagram generation

Several varieties of Greechie diagrams can be created with a given number of vertices, and all of the previous works only cover a small portion of diagrams that interest them. For example, [12] only covers Greechie diagrams with blocks of exactly 3 atoms each. Figure 1.10 in Section 1.3.1 is an example of the type of diagrams they are interested in.

Even this project covers a portion of the possibilities, accomplished by several functions, each generating a particular portion of diagrams. There are a total of 5 functions, and they will be explained in the subsections below. Before that, there are a few terminologies to know to understand the algorithms:

1. Atom: Vertex; Block: Hyperedge; Diagram: Greechie Diagram
2. display(Graph): Displays a Greechie diagram.
3. Block($V_0, V_1, ...$): Connects all specified vertices to form a block.

4. $Loop(V_i, V_j)$: Connects all atoms from V_i to V_j to form a loop. The assumption is that each block has precisely 3 atoms. This means that a loop of order n would require 2n atoms.

5. Also note that the blocks may be outlined with colours to distinguish between blocks or explain new connections formed by the Algorithm.

2.1.1 Disconnected Blocks without loops

Algorithm 1: Greechie Diagram Function 1

```
Data: N ≥ 3
// Create an array, Vertices:V_0 to V_{N-1}
Function Disconnected_Greechie_Diagrams(Vertices):
    if N is not a prime number then
        for n ← 2 to N/2 1 do
            if N%n = 0 then
                for V ← 0 to N − 1 n do
                    // Create multiple n-atom-blocks
                    Block(Vertices[V] to Vertices[V + n])
                end
                display(Graphs);
            end
        end
    end
    else
        for n ← 0 to N − 4 2 do
            // Create multiple 2-atom-blocks
            Block(Vertices[n] to Vertices[n + 1])
        end
        Block(Vertices[N − 3] to Vertices[N − 1]);
        display(Figure);
    end
    Block(Vertices[0] to Vertices[N − 1]);
    display(Graph);
    return;
```

The algorithm has been summarized in **Algorithm 1**. The first pair of Greechie

diagrams are the disconnected diagrams. Given the value of N, the above algorithm proceeds by first checking if N is a prime number.

If N is not a prime number, the algorithm finds all divisors of N and for each divisor (let's say **n**) generates a bunch of individual blocks each of size **n**. Figures 2.1 and 2.2 are the outputs we get for $N = 15$. Since the divisors of 15 are 3 and 5, we get 3-atom and 5-atom blocks, respectively.

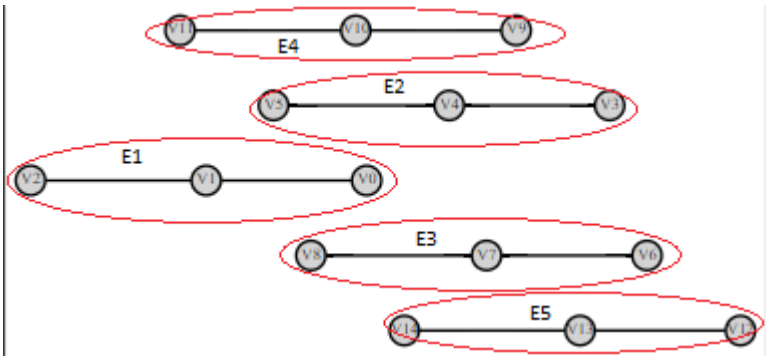

Figure 2.1: $N = 15$: Output for Algorithm 1: 3-atom blocks

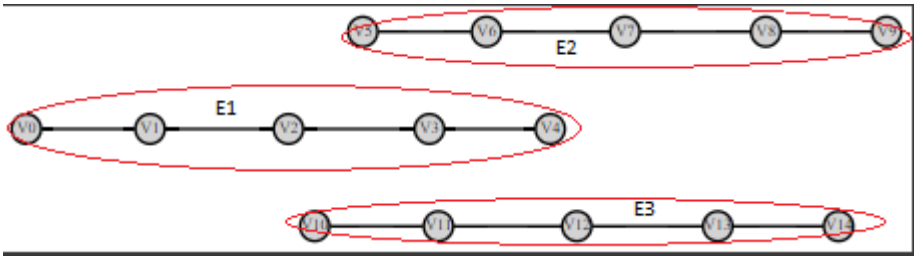

Figure 2.2: $N = 15$: Output for Algorithm 1: 5-atom blocks

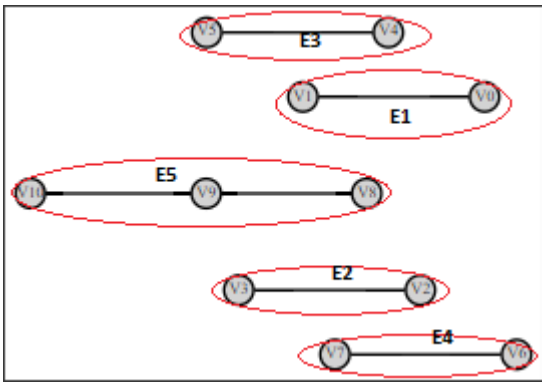

Figure 2.3: $N = 11$: Output for Algorithm 1: Four 2-atom blocks and one 3-atom block

If N is a prime number, we would get a single diagram with one 3-atom block and

the rest being 2-atom blocks, like in Figure 2.3. $N = 11$ gives four 2-atom blocks and one 3-atom block.

2.1.2 Connected Greechie diagrams without loops

Algorithm 2: Greechie Diagram Function 2

Data: $N \geq 5$

`// Create an array, Vertices:` V_0 `to` V_{N-1}

Function `Noloops_Connected_GDs(`*Vertices*`)`**:**

- **for** $i \leftarrow 3$ **to** $N-2$ 1 **do**
  - $Block(Vertices[0]$ to $Vertices[i-1])$;
  - $Block(Vertices[i]$ to $Vertices[N-1])$;
- **end**
- **for** $j \leftarrow 1$ **to** $i/2$ 1 **do**
  - **if** *N is even* **then**
    - Skip $j = i/2$ iteration;
  - **end**
  - `// Connect block2 with block1 at the jth position`
  - $display(Graph)$
- **end**
- **if** *N is odd* **then**
  - **for** $j \leftarrow 1$ **to** $N-1$ 2 **do**
    - $Block(Vertices[0]$ to $Vertices[j-1])$;
    - $c \leftarrow 0$;
    - **for** i **to** $N-1$ 2 **do**
      - **if** $c > j-1$ **then**
        - $c = c\%j$;
      - **end**
      - $Block(Vertices[c], Vertices[i], Vertices[i+1])$;
      - $c \leftarrow c+1$;
    - **end**
  - **end**
  - $display(Graph)$;
- **end**
- **return**;

The next category of Greechie diagrams is the well-connected ones, which follow **Algorithm 2**. First, The logic is to create block 1, which is of minimum size 3 and a maximum size of N-2. The remaining atoms are used to create another block, block 2. The next goal is to connect block 2 to block 1 at all possible atoms.

At the same time, the aim is also to eliminate isomorphisms. For this reason, only a part of the vertices in block 1 will be considered to connect block 2 with block 1.

If the number of vertices in block 1 is assumed to be *i*, then block 2 will intersect block 1 in 1 to i/2 positions. The rest of the connections would produce the same diagrams with a different combination of atoms. If N is even, then j=(i/2) is another redundant case, and thus we only go as far as (i/2)-1. An example of N=6 is in Figure 2.4 (blocks highlighted in **red**), with the blocks as follows:

- Figure 2.4 (a): $E_1 = \{V_0, V_1, V_2\}$, $E_2 = \{V_1, V_3, V_4, V_5\}$
- Figure 2.4 (b): $E_1 = \{V_0, V_1, V_2, V_3\}$, $E_2 = \{V_3, V_4, V_5\}$
- Figure 2.4 (c): $E_1 = \{V_0, V_1, V_2, V_3, V_4, V_5\}$

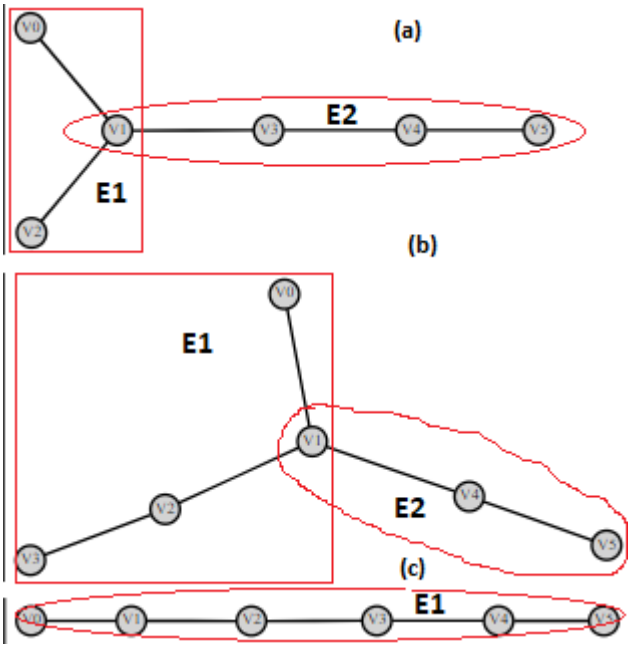

Figure 2.4: $N = 6$: Output by Algorithm 2

If Figure 2.4(b) is considered, then connecting V_4 to V_2 instead of V_1 would have produced its isomorphism. These are the possibilities that the algorithm avoids.

One more type of Greechie diagram this algorithm covers (given that 'N' is odd and at least 7) takes the form of a **star**. This can simply be formed using 'j' (j is odd) atoms to form the centre. Then the N-j atoms can all be connected to the centre two at a time.

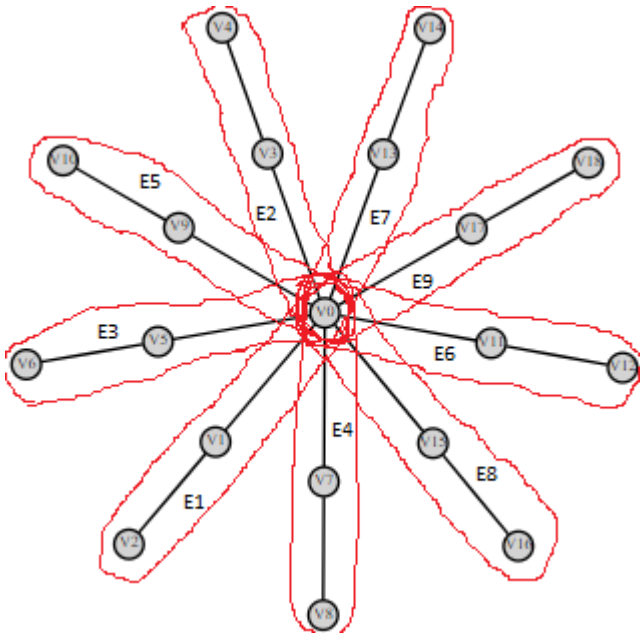

Figure 2.5: $N = 19$: One of the Outputs by Algorithm 2

Figures 2.5 and 2.6 are examples of this. For instance, Figure 2.6 has j=3 atoms as the centre (in **red**). The rest of the blocks are connected to it in cyclic order (in **green**).

2.1.3 Greechie Diagrams with connected loops

Algorithm 3: Greechie Diagram Function 3

Data: $N \geq 10, x = 8, y = 16$ and $N - y \leq x/2$

// Create an array, Vertices:V_0 to V_{N-1}

Function Connecting_Loops_Greechie_Diagrams($Vertices$):

    $Loop(Vertices[i], Vertices[i+x-1])$;
    $Loop(Vertices[i+x], Vertices[i+1+x])$;
    $c \leftarrow 0$;
    for $i \leftarrow 0$ **to** $x-1$ 1 **do**
        // Use the remaining vertices to connect the loops
        $Block(Vertices[c], Vertices[i], Vertices[c+x])$;
        $c+=2$;
    end
    $display(Graph)$;
    return;

From now, it is time to examine Greechie diagrams with loops. By definition, the minimum order of a loop is 4 and intersecting blocks have no more than one atom in

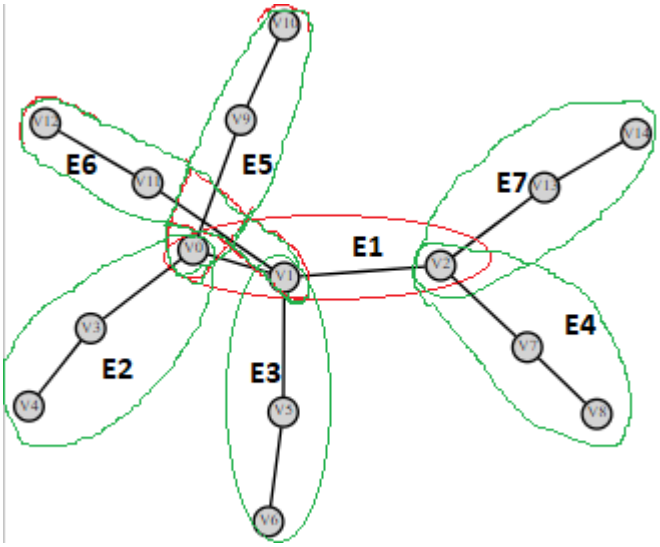

Figure 2.6: $N = 15$: One of the Outputs by Algorithm 2

common. Another strict convention that will be followed in this study when dealing with loops is that the size of each block is 3, or each block has exactly 3 atoms.

In this scenario, we will only consider connections between 2 loops of equal order; thus, the order of the loops will be adjusted accordingly. This ensures that no more than 2 loops are generated, and the rest of the atoms are used to connect these loops. These diagrams are generated by **Algorithm 3**.

Figure 2.7 is the diagram we get for N=19. Two loops: Loop1 = $\{V_0, V_1, ..., V_7, V_0\}$ of order 4, made of edges E1 to E4 (in **red**); Loop2 = $\{V_8, V_9, ..., V_{15}, V_8\}$ also of order 4 and made of blocks E5 to E8 (in **red**) are created and are connected using the other 3 atoms in the form of 3 blocks: E9, E10 and E11 (in **green**).

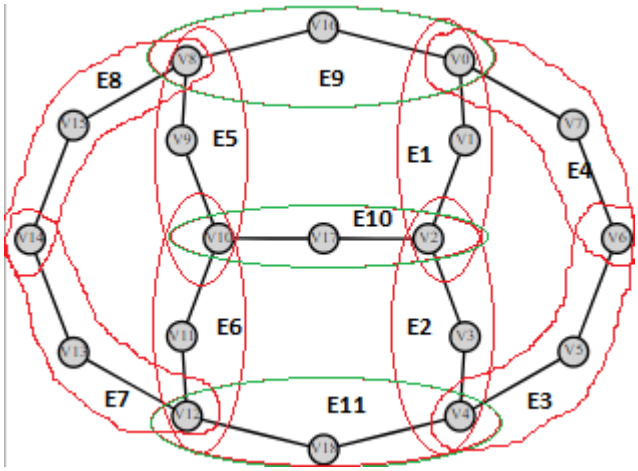

Figure 2.7: $N = 19$: Output for Algorithm 3

2.1.4 Greechie diagrams with loops with spider patterns

Algorithm 4: Greechie Diagram Function 4

Data: $N \geq 8$

// Create an array, Vertices:V_0 to V_{N-1}

Function Spider_Loops_GDs(*Vertices*)**:**

- **if** $N > 7$ *and* $N\%2 = 0$ **then**
  - **if** $N = 8$ **then**
    - $Loop(Vertices[0], Vertices[N-1])$;
  - **end**
  - **else**
    - $i \leftarrow 8$;
    - // $N - i > 2i$ and if not: $i+=2$ until condition satisfies.
    - **for** $x \leftarrow 0$ **to** $N\ 2$ **do**
      - $Loop(Vertices[0], Vertices[x-1])$;
      - $c \leftarrow 0$;
      - **for** $k \leftarrow x$ **to** $N\ 2$ **do**
        - $Block(Vertices[c], Vertices[k], Vertices[k+1])$;
        - $c \leftarrow c+1$;
      - **end**
      - $display(Graph)$
    - **end**
  - **end**
- **end**
- **return**;

The following Greechie diagrams have a single loop of minimum order 4 with connecting blocks. These follow **Algorithm 4**, and an example is in Figure 2.8. N=24, which can be considered $24 = Loop(8Atoms) + 16$. The first 8 atoms V_0 to V_7 form a loop, while the other 16 atoms are used to form 8 connecting blocks. The algebra is as follows:

- Loop = {E1, E2, E3, E4} (highlighted in **red**)
- Connecting Blocks: {E5, E6, E7, E8, E9, E10, E11, E12} (highlighted in **green**)

Apart from a loop of order 4, there are a few more possibilities. The loop can also be

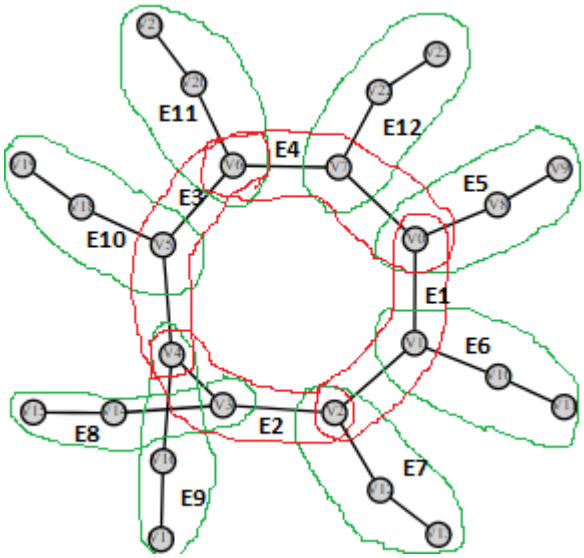

Figure 2.8: $N = 24$: One of the Outputs for Algorithm 4

of order 5, 6, 7, 8, 9, 10 and 11, respectively and will produce 7 more diagrams. This algorithm only works for even numbers ≥ 8.

2.1.5 Internally Connected Loops

This Greechie diagram will have a single loop with additional blocks connected internally. **Algorithm 5** is an extension of Algorithm 4 in many ways. It takes a minimum N value of 11 and will not work for specific values.

The first step would be determining the number of vertices to form a loop. Lines 2 to 10 accomplish this task. The simple idea here is to have an even number of atoms to form a loop (at least 8 to form a loop of order 4), and then the remaining number of atoms must be a multiple of 3.

- Mathematically N should be expressible in the form $2x+3y$ (where $x \geq 4$ and $y \geq 0$) and 2x would generate a loop of order x.

If the value of N is assumed to be 17, then the possibilities are $\{8,14\}$. This is because $17 = 2*(4)+3*(3)$ and $17 = 2*(7)+3*(1)$. The function would not work for values like 12. For simplicity, if we assume N=14 as in Figures 2.10, the possibilities are L = $\{8,14\}$ as $14 = 2*(4)+3*(2)$ and $14 = 2*(7)+3*(0)$. Figure 2.10(a): (L=14) simply generates a single loop {E1,...,E7} of order 7. Figure 2.10(b): (L=8) has one loop of order 4 formed with atoms V0 to V7 (in **red**); the remaining atoms form connecting blocks E5 to E8 (in **green**). Figure 2.9 is a similar example. **Algorithm 5** is as shown below:

Algorithm 5: Greechie Diagram Function 5

Data: $N \geq 11$

// Create an array, Vertices:V_0 to V_{N-1}

Function Complex_Loop_Structured_GDs(*Vertices*)**:**

- $L \leftarrow []$;
- **if** *N is even and* $N \geq 8$ **then**
  - $L.append(N)$;
- **end**
- **if** $N > 10$ **then**
  - $i \leftarrow N$;
  - **while** $i > 8$ **do**
    - **if** $(i-3)\%2 = 0$ *and* $i-3 \geq 8$ **then**
      - $L.append(i-3)$
    - **end**
    - $i \leftarrow i-3$;
  - **end**
  - **foreach** $x \in L$ *and* $L \neq \phi$ **do**
    - *Loop(0,x-1);*
    - $c \leftarrow 0$;
    - **for** $y \leftarrow x$ **to** $N-1$ 3 **do**
      - $Block(Vertices[c], Vertices[y], Vertices[y+1], Vertices[y+2])$;
      - **if** $c+4 > x-1$ **then**
        - $Block(Vertices[(c+4)\%x], Vertices[y+2])$;
        - **if** $c+2 > x-1$ **then**
          - $c \leftarrow (c+2)\%x$;
        - **end**
        - **else**
          - $c \leftarrow c+2$;
        - **end**
      - **end**
      - **else**
        - $Block(Vertices[c+4], Vertices[y+2])$;
        - $c \leftarrow c+2$;
      - **end**
    - **end**
    - $display(Graph)$
  - **end**
- **end**

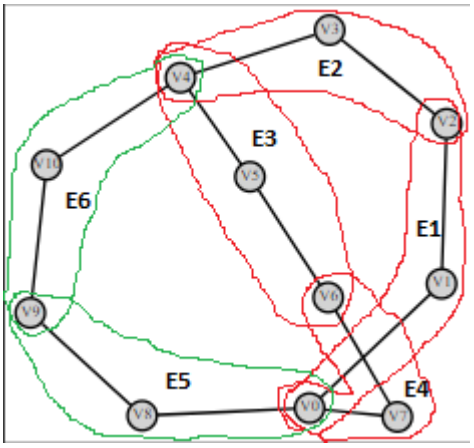

Figure 2.9: N=11: Output for Algorithm 5

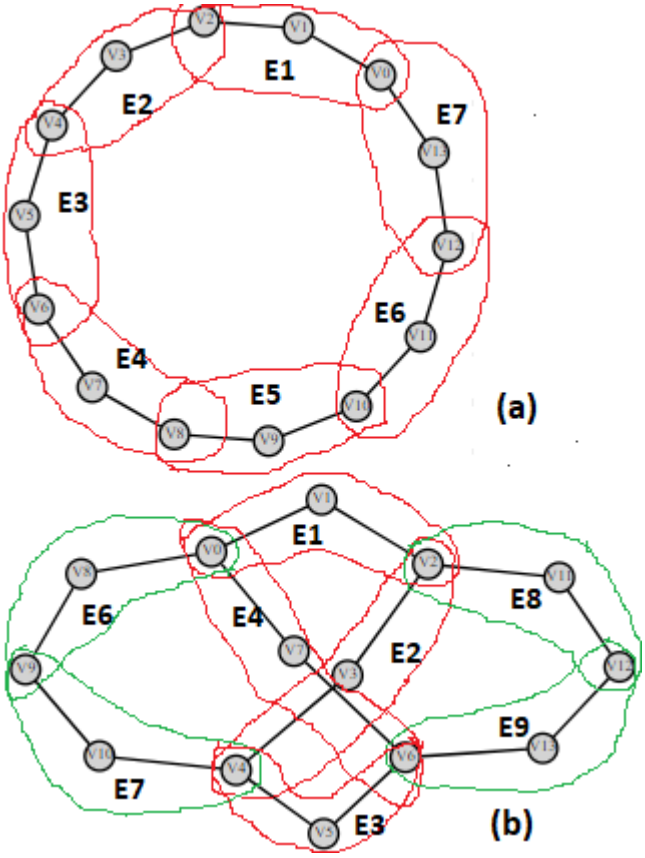

Figure 2.10: N=14: Output by Algorithm 5

2.1.6 Greechie Diagram Validation

Algorithm 6: Greechie Diagram Validation

Data: List L and $L \neq []$

```
// Create a list of edges with each edge represented in a set.
```

1 **Function** `Greechie_Diagram_Validation(`L`)`**:**
2 $N \leftarrow length(L), Flag = 0;$
`// If Flag=0 till the end: Valid Greechie diagram`
3 **for** $i \leftarrow 0$ **to** $N-1$ 1 **do**
4 **if** *L[i] is not a set* **then**
`// Property 1 violation, Flag=1 and break`
5 **end**
6 **else if** $V(L[i]) < 2$ **then**
`// Property 2 violation, Flag=1 and break`
7 **end**
8 **end**
9 **while** $i \in \{0,...,N-1\}, j \in \{i+1,...,N-1\}$ **do**
10 **if** $V(L[i]) = 2$ *and* $V(L[i] \cap L[j]) > 0$*)* **then**
`// Property 3 violation, Flag=1 and break`
11 **end**
12 **else if** $V(L[i] \cap L[j]) > 1$ **then**
`// Property 4 violation, Flag=1 and break`
13 **end**
14 i++,j++;
15 **end**
16 **while** $i \in \{0,...,N-1\}, j \in \{i+1,...,N-1\}, k \in \{j+1,...,N-1\}$ **do**
17 **if** $V(L[i] \cap L[j])$ = $V(L[j] \cap L[k])$ = $V(L[i] \cap L[k])$ = *1* **then**
`// Property 5 violation, Flag=1 and break`
18 **end**
19 i++,j++,k++;
20 **end**
21

Algorithm 6 can be used to validate Greechie diagrams. The input is a list of all blocks; each represented as a set. An example is as follows:

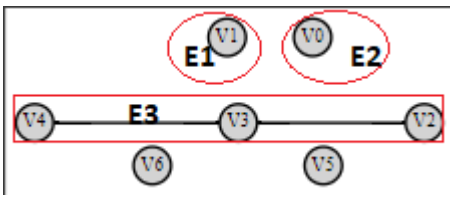

Figure 2.11: Property 1 violation example

- $L = [\{"V0","V1","V2"\},\{"V2","V3","V0"\},\{"V7"\}]$
- E1 = {"V0", "V1", "V2"}, E2 = {"V2", "V3", "V0"} and E3 = {"V7"}

Note: If an atom is not enclosed in a set, it is not associated with any block. And V(L[i]) in the algorithm refers to the number of atoms in block L[i]. Ex: $V(\{"V2","V3","V4"\}) = 3$.

The algorithm maintains a flag initially set to 0. If the diagram does not follow any of the 5 properties for Greechie diagrams (specified in Section 1.3.1), the flag will be set to 1, and the appropriate warning message will be printed. If it does follow all properties, then the flag will continue to be 0, and the output will be "Valid Greechie Diagram".

Figure 2.11: First, each element in the list will be checked to identify any atoms not enclosed in a set. For example, consider the list:

- L=[{V2,V3,V4},{V0},{V1},V5,V6]
- E1 = {V0}, E2 = {V1}, E3 = {V2,V3,V4}

It is noticeable that V5 and V6 are not enclosed in any sets. This means that they are not a part of any block. This violates Property 1 (checked by Line 4), as each atom must be a part of at least 1 block. And also, note that this algebra violates Property 2 (discussed next).

Figure 2.12: The next check is to Property 2, and a sample test case is:

- $L = [\{V0,V1,V2\},\{V2,V3,V4\},\{V5\},\{V6\}]$.
- E1= {V0, V1, V2}, E2 = {V2, V3, V4}, E3 = {V5}, E4 = {V6}

E3 and E4 clearly violate Property 2 (V(E3)=1 and V(E4)=1) as each block should have at least 2 atoms, which is checked in Line 6.

The next check is to Property 3, and a sample test case is:

- $L = [\{V3,V1\},\{V0,V1,V2\}$.

Figure 2.12: Property 2 violation example

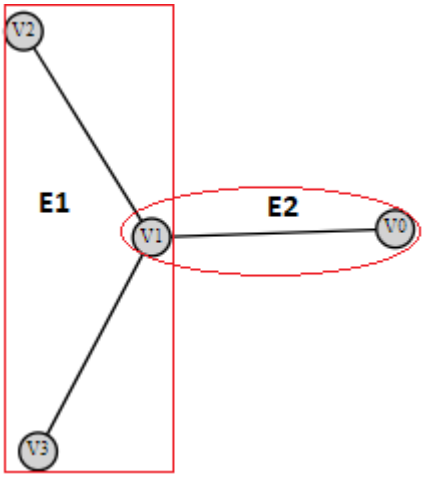

Figure 2.13: Property 3 violation example

- E1= {V3,V0}, E2 = {V0,V1,V2}

Figure 2.13: By Property 3, a block should at least have 3 atoms to participate in a connection. In the above example, block E1 has 2 atoms (V(E1)=2) but is connected to E2, thus violating Property 3. This is checked by Line 10.

The next check is to Property 4, and a sample test case is:

- $L = [\{V2, V3, V4\}, \{V4, V5, V6\}, \{V6, V7, V0\}, \{V6, V7, V1\}]$.
- E1= {V2,V3,V4}, E2 = {V4,V5,V6}, E3 = {V6,V7,V0}, E4 = {V6,V7,V1}

Figure 2.14: By Property 4, the blocks should not have more than 1 atom in common. In the above example, E3 and E4 have 2 atoms in common ($E3 \cap E4 = \{V6, V7\}$), thus violating Property 4. This is checked by Line 12.

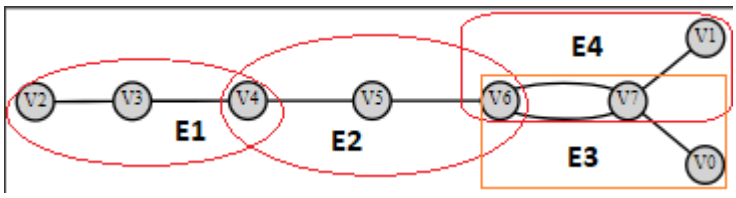

Figure 2.14: Property 4 violation example

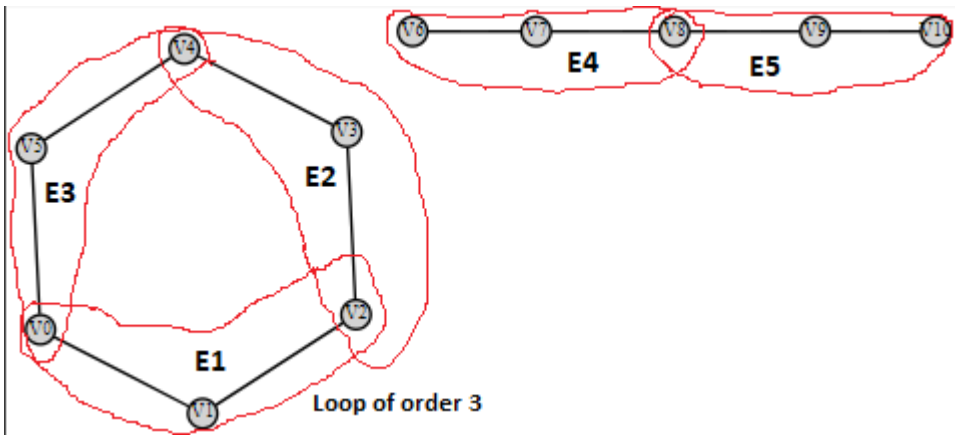

Figure 2.15: Property 5 violation example

The next check is to Property 5, and a sample test case is:

- $L = [\{V6, V7, V8\}, \{V8, V9, V10\}, \{V0, V1, V2\}, \{V2, V3, V4\}, \{V4, V5, V0\}]$.
- E1= {V6,V7,V8}, E2 = {V8,V9,V10}, E3 = {V0,V1,V2}, E4 = {V2,V3,V4}, E5 = {V4,V5,V0}

Figure 2.15: By Property 5, the minimum order of a loop allowed is 4. In the above example, $E3 \Rightarrow E4 \Rightarrow E5 \Rightarrow E3$ is a loop of order 3, thus violating Property 5. This is checked by Lines 16 and 17.

2.2 Orthoalgebra

2.2.1 Orthoalgebra Validation

The next section after Greechie diagram generation and validation is Orthoalgebra Validation. Given an Orthoalgebra, the validity of it can be computed with the help of 3 arrays:

- CMP: This is a symmetric matrix ($A[i, j] = A[j, i], \forall i, j$) of order 'N'. This matrix stores the list of compatibilities between elements of an Orthoalgebra. If i and j are compatible, then CMP[i,j]=1 and 0 otherwise.
- NELE: If the number of elements in an Orthoalgebra is N, then $NELE = [N-1, N-2, ..., 0]$.
- OAPLUS: This is also a symmetric matrix computed using CMP. If CMP[i,j]=1 then OAPLUS[i,j]= $i \oplus j$ and 0 otherwise.

Algorithm 7: Orthoalgebra Validation

Data: NELE, CMP, OAPLUS

```
1  Function Ortho_Algebra_Validation(NELE, CMP, OAPLUS):
2      for a, b ← 0 to N − 1 1 do
3          if CMP[a,b] = 1 and CMP[b,a] = 0 then
4              return False
5          end
6      end
7      for a, b, c ← 0 to N − 1 1 do
8          if CMP[a,b] = 1 and CMP[a,OAPLUS[b,c]]=1 then
9              if CMP[b,c] ≠ 1 or CMP[a,OAPLUS[b,c]] ≠ 1 then
10                 return False
11             end
12             if OPLUS[OPLUS[a,b],c] ≠ OPLUS[a,OPLUS[b,c]] then
13                 return False
14             end
15         end
16     end
17     for a ← 0 to N − 1 1 do
18         NC ← 0;
19         for b ← 0 to N − 1 1 do
20             if CMP[a,b] = 1 and OAPLUS[a,b] = N − 1 then
21                 NC ← NC + 1;
22                 if NC > 1 and NELE[a] ≠ b then
23                     return False
24                 end
25             end
26         end
27     end
28     for a ← 0 to N − 1 1 do
29         if a ≠ 0 and CMP[a,a] = 1 then
30             return False
31         end
32     end
```

Algorithm 7 checks a given orthoalgebra against the 4 properties specified in Section 1.1.2, taking the above parameters as input. Programmatically, the following analogies will help to understand the algorithm easier:

1. $a \perp b \Rightarrow CMP[a,b] = 1$
2. $a \oplus b \Rightarrow OAPLUS[a,b]$
3. $a \perp (b \oplus c) \Rightarrow CMP[a, OAPLUS[b,c]] = 1$

The first property is commutativity (Lines 2 to 6). The commutative law states that if $a \perp b$, then $a \perp b$. Programmatically, if CMP[a,b]=1 then CMP[b,a]=1.

The second property is associativity (Lines 7 to 15). The associative law states that if $b \perp c$ and $a \perp (b \oplus c)$ then, $a \perp b$, $a \oplus b) \perp c$, $a \oplus (b \oplus c) = (a \oplus b) \oplus c)$. Programmatically, if CMP[a,b] = 1 and CMP[a,OAPLUS[b,c]]=1 then,

1. $CMP[b,c] = 1$
2. $CMP[a, OAPLUS[b,c]] = 1$
3. $OPLUS[OPLUS[a,b],c] = OPLUS[a, OPLUS[b,c]]$

Similarly, the third property, orthocomplementation (Lines 17 to 27), and the fourth property, consistency (Lines 28 to 32), are checked. "False" is printed otherwise if any of the properties fail to meet.

2.2.2 Hasse Diagram for Valid Orthoalgebra

An example of a valid orthoalgebra would be: CMP = $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ which gives $NELE = [3,2,1,0]$ and OAPLUS = $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 0 \\ 2 & 3 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$. The Hasse Diagram (Figure 2.16) is generated by **Algorithm 8**. It simply creates an edge between every pair of compatible elements ($CMP[j,k] = 1$) using the OAPLUS matrix(Line 7). At the same time, duplicate edges are avoided by ensuring $j \neq OAPLUS[j,k]$. The edges are generated in the order: $(0) \rightarrow (1)$, $(0) \rightarrow (2)$, $(0) \rightarrow (3)$, $(1) \rightarrow (3)$ and $(2) \rightarrow (3)$.

Algorithm 8: Hasse Diagram Generation

Data: NELE, CMP, OAPLUS

```
1  Function Hasse_Diagram_OA(NELE, CMP, OAPLUS):
2      N ← length(NELE);
3      for j ← 0 to N − 1 1 do
4          for k ← 0 to N − 1 1 do
5              if CMP[j,k]=1 then
6                  if j ≠ OAPLUS[j,k] then
7                      Edge(Vertices[j],Vertices[OAPLUS[j,k]]);
8                  end
9              end
10         end
11     end
12     return graph_obj
```

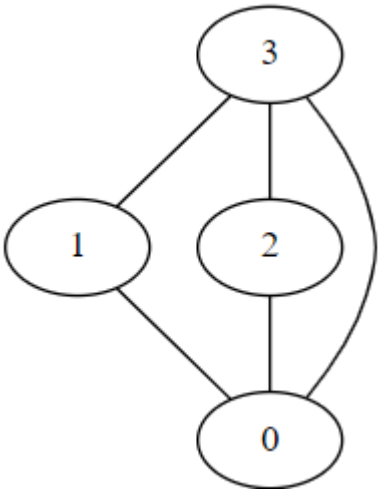

Figure 2.16: Hasse Diagram for the specified Orthoalgebra

Chapter 3

Conclusions

3.1 Discussion

The following factors influence the time complexity of Greechie diagrams:

- Choice of programming language: Interpreted languages like Python take longer than compiled languages like C/C++.
- Number of atoms and thus the number of blocks: Time complexity increases exponentially as these parameters increase.
- Manual Generation vs Automated Generation: Manual generation is much faster.
- Number of Greechie diagrams possible: The more diagrams, the greater the time.
- Certain compound operations precomputed, and usage of heuristics can speed up the code[12].

The code generates a variety of Greechie Diagrams. Given that there are N-atoms, the emphasis has been placed on only sticking to 3-atom blocks, given that the diagram has loops which is generally the case in most research works. Even 2-atom blocks are generally omitted. Few studies deal with a 4-vertex hyperedge(like in [10, 14]).

The diagrams without loops would have blocks of sizes 2 to N. The time taken to generate all of the Greechie Diagrams (as in Algorithms 1 to 5) and the number of diagrams generated are summarized in Table 3.1.

Table 3.2 generalizes Table 3.1 and provides the **Big-oh** time complexity for the algorithms. The table shows that the Greechie diagram generation has an overall time complexity of $O(3N^2 + 2N)$, which can be generalized as $O(N^2)$.

The limitations of this project are as follows:

- The programming language of choice was Python, the most commonly used and easily understandable language for researchers of any discipline. However, it is expected to work slower than the same Algorithm coded in C or C++.
- Another primary emphasis in this project is not to generate isomorphisms. Isomorphisms could be redundant diagrams that add up additional time. Although the code used for this experiment has been designed with a lot of intuition and care to eliminate such cases (such as eliminating iterations that produce diagrams that are mirror images of each other), it does not provide a way to check isomorphisms.
- The diagrams containing loops are generated by assuming that each block has exactly 3 atoms (Each hyperedge has exactly 3 vertices). The other possibilities are neglected.
- There are numerous possibilities for valid Greechie diagrams without loops, and this project only covers a portion of these possibilities.
- There are much more complex Greechie diagrams that are bigger and more convoluted, and this project does not cover those cases, given the time constraint. An example of such a diagram of order (36, 36) is in [12].

3.2 Conclusion and Future Direction

Greechie diagrams do not follow a unique algorithm. As the number of vertices increases, the possible number of Greechie diagrams increases exponentially, and they do not necessarily follow a particular pattern.

Although this thesis provides 5 algorithms that generate valid Greechie diagrams, most of the research in this field generally uses Manual Generation alongside automated generation following a certain Pseudocode.

However, the future direction for the portion of diagrams following automated generation would be to survey such algorithms and discuss all of them in an article or a short book. Since Python is one of the emerging languages, a module dedicated to Greechie diagrams and Orthoalgebras would be a considerable advantage for upcoming Mathematical researchers.

Most of the algorithms specified provided minimal detail, and there is no say on the maximum possible number of atoms or blocks, as the expectation is that they can be set to ∞.

Atoms	Time to exec	Number of Diagrams
2	26.4ms	1
3	72.9ms	1
4	51.4ms	2
5	129ms	3
6	188ms	5
7	176ms	8
8	270ms	10
9	446s	14
10	425ms	17
11	822ms	23
12	684s	28
13	876ms	33
14	1.04s	38
15	1.31s	46
16	1.87s	53
17	1.82s	61
18	1.93s	69
19	2.96s	76
20	2.95s	87
21	3.39s	96
22	3.83s	99
23	4.16s	116
24	5.07s	129
25	4.53s	138

Table 3.1: Time to execution and the number of diagrams by the number of atoms. Time to execution was measured using the $\%load\_extautotime$ utility in the **ipython-autotime** module.

Algorithm	**Big-oh**
1	$O(N^2)$
2	$O(N^2)$
3	$O(N)$
4	$O(N^2)$
5	$O(N)$
Overall:	$O(3N^2+2N)$

Table 3.2: Big-Oh time complexity of Greechie diagram generation where N is the number of atoms: $O(3N^2+2N) \Rightarrow O(N^2)$

Greechie diagrams represent ortho-algebras and orthomodular lattices that comprise a significant portion of the theory of Quantum Logic. However, the amount of resources that cover the advantages, disadvantages, and applications of Greechie diagrams and the different ways to represent an Orthoalgebra apart from Greechie diagrams are extremely limited. Thus a lot of survey articles are required to keep researchers updated.

Bibliography

[1] Numpy. https://numpy.org/, 2005.

[2] Are these 2 graphs isomorphic? https://math.stackexchange.com/questions/393416/are-these-2-graphs-isomorphic, 2013.

[3] Discrete mathemathics—hasse diagrams. https://www.geeksforgeeks.org/discrete-mathematics-hasse-diagrams/, 2022.

[4] graphviz: User guide. https://graphviz.readthedocs.io/en/stable/manual.html, 2023.

[5] Hypergraph. https://en.wikipedia.org/wiki/Hypergraph, 2023.

[6] PD Finch. On orthomodular posets. *Journal of the Australian Mathematical Society*, 11(1):57–62, 1970.

[7] Mustafa Mohammadi Garasuie, Mahmood Shabankhah, and Ali Kamandi. Improving hypergraph attention and hypergraph convolution networks. In *2020 11th International Conference on Information and Knowledge Technology (IKT)*, pages 67–72. IEEE, 2020.

[8] Richard J Greechie. Orthomodular lattices admitting no states. *Journal of Combinatorial Theory, Series A*, 10(2):119–132, 1971.

[9] Sebastian Horvat and Iulian Danut Toader. Quantum logic and meaning. *arXiv preprint arXiv:2304.08450*, 2023.

[10] Gudrun Kalmbach. *Orthomodular lattices*. Number 18. Academic press, 1983.

[11] Ying Liu, Jiabin Yuan, Bojia Duan, and Dan Li. Quantum walks on regular uniform hypergraphs. *Scientific reports*, 8(1):9548, 2018.

[12] Brendan D McKay, Norman D Megill, and Mladen Pavičic. Algorithms for greechie diagrams. *International Journal of Theoretical Physics*, 39:2381–2406, 2000.

[13] Xavier Ouvrard. Hypergraphs: an introduction and review. *arXiv preprint arXiv:2002.05014*, 2020.

[14] Pavel Pták and Sylvia Pulmannová. Orthomodular structures as quantum logics. *INIS-mf14676*, page 27, 1991.

[15] Karl Svozil. *Quantum logic*. Springer Science & Business Media, 1998.

[16] Karl Svozil and Josef Tkadlec. Greechie diagrams, nonexistence of measures in quantum logics, and kochen–specker-type constructions. *Journal of Mathematical Physics*, 37(11):5380–5401, 1996.

[17] Voráček Václav. Graph theory and quantum structures. `https://dspace.cvut.cz/bitstream/handle/10467/82343/F3-BP-2019-Voracek-Vaclav-Thesis.pdf`, 2019.

[18] Alexander Wilce. Topological orthoalgebras. *arXiv preprint math/0301072*, 2003.

Appendix A

Implementation

A.1 Code

The following link has the working code: `https://github.com/YeshwanthZ/Greechie_Diagrams`