AMSI SUMMER RESEARCH SCHOLARSHIPS 2023–24

SET YOUR SIGHTS ON RESEARCH THIS SUMMER

Brauer Monoid Factorisations using Machine Learning Techniques

Saakshi Singh

Supervised by Assoc. Prof. Oliver Obst and Matthias Fresacher

Western Sydney University

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

Contents

1 Abstract

This project explores the factorisation of Brauer monoids, a unique class of algebraic structures with deep connections to representation theory and finite group representation. The factorisation process is traditionally deterministic, however, this project aims to investigate the feasibility of utilising machine learning techniques, particularly reinforcement learning, to enhance the factorisation process. Despite no inherent superiority of machine learning over deterministic approaches, the project seeks to explore its potential benefits. The initial phase of the project involved building the theoretical framework for Brauer diagram factorisation. Moving forward, the next steps include implementing machine learning techniques into the existing framework to improve the factorisation process.

2 Introduction

The Brauer monoid, B_n, is a diagram algebra introduced by Richard Brauer in 1937 [1], representing a class of algebraic structures closely related to symmetric groups and braid groups. Its relevance spans various mathematical disciplines, from representation theory to combinatorics, offering rich avenues for exploration. This project aims to develop an algorithm for decomposing Brauer diagrams, which are elements of B_n, by utilising machine learning techniques to factorise and gain deeper insights into the structure and composition of these monoids.

In this report, we will introduce the notation and background of Brauer monoids, including their definitions, properties, and significance to mathematics. We will then discuss the presentation of these monoids and explore the factorisation of Brauer diagrams. This includes studying the properties relevant to diagram multiplication, laying the groundwork for our factorisation algorithm. While our immediate focus lies in algorithm development, our ultimate objective is to integrate machine learning techniques, particularly reinforcement learning, to enhance the factorisation process. The report presents our progress towards this goal, highlighting the methodologies employed and discussing potential directions for advancement.

Moving forward, our next steps include implementing machine learning techniques into the existing framework to improve the factorisation process.

Related Resource: For a more detailed and comprehensive outline of the theoretical framework of the Brauer monoids, see the paper by Kudryavtseva and Mazorchuk [4].

3 Statement of Authorship

Saakshi Singh developed all of the code used to generate the studied factorisations for the Brauer monoid (written in GAP [3]), and produced all figures, results and interpretations written in this report. Associate Professor Oliver Obst and Matthias Fresacher supervised the project, provided input on the project direction and outcomes achieved, and proofread this report.

4 Notation

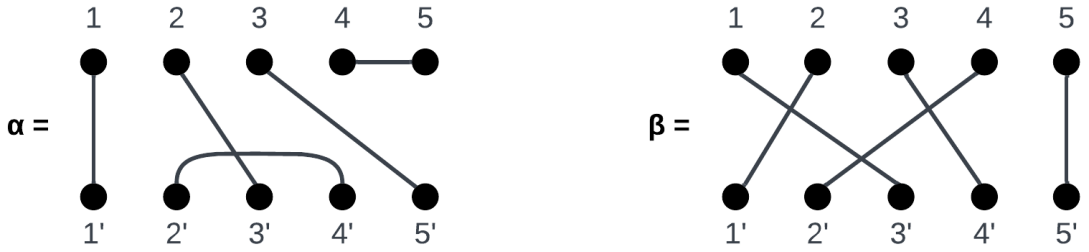
4.1 Brauer Monoid

Let n be a positive integer, and define the Brauer monoid, B_n, as the set of all set partitions of $[n] \cup [n]' = \{1, 2, \ldots, n\} \cup \{1', 2', \ldots, n'\}$. For example, here are two elements of B_5:

$$\alpha = \{\{1, 1'\}, \{2, 3'\}, \{3, 5'\}, \{4, 5\}, \{2', 4'\}\},$$
$$\beta = \{\{1, 3'\}, \{2, 1'\}, \{3, 4'\}, \{4, 2'\}, \{5, 5'\}\}.$$

An element of B_n can be uniquely represented by a graph on the vertex set $[n] \cup [n]'$, where each distinct pair of vertices is connected by a single edge, denoted as $e = (x, y)$ for $x, y \in [n] \cup [n]'$. These edges are categorised as either *transversal* or *non-transversal*. Transversal edges connect vertices from different rows, without loss of generality, $x \in [n]$ and $y \in [n']$, while non-transversal edges connect vertices within the same row, such that $x, y \in [n]$ or $x, y \in [n']$. The number of transversal edges in a given diagram is, by definition, its rank p.

We typically identify $\alpha \in B_n$ with its corresponding graph. When drawing such a graph, the vertices $1, 2, \ldots, n$ are arranged in a horizontal line, numbered from left to right, with vertex i' situated directly below i for each $i \in [n]$, and the edges are always drawn within the bounds of the rectangle formed by the vertices [2]. A graph drawn in this manner is called a **Brauer n-diagram**, or simply a **Brauer diagram** if n is implied by the context. The degree of the monoid, n, indicates the number of vertices in each row. So, with $\alpha, \beta \in B_5$ as above, we have:



The multiplication of elements in a Brauer monoid, sometimes referred to as composition or concatenation, denoted by $\bullet$, combines two diagrams α and β, to form their product $\alpha \bullet \beta$. This process involving stacking α on top of β, and aligning the bottom row of α with the top row of β. This process involves adding vertical edges between the two diagrams, while eliminating the inner row of vertices and any internal loops (collection of edges not connected to the top or bottom rows). The resulting diagram is then adjusted by compressing the two rows back to their original size and drawing clean edges where they exist. Figure 1 illustrates this process.

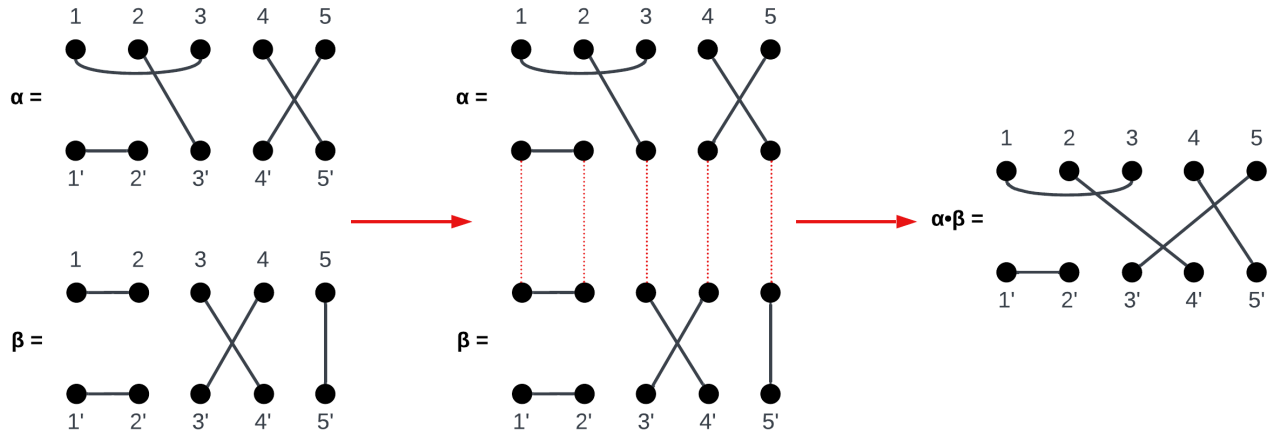
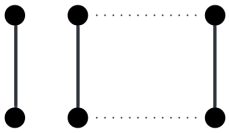


Figure 1: Process of multiplying Brauer diagrams.

The operation $\bullet$ is associative [6] and there exists an identity diagram, specifically all vertical edges between the two rows of vertices. This is denoted by:

$$1 = \{\{1,1'\},\{2,2'\},\cdots,\{n,n'\}\} = \quad \in B_n$$



Moreover, when multiplying Brauer diagrams, the rank can only decrease or stay the same. Factorisations within B_n are non-commutative, so the order of multiplication affects the result.

4.2 Presentation

In general, a monoid M is defined by its set of generators S and a set of relations R that the generators satisfy. These relations define how the generators interact with each other, ensuring that the resulting products adhere to certain rules or properties. Any element x of the monoid can be expressed as a product of the generators, possibly with repetitions.

The presentation we are using specifically pertains to the Brauer monoid B_n. This is defined by a generating set, $G = T \cup U$, where T and U are sets defined as follows:

- $T = \{T_i \mid i \in \{1,2,\ldots,n\}\}$, where each T_i represents a diagram in B_n
- $U = \{U_i \mid i \in \{1,2,\ldots,n'\}\}$, where each U_i represents a diagram in B_n

Each T_i and U_i in the sets T and U respectively, represents a specific type of diagram in B_n. More formally, we define T_i and U_i as follows:

- $T_i = \{\{1,1'\},\{2,2'\},\cdots,\{i,i'+1\},\{i+1,i'\},\cdots,\{n,n'\}\}$
- $U_i = \{\{1,1'\},\{2,2'\},\cdots,\{i,i+1\},\{i',i'+1\},\cdots,\{n,n'\}\}$

Here, i ranges from 1 to $n-1$, and it denotes the index of the generating set. These definitions capture the essential structure of the diagrams in B_n, as shown by Figure 2 and Figure 3, and serve as the basis for defining the relations that govern their composition.

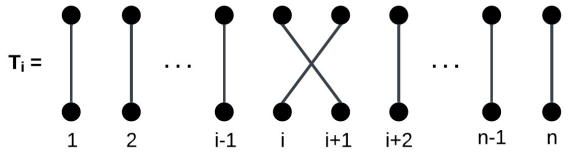


Figure 2: Diagram for T_i.

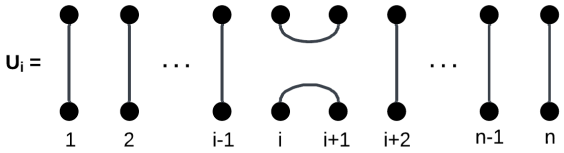


Figure 3: Diagram for U_i.

These generators satisfy a set of relations, as per Table 1. These relations define constraints or rules that the generators must adhere to within B_n. Essentially, they govern how the generators interact with each other during multiplication, ensuring that the resulting elements remain within the defined structure of B_n. We classify these relations three types [5]:

- *Delete relations*: decrease the number of generators to be composed (remove redundancy)
- *Braid relations*: these resemble the braid relation in the symmetric group, S_n
- *Swap relations*: allow generators to commute without changing the resulting diagram

Delete

1.	$T_i \bullet T_i$	$=$	I_n
2.	$U_i \bullet U_i$	$=$	U_i
3.	$T_i \bullet U_i$	$=$	U_i
4.	$U_i \bullet T_i$	$=$	U_i
5.	$U_i \bullet U_j \bullet U_i$	$=$	U_i for $\lvert i-j\rvert = 1$
6.	$T_i \bullet U_j \bullet U_i$	$=$	$T_j \bullet U_i$ for $\lvert i-j\rvert = 1$
7.	$U_i \bullet U_j \bullet T_i$	$=$	$U_i \bullet T_j$ for $\lvert i-j\rvert = 1$

Braid

8.	$T_i \bullet T_j \bullet T_i$	$=$	$T_j \bullet T_i \bullet T_j$ for $\lvert i-j\rvert = 1$

Swap

9.	$T_i \bullet T_j$	$=$	$T_j \bullet T_i$ for $\lvert i-j\rvert > 1$
10.	$T_i \bullet U_j$	$=$	$U_j \bullet T_i$ for $\lvert i-j\rvert > 1$
11.	$U_i \bullet T_j$	$=$	$T_j \bullet U_i$ for $\lvert i-j\rvert > 1$
12.	$U_i \bullet U_j$	$=$	$U_j \bullet U_i$ for $\lvert i-j\rvert > 1$

Table 1: Relations for the generators, where $i, j \in \{1, 2, \ldots, n-1\}$.

The presentation of B_n is denoted by $B_n = \langle T_1, \ldots, T_{n-1}, U_1, \ldots, U_{n-1} | R \rangle$, where T and U are the generators of B_n and R denotes the relations as per Table 1. These form the basis for expressing elements of B_n as products of these generators.

One way to understand these relations is through a visual representation, as shown by Figure 4 and Figure 5. These demonstrate how certain operations, such as deletion and simplification apply to the diagrams, affecting their composition within the monoid. However, due to the complexity of these operations, a visual depiction is more intuitive than a verbal explanation.

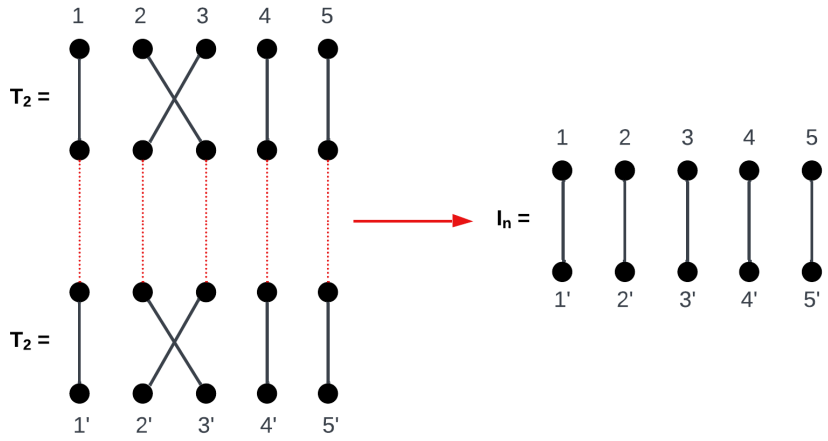


Figure 4: Multiplying T_i generator by itself.

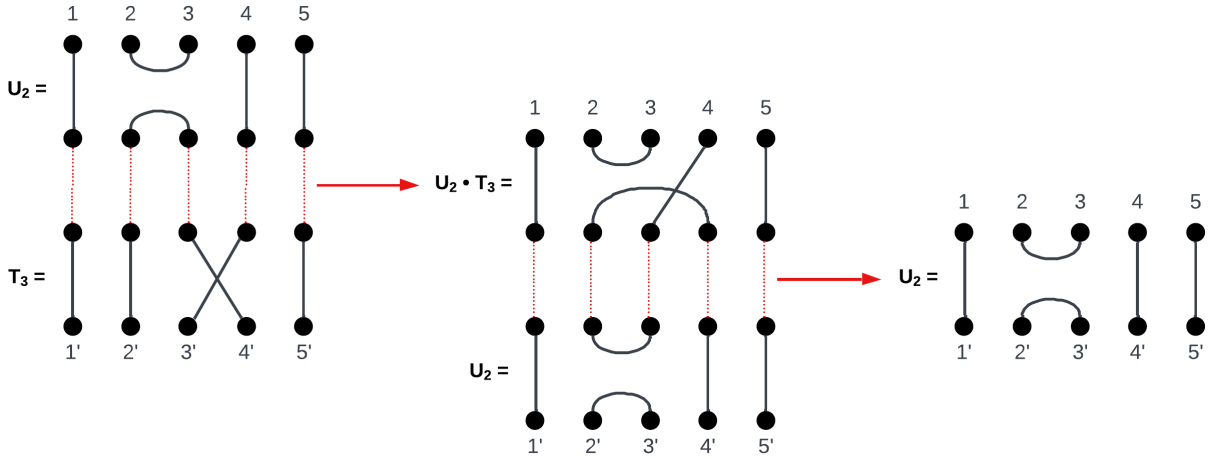


Figure 5: Simplification example.

In summary, the presentation of B_n provides a framework for understanding the structure and composition of the monoid, offering insights into the behavior of its generators and their compositions. This understanding is crucial for developing algorithms, such as those involving machine learning, to efficiently factorise Brauer diagrams and explore the properties of B_n.

4.3 Factorisation

The factorisation process in B_n involves breaking down a given diagram into a product of simpler diagrams, typically using the generators and relations defined in the presentation of B_n. This process is akin to prime factorisation in integers, where each integer can be uniquely expressed as a product of prime numbers. In the case of B_n, the goal is to express each diagram as a product of "prime" diagrams, which in this context are the generators T_i and U_i.

We employ a deterministic algorithm to perform the factorisation process, utilising the generators and relations defined in the presentation of B_n to decompose diagrams into their prime components. This process in B_n is non-commutative, meaning the order of multiplication matters. This is a key characteristic that distinguishes it from other algebraic structures. The relations among the generators play a crucial role in determining how diagrams can be simplified or decomposed. For example, the delete relations in Table 1 allow us to eliminate certain generators based on specific patterns, reducing the complexity of the diagram.

Additionally, the factorisation of Brauer diagrams is not always unique, implying that a given diagram can often be expressed as a product of generators in multiple ways. This non-uniqueness adds another layer of complexity to the factorisation process, requiring careful consideration of the relations and their implications.

5 Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning technique inspired by behavioural psychology [7], where an agent learns to make decisions by interacting with an environment. The goal of the agent is to maximise a cumulative reward signal over time.

In RL, the agent learns through trial and error, receiving feedback in the form of rewards or penalties for its actions. The agent takes actions in the environment, which transitions it to a new state, and receives a reward based on the action take and the new state. Over time, the agent learns which actions lead to the highest rewards and adjusts its strategy accordingly.

The main elements of an RL system are:

1. **Agent**: learner or decision-maker that interacts with the environment
2. **Environment**: external system with which the agent interacts
3. **Actions**: choices available to the agent
4. **States**: conditions or situations the agent can find itself in
5. **Rewards**: feedback from the environment that guides the agent's learning process

The agent's goal is to learn a policy, which is a mapping from states to actions, that maximises the expected cumulative reward. This is achieved through exploration, where the agent tries different actions to learn about the environment, and exploitation, where the agent uses its current knowledge to select actions that are likely to lead to high rewards [7].

6 Algorithm

In our algorithm for factorising diagrams in the Brauer monoid, we employ a deterministic approach that aims to integrate machine learning, specifically reinforcement learning. It involves decomposing a given diagram into a product of simpler diagrams, i.e., generators.

The basic idea is to loop from $k = 1$ to a maximum value, attempting to factorise a given diagram a using the generators and relations defined in the presentation of B_n. At each iteration, we pick a generator g_{i_k} from our generating set G and add it to the current attempt b_k. We hope that the factorisation is correct and that $b_k = a$ at some point. However, this process can fail if the rank of a exceeds the rank of b, in which case we stop the loop.

An example of rank failure can be seen in Figure 6, where the input diagram α has rank 5 and the resulting diagram β_3 has rank 3, causing the factorisation process to fail.

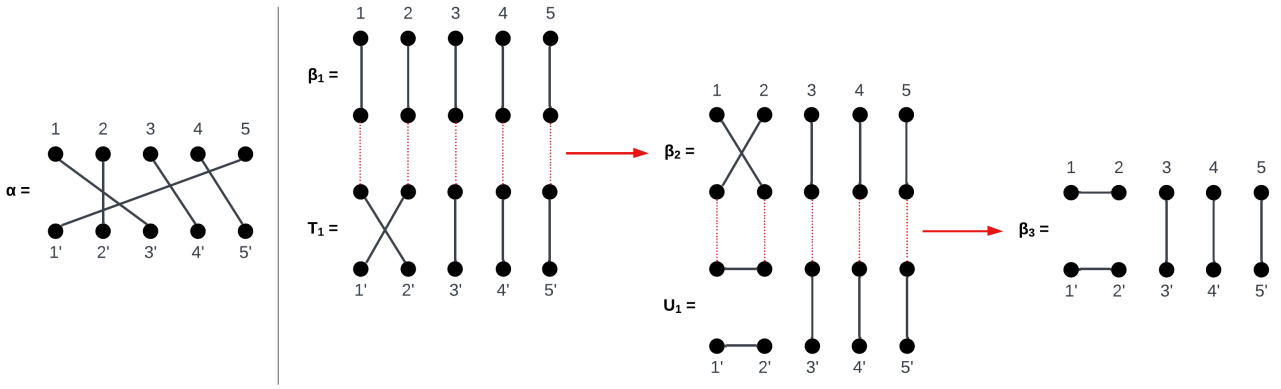


Figure 6: Example of rank failure.

The choice of g_{i_k} is crucial as it directly influences the resulting factorisation. To decide which generator g_{i_k} to pick at each step, we assign weights to the generators and randomly select one according to these weights. RL is used to choose these weights, allowing the algorithm to learn and adapt its selection strategy over time based on the rewards received for different choices of g_{i_k}. Here, the algorithm learns the optimal weights through exploration and exploitation;

- **Exploration** refers to the algorithm's strategy of testing out different generators to understand their effectiveness in achieving a successful factorisation.
- Alternatively, **exploitation** means refers to the algorithm using its current knowledge to make decisions. It involves leveraging RL to bias the selection towards generators that have previously contributed to successful factorisations.

In our algorithm, a general generator is denoted as g_k, but for the purpose of our factorisation process, we use g_{i_k} to indicate the specific generator selected at the k-th step.

Here is the pseudocode for our factorisation algorithm:

Algorithm Factorisation algorithm

Require: $n > 0$

function FACTORISATIONPROCESS(a)

$b_k \leftarrow 1$ ▷ identity element in monoid

generatorsUsed $\leftarrow$ empty list

for $k = 1$ to some max value **do**

if $b_k = a$ **then** ▷ a is the input diagram

break ▷ stop loop if factorisation is found

end if

if $rank(a) > rank(b)$ **then**

break ▷ hard stopping for loops

else

$g_{i_k} \leftarrow$ Random(G) ▷ pick a random generator from the set G

$b_k \leftarrow b_{k-1} \times g_{i_k}$

Add g_{i_k} to generatorsUsed list

end if

end for

return [b_k, generatorsUsed] ▷ return resulting diagram and list of generators used

end function

Note, our **generatorsUsed** will return some result in the form $a = g_{i_1} \times g_{i_2} \times g_{i_3} \times \cdots$

This algorithm aims to iteratively factorise a given diagram a into a product of generators, by employing deterministic rules and machine learning-based weight selection for the generators. This deterministic approach is shown visually in Figure 7 for ease of understanding.

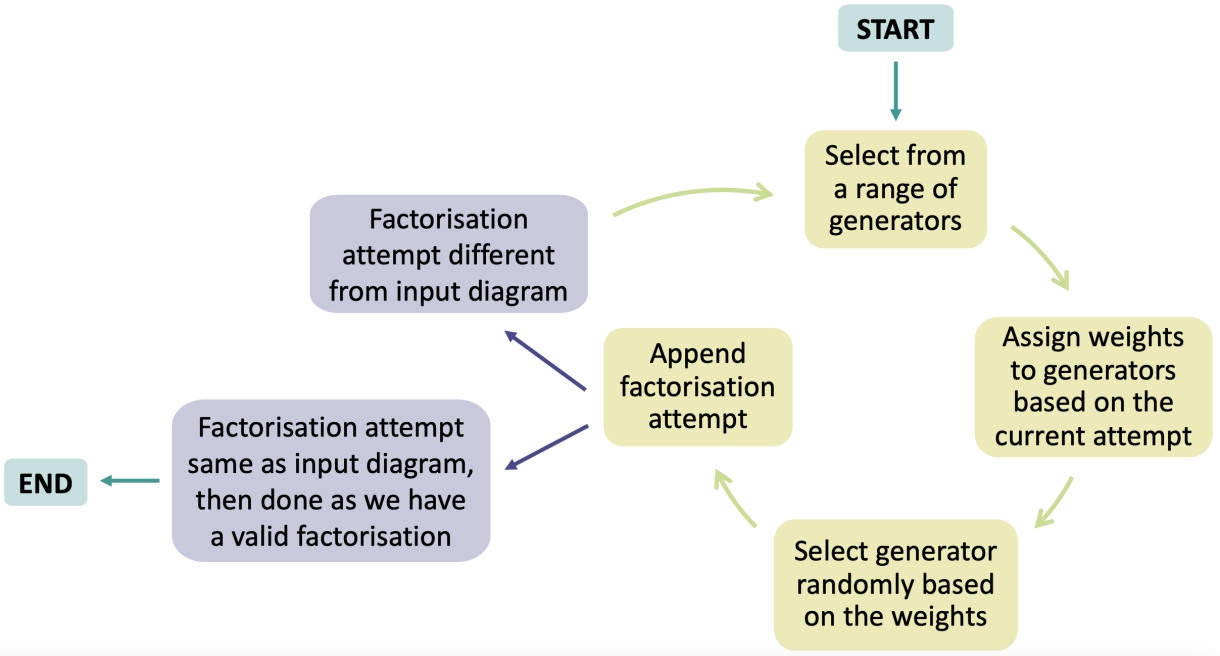


Figure 7: Reinforcement learning process.

7 Code and Implementation

In our implementation, we used the GAP system, a software package for computational discrete algebra [3], to faciliate our factorisation algorithm. GAP offers a rich set of functions for group and monoid computations, making it well-suited for our project.

We first defined a function that generates the set of generators G for the Brauer monoid B_n. This function returns a list of generators, which are diagrams in B_n. It achieves this by iterating through the values of i to create the T and U generators, as per Code 1.

Note, given prime, ′ is used as a delimiter for character constants in GAP [3], we modified the notation, replacing primes with negative integers to represent the bottom row. This was

employed for all calculations to ensure consistency and make sure it can be easily understood.

```
## Function to create the generators for Bn given n
BrauerGenerators := function(n)
    local generators, Ti, Ui, i, j;
    generators := [];  # initialise the generators list

    for i in [1..n-1] do
        # generate Ti generators
        Ti := (i, i+1);
        Add(generators, AsBipartition(Ti, n));  # converts to partition
    od;

    for i in [1..n-1] do
        # generate Ui generators
        Ui := [[i, i+1], [-i, -(i+1)]];
        for j in [1..n] do
            if j <> i and j <> i+1 then
                Add(Ui, [j, -j]);
            fi;
        od;

        Add(generators, Bipartition(Ui));
    od;

    return generators;
end;
```

Code 1: Implementing generators function in GAP.

In our algorithm, to avoid infinite looping and ensure a reasonable computational run time, we imposed a limit on the maximum number of generators chosen in any given attempt. This was particularly useful for testing, as the parameter could be changed to adjust the algorithm's behaviour and performance. In our experiments, we found that setting this to 1000 generators per attempt provided a good balance between exploration and efficiency.

Additionally, we assumed uniform weights for the generators, meaning each generator had an equal probability of being selected at each step of the factorisation process. This choice simplified the implementation and allowed us to focus on the core algorithmic aspects, namely, attaining a valid factorisation, without introducing additional complexities.

Here is the source code for our factorisation algorithm, with implementation and testing:

```
LoadPackage("semigroups");  # load in required monoid package

max_iterations := 1000;  # hard-coded limit to avoid infinite looping
n := 50;  # replace with the desired value of n
monoid := BrauerMonoid(n);  # initialise Brauer monoid
a := Random(monoid);  # input diagram

## Function to create the generators for Bn given n
BrauerGenerators := function(n)
    local generators, Ti, Ui, i, j;
    generators := [];  # initialise the generators list

    for i in [1..n-1] do
        # generate Ti generators
        Ti := (i, i+1);
        Add(generators, AsBipartition(Ti, n));  # convert to partition
    od;

    for i in [1..n-1] do
        # generate Ui generators
        Ui := [[i, i+1], [-i, -(i+1)]];
        for j in [1..n] do
            if j <> i and j <> i+1 then
                Add(Ui, [j, -j]);
            fi;
        od;

        Add(generators, Bipartition(Ui));
```

```
    od;

    return generators;
end;

## Function to perform the factorisation process
FactorisationProcess := function(a)
    local itr, step, b, g_k, generatorsUsed;
    b := Identity(monoid);
    generatorsUsed := [];  # list to store generators

    for step in [1..max_iterations] do
        if b = a then
            break;  # stop loop if factorisation is found
        fi;

        if RankOfBipartition(a) > RankOfBipartition(b) then
            break;  # hard stopping for loops
        else
            # pick a random generator g_k from the set of generators G
            g_k := Random(BrauerGenerators(n));
            b := b * g_k;
            Add(generatorsUsed, g_k);
        fi;
    od;

    return [b, generatorsUsed];  # return final diagram and generators list
end;

## Testing the algorithm, i.e. sample run
num_trials := 1000;  # hard-coded limit

if a = Identity(monoid) then  # check if input is identity
    Print("Identity element, no factorisation needed.");
else
    for attempt in [1..num_trials] do
```

```
        b := FactorisationProcess(a)[1];
        result := FactorisationProcess(a)[2];

        if b = a then
            Print("Factorisation Found After Attempt ", attempt, ":\n\n");
            Print("Input Diagram: \n", a, "\n\n");
            Print("Resulting Diagram: \n", b, "\n\n");
            Print("Factorisation Result: ", result);
            break;  # exit loop if factorisation is found
        fi;
    od;

    if b <> a then
        Print("Factorisation not found after 1000 attempts.");
    fi;
fi;
```

Code 2: GAP Algorithm

The success rate of our algorithm (Code 2) depends on various factors, including the complexity of the input diagram a and the chosen maximum number of iterations.

When evaluating our algorithm with the parameters; $n = 50$, a maximum of 1000 generators, and 1000 trials, we observed an average success rate of 0.53%. This result suggests the factorisation process is challenging and may need further refinements to improve its effectiveness.

8 Next Steps

In the next steps for our project, we plan to enhance our factorisation algorithm by incorporating reinforcement learning to dynamically adjust the weights of the generators. Unlike our current implementation with uniform weights, RL will enable the algorithm to learn optimal weights based on the current state of factorisation, potentially improving its performance.

With RL, we aim to explore non-uniform weight distributions for the generators, allowing the algorithm to prioritise certain generators over others based on their effectiveness in achieving successful factorisations. This dynamic approach can lead to more efficient factorisation processes by biasing the selection towards generators that contribute more to the desired outcome.

In terms of improvements, there are two keys areas to focus on:

1. **Increasing Success Rate**: We aim to enhance the algorithm's success rate by leveraging specific properties of B_n multiplication. By utilising operations like delete, braid, and swap, we can guide the factorisation process towards more successful outcomes.

2. **Optimising Factorisation Length**: Producing shorter factorisations, implies a faster factorisation process. To achieve this, we intend to to make better use of the relations within B_n, ensuring that each step of the factorisation significantly reduces complexity. By incorporating these relations more effectively into the algorithm, we aim to enhance the factorisation process, resulting in more concise factorisations.

These improvements aim to strength our factorisation algorithm, improving its effectiveness and adaptability to various scenarios. This contributes to advancing our understanding and application of machine learning techniques in the context of Brauer monoid factorisations.

9 Conclusion

In conclusion, we have developed a factorisation algorithm for the Brauer monoid B_n, which integrates deterministic rules with machine learning, specifically reinforcement learning. Our algorithm aims to decompose Brauer diagrams into a product of their generators, with the ultimate goal of achieving successful factorisations. While our algorithm shows promise, further refinements are needed to improve its success rate and efficiency.

10 References

[1] Brauer, R. [1937], 'On algebras which are connected with the semisimple continuous groups', *Annals of Mathematics* pp. 857–872.

[2] Dolinka, I., East, J. and Gray, R. D. [2017], 'Motzkin monoids and partial Brauer monoids', *Journal of Algebra* **471**, 251–298.

[3] Group, G. et al. [2007], 'Gap system for computational discrete algebra'.

[4] Kudryavtseva, G. and Mazorchuk, V. [2006], 'On presentations of Brauer-type monoids', *Open Mathematics* **4**(3), 413–434.

[5] Marchei, D., Merelli, E. and Francis, A. [2024], 'Factorizating the Brauer monoid in polynomial time', *arXiv preprint arXiv:2402.07874* .

[6] Raynor, S. [2021], 'Brauer diagrams, modular operads, and a graphical nerve theorem for circuit algebras', *arXiv preprint arXiv:2108.04557* .

[7] Synopsys Inc. [2024], 'What is reinforcement learning? – overview of how it works'.
URL: *https://www.synopsys.com/ai/what-is-reinforcement-learning.html*