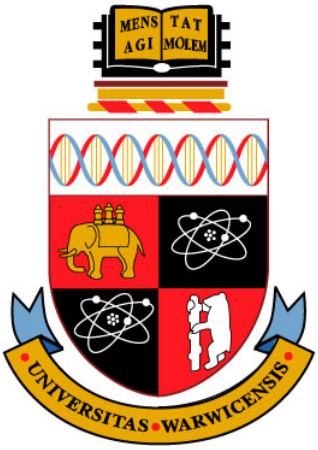

An Analysis Of Strategies In The Re-Pairing Game

CS344 Discrete Mathematics Project

Laveen Chandnani

Supervisors: Dr. Dmitry Chistikov & Dr. Matthias Englert

Department of Computer Science

Abstract

The project expands on the work of Chistikov and Vyali, which introduced a simple one-player game; The re-pairing game can be played on any well-formed sequence of opening and closing brackets (a Dyck word). A move consists of "pairing" any opening bracket with any closing bracket to the right of it, and "erasing" the two. The process is repeated until we are left with no remaining brackets. Such a game can have many strategies, but the effectiveness of a strategy is measured by it's width, which is the maximum number of nonempty segments of symbols seen during a play of the game.

Keywords: *Dyck language, Re-pairing brackets, Combinatorics, Web application, Python, ReactJS*

Acknowledgements

I'd like to thank my dissertation supervisors, Dr. Dmitry Chistikov and Dr. Matthias Englert, for their invaluable guidance, support and feedback throughout this project.

I'd also like to thank Melany Henot, Brendan Bell, Raumaan Ahmed, Varun Chodanker and Devon Connor for engaging in insightful discussions on some of the problems I tackled, and for their continued support which enabled me to complete this project throughout a difficult year.

Contents

Chapter 1

Introduction

We start with a simple one-player game. Take any Dyck word (a balanced sequence of opening and closing brackets), for example:

```
(()())
```

A **move** in this game consists of pairing any opening bracket with any closing bracket to its right, and replacing the two with a blank space/gap (denoted with the _ symbol). This process is repeated until there are no more brackets to pair up, resulting in the empty string, and the sequence of moves made is called a **re-pairing**. An example of a re-pairing is as follows:

```
(()())
(_)(_)
___(_)
______
```

Note that our first move pairs up two brackets that are not matched to each other in the initial word. We allow such moves in a re-pairing.

Finding any re-pairing following these rules is a trivial task, but we want to measure how good a re-pairing is. For this we introduce a property called the **width**, defined as the maximum number of non-empty segments of brackets seen during the re-pairing. We can see that the previous re-pairing had width 2. A natural question then arises; can we obtain a

re-pairing with a smaller width? The answer for this particular string is yes, and such a re-pairing is shown below:

```
(()())
_()()_
_()___
______
```

Here we see this re-pairing has width 1. Our goal is to try and minimise this width as much as possible.

1.1 Objectives

This project aims to take both a theoretical and practical approach to this game. We split the project into two parts; a literature review and software development.

The literature review aims to assess the current knowledge available on the game by understanding and expanding on the relevant background knowledge, context and results. This will also help gain a more intuitive grasp on the topic.

The software development aspect aims to implement strategies in the re-pairing game to allow for visualisation and experimentation. A subgoal of this is to attack the goal of minimising width with software, by analysing certain strategies and subcases of Dyck words. More specifically, the objectives of the project can be outlined as follows:

1. An introduction to the relevant definitions and content to lay out background knowledge.

2. Expanding on and providing potential novel insights on well established results.

3. Establishing the research landscape on the re-pairing game and relevant topics.

4. Creating software to demonstrate re-pairing strategies from literature and allow manual experimentation on Dyck word re-pairings.

5. Using software to analyse subcases of Dyck words and exhaustively find their width in an attempt to pin down the formula of a general

Dyck word.

1.2 Related work

This project builds on the work from Chistikov and Vyalyi [2], whose paper serves as a basis for this project. The paper introduces this game as a way to prove a lower bound on translating one-counter automata (OCA) into Parikh-equivalent nondeterministic finite automata (NFA). It provides lower bounds on the width for simple and non-simple re-pairings, which is then used to prove lower bounds on these translations.

Due to the specialised nature of this game's origins, there are no other works on this game at the time of writing.

Chapter 2

Literature Review

This section aims to cover the relevant definitions, theorems, and results from literature. We will also attempt to provide insight on and formalise some well known ideas in the context of this project.

2.1 Background

Before we delve into the re-pairing game itself, it's important to understand the objects that make up the game itself. We provide a brief but rigorous background on ideas relevant to this game below.

2.1.1 Dyck Paths

We start with the concept of Dyck paths. These are paths of length $2n$ where the path starts at $(0, 0)$ and finishes at $(2n, 0)$, the only legal moves are $(1, 1)$ or $(1, -1)$, and the path does not cross below the x-axis at any point. More formally:

Definition 1. *A **Dyck Path** is a sequence of vectors (a, b) starting at $(0, 0)$ and finishing at $(2n, 0)$, such that every step on the walk goes from some (a, b) to either $(a + 1, b + 1)$ or $(a + 1, b - 1)$.*

Here's an example of such a path:

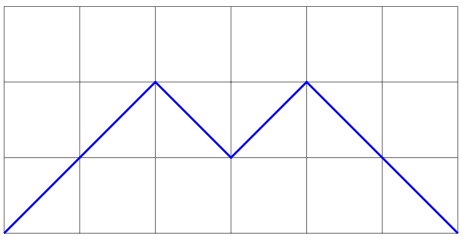

Figure 2.1: An example Dyck path

The move $(1, 1)$ is called a "rise", and the move $(1, -1)$ a fall. Notice that all Dyck paths must have more rises than falls at any point during the path to stay above the x-axis, and to reach $(2n, 0)$ we must have had as many rises as falls in the whole sequence.

We can describe a Dyck path as a string of opening and closing brackets, where `(` corresponds to a rise, and `)` corresponds to a fall. We call such strings Dyck words.

2.1.2 Dyck Words

Definition 2. *A **Dyck word** is a string that satisfies the following:*

1. *The only characters in the string are "`(`" or "`)`".*
2. *At any point in the string, there are no more "`(`" brackets than "`)`" brackets.*
3. *The string contains the same number of "`(`" and "`)`" brackets*

It should be clear that every Dyck word corresponds to a unique Dyck path.

Alternatively, we can define a Dyck word through the convention "`(`" $= 1$ and "`)`" $= -1$:

Definition 3 ([2])**.** *A **Dyck word** is a sequence $\sigma = (\sigma(1), \ldots, \sigma(n))_{n\geq 1}$ such that for all $1 \leq i \leq n$, we have $\sigma(i) \in \{1, -1\}$ and the following is satisfied:*

1. *For every $1 \leq k \leq n$, we have $\sum_{i=1}^{k} \sigma(i) \geq 0$*
2. *$\sum_{i=1}^{n} \sigma(i) = 0$*

These definitions are equivalent, and will be used interchangeably throughout this report.

2.1.3 North-East Lattice Walks

These words have interesting combinatorial properties, one of which is their unsurprising link to Lattice Walks:

Definition 4. *A **North-East lattice walk** is a sequence of vectors (a, b) starting at $(0, 0)$ and finishing at (n, n), such that every step on the walk goes from some (a, b) to either $(a + 1, b)$ or $(a, b + 1)$.*

In other words, every step is exactly one unit vector right or one unit vector up.

Theorem 1. *Every Dyck word of length $2n$ corresponds to a unique North-East lattice walk from $(0, 0)$ to (n, n) that does not cross above the line $y = x$.*

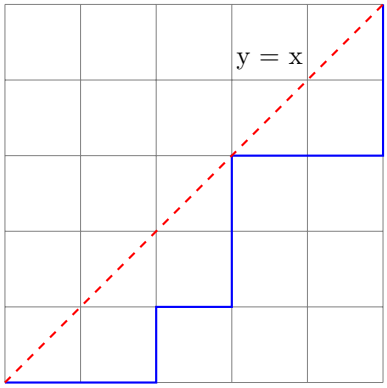

Figure 2.2: An example North-East lattice walk for `(()())(())`

This is a trivial mapping, as we can take a move east $(1, 0)$ to be "`(`", and a move north $(0, 1)$ to be "`)`". This works out as expected since we have $2n$ brackets to match, and n moves east and north each during such lattice walks. Our restriction on Dyck words that ensures there are never more "`)`" brackets than "`(`" brackets corresponds to the restriction of the walk being below (or on) the line $y = x$. This connection gives us another result:

Theorem 2 ([12])**.** *The number of Dyck words of length $2n$ corresponds to the n^{th} Catalan number $C_n = \sum_{i=1}^{n-1} C_i C_{n-1-i}$.*

Proof. Let W_n be the number of North-East lattice walks from $(0, 0)$ to (n, n) that do not cross above the line $y = x$. Given any such walk, it may touch this line at any (i, i) where $0 \leq i \leq n - 1$. This gives us a natural recursion; every such North-East lattice walk can be broken down into a North-East lattice walk from $(0, 0)$ to (i, i), and from (i, i) to (n, n). Using this idea, we see that $W_n = \sum_{i=0}^{n-1} W_i W_{n-1-i}$. This is precisely the recursion for the n^{th} Catalan number. □

2.1.4 Binary Trees

However, we can also relate this result to the concept of binary trees as well. Suppose we had four factors of a number $x = x_1x_2x_3x_4$. How many ways can we *completely parenthesize* this expression of factors (i.e. place brackets around this to group factors)?

1. $(x_1(x_2x_3))x_4$
2. $((x_1x_2)x_3)x_4$
3. $x_1(x_2(x_3x_4))$
4. $x_1((x_2x_3)x_4)$
5. $(x_1x_2)(x_3x_4)$

There are 5 ways, but this is the same as $C_3 = \frac{1}{3+1}\binom{2(3)}{3} = 5$. To understand why, let G_n be the number of ways to place brackets around $x = x_1 \dots x_n$. Note that the outermost factor will not be within any pair of brackets.

We look at the leftmost "(", and its matching ")". Suppose this surrounds $0 \leq i \leq n-1$ factors. We can completely parenthesize this expression of factors in G_i ways. Since the outermost factor will not be within any pair of brackets, this leaves a further $n-i-1$ factors to completely parenthesize, which can be done in G_{n-i-1} ways. Therefore the number of ways to completely parenthesize an expression of n factors is $G_n = \sum_{i=0}^{n-1} G_iG_{n-1-i}$, which is precisely the n^{th} Catalan number.

Theorem 3. *The number of proper binary trees with $n+1$ leaves is C_n.*

Proof. The idea is to show that every proper binary tree of $n+1$ leaves corresponds to a unique completely parenthesized expression of n factors. Recall that a proper binary tree is a tree such that every non-leaf node has 2 nodes. □

2.2 Re-Pairing Game Definitions

Now that we've formalised some of the background knowledge, we can start to discuss the re-pairing game.

2.2.1 Re-Pairings

Definition 5 ([2])**.** *Formally, a **re-pairing** of a Dyck word of length n is a sequence of ordered pairs of indices $m = (m_1, \dots, m_{\frac{n}{2}})$, where each*

$m_i = (l_i, r_i)$ *for* $l_i, r_i \in \{1, \ldots, n\}$*, and the following properties are satisfied:*

1. *For all* $1 \leq i \leq \frac{n}{2}$*, we have* $l_i \leq r_i$*,* $\sigma(l_i) = 1$ *and* $\sigma(r_i = -1)$*.*

2. *Indices* l_i *and* r_i *appear in no other ordered pair* m_j *for* $j \neq i$*.*

This says that a re-pairing consists of ordered pairs of indices of the Dyck word, where each pair consists of an index for a left bracket and an index for a right bracket to its right, and these indices do not appear in any other ordered pair.

Since a Dyck word of length n must be even, we must have $\frac{n}{2}$ ordered pairs, and it is clear this will be a positive integer.

To play the game, we define any valid re-pairing m on a valid Dyck word, and start at the ordered pair m_1. We take this ordered pair, match the left bracket at position l_i with the right bracket at position r_i of the Dyck word. We then **pair** these brackets up by erasing them and placing a gap "_" in their place. We repeat this process until we've reached the end of the re-pairing.

We've already seen two possible re-pairings on the Dyck word `(()())`. Both of these resulted in the final string of `______`. For the sake of brevity we refer to the string of n gaps as ε_n.

Let us consider the moves given by the sequence $((1, 6), (2, 5))$:

```
(()())
_()()_
__)(__
```

With this sequence of moves, we see that there is no more moves we can make which gives us the final string ε_n. We formalise this notion.

Definition 6. *A* ***partial re-pairing*** *on a Dyck word of length* n *is a sequence of moves* $p = (p_1, \ldots, p_k)$*, where* $k \leq \frac{n}{2}$*, such that applying all moves in order does not result in* ε_n*.*

Definition 7. *A* ***partial Dyck word*** *is a string which has been partially re-paired.*

Observe that every Dyck word will also be a partial Dyck word, as its partial re-pairing is the empty sequence. From here onwards, a "complete

re-pairing" will refer to a re-pairing that is not parial, but if not specified, assume "re-pairing" to mean a complete one.

2.2.2 Height

Recall that a Dyck word corresponds to a unique Dyck path, and each bracket in the word is either a rise or a fall.

Therefore, each bracket can be associated with a **height** property, which we'll define as the height of the path after we've taken the rise/fall move corresponding to this bracket. Naturally, we can extend this to talk about the height of a Dyck word, which is be the maximum height of any bracket in the Dyck word. By our sequential definition of a Dyck word, we can formalise this idea.

Definition 8. *The **height of a position i** in a Dyck word σ is defined as $h_\sigma(i) = \sum_{j=1}^{i} \sigma(j)$.*

*Similarly, the **height of a Dyck word** is $h(\sigma) = \max_{1 \le i \le n} \sum_{j=1}^{i} \sigma(j)$.*

2.2.3 Width

As mentioned previously, we want some notion of how *good* a given re-pairing is. Every complete re-pairing has $\frac{n}{2}$ moves, so there is no use in analysing the asymptotic run-time of a re-pairing. We'll instead define a new property, called the **width**. This refers to the maximum number of non-empty segments seen during the re-pairing.

Definition 9. *Let M be a re-pairing of a Dyck word σ. The **width of a re-pairing** is the maximum number of non-empty segments observed during the re-pairing. We write this as width(M).*

However, we also mentioned that given a Dyck word and some re-pairing, there could exist other re-pairings with lower widths. This leads us to ask, what is the smallest width we need to re-pair a given Dyck word?

Definition 10. *Let σ be a Dyck word, and $M(\sigma)$ be the set of all possible re-pairings of σ. Then, the **width of a Dyck word** is the minimum width required to re-pair it.*
Formally, this is $width(\sigma) = \min_{\forall M \in M(\sigma)} width(M)$

To avoid confusion, we'd like to differentiate between the width of a word, which is a property over all re-pairings, and the width of the word during a re-pairing, which is just the current number of non-empty segments. During

a re-pairing, the string is guaranteed to be strictly partial. We'll call such strings **proper** partial Dyck words.

Definition 11. *The **width of a proper partial Dyck word** is the number of non-empty segments currently visible in the word.*

2.2.4 Dyck Primes

We noticed earlier that not all re-pairings will give us ε_n. In particular, the partial re-pairing we saw could not be extended to a complete re-pairing. We backtrack to the last partial Dyck word seen in the re-pairing where a complete re-pairing was possible. This was at `_()()_` . Here, the only sequence of moves that allows us to completely re-pair this is $(2, 3)$ then $(4, 5)$.

A natural question arises; what was different about this sequence of moves that allows us to completely re-pair the word? To answer this, we define a special subset of Dyck words, called Dyck primes.

Definition 12. *A **Dyck prime** (or prime word) of length n is a Dyck word σ such that $\sum_{i=1}^{k} \sigma(i) > 0$.*

*Similarly, a **composite Dyck word** (or composite word) is a Dyck word that is not prime.*

This defines a Dyck word which cannot be constructed by only concatenation of other existing Dyck words. In terms of Dyck paths, this means the path will never touch the x-axis.

We make a claim about words composed of Dyck primes.

Claim 1. *Let σ be a composite Dyck word, and M be a re-pairing of σ. Then applying the re-pairing M will yield ε_n if and only if M does not contain any moves which pair brackets from different Dyck primes within σ.*

To understand why this must hold, notice that if A and B are two Dyck primes, then by choosing a left bracket from A and a right bracket from B, there are no longer an equal number of left and right brackets within A and B.
In particular, A contains fewer left brackets than right brackets now. Since a pairing requires a right bracket to be paired with a left bracket to its left, there will be an extra right bracket which cannot be matched with anything. A similar argument for B shows there will be an extra left bracket. Since these cannot be matched with each other either, we cannot obtain ε_n.

2.3 Re-Pairing Strategies

We can now begin to discuss strategies for playing the re-pairing game. By strategy, we refer to a set of rules that we can follow to generate a sequence of moves which will always yield ε_n. We'll describe all strategies from a high level, and then give algorithms for running the strategies on a given input. All algorithms will use the sequential definition of a Dyck word.

2.3.1 Simple

We start with the most simple strategy. Given any Dyck word, we already have a naturally existing pairing, namely the pairs of matching brackets. However, given any simple re-pairing, we do not have a clear order on which matching brackets to pair first, and we have seen that the order matters greatly when attempting to minimise the width.

We'll begin with the most natural ordering, namely pairing each leftmost bracket with its matching right bracket. This means for two moves (l_i, r_i) and (l_j, r_j), if $l_i < l_j$ then we perform (l_i, r_i) first.

Algorithm 1 Leftmost Simple Re-Pairing

$c \leftarrow 0$
$p \leftarrow 0$

Chapter 3

A Web Application For The Re-Pairing Game

This section discusses the process of implementing a web application that allows for visualisation and experimentation of the re-pairing game.

3.1 Software Requirements

Before any software was written, a list of requirements for the software was made. These are features that the software must have in order to fulfill the relevant objectives, and were subsequently tested throughout the development of the web application:

1. Allow a user to enter Dyck words.
2. Validate a given input and provide a relevant error message if required.
3. Display the given Dyck word with the ability for manual interaction via a mouse.
4. Verify that the characters selected are a valid move before allowing them to be paired.
5. Allow for the use of strategies from literature (Simple, Non-Simple recursive), as well as our own strategies (Simple Greedy, Brute Force).
6. If selecting a strategy, generate a list of moves before visualising the play.

7. Display the list of moves, and allow the user to jump to any move in the play.

8. Display a running width counter from the moves seen so far, and the width of the overall play if a strategy has been used.

9. Allow the user to brute force a given list of Dyck Words, and return the optimal widths of each along with the re-pairing.

3.2 Technologies

The development of the web application went through multiple iterations before settling on the current choice of technologies used. The application uses the JavaScript library React for the frontend user interface with the Material UI library for components, and the Python web framework Flask for the backend algorithms with PyPy for faster runtimes. This section discusses the decisions made when choosing the main technologies (frameworks, libraries etc).

3.2.1 Initial Ideas

The software was initially planned to be a local Python only application, using the Tkinter GUI library. This was because the author is most familiar with the Python programming language, and has some limited experience with the Tkinter library. However, early prototypes of this software using Tkinter proved to be cumbersome to work with and have a dated visual style. Alternative Python UI libraries (such as PyQt and Kivy) were also considered, but proved to have learning curves no steeper than using an entirely different language.

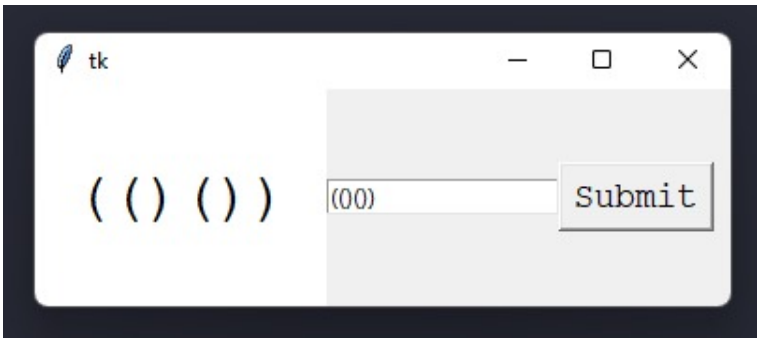

Figure 3.1: An early iteration of the software using Tkinter

We then considered the prospect of a web application; the author again had some limited experience with HTML/CSS, but using these languages

to control the layout and sizing of the UI seemed much more intuitive. This approach also naturally allowed for separation between the UI and the strategies, and the author's familiarity with Python could still be leveraged as a backend language. The author had also wished to learn the technologies used for web development, so this seemed like the natural approach to take.

3.2.2 Backend

The strategies to run on Dyck words are implemented in Python but run using PyPy, and communicated with the frontend using Flask.

Flask

Flask is a web framework that allows users to build a web server with Python that they can communicate with using HTTP requests, allowing Python to be used as a backend language [10]. One of the reasons Flask was chosen over potential alternatives, such as Django, is due to its simplicity. Flask is classed as a microframework, meaning it does not have a lot of dependencies or libraries required to use it. This makes Flask fairly lightweight and efficient for our purposes, reducing overhead when communicating requests back and forth.

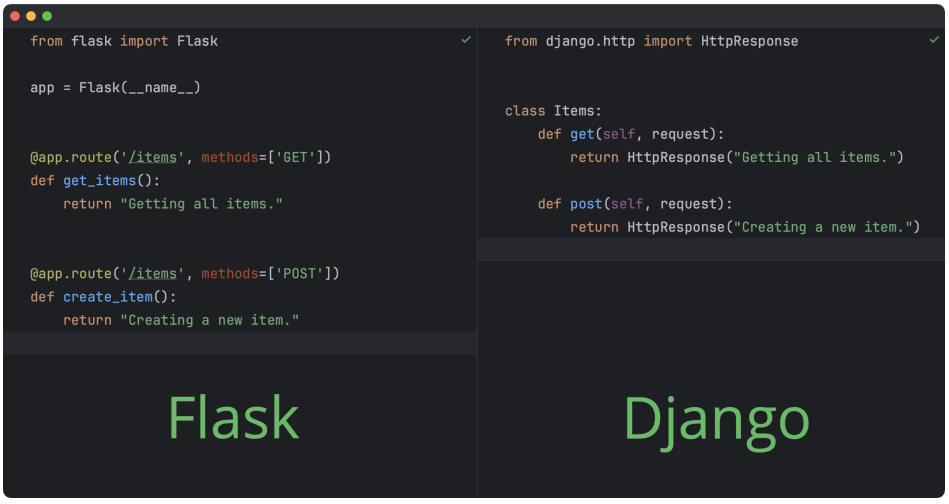

Figure 3.2: Implementing the same functionality in Flask and Django [5]

Figure 3.2 shows the difference between using Flask and Django. In the Flask example, the same functions could be used with the only modification being an app route, which is added directly above the function. This binds each function to the route URL plus its given route, and depending on the type of request either "get_ items()" or "create_ items()" would be called.

Thus, any work done to translate a re-pairing strategy into an algorithm could easily be implemented using Flask by simply adding an app route at the top.

PyPy

During the implementation of the web application, the author discovered PyPy. This is an alternative implementation of Python, providing faster execution times and less memory usage than the standard implementation.

In particular, PyPy uses a Just-In-Time compiler, which converts Python code into machine code at runtime. This makes PyPy have the greatest speed-up when executing long-running programs where a significant fraction of the time is spent executing Python code [13], which is exactly what the backend will be doing (particularly when brute forcing for the width of a Dyck word).

Exceution of the program below, which computes the factorial of 20000 and repeats this 500 times, demonstrates the potential speed-up that PyPy can provide:

```
1   import timeit
2
3
4   def factorial(x):
5       f = 1
6       for i in range(x):
7           f *= x - i
8       return f
9
10
11  x = 20000
12  t = timeit.timeit(
13      "factorial(x)", "from __main__ import factorial", globals={"x": x}, number=500
14  )
15  print(f"Time taken: {t:.3f}s")
```

Figure 3.3: Computes 20000! 500 times

Executing this program using Python takes 77.930s, but using PyPy takes 31.872s. The program also requires no modifications to run on PyPy, since PyPy is a restricted subset of Python called RPython. However, these restrictions have had no impact on the development of this web application.

All of the re-pairing strategies (Simple, Non-Simple, Simple Greedy) were trivial implementations into algorithms due to the nature of Python's syntax.

3.2.3 Frontend

React

Naturally, the components of the UI would need ways of interacting with one another. This led to using React, which is a JavaScript library used to simplify the process of creating user interfaces [8]. It allows modular user interface components to be built, and later nested and combined to create a dynamic and interactive webpage.

The web application is designed as a single page application (SPA), meaning it loads only a single web document and updates its contents dynamically. In contrast, typical websites will make the web browser load entire new pages when the contents of the page need to be modified [7]. React fits the development of SPAs well, since each component has its own state and properties, and React will only automatically re-render components if there is a change to one of these. This allows for greater performance gains due to the lower overhead and a more dynamic experience.

Material UI

Rather than styling the components of the web application from scratch, the author opted to use Material UI. This is a React component library that implements the design language made by Google, and contains a collection of prebuilt components with customisation options.

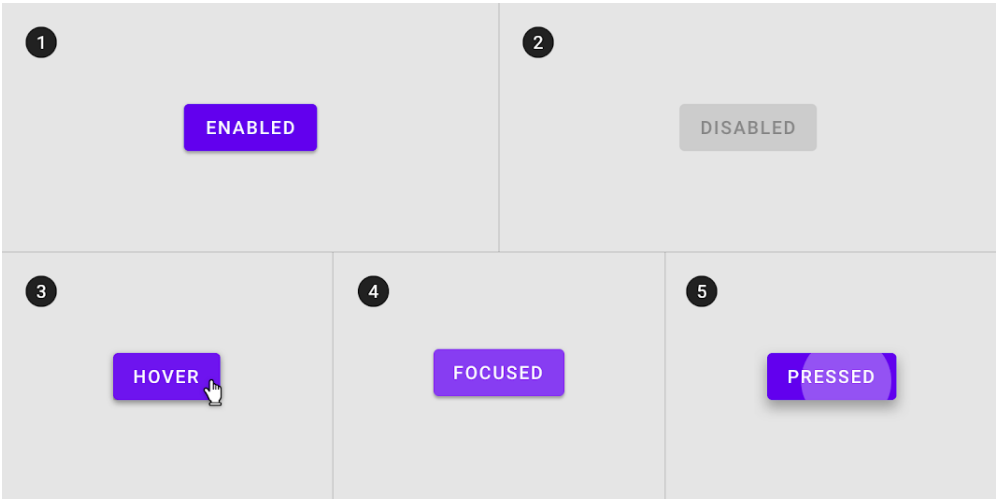

Figure 3.4: An example of how a Material UI button varies depending on its state [3]

Through this library, we have been able to more easily modify the visuals of UI components to better reflect what can and cannot be accessed on the

web application with the current states set.
For example, if an input is invalid, we do not want to allow generating steps from a selected strategy. Instead of always keeping the button available to be clicked and having to check if the input is valid when we try to generate steps, we simply map the state of the "disabled" property to the inverse of a "valid" boolean state variable we set. React makes the visual change automatically depending on the value of this property, which further streamlines the development process.

3.3 Implementation

This section concentrates on the usage of the previously mentioned technologies during the development of this web application. In particular, we discuss what parts have been developed in the backend and the frontend, and how each of these have been implemented.

3.3.1 Backend

This contained the bulk of the computational work for the re-pairing game. This includes any validations on inputs, calculations on the width property for a re-pairing and the generation of re-pairing moves using strategies.

The majority of functions in the backend were developed to handle **POST** requests rather than **GET** requests. This is because the backend is used more so as a way to process any incoming data about Dyck words from the frontend rather than to store data. This decision is a deliberate one, and elaborated on further in the frontend section.

As a result, almost all backend functions with app routes will return data using the **jsonify** function from the Flask library. This ensures any data is properly formatted in the JSON (JavaScript Object Notation) format [4]. It can take Python data structures, such as lists and dictionaries, and convert them into a string format that can be easily transmitted to and understood by React. It's important to note that not all Python objects can be converted into JSON format. However, for the purposes of this project, all relevant data structures can be converted by jsonify.

We also make use of the **Flask-CORS** library. CORS stands for Cross-Origin Resource Sharing, and is mechanism used to allow a web application to indicate any domains or ports other than its own from which a browser should permit loading resources [6]. The backend runs locally on '127.0.0.1:8080', which is different to the frontend's location at 'localhost:5173'. This separates the development of the frontend and the back-

end, allowing each of them to be tested separately before being combined together. However, this means they both run on different origins, and therefore we must use Flask-CORS.

Validation

Given any input, we want to ensure that it is indeed a valid Dyck word. To do so, we simply iterate through a given string, checking that it meets the definition of a Dyck word. In particular, since Python strings can be treated as lists, we pass the input into Python, which then allows us to treat each character as an element of a list. We then treat the list of elements as a sequence, and use the sequential definition of a Dyck word (see Definition 3).

Generating Moves

When we want to generate the re-pairing moves using a certain strategy on some given valid input, we use Python. This is done through the following steps:

1. Call the selected strategy's relevant function.

2. To calculate each move, iteratively apply the strategy to the input until an empty string is reached, appending each calculated move into a list.

3. Send the list of moves to a separate function which calculates the width after applying each move, and the maximum width seen during the play.

4. Return a final 2D list $[m_1, \ldots, m_{\frac{n}{2}}]$, where each m_i contains:

   (a) the i^{th} move $[l_i, r_i]$ corresponding to the index for the left bracket and right bracket respectively

   (b) the width of the proper partial word after m_i

   (c) the string of the proper partial word

   in that exact order.

After generating moves, we want to display each move with the relevant information about width in the UI. Returning a list containing all this information, instead of just the moves itself, allows us to easily convert this into a list of interactive moves with relevant information to display.

Factors

TALK ABOUT CALCULATING FACTORS OF THE WORD. THIS RESTRICTS THE CHOICE FOR MANUAL REPAIRINGS BY ENSURING ONLY BRACKETS FROM THE SAME COMPONENTS ARE CHOSEN.

Strategies

Each of the Simple, Non-Simple, and Greedy strategies are trivially translated into Python from their algorithmic descriptions seen previously.

Width Calculation

As a result of generating moves in the backend, the width calculation is also done here. We iterate over the initially generated list of moves, and calculate the width after applying each move. Given a move m_i, we have two possible approaches to calculating the width.

1. **Approach 1:** A naive approach, where we apply the move to the current word, and run through the word counting the number of non-empty segments encountered. This can be trivially shown to have $\mathcal{O}(n^2)$ (each move requires running through the word of length n, and there are $\frac{n}{2}$ moves).

2. **Approach 2:** We can speed this up by analysing adjacent characters around the brackets to be re-paired, rather than calculating a new width value each time. This will require storing the previous width, but since we're iterating over the list of moves in order and calculating the width after each move anyway, this requires no extra effort as we can simply take the previous result each time.

We used approach 2, which requires the following observation.

Observation 1. *Replacing a single bracket with the* `_` *symbol will either maintain the current width, increase the width by one, or decrease the width by one.*

Proof. WLOG consider the sequence of characters "A(B" (the same logic applies if the character is a right bracket instead). Then we have 3 cases:

1. A and B are both gaps. Here, replacing a bracket with a gap will mean this sequence of characters will no longer contain a non-empty segment of brackets, so the width will decrease by one.

2. A and B are both brackets. Here, replacing the bracket with a gap will mean this sequence non-empty segment of brackets will be divided

by a gap, breaking this into two segments. Therefore, the width will increase by one.

3. Exactly one of A and B is a bracket. Suppose A was a bracket (we'll represent this as A), and B was a gap (represented as `_`). Then, the re-pairing will turn this sequence of characters from "A`(_`" to "A`__`". In this case, there has not been a removal of an existing segment like the first case, nor has there been an introduction of a new gap like in the second case. We can think of the gap on the right as *expanding* out into the middle, shortening the segment of brackets on the left. Therefore the width stays the same.
   Note that by symmetry, this case is the same as A being a gap and B being a bracket, as our gap now *expands* in the opposite direction.

This concludes the proof. □

From here, it suffices to analyse the characters adjacent to the brackets being paired. We visualise a move as turning "A`(`$B_1 \dots B_2$`)`C" into "A`_`$B_1 \dots B_2$`_`C", where A, B_1 are characters adjacent to "`(`" and B_2, C are characters adjacent to "`)`". We take the width of the subsequence before the move is made and, by using the adjacency analysis from above, add and subtract to this width as necessary.

However, suppose we were re-pairing the Dyck word "`(())`" using the simple strategy. We immediately notice two edge cases with this analysis that cause some issues:

1. The first move would be the indices (0, 3). Using our analysis as above, we'd be considering the subsequence "A`(`$B_1 \dots B_2$`)`C". But since this move takes the outermost brackets, A and C do not exist!

2. The second move would be the indices (1, 2). However, since this move pairs adjacent brackets, here B_1 and B_2 do not exist either!

For the first case, if a pairing takes an outermost bracket we need only consider its respective inner adjacent character. If this is a gap, the width increases by 1, otherwise the width stays the same.

To implement this in Python, we could use boolean logic to check if our move indices are adjacent or outermost. This tells us which adjacent characters exist, and then we can consider what effects these have on the current width. However, we can circumvent the first issue entirely by the following observation:

Observation 2. *Consider the Dyck word $D' =$ "_D_" created by padding an existing Dyck Word D with gaps on the outside. Then given a re-pairing of D with width w_i at move i, applying these moves as a partial re-pairing to D' gives width $w'_i = w_i$ for each move i.*

This simplifies the first edge case; instead of having to check if a move takes an outermost bracket, we can simply pad the word from the start with gaps on the outside and apply the same analysis as before.

For the second case we simply check if a pairing takes adjacent brackets or not. If so, we can consider these to be a single bracket like our analysis in the proof of Observation 1, and the same idea applies.

Using approach 2 over the naive idea from approach 1 means we do not have to iterate over the string of length n for each of the $n/2$ moves, and instead we require a constant number of steps. This means approach 2 has $\mathcal{O}(n)$, which is a quadratic speedup over approach 1.

3.3.2 Frontend

This section explores the higher-level design aspects of the yehweb application. We'll discuss the different components that make up the web application in both the UI and the backend.

It's important to note that most of the data and calculations from the backend on Dyck words are stored in state variables. This allows for a simpler development process; by checking for changes in state variables, re-renders can be made more streamlined since React will re-render automatically when a change to its state variables is made.

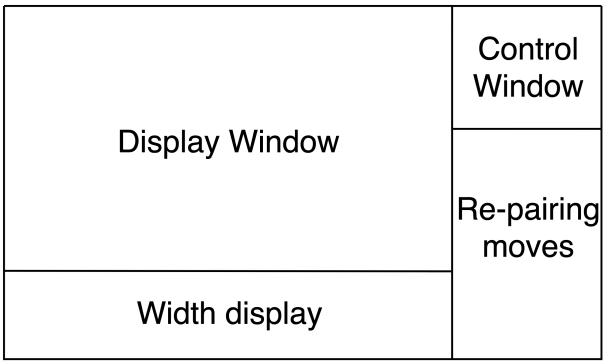

Figure 3.5: A high-level overview of the components that make up the web application

Control Window

This is where all input and controls will exist. Here the Dyck word will be taken in as input, and all options for runnings strategies

Chapter 4

Project Management

4.1 Methodology

4.1.1 Development Methodology

This project used the Agile methodology, as the goals and limitations of the project aligned closely with the principles of Agile [1]:

- Creating functional software was of a higher priority than comprehensive documentation. The faster that the software was made functional, the more time could be devoted to its refinement. Given that the author was also new to writing web applications and many of the libraries chosen at the start of this project, making such refinements proved to have a steeper learning curve.

- Responding to changes in requirements was of a higher priority than a strict plan and implementation. Throughout the process of absorbing the seminal work [2], more elegant and useful ideas arose and became the clear option to implement over the current option.

Note that the software requirements were written and strucutred in such a way that each one could mostly be handled sequentially, without needing any of the later requirements. Approaching the development in this manner greatly enhanced efficiency and also fit well into the Agile development methodology; functionality could easily be linked to a requirement, and the sequential nature of requirements meant that there was a clear roadmap to follow at all times.

4.1.2 Testing Methodology

As development was done in an Agile manner, it made sense for all testing to be manual, since the functionality of the software was rapidly changing as it was being developed. This meant writing automated tests each time new functionality was required would have been more time-consuming than manual tests. The theoretical nature of the game for which the web application was designed also made manual testing a viable option; as long as definitions were translated correctly into code and certain edge cases were verified, the software could be expected to perform as required.

All testing was done using a combination of unit testing and incremental integration testing. We elaborate on this process below.

Unit Testing

During the development, each of the software requirements was decomposed into smaller functions and subgoals needed to fulfill their relevant requirements. This naturally leads into unit testing; we treated each function as its own unit, and tested it according to the expected functionality.

For example, to ensure a Dyck word is valid and provide error messages if necessary, we break this down into multiple units as follows:

1. A function for the frontend to call and receive data from
2. A function to check the input against the definition of a Dyck word, and return a yes or no answer along with a relevant error message
3. Logic to check if an error was returned
4. Logic to display the error or the input depending on the result

We then write a function (or in some cases with the frontend, a new component or a new state variable) and test each of them work as expected.

As previously mentioned, by using Flask as the framework of choice for the backend, the Python code required minimal modification to allow functions to serve requests. This made unit testing much easier to perform.

Incremental Integration Testing

After all the units for a requirement have been written and tested, we incrementally integrate them, again manually verifying that inputs and edge cases provide expected outputs. In particular, this approach falls under the concept of "Bottom-Up Integration" [11], since we started at

the smallest possible units and progressively added units together to create larger components.

4.2 Setup

The start of any software development process requires some setup to ensure the required technologies are functional. We want to make sure the frameworks, libraries and tools we intend to use are in a good state, and any necessary preqrequisites and dependencies are fulfilled before development begins to ensure a more streamlined process.

Some methods of testing may also require initial setup, and this is important to account for as well.

We also want to ensure that any contingencies are already in place for any non-negligible problematic events which could occur.

This section discusses the concerns above in more detail, as well as how they were resolved or mitigated.

4.2.1 Git and Github

One of the most obvious issues which may occur is the loss of progress due to hardware failure. During the development process, we may also find that an approach taken for fulfilling a requirement causes issues with pre-existing functionality, but manually reverting and removing every new line of code is a time-consuming process.

A simple solution to both of these is the use of **Git** and **Github**, for version control and cloud-based backups. In particular, both the codebase and the report were constantly kept up to date in Github, with distinct commit messages to ensure that progress could easily be tracked.

4.2.2 Virtual Environments

Using Python as the backend for the project required many dependencies and additional libraries for certain functionality to exist. One previously discussed example of this is Flask-CORS, which is the library required for CORS functionality. However, simply using `pip` to install the required libraries may cause conflict issues; only a single version of a given package can be installed with a given Python interpreter, and this can lead to issues when different pre-existing projects require different versions of packages.

To solve this, we use a **virtual environment**. This creates an isolated

Python environment, where specific versions of a Python interpreter, software libraries and binaries are held [9]. By isolating it from the global installation of Python, we can install alternative versions of libraries, and ensure that there are no conflicts with any global installation of packages. This also ensures that we always use the same version of each package when testing the web application.

For certain periods of time the author was also required to relocate, and so developing on different machines and operating systems was also something to consider. It is not enough to simply clone the virtual environment between machines as the scripts within it may refer to system locations. Using virtual environments also helps with this, since the libraries and dependencies required for the software could be automatically listed in a `requirements.txt` file. This is done by the command `pip freeze > requirements.txt`. Then, we can use a `.gitignore` file to tell Git not to push the virtual environments, but to push this requirements file, which can be used to easily install all required libraries and dependencies.

4.2.3 Goodnotes

For much of the literature review, written notes were taken to help the author absorb its contents. This was chosen over typed notes due to the ease of noting down both diagrams for visually interpreting concepts and mathematical formulas. To do this, we used the Goodnotes app; this is an iPadOS application which allows us to take handwritten notes and annotate pdfs. These can then be backed up to the cloud, to ensure that any observations and ideas written down are not lost in the event of hardware failure.

Below is an example page to illustrate the usage of note-taking during this project.

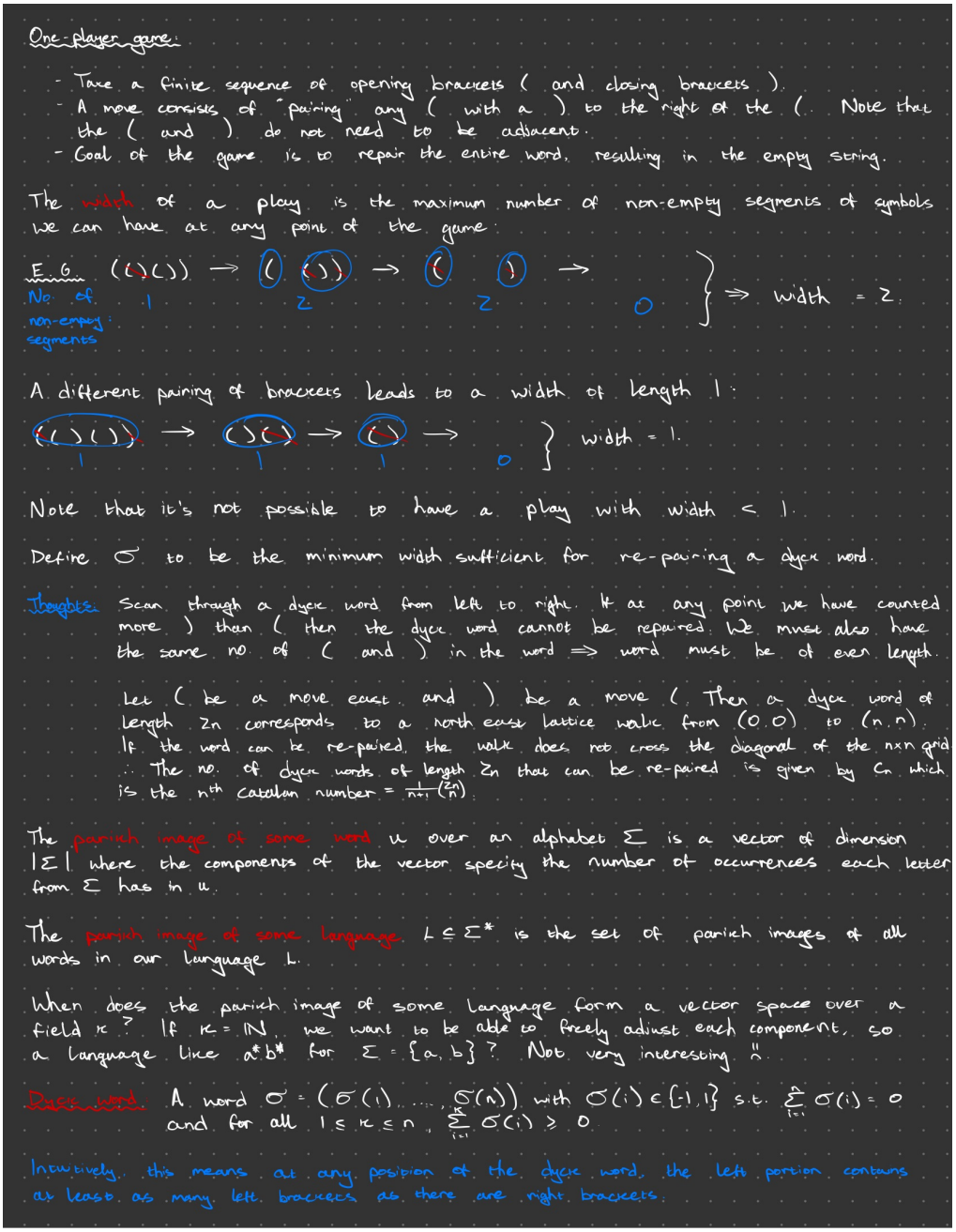

Figure 4.1: An example page of the written notes taken

4.3 Issues Encountered

One of the largest risks from the beginning of this project has been the possibility that obtaining new findings on the open problem from the seminal work [2] turns out to be an infeasible task. When the project first

begun, one of the goals was to attack this problem from a more theoretical approach. However, after further effort during term 1 this approach was found to be unproductive. Fortunately, this was taken into account early on, and the goal shifted from a theoretical approach to a more software oriented approach. Further details of this can be found in the progress report.

An unforeseen issue on the author's health made it difficult to complete this report for the original timeline. However, after some deliberation with an incredibly supportive department, the author was granted an opportunity to extend the timeline for this report. The resulting timeline can be seen in the next section.

TALK ABOUT DIFFICULTY OF IMPLEMENTING BRUTE FORCE?

4.4 Timeline

Bibliography

[1] M. Beedle *et al.* "The agile manifesto." (2001), [Online]. Available: https://agilemanifesto.org/principles.html.

[2] D. Chistikov and M. Vyalyi, "Re-pairing brackets," in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2020, pp. 312–326.

[3] Google. "Buttons - material design." (2018), [Online]. Available: https://m2.material.io/components/buttons.

[4] M. Makai. "Flask.json jsonify." (2021), [Online]. Available: https://www.fullstackpython.com/flask-json-jsonify-examples.html.

[5] D. Mashutin. "Django vs flask." (2023), [Online]. Available: https://blog.jetbrains.com/pycharm/2023/11/django-vs-flask-which-is-the-best-python-web-framework/.

[6] MDN Web Docs. "Cross-origin resource sharing (cors)." (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[7] ——, "Spa (single-page application)." (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[8] Meta Platforms. "React." (2023), [Online]. Available: https://react.dev/.

[9] Python. "Venv - creation of virtual environments." (2024), [Online]. Available: https://docs.python.org/3/library/venv.html.

[10] Python Basics. "What is flask?" (2021), [Online]. Available: https://pythonbasics.org/what-is-flask-python/.

[11] M. Rouse. "Bottom-up testing." (2014), [Online]. Available: https://www.techopedia.com/definition/18035/bottom-up-testing.

[12] J. Rukavicka, "On generalized dyck paths," *The Electronic Journal of Combinatorics*, vol. 18, 2013.

[13] The PyPy Team. “Pypy - features.” (2024), [Online]. Available: https://www.pypy.org/features.html.