An analysis of strategies in the re-pairing game

Laveen Chandnani

Introduction and Motivation

Dyck words are strings containing only “ (” and “) ” characters that satisfy the following conditions:

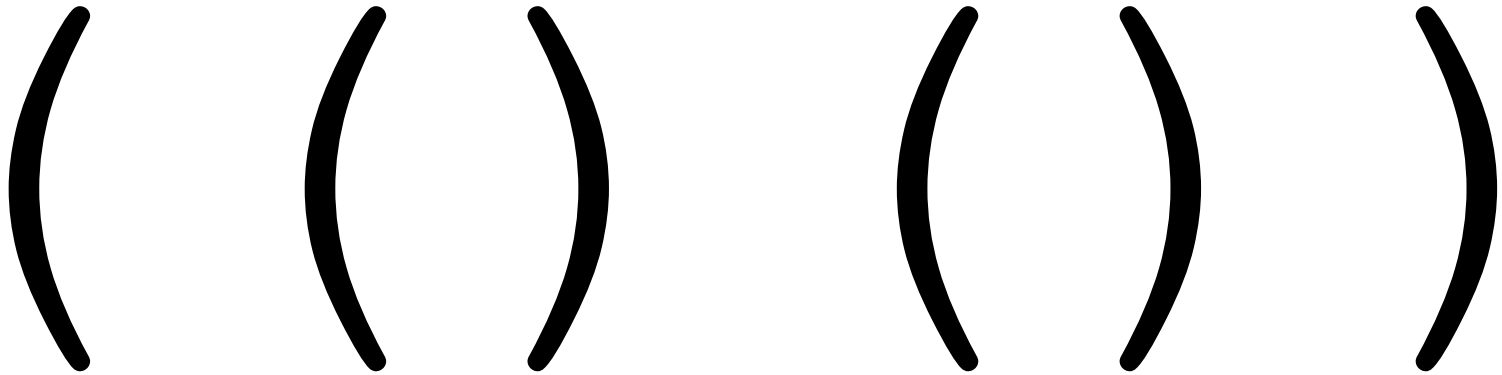
- The string contains an equal number of opening and closing brackets
- All prefixes of the string contain no more closing brackets than opening brackets

The string “ (( ) ( )) ” is a valid Dyck word, but “ ( )) (” and “ (( ) (( ) ” are not valid Dyck words.

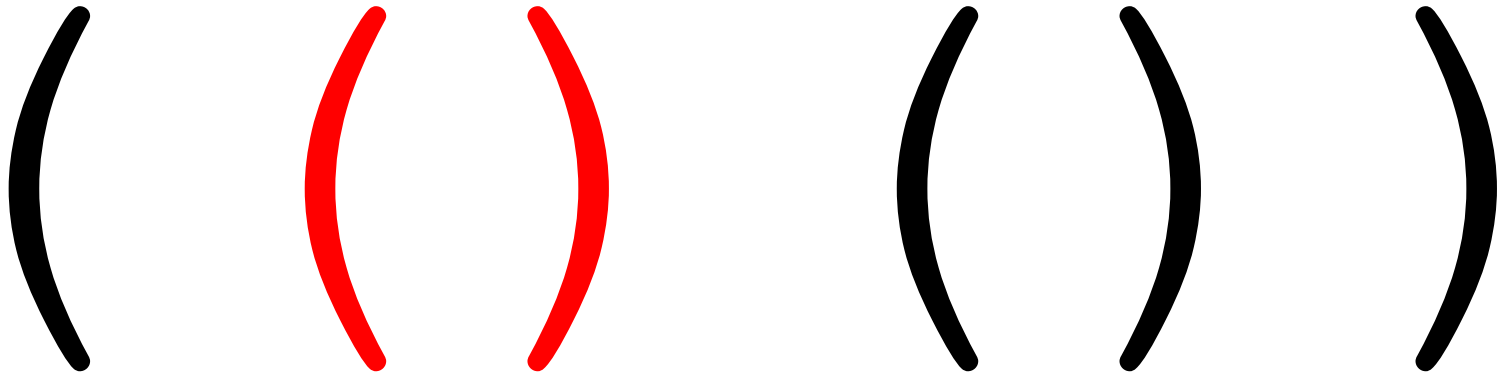
We describe a one-player game on Dyck words as follows:

- Start with a Dyck word.
- Pick an opening bracket, and any closing bracket to the right of this opening bracket.
- Pair up these chosen brackets, and erase them.
- Repeat until we have the empty string.

Introduction and Motivation



Introduction and Motivation



Introduction and Motivation

Introduction and Motivation

Introduction and Motivation

Introduction and Motivation

Introduction and Motivation

Introduction and Motivation

We define the width of a play to be the maximum number of non-empty segments present during a play of the game[1].

We define the width of a Dyck word to be the minimum width sufficient for re-pairing the word.

The previous play had width 2. Can we do any better than this?

(( ) ( )) → ( ) ( ) → ( ) →

The above play has width 1, so yes we can do better! The width of a Dyck word is always greater than 0 since the word itself is one segment.

We wish to analyse strategies that attempt to give us widths as close as possible to the width of any Dyck word.

Introduction and Motivation

Objectives:

- An introduction to the relevant definitions and content to lay out some background knowledge
- Expanding on and providing potential novel insights/perspectives on well established results
- Establishing the research landscape on the re-pairing game and relevant topics
- Creating software to demonstrate re-pairing strategies from literature and allow manual experimentation on Dyck word re-pairings
- Using the software to analyse subcases of Dyck words and exhaustively find their width in an attempt to pin down a formula for the width of a general Dyck word

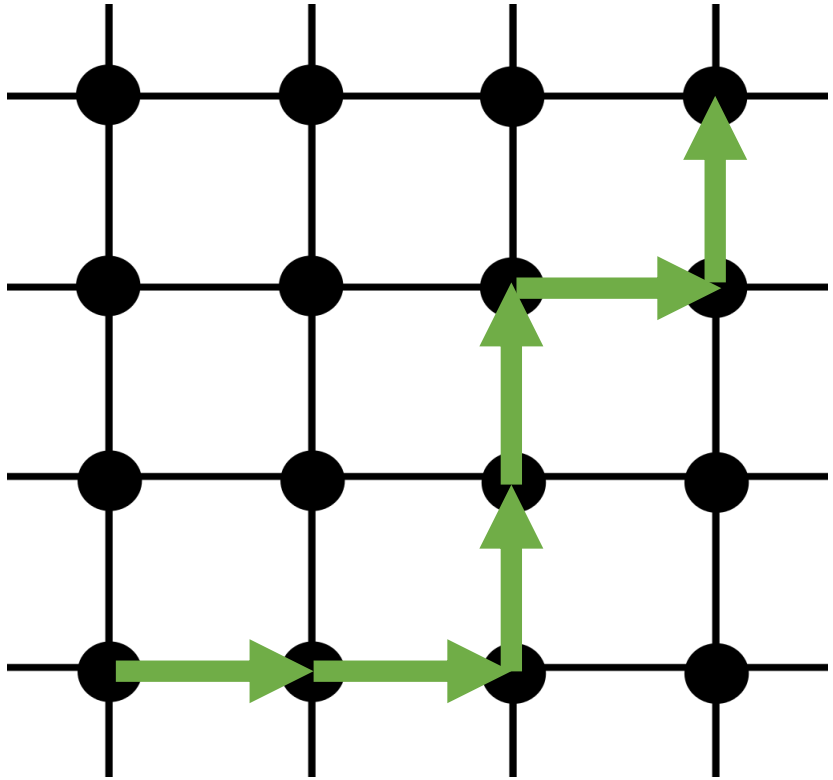
Motivation:

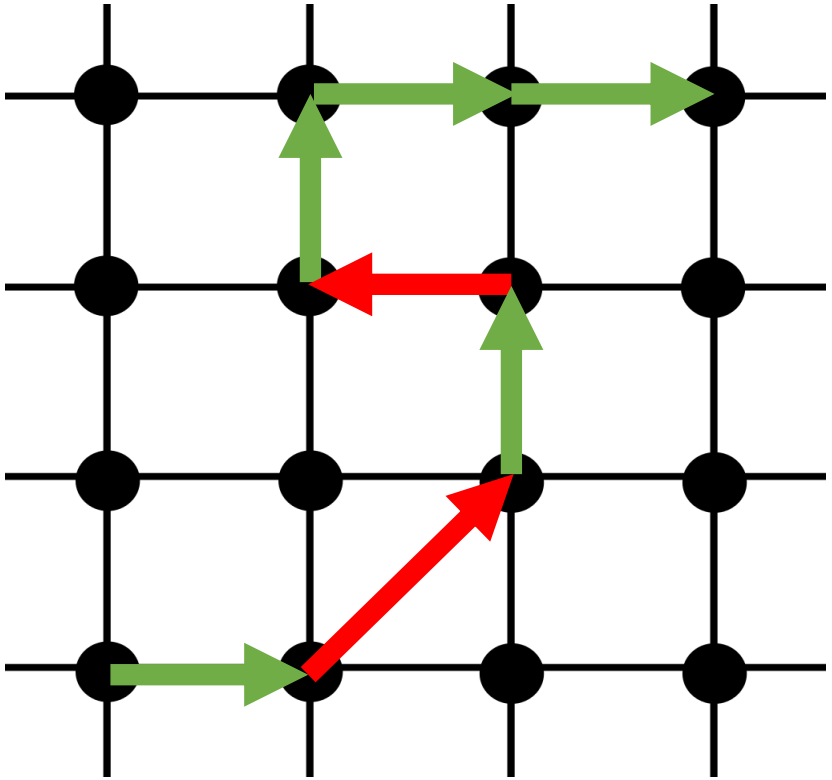
- Interesting combinatorial problem; simple to understand and challenging
- Gain experience with theoretical computer science problems
- Attempting to make progress towards an open problem at the end of the paper by Chistikov and Vyalyi

Background content and definitions

North-east lattice walks

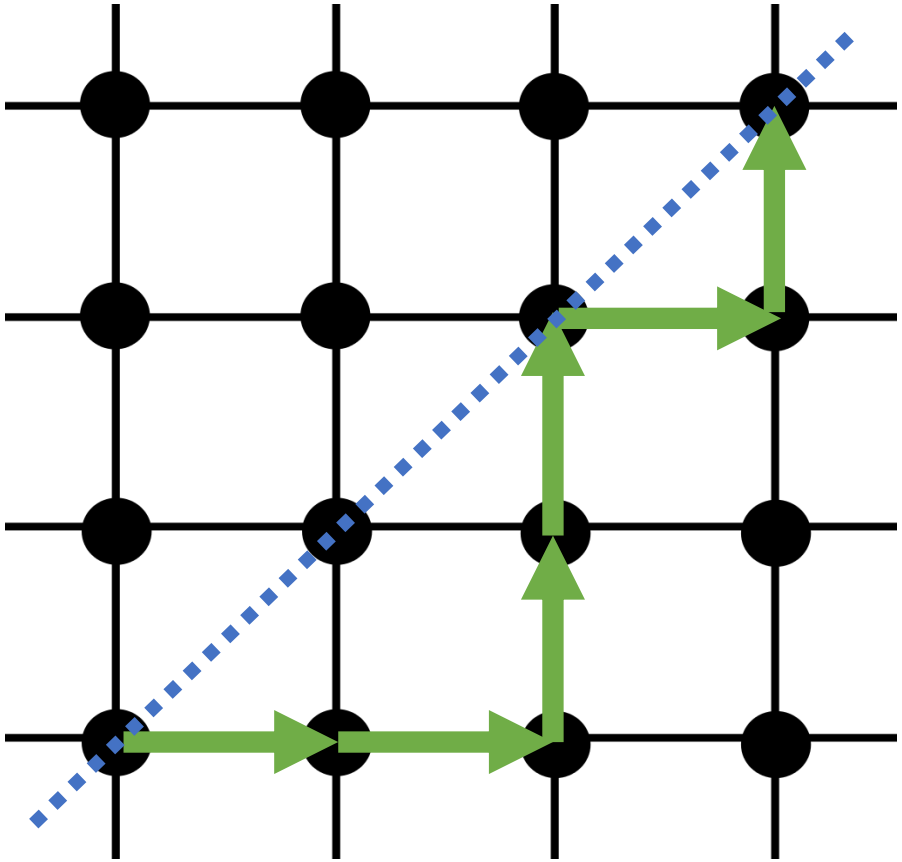
North-east lattice walks are defined on a grid of $(n+1)$-by-$(n+1)$ points by a sequence of points $P_0, P_1, \ldots, P_{2n}$ with $n \geq 0$, where each P_i is a lattice point and P_{i+1} is obtained by offsetting one unit east or one unit north on the grid. Every path starts at $(0,0)$ and finishes at (n,n)[2].

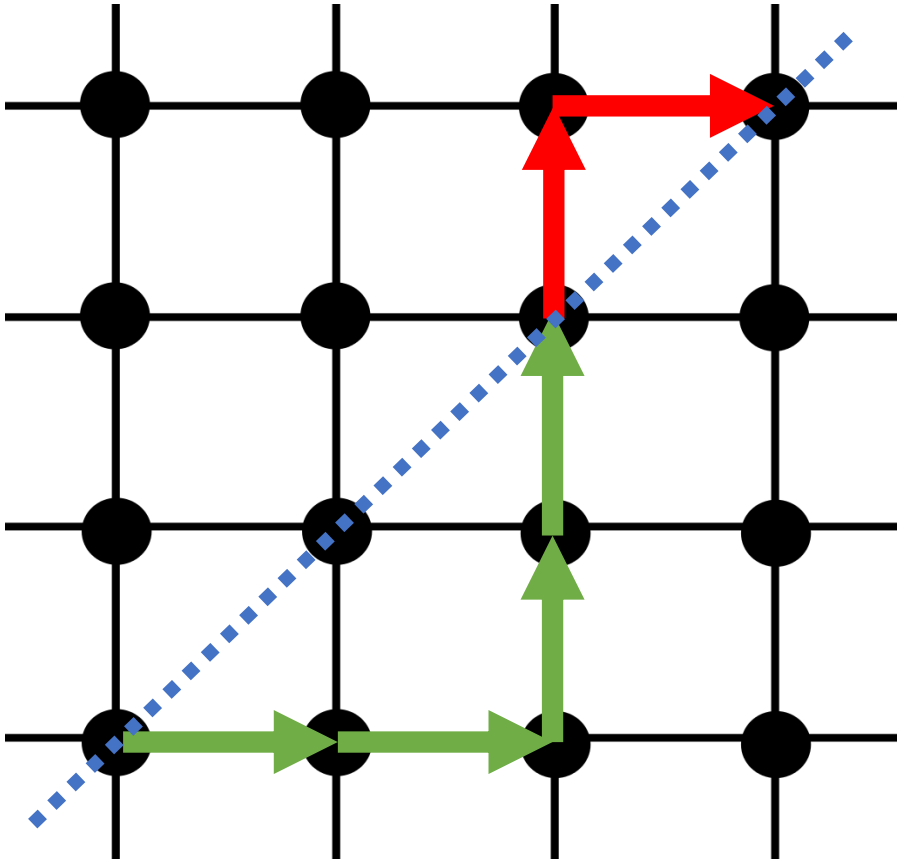




Dyck Paths

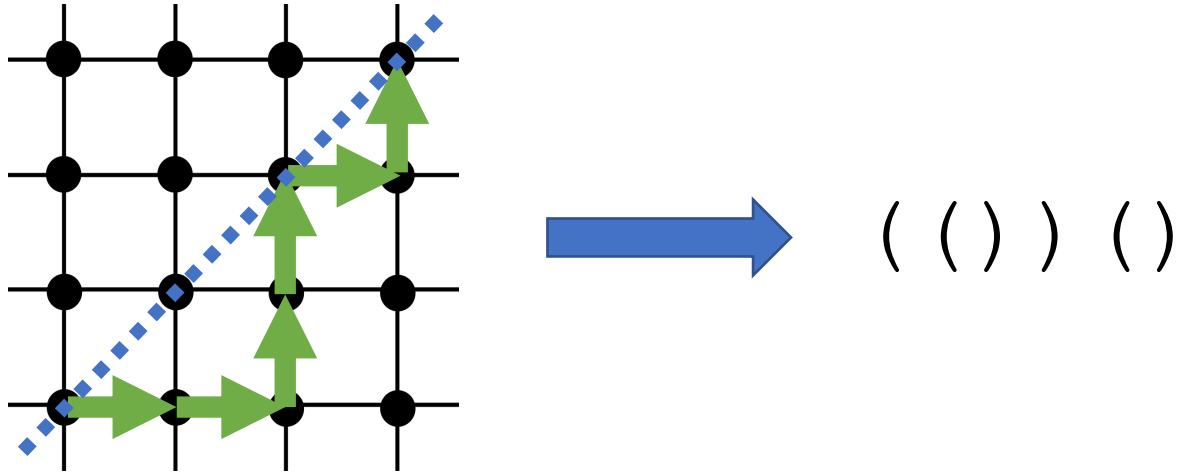
Dyck paths are north-east lattice paths where the walk does not cross the main diagonal going from $(0,0)$ to (n,n). We call these Dyck paths of length $2n$.





Dyck words

Dyck words are strings which can be used to describe Dyck paths, where a move east east is “ (” and a move north is “) ”.



From this we obtain a bijection between all Dyck paths of length $2n$ and all Dyck words of length $2n$.

$\Rightarrow$ no. of Dyck words of length $2n = C_n = \frac{1}{n+1}\binom{2n}{n}$

Binary forests

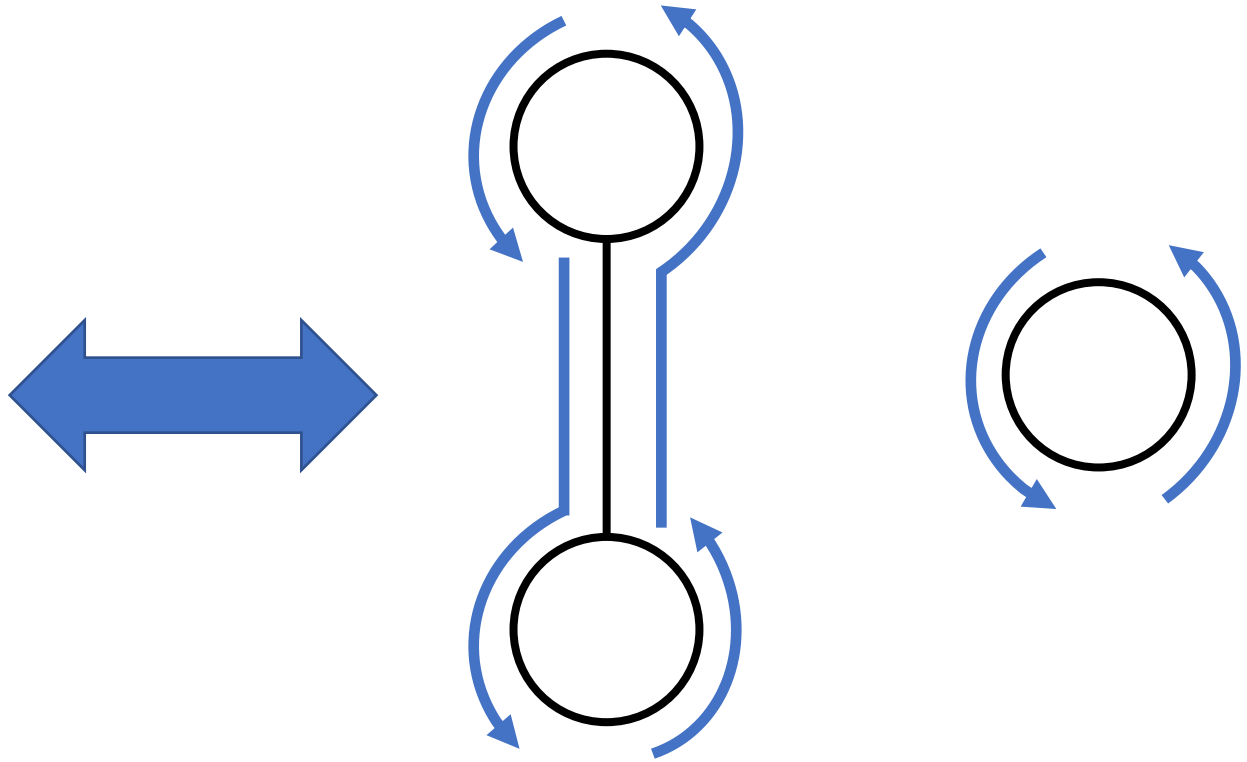
We can also construct a bijection between Dyck words and binary forests as follows:

- Given a binary forest, conduct a pre-order traversal of a tree:
  - During the traversal, if we move down in depth to the next node, write（
  - If we move up in depth to the next node, write ）
  - If we reach the end of the traversal for this tree but there are further disjoint trees to traverse, repeat this process and concatenate the resulting Dyck words together
  - The resulting tree is a unique Dyck word representing this binary forest

Since the Dyck word describes a pre-order tree traversal (or multiple if the forest consists of more than one binary tree), we just follow out the traversal described by a Dyck word to obtain our binary forest.

Binary forests

(( )) ( )



This tells us that the number of binary forests we can construct using n nodes is defined by the number of Dyck words of length $2n$

$\Rightarrow$ Number of possible binary forests using n nodes $= \ Cn$

Current results on the re-pairing game

Simple vs. Non-simple re-pairing

For any Dyck word σ, define a simple re-pairing to be any re-pairing which pairs up two matching brackets in σ :

- It is shown in Chistikov et al. that simple re-pairing strategies yield a width bounded by $O(\log|\sigma|)$.

E.G. (( ) ( )) → ( ) ( ) → ( ) →

Similarly, a non-simple re-pairing is any re-pairing which does not follow this constraint.

E.G. (( ) ( )) → ( ) ( ) → ( ) →

An example simple re-pairing strategy

- Scan through the word starting from the left until we encounter as a substring “ (_) ”, where “_” is an arbitrary number of gaps
- Erase this “ (_) ”
- Repeat until we obtain $\sigma = \varepsilon$

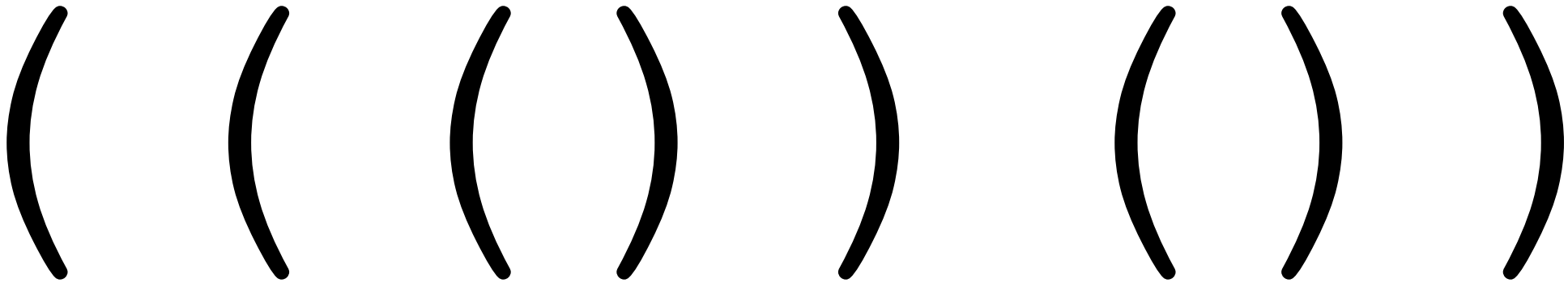
To show this algorithm is correct we claim the following:

- Every Dyck word must contain “ ( ) ” as a substring (or we cannot start the re-pairing):
  - Suppose we have a Dyck word φ that does not satisfy this
  - Dyck words always start with opening brackets, so $\varphi =$ “ (…”
  - This cannot be adjacent to a closing bracket, so $\varphi =$ “ ((…”
  - This repeats infinitely, so if φ does not satisfy this property, then $\varphi =$ “ (((…” must be infinitely long, which is not possible

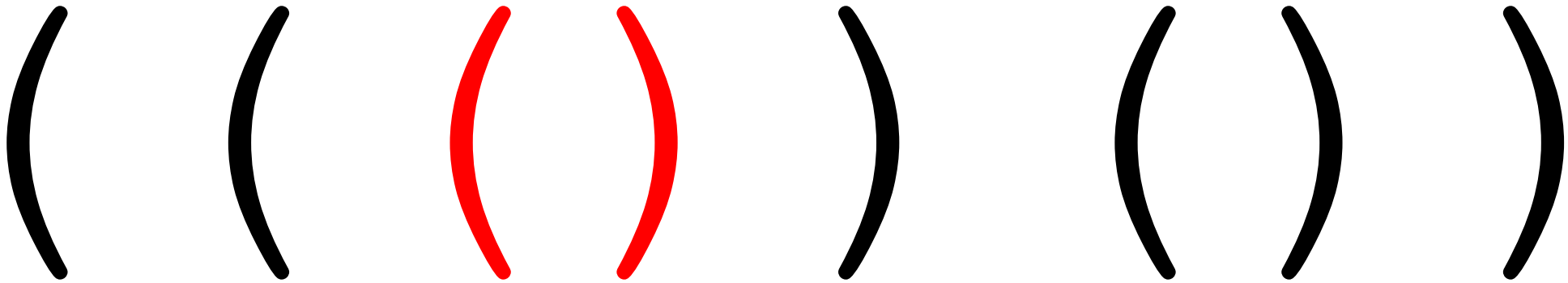
A similar argument can also be used for showing “ (_) ” is a substring during a re-pairing.

Dyck words are finite, so this algorithm must terminate.

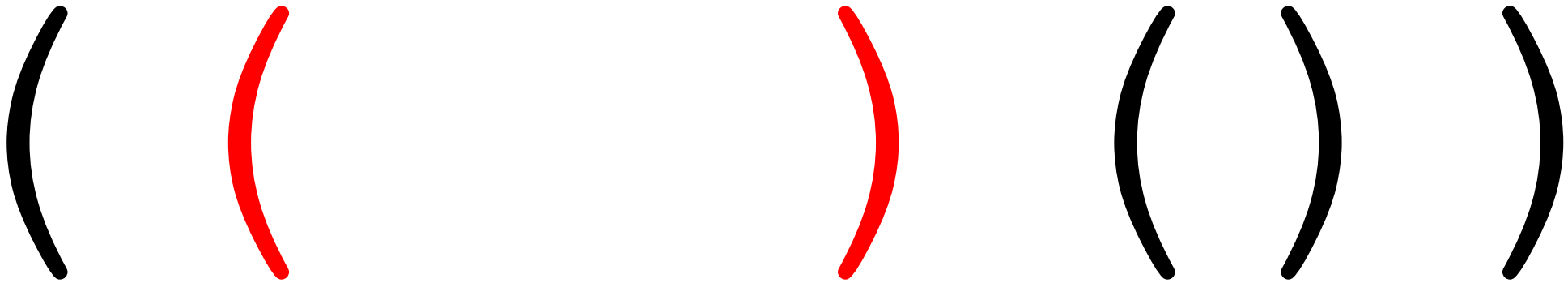
An example simple re-pairing strategy



An example simple re-pairing strategy



An example simple re-pairing strategy



An example simple re-pairing strategy

An example simple re-pairing strategy

A greedy recursive approach

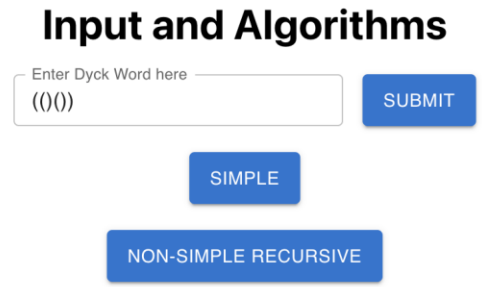
- Find the 0 levels in the Dyck word. These split the word into substrings which must also be Dyck words.
- Starting from the edges, take the Dyck word of smallest length and pair its outermost brackets.
- Treat the substring obtained after doing this as a new Dyck word, find its 0 levels and repeat
- Continue this until the empty string is obtained

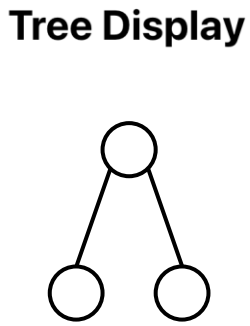
E.G. (( )) ( ) ( ) = (( )) ( ) = (( )) = ( ) =

- This strategy is not always optimal! An alternative strategy would be to take the Dyck word of smallest width from the edge (this is expensive)
- Since this is simple, it's bounded by $O(\log|n|)$

Web-based application

(()())





Project Management

- Lots of time spent researching and tweaking objectives to get a clear sense of what direction to go in
- Some difficulties have come up causing certain tasks to take longer than expected:
  - Attempting to tackle the open problem of computing the general width of a Dyck word directly seemed far too ambitious
  - Python GUI approach seemed cumbersome and dated to work with
  - Digesting content from literature has been particularly difficult
- These have been mitigated:
  - Objectives altered to allow for a more software and analytically oriented approach
  - Using a web based approach with React for the front-end and Flask for the back-end
  - More time allocated to reading literature
- Combining the front-end and back-end code is still in progress due to it taking longer than expected to learn web development and React

Further work

- Get the front-end linked up with the back-end and implement manual re-pairing
- Implement non-recursive re-pairing algorithm
- Potentially analysing the asymptotic width of the simple greedy recursive strategy (width variant only)
- Brute force Dyck words to see if a general formula for the width seems plausible

Possible extensions could be:

- Approximation strategies for the re-pairing game
- Adding custom strategies into the website for visual aid

References

1. D. Chistikov and M. Vyalyi, “Re-pairing brackets,” in Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 312–326, 2020.
2. Weisstein, Eric W. "Lattice Path." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/LatticePath.html