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

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 (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. It is split 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

3

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

2.1.1 Dyck Words

Definition 1. *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*

We can alternatively define a Dyck word through the convention "(" $= 1$ and ")" $= -1$:

Definition 2 ([2]). *A **Dyck word** is a sequence $D = (d_n)_{n \geq 1}$ such that $d_n \in \{1, -1\}$ and the following is satisfied:*

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

2. *$\sum_{i=0}^{n} d_i = 0$*

2.1.2 North-East Lattice Walks

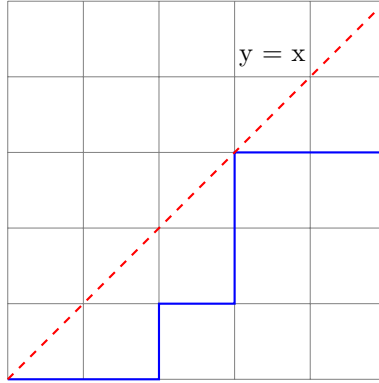These words have interesting combinatorial properties, one of which is their unsurprising link to Lattice Walks:

Figure 2.1: North-East lattice walk for `(()())(())`

Definition 3. *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 $y = x$.*

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 ([8])**.** *The number of Dyck words of length $2n$ corresponds to the nth 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 nth Catalan number. $\square$

2.1.3 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 $x = x_1 x_2 x_3 x_4$. 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 $x = x_1 \ldots x_n$. 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 $G_n = \sum_{i=0}^{n-1} G_i G_{n-1-i}$, which is precisely the nth 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. $\square$

Chapter 3

A web application for the 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.
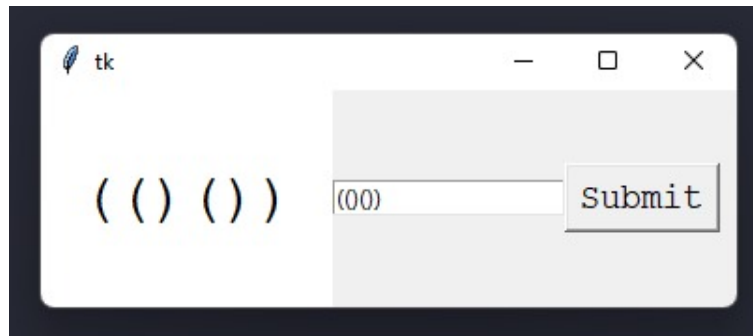

Figure 3.1: An early iteration of the software using Tkinter

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 [7]. 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.


```python
from flask import Flask

app = Flask(__name__)


@app.route('/items', methods=['GET'])
def get_items():
    return "Getting all items."


@app.route('/items', methods=['POST'])
def create_item():
    return "Creating a new item."
```
```python
from django.http import HttpResponse


class Items:
    def get(self, request):
        return HttpResponse("Getting all items.")

    def post(self, request):
        return HttpResponse("Creating a new item.")
```
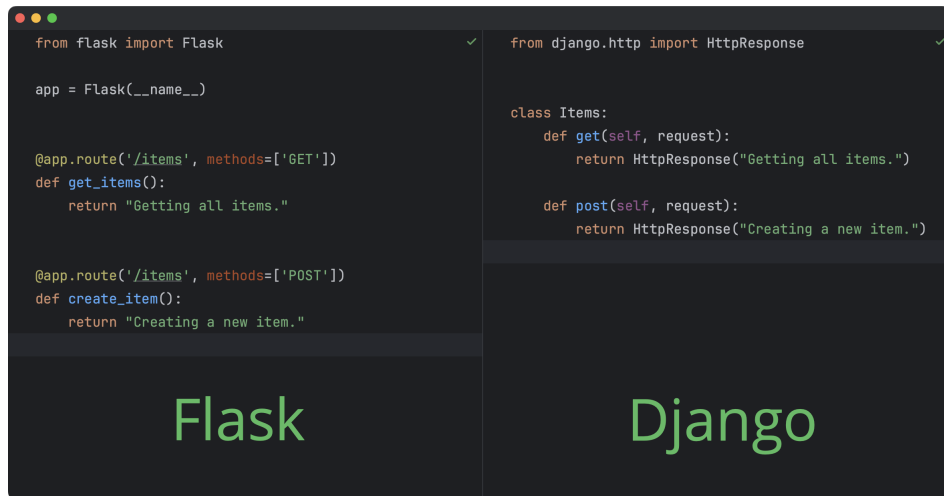
Flask   Django

Figure 3.2: Implementing the same functionality in Flask and Django [4]

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 [9], 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:

```python
import timeit


def factorial(x):
    f = 1
    for i in range(x):
        f *= x - i
    return f


x = 20000
t = timeit.timeit(
    "factorial(x)", "from __main__ import factorial", globals={"x": x}, number=500
)
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 [6]. 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 [5]. 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.
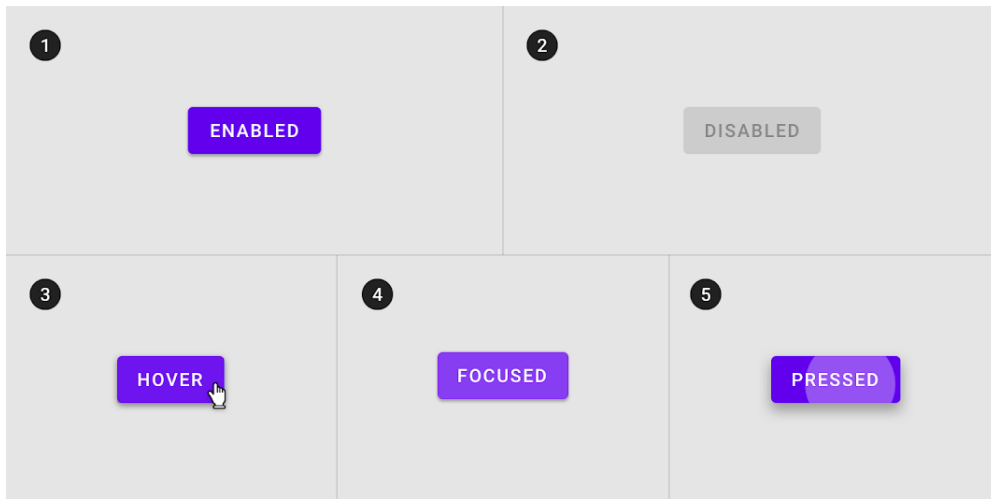

Figure 3.4: An example of how a Material UI button varies depending on its state [3]

Javascript

Why do we use JS over TS?

3.3 Implementation

This section explores the higher-level design aspects of the web application. We'll discuss the different components that make up the web application in both the UI and the backend.
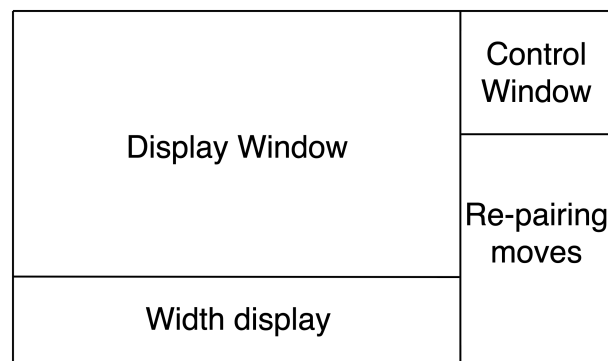
Display Window	Control Window
	Re-pairing moves
Width display	

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

3.3.1 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

3.4 Project Management

3.4.1 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.

The libraries and languages the author was most familiar with were developed first. We begun with the backend

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] 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/`.

[5] MDN Web Docs. "Spa (single-page application)." (2023), [Online]. Available: `https://developer.mozilla.org/en-US/docs/Glossary/SPA`.

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

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

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

[9] The PyPy Team. "Pypy - features." (2024), [Online]. Available: `https://www.pypy.org/features.html`.