Permutation pattern matching

Both Emerite Neou

UNIVERSITÉ PARIS-EST
Université Paris-Est Marne-la-Vallée

A thesis presented for the degree of
Docteur de l'Université Paris-Est Marne-la-Vallée

The Permutation Pattern Matching Problem

Both Emerite NEOU

Soutenue le 18 décembre 2017

Jury:
BASSINO Frédérique
HAMEL Sylvie
NICAUD Cyril
PIERROT Adeline
ROSSIN Dominique
VIALETTE Stéphane

Contents

Remerciements

Je remercie mon directeur de thèse Stéphane Vialette pour m'avoir accompagné durant ces 4 années, pour sa bienveillance, son aide et sa patience envers moi. De même, je remercie Romeo Rizzi pour les opportunités qu'il m'a offert.

Je remercie également les membres de mon jury, Frédérique Bassino, Sylvie Hamel, Cyril Nicaud, Adeline Pierro et Dominique Rossin. Je voudrais en particulier exprimer ma gratitude à Frédérique Bassino et Sylvie Hamel pour la qualité, le détail et la précision de leurs corrections sur ma thèse, mais également tous les correcteurs anonyme pour leurs commentaires sur mes articles.

Je remercie aussi mes collègues pour toutes l'aides diverse qui m'ont fournis, Cyril Nicaud, Jean-Christophe Novelli, Claire David, Corinne Palescandolo, Philippe Gambette, Antoine Meyer et Xavier Goaoc. Ma reconnaissance va aussi à LIGM et à l'UPEMLV qui m'ont accueilli durant ces 10 années.

Finalement, je remercie ma famille et mes amis. A Jun et Yvon, merci de m'avoir aidé à corriger mon mémoire. A Richard, Alain, Anais et mes nièces, merci pour le chakra.

Merci à tous, Emerite.

Résumé

Cette thèse s'intéresse au problème de la recherche de motif dans les permutations, qui a pour objectif de savoir si un motif apparaît dans un texte, en prenant en compte que le motif et le texte sont des permutations. C'est-à-dire s'il existe des éléments du texte tel que ces éléments sont triés de la même manière et apparaissent dans le même ordre que les éléments du motif. Ce problème est NP complet. Cette thèse expose des cas particuliers de ce problème qui sont résoluble en temps polynomial.

Pour cela nous étudions le problème en donnant des contraintes sur le texte et/ou le motif. En particulier, le cas où le texte et/ou le motif sont des permutations qui ne contiennent pas les motifs 2413 et 3142 (appelés permutations separables) et le cas où le texte et/ou le motif sont des permutations qui ne contiennent pas les motifs 213 et 231 sont considérés. Des problèmes dérivés de la recherche de motif et le problème de la recherche de motif bivinculaire sont aussi étudiés.

Abstract

This thesis focuses on permutation pattern matching problem, which asks whether a pattern occurs in a text where both the pattern and text are permutations. In other words, we seek to determine whether there exist elements of the text such that they are sorted and appear in the same order as the elements of the pattern. The problem is NP-complete. This thesis examines particular cases of the problem that are polynomial-time solvable.

For this purpose, we study the problem by giving constraints on the permutations text and/or pattern. In particular, the cases in which the text and/or pattern are permutations in which the patterns 2413 and 3142 do not occur (also known as separable permutations) and in which the text and/or pattern are permutations in which the patterns 213 and 231 do not occur (also known as wedge permutations) are also considered. Some problems related to the pattern matching and the permutation pattern matching with bivincular pattern are also studied.

Chapter 1

Introduction

A permutation σ is said to *occur* in another permutation π (or π *contains* σ), denoted $\sigma \preceq \pi$, if there exists a subsequence of elements of π that has the same relative order as σ; otherwise, π is said to *avoid* the permutation σ. For example, a permutation contains the permutation 123 (resp. 321) if it has an increasing (resp. a decreasing) subsequence of size 3. Similarly, 213 occurs in 6152347, although 231 does not occur in 6152347. In the last decade, the study of patterns in permutations has become a very active area of research [37] and a conference (Permutation Patterns) focuses on patterns in permutations. We consider here the so-called *permutation pattern matching* (noted PPM), which is also sometimes referred to as the *pattern involvement problem*: Given two permutations σ of size k and π of size n, the problem is to decide whether $\sigma \preceq \pi$ (the problem is attributed to Wilf in [18]). The PPM is known to be **NP**-hard [18]; however, it is polynomial-time solvable by brute-force enumeration if σ has a bounded size. Improvements to this algorithm were presented in [1] and [4], the latter describing a $O(n^{0.47k+o(k)})$-time algorithm. Bruner and Lackner [22] gave a fixed-parameter algorithm solving the PPM problem with an exponential worst-case runtime of $O(1.52^n nk)$. This is an improvement upon the $O(k\binom{n}{k})$ runtime required by brute-force search without imposing restrictions on σ and π, in which one has to enumerate all of the $\binom{n}{k}$ different subsequences of size k in π and test if one of them has the same relative order as σ. Guillemot and Marx [32] showed that the PPM problem is solvable in $2^{O(k^2 \log k)}n$, and hence is fixed-parameter tractable with respect to the size k of the pattern σ (standard parameterisation). However, Mach proved that the permutation involvement problem under the standard parameterisation does not have a polynomial size kernel (assuming $\mathbf{NP} \not\subseteq \mathbf{coNP}/\text{poly}$ [41]). A few particular cases of the PPM problem have been attacked successfully. The case of increasing patterns is solvable in $O(n \log \log k)$-time [26], improving the previous 30-year bound of $O(n \log k)$. Furthermore, the patterns 132, 213, 231 and 312 can all be handled in linear-time by stack sorting algorithms. Any pattern of size 4 can be detected in $O(n \log n)$ time [4]. Algorithmic issues related to 321-avoiding patterns matching for permutations have been investigated in [33], [3] and more recently in [35].

This thesis studies specific cases of the PPM problem. We focus especially on the cases in which the text and/or pattern are separable permutations. We expose different solution for both cases and propose solutions of our own. Re-

lated subjects to PPM are also solved, such as the union and consensus of two permutations. Moreover, we look at the PPM when considering bivincular pattern, which we believe, is the most valuable contribution as this is the first time that these patterns are studied. We delve deeper in the separable permutation as we also study a subclass of separable permutations, namely the wedge permutations. In particular, the case in which both the pattern and the text are wedge permutations, only the text is a wedge permutation and the pattern is a bivincular for the PPM problem are considered. We present polynomial solutions for these cases. An algorithm for the longest wedge permutation is also given.

This thesis has two main motivations. Our first motivation is to get closer to the PPM's application, especially for bioinformatic, which focuses on sequences of genetic material such as DNA, RNA, or protein sequences. In particular for the DNA sequence, its transmission is studied. However, this transmission may result in the modification of the sequence: some parts can be added, removed or swapped. As such, it is a problem to decide whether two sequences are related. More generally, deciding whether a trait (which is a characteristic of an individual), given as a sequence, is present in a DNA sequence is a problem. Nonetheless, not all modifications occur, as some are lethal to the descendant and cannot thus be observed. As such, there is a logic in a sequence and it can thus split into blocs. Moreover, we know that some blocs are conserved in order for the trait to be present. So for a trait to be present in a sequence, it must contains blocs such that they appear in the same order as the blocs of trait. The PPM is found when labelling each bloc by a number. Additionally, the blocs cannot be ordered arbitrarily. There may be some dependencies between blocs (depending on the trait), such as, some blocs cannot be in the same trait and some blocs need to be next to each other. We represent this dependencies with the avoiding classes and the bivincular patterns. Our second motivation is to grasp a better understanding over some objects. Obviously, the first objects are separable permutations and wedge permutations, in which we explore their structures. This can be used in random generation and combinatorics. The second object is the bivincular patterns, which is tool not yet well-known. The study of bivincular patterns is the continuity of the study of permutation pattern as bivincular pattern generalise permutation pattern.

The thesis is organised as follows. The second chapter outlines the justification for this thesis. The third chapter defines the principal notions needed for the study. The fourth chapter presents the results known for the PPM problem and some variants of the problem at the time of writing. The next chapter offers results related to problems based on the PPM. Chapter five is devoted to the study of the PPM over separable permutations, whereas chapter six focus on the wedge permutations. Finally, the conclusion proposes research leads that we consider relevant.

Chapter 2

Context of the Thesis

2.1 Non-deterministic Polynomial Problem

A computational complexity theoretician classifies the problem by how hard solving it is and measures the difficulty of the problem by scaling the "size of the problem" to the resources needed to solve it. Such theoreticians consider two types of resources: the number of steps needed to go from the information given in the problem to the solution and the space consumption, which is how much information is needed to find the solution. The number of steps is tied to the time spent solving the problem, so by convention, the number of steps is refereed as the time consumption. A prime piece of information lies at the end of the theoretician's work: whether we can find a practical method to solve the problem. All the tools and knowledge related to computational complexity theory that we need for this thesis are presented below.

2.1.1 The Notion of Problem

In computer science, a problem is a question or task over a class of object. An example of a problem over subsets of integers is to find the maximal value contained in the set. A problem paired with an object of the class is an instance of a problem. A possible instance of the above problem is finding the maximal value contained in the set $\{3, 4, 8, 2\}$. The scaling is undertaken with the size of the object, with what we call size depending on the object. In the above example, the input's size would be the number of integers, so 4. A problem is said to be solvable if an algorithm can find the solution of any instances of that problem.

2.1.2 Reduction: Transforming one Problem into Another

A reduction is the act of transforming any instance of a first problem to an instance of another problem. Informally, this corresponds to representing a problem in a different way. The original and obtained instances must have the same solution. To understand a reduction, imagine that we want to throw a six-sided die, but do not have one. Luckily, we have six different cards. We can draw randomly one of the six cards to simulate a throw of a six-sided die.

The problem of throwing a die can thus be reduced to the problem of drawing a card. Reduction is used for two purposes, as described below.

Solving a Problem with a Reduction

The first role of reduction is to solve a problem, as in the above examples. The principle is that if we have an original problem that is not yet solvable, we can use a second problem that is solvable to solve the first one. If we know how to reduce any instances of the original problem into an instance of the second problem, then an algorithm to solve the original problem is to transform any instance of the original problem into an instance of the second problem and to solve the created instance. A simple example is a reduction of the problem of sorting a set of words in the shortlex index (which corresponds to sorting words by the number of letters and then sorting in lexical order) to the problem of sorting a set of integers; the reduction associates a word $w_1 w_2 \dots w_n$ to the number $\Sigma_{1 \leq n} \ p(w_i) * s^i$ where $p(w_i)$ is the position of the letter w_i in the alphabet, and s is the number of letters in the alphabet.

Classifying a Problem

The second purpose of reduction is to classify a problem. Indeed, if a problem can be reduced to a second one, then the second problem needs at least as many resources to be solved as the original (according to, the above strategy, and under the condition that the reduction costs less than or is proportionally the same as the cost of computing the second problem's solution). As an informal example, we can simulate a unique throw of a die by drawing in twelve different cards, by associating one side of a die to two different cards. This result in drawing in twelve different cards being at least as difficult as throwing a die once.

2.1.3 The Classes of Problems

As example above shows, we can simulate a unique throw of a die by drawing twelve different cards. However, we cannot simulate a drawing from twelve different cards with a unique throw of a die. This example shows that some problems are harder than others. This gives the intuition that we can classify problems, by what a problem can be reduced to. There is 2 different main classes that we are interested in.

The Class of Polynomial Problems

The class of polynomial problems is the class of problem that can be solved, and the time consumption is polynomial by the size of the input.

The Class of NP-Complete Problems.

The class of NP (non-deterministic polynomial-time) problem is the class of problem where we can check, in polynomial time, if an answer is a solution of a problem. A problem $\mathcal{M}$ is an NP-Hard problem if there exists an NP-Hard problem $\mathcal{N}$ and if $\mathcal{N}$ can be reduced to $\mathcal{M}$. A problem is NP-Complete if it is NP and NP-Hard. Under the assumption that $P \neq NP$, the class of NP-Complete problems can be understood as the class of problems that can be solved at

best in exponential time by the size of the input. Basically, when a problem is NP-complete, no algorithm can be used in practice. The following example illustrates this point. Imagine that we have a problem and an algorithm that solves that problem in 2^n, where n represents the size of the input. If the size of the input is 60, this instance will be roughly solved in 1.10^{18} steps. Today's computers (a one Ghz CPU) can make 1000000000 steps per second, which means that this problem will be solved in 1.10^9 seconds, or 277777 hours, or 11574 days, or 31 years. Not anyone can afford to wait 31 years to have a question answered.

2.1.4 When a Problem is NP-Complete

Solutions exist whenever we need to solve an instance of an NP-Complete problem.

In practice, not every instance can appear and we use this to our advantage. An extreme case to consider is whether there is a finite number of instances; in that case, one can consider computing every instance (if temporally possible) and keeping the solutions somewhere. Another case to consider, is whether the instances have constraints; in that case, the constraints may allow to compute a solution faster than a general instance. This case can be illustrated with the problem of sorting a sequence of numbers. This problem is known to be solvable in $O(n \log n)$ time and constant space memory, where n is the size of the sequence for any sequence. However this problem is solved by the counting sort algorithm, which solves this problem in $O(mn)$ time where m is the maximal value and n is the size of the sequence but in $O(m)$ space. So, if we we give the constraint for the maximal element to be lower than $\log n$, the latter algorithm is the best algorithm.

Another possibility is the class of Fixed Parameter Tractable (noted FPT) problems. The class of FPT problems is a subclass of the NP-complete problems. A problem is FPT if its input can be split into two parts and can be solved in exponential time by the size of the first part of the input while polynomial time by the size of the second part of the input. When a problem is FPT, one can hope for a practical algorithm; indeed, if the first part of the input is known to be always very small then even if the algorithm is exponential, the algorithm would still be runnable in human time. There exist others classes of NP-Complete problems other than the class of FPT problems. A notable hierarchy of class is the W-hierarchy (formed by the classes $W[1]$, $W[2]$, ...). These classes are used to represent how hard it is to check whether an answer is a solution to a problem.

2.1.5 The Relation With This Thesis

The main problem studied in this thesis is NP-Complete. The goal of this section was to examine the intuition that using brute force to solve an instance of the problem may not be the best solution. We refer the reader to [21] We adopted the first point of the above paragraph, adding constraints to the input to define classes and find a polynomial algorithm to solve the problem for this class.

Chapter 3

Definitions

This chapter provides general definitions needed for this thesis.

3.1 Permutations

3.1.1 Definitions for Permutation

Definition 1. *A permutation of size n is a linear order of a ordered set of size n.*

Most of the time, the set used in Definition 1 is $\{1, 2, \ldots, n\}$, the set of the n first integers. We write a permutation as a word, the linear order of the permutation is encoded in that word: pick a letter in that word, any letter on its left is smaller than the picked letter and any letter on its right is larger than the picked letter. To work with permutation, we use notation borrowed from words: Given an element e, we refer by its index (usually represented by the letters i, j, ℓ) its position in the word. Moreover, given an i, $\pi[i]$ denotes the element with index i. For example, given that $\pi = 52143$, $\pi[5] = 3$ and the element 5 has 1 for its index.

3.1.2 The Representation of a Permutation.

Associating a permutation with a figure that we call a plot is helpful for understanding some algorithms. A plot represents a permutation by associating every element to a point of coordinate $(i, \pi[i])$. The plot reveals the linear orders: the set's linear order is the elements read from bottom to top and the permutation's linear order is the elements read from left to right. See Figure 3.1 for the plot of the permutation $\pi = 52143$.

3.1.3 Comparing the Elements of a Permutation

Value and Position of an Element

We need to consider both the set's linear order and permutation's linear order. When we talk about an element, we refer to its relative position in the set's linear order as its value, and its relative position in the permutation's linear order as its position. We usually use i, j and ℓ to represent an element's position and

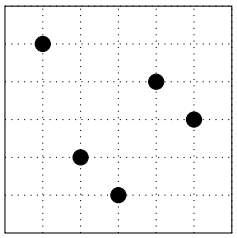


Figure 3.1 – The plot of the permutation $\pi = 52143$.

$\pi[i]$ to indicate the element's value at position i. For example, $\pi[3] = 1$ implies that the element of value 1 has position 3, and equivalently that the element at position 3 has value 1.

Comparing Two Elements by Value

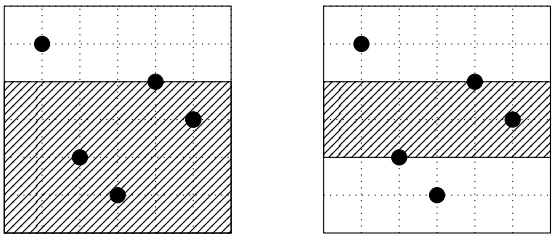
When an element e_1 is smaller (resp. larger) than an element e_2 by value (in the set's linear order), we say that e_1 is below (resp. above) e_2. Moreover, the element with the largest value is called the topmost element and the element with the smallest value is referred to as the bottommost element. For example, in the permutation $\pi = 52143$, 2 is below 4, the top most element is 5 and the bottommost element is 1. We can find the natural linear order of the set by reading the element from bottom to top, which explains why we say that an element is above or below another element.

Comparing Two Elements by their Positions

When an element e_1 is smaller (resp. larger) than an element e_2 by position (in the permutation's linear order), we say that e_1 is on the left (resp. right) of e_2. Besides, the element with the largest position is called the rightmost element and the element with the smallest position is called the leftmost element. For example, in the permutation $\pi = 52143$, 5 is on the left of 2, the leftmost element is 5 and the rightmost element is 3. We can find the linear order given by the permutation by reading the element from left to right, which explains why we say that an element is on the left or on the right of another element.

3.1.4 A Subsequence of a Permutation

It is useful to consider only part of a permutation. To do so, we are allowed to remove elements of that permutation. For instance, in the permutation 52143, one may want to consider only the elements with a value above 1; intuitively, the subsequence permutation is 5243. Formally speaking, a subsequence of a permutation π is any permutation that can be obtained from π by deleting some elements without changing the linear order of the remaining elements. Note that when subsequences of a permutation are being discussed, "of a permutation" is omitted.



(a) The elements of $\pi\{: 4\}$ are inside the dashed rectangle.

(b) The elements of $\pi\{2 : 4\}$ are inside the dashed rectangles.

Figure 3.2 – Omitting the element of $\pi = 52143$ by values.

Omitting Elements by Values with Bounds

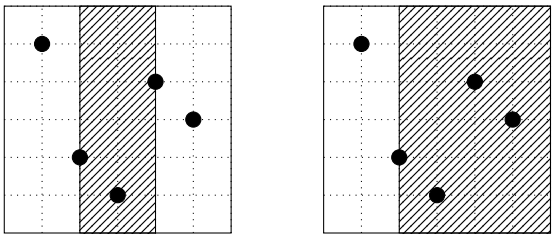
When we want to delete elements by their values, we write $\pi\{\text{inf} : \text{sup}\}$ to indicate that we are considering only the elements with values between inf and sup. We write $\pi\{\text{inf} :\}$ if we are considering elements with values above to inf (this is equivalent to writing $\pi\{\text{inf} : n\}$ if π has n elements) and $\pi\{: \text{sup}\}$ if we are considering elements below sup (this is equivalent to writing $\pi\{1 : \text{sup}\}$). For example, in the permutation $\pi = 52143$, $\pi\{: 3\} = 213$, $\pi\{2 :\} = 543$ and $\pi\{2 : 4\} = 243$. The disregarding of elements by their values can be understood by drawing two horizontal lines and omitting the elements between the two lines. See Figure 3.2 for an example.

Omitting Elements by Position with Bounds.

When we want to omit part of a permutation by the position of elements, we write $\pi[i, j]$ to indicate that we are considering the elements that are on the right of the element $\pi[i]$ and on the left of the element $\pi[j]$. In the same fashion, we write $\pi[i :]$ when we are considering the elements that are on the right of the element $\pi[i]$, (which is a short cut for $\pi[i : n]$) and $\pi[: j]$ when we are considering the elements that are on the left of the element $\pi[j]$, (which is a short cut for $\pi[1 : j]$). For example, in the permutation $\pi = 52143$, $\pi[3 :] = 143$, $\pi[: 4] = 5214$ and $\pi[2 : 4] = 214$. The disregard of elements by their positions can be understood by drawing two verticals lines and considering only the elements between the two lines. See Figure 3.3 for an example.

Rectangle of a Permutation

Generally speaking, whenever we consider a subsequence of elements of a permutation by selecting the elements by bounding them by value and/or by position, the subsequence is called a rectangle. The name comes from the plot of a permutation as, a simple way to see all the elements of a rectangle is to draw a rectangle with each edge defined with the bounds; every element that is in the rectangle is part of the subsequence. To stay constant with the general definition of a rectangle, we describe a rectangle as two points (A, B), where A is the bottom left corner and B is the top right corner. For instance, the rectangle $((2, 2), (4, 4))$ of the permutation 52143 is 24. See Figure 3.4 for an example.



(a) The elements of $\pi[2:4]$ are inside the dashed rectangle.

(b) The elements of $\pi[2:]$ are inside the dashed rectangle.

Figure 3.3 – Omitting element of $\pi = 52143$ by positions.

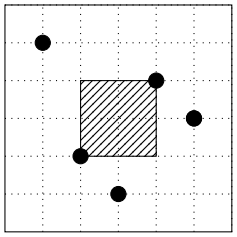


Figure 3.4 – The rectangle $((2,2),(4,4))$ of the permutation 52143.

We will be led to comparing two rectangles in this thesis, however not all pairs of rectangles are comparable: two rectangles are comparable if and only if they are not empty, the first rectangle's top right corner is to the left of and below the bottom left corner of the second rectangle, or its bottom right corner is on the left of and above the top left corner of the second rectangle. In the plot, two rectangles are comparable if and only if there do not exist any horizontal or vertical splices that contain both rectangles. Based on those constraints, when two rectangles are compared, one is always completely above/below the other and one is always completely on the right/left of the other.

Remark 2. *This thesis does not use the notation to bound the elements by value. Only the notation to bound the elements by position and the notation of a rectangle are used.*

3.1.5 Elements of a Subsequence

In relation to subsequences of a permutation and comparing elements, the following definitions of element are needed for this thesis.

Definition 3. *We call the left-to-right maximum (noted* LRMax*) of a permutation π at index i, the maximal element in the permutation $\pi[:i]$.*

Definition 4. *We call the left-to-right minimum (noted* LRMin*) of a permutation π at index i, the minimal element in the permutation $\pi[:i]$.*

Definition 5. *We call the right-to-left maximum (noted* RLMax*) of a permutation π at index i, the maximal element in the permutation $\pi[i:]$.*

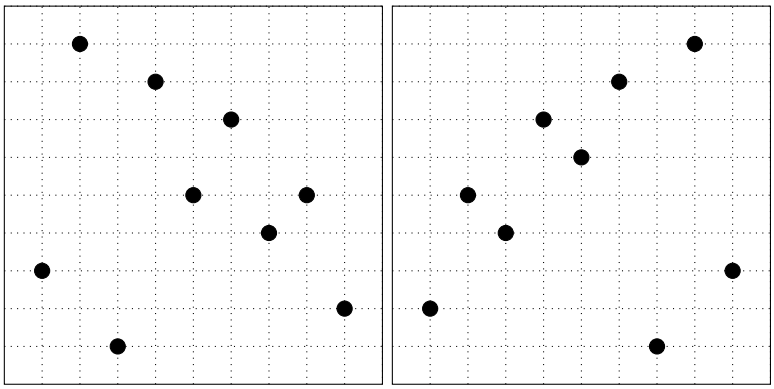


Figure 3.5 – The permutation 391867452 and its reverse 254768193.

Definition 6. *We call the right-to-left minimum (noted* RLMin*) of a permutation π at index i, the minimal element in the permutation $\pi[i:]$.*

3.2 Operations on Permutation

3.2.1 Transforming a Permutation

The following operations are not used directly in this thesis, but they allow to generalise result proved for one class of permutation to others classes. These operations create bijection between classes of permutations; as such, an algorithm that treats only one class of permutations can be used if we know how to transform a permutation of another class into a permutation of this class.

The Reverse of a Permutation

Definition 7. *The reverse of a permutation π, is the permutation τ, where the set's linear order is the same as π but the permutation's linear order is reversed.*

Reversing a permutation corresponds to reading the permutation from right to left. In a plot, this corresponds to a reflection on the y-axis. For example, the reverse of the permutation 391867452 is 254768193. See Figure 3.5.

The Complement of a Permutation

Definition 8. *The complement of a permutation π, is the permutation τ where the linear order of the permutation is the same as π but the linear order of the set is reversed.*

Moreover, when the elements are the first n integers, we rename the elements such that the elements are still read in their natural linear order, for comprehension purposes. In a plot, this corresponds to a reflection on the x-axis. A simple algorithm that computes the complement of permutation of size n, is putting at position i the element $n-i+1$. For example, the complement of the permutation 391867452 is 719243658. See Figure 3.6.

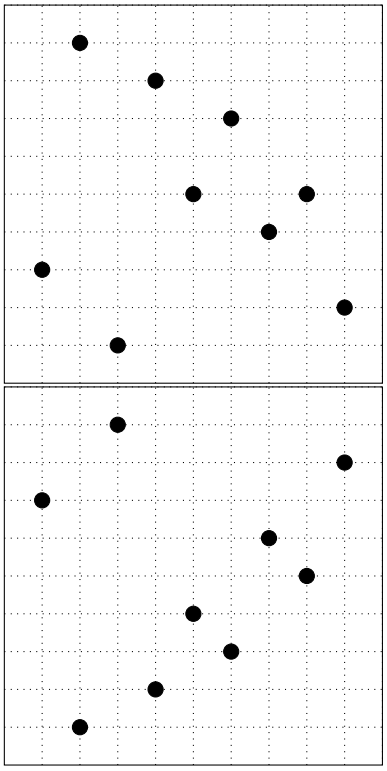


Figure 3.6 – The permutation 391867452 and its complement 719243658.

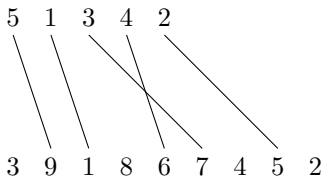


Figure 3.7 – A mapping from the permutation $\sigma = 51342$ to the permutation $\pi = 391867452$.

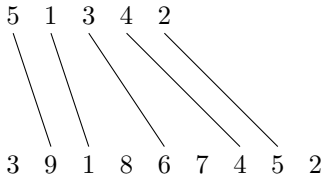


Figure 3.8 – An increasing mapping from the permutation $\sigma = 51342$ to the permutation $\pi = 391867452$.

3.2.2 Mapping

Definition 9. *Given a permutation σ of size k and a permutation π of size n, a mapping ϕ from σ to π is an injective function define as $\phi : \{1, 2, \ldots, k\} \to \{1, 2, \ldots, n\}$.*

A mapping from a permutation σ to a permutation π is an usual mapping: Each element of σ is associated with an element of π, and no element of π has two different inverse images. We map the elements with their positions in σ to the position of elements in π, as positions reveal an element's value. See Figure 3.7 for an example of mapping.

Remark 10. *The mapped elements form a permutation: The elements have their natural linear order and reading the mapping from left to right yields a total linear order.*

Definition 11. *Given a permutation σ of size k and a permutation π of size n, an increasing mapping is a mapping ϕ from σ to π such that if $i < j$ then $\phi(i) < \phi(j)$.*

For our purposes, we only need to consider increasing mapping. In other words, if two elements are increasing by position then the positions of their images by the mapping are also increasing. As we map elements by their positions, we only require the mapping to be increasing. In the Figure 3.7 this corresponds to not having crossing lines. See Figure 3.8 for an example of an increasing mapping.

We write increasing mapping as a word, where the i^{th} letter is the image of i by the mapping in π : $\phi = \pi[\phi(1)]\pi[\phi(2)] \ldots \pi[\phi(k)]$. We do not loose any information by doing so, as the inverse of a letter is found in the position of that letter. For example, the mapping of Figure3.8 is written as 91645. An increasing mapping can be accomplished in two ways: If the permutation is written as a word, making an increasing mapping consists of selecting as many elements in

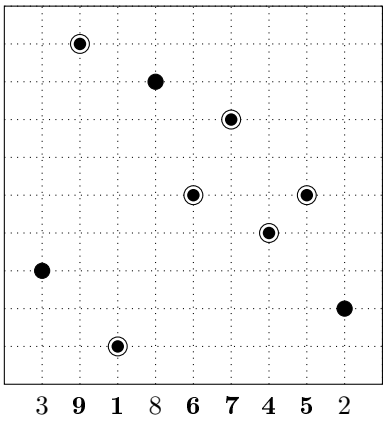


Figure 3.9 – Two ways to represent the mapping 916745 in 391867452: The first one is to select the points and the second is to bold the selected elements in the permutation.

π as there are elements in σ. In other words, we want a subsequence of π of the same size as σ. For this reason, we represent a mapping by a subsequence. If we have the plot of π, an increasing mapping consists of selecting as many points in the plot of π as there are elements in σ. See Figure 3.9 for an example.

3.2.3 Reduction of a Permutation

The set of a permutation is not always the set $\{1, 2, \ldots, n\}$. For example, the permutation given by the images of a mapping and the permutation given by a rectangle are permutations over a set which is not the first integers. We can also imagine a permutation over any set, as long as we have a linear order over this set. We called those permutations "non-canonical".

The first case of non-canonical permutations described above turns out to be a problem: When reading the permutation from left to right, we cannot guess the relative order (in the linear order by value) of an element until we read all of the elements. For example, in the permutation 91645, we cannot know that 6 is the second largest element from having read 916.

It should also be noted that we only need the linear order between the value of the elements. So the actual values of the elements do not matter as long as this linear order is preserved.

It is in our best interest to work with a canonical permutation, that is a permutation of size n with domain $\{1, 2, \ldots, n\}$. As we do many operations on a permutation, it is better to spend some time computing a permutation that is easy to handle and understand rather than using a "hard" permutation and spending time to make operations on it. We call the operation that goes from a non-canonical permutation to an canonical permutation a reduction.

Definition 12. *Given a permutation π on set $\{e_1, e_2, \ldots, e_n\}$ such that $e_1 < e_2 < \ldots < e_n$ the reduction of π, noted as reduction(π), is the permutation in which we replace every element e_i by i.*

For example, 91645 has for natural linear order $1 <_y 4 <_y 5 <_y 6 <_y 9$ so

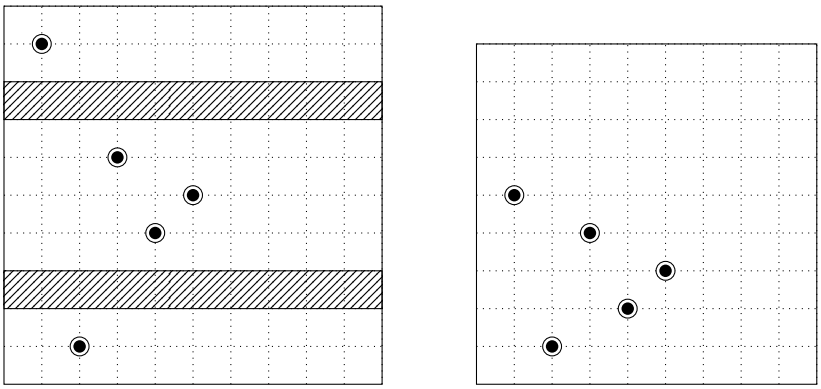


Figure 3.10 – The permutation 91645 and the lines to remove on the left and its reduction on the right.

1 becomes 1, 4 becomes 2, 5 becomes 3, 6 becomes 4 and 9 becomes 5. This yields us the permutation 51423 and thus, $reduction(91645) = 51423$.

In a plot of a permutation this corresponds to removing every empty line of the figure. See Figure 3.10 for the reduction of 91645.

3.2.4 Direct Sums and Skew Sums

Definition 13. *Given a permutation π_1 of size n_1 and a permutation π_2 of size n_2, the direct sum of π_1 and π_2 is defined by:*

$$\pi_1 \oplus \pi_2 = reduction(\pi_1[1]\pi_1[2]\dots\pi_1[n_1](\pi_2[1]+n_1)(\pi_2[2]+n_1)\dots(\pi_2[n_2]+n_1)).$$

The direct sum of two permutations is found by taking the left permutation's elements and adding the right permutation's elements such that the latter are to the right of and above the former. In other words, we take the elements of the left permutation and then add the elements of the right permutation shifted by the size of the left permutation. For example, $312 \oplus 3214 = 321(3+3)(2+3)(1+3)(4+3) = 3126547$.

From the plots of the permutations, this corresponds to taking the points of the right permutation and putting them to the right and above the left permutation. See Figure 3.11 for the direct sum of 312 and 3214.

Definition 14. *Given a permutation π_1 of size n_1 and a permutation π_2 of size n_2, the skew sum of π_1 and π_2 is defined by:*

$$\pi_1 \ominus \pi_2 = (\pi_1[1]+n_2)(\pi_1[2]+n_2)\dots(\pi_1[n_1]+n_2)\pi_2[1]\pi_2[2]\dots\pi_2[n_2].$$

The skew sum of two permutations is formed by taking the right permutation's elements and adding the left permutation's elements such that latter are to the left of and above the former. In other words, take the elements of the right permutation and then add on the left the elements of the left permutation shifted by the size of the right permutation. For example, $312 \ominus 3214 = (3+4)(1+4)(2+4)3214 = 7563214$.

From the plots of the permutations, this corresponds to taking the points of the left permutation and putting them to the left and above the right permutation. See Figure 3.12 for the skew sum of 312 and 3214.

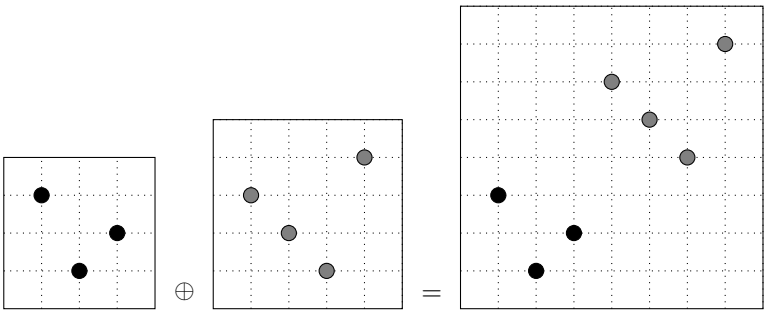


Figure 3.11 – The direct sum of permutations 312 and 3214: $312 \oplus 3214 = 3126547$.

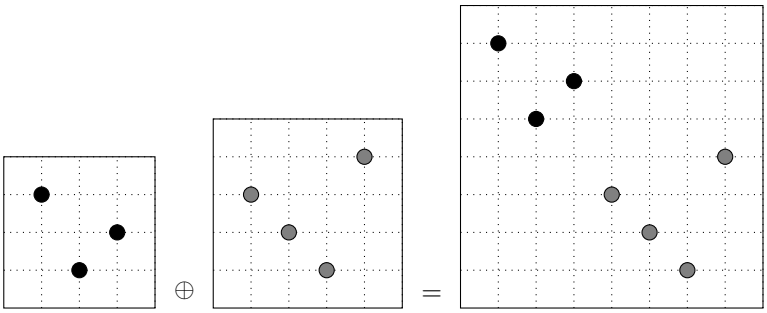


Figure 3.12 – The skew sum of permutations 312 and 3214: $312 \ominus 3214 = 3126547$.

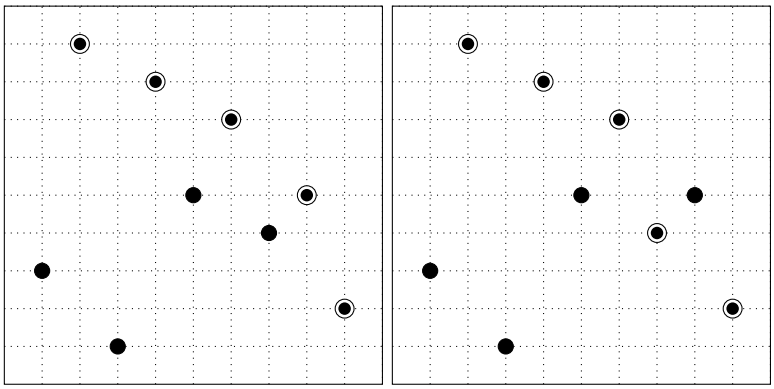


Figure 3.13 – Two occurrences of 54321 in 391867452

3.3 The Notions of Occurrence and Avoidance

3.3.1 Order isomorphism

Two definitions of order isomorphism are presented first, as they are needed to the definition of an occurrence. Intuitively, two permutations are order isomorphic if their linear orders are equivalent. As the natural linear order of the elements remains unchanged, this corresponds to asking for the second linear orders to be equivalent.

Definition 15. *Given two permutations σ and π, σ and π are order isomorphic if for all i, j, $i \neq j$, $1 \leq i, j \leq n$, $\sigma[i] < \sigma[j]$ if and only if $\pi[i] < \pi[j]$.*

From the plot of two permutations, one can guess if those two permutations are order isomorphic if and only if the points of the elements of the first permutation are in the same disposition as the points of the elements of the second permutation without regard to the distance between the points.

Definition 16. *Given two permutations σ and π, σ and π are order isomorphic if $reduction(\sigma) = reduction(\pi)$.*

3.3.2 An Occurrence

The notion of an occurrence is central for the PPM problem as it is needed to define the problem.

Definition 17. *An occurrence of σ in π is a subsequence s of π such that $reduction(s) = reduction(\sigma)$.*

If π contains an occurrence of σ, then we say that σ occurs in π, which is denoted usually as $\sigma \prec \pi$; otherwise, we say that π avoids σ. For example, 98752 and 98743 are occurrences of 54321 in 391867452, so 54321 occurs in 391867452, however, 391867452 avoids 1234, as there is no increasing subsequence of size 4. See Figure 3.13 for the plot of the example.

3.4 Set of Pattern Avoidance

3.4.1 Definition

The notion of avoidance allows us to define a class of permutations $\mathscr{C}$: Given a set $\mathcal{S}$ of permutations, the class $\mathscr{C}$ corresponds to all permutations that do not contain any permutation in $\mathcal{S}$. This class is denoted by $Av(\mathcal{S})$ or $\mathcal{S}$-avoiding; only the latter is used in this thesis. For the first notation, the convention is to remove the brackets of the set. For the second notation, when the set $\mathcal{S}$ contains a unique permutation, we do not put parentheses.

Remark 18. *This thesis only considers cases in which $\mathcal{S}$ is finite.*

3.4.2 The Utilities of Classes of Permutation

The usefulness of permutations classes seems to be limited. It appears that some sets of permutations defined by their structures can be seen as sets of permutations by avoidance. This has generate interest in the field of combinatorics, especially in enumerating a certain classes and finding a transformations of a permutation into other objects. For example, the 321-avoiding set is known to be a set of permutations that can be partitioned into two increasing subsequences.

3.4.3 Comparing Avoiding Sets

Some classes of permutations are more impactful for our purposes than some others. Indeed, knowing that a problem is NP-complete when considering permutations of a given class and that this class is included in another one implies that the problem is also NP-complete when considering permutations of the second class. For this purpose we need to be able to compare avoiding sets of permutations. Comparing avoiding sets can be easy: Avoiding classification forms a downset. The set of all avoiding sets is partially ordered for the inclusion. Based on what is more important, one can decide whether two sets are comparable and in this case which one is included in the other. To decide whether $\mathcal{S}_1$-avoiding is included in $\mathcal{S}_2$-avoiding, one has to decide whether every permutation of $\mathcal{S}_1$ occurs in at least one permutation in $\mathcal{S}_2$. Indeed, if a permutation π_1 avoids a permutation π_2, and π_2 occurs in π_3, then π_1 avoids π_3. This is because any occurrence of π_3 would contain an occurrence of π_2, which implies that π_1 would contain an occurrence of π_2. The downside is that one has to decide whether a permutation occurs in another permutation, which is our main problem. This approach only allows us to compare a few sets, but it is the most direct option and is sufficient for our purposes.

Another possible approach is to use the structure of the two sets. For example, the set of 213-avoiding permutations is the set of permutations that are sortable by a unique stack (See [38]) and the set of $(2413, 3142)$-avoiding permutations is the set of permutations that are sortable by an arbitrary number of stack (See [12]). By definition, the set of 213-avoiding permutations is included in the set of $(2413, 3142)$-avoiding permutations.

3.5 The Permutation Pattern Matching Problem

This section presents the main problems studied in this thesis.

3.5.1 The Permutation Pattern Matching Problem

We define the PPM problem as the following decision problem:

Problem 19. *Given a permutation σ of size k and a permutation π of size n, $k \leq n$, the PPM problem asks whether σ occurs in π.*

By analogy to pattern matching for words, we call σ the pattern and π the text.

3.5.2 The Permutation Bivincular Pattern Matching Problem

A more general problem than the PPM problem adds constraints to the occurrence. In this thesis, we study a generalisation known as permutation bivincular pattern (noted BVP) matching.

A BVP $\widetilde{\sigma}$ is a pattern permutation where we add constraints on the occurrence. As such a BVP $\widetilde{\sigma}$ of size k can be visualized as a "regular" permutation of size k and a set of constraints. With this visualization in mind, given a BVP $\widetilde{\sigma}$, we let σ denote the "regular" permutation. Intuitively, for a sequence to be an occurrence of $\widetilde{\sigma}$, it must be an occurrence for σ and respect the given constraints.

Definition 20. *An occurrence of $\widetilde{\sigma}$ in π is a subsequence s of π such that $reduction(s) = reduction(\sigma)$ and the constraints of $\widetilde{\sigma}$ are respected.*

We differentiate two types of constraints. The first type is constraint on the values of an occurrence. We can ask an element to be consecutive to another one, to be the bottommost element or the topmost element.

Example 21. *Given the pattern* 213 *with the condition that the elements corresponding to* 2 *and* 3 *are consecutive,* 729 *is not an occurrence because* 7 *and* 9 *are not consecutive but* 728 *is an occurrence.*

Example 22. *Given the pattern* 213 *with the condition that the element corresponding to* 1 *is the bottommost element,* 729 *is not an occurrence because* 2 *is not the bottommost element but* 718 *is an occurrence.*

It should be noted that some constraints are not "compatible" with some patterns, in the sense that no sequence can be an occurrence for this BVP . For example, given the pattern 213 with the condition that the element 1 is the topmost element will never hold an occurrence because if the element representing 1 is the topmost element, no element can represent the 2 or 3.

The second type of constraint is on the position (in the text) of the element of an occurrence. We can ask an element to be next to another one in the text, to be the leftmost element or the rightmost element.

Example 23. *Given the pattern* 213 *with the condition that the elements corresponding to* 2 *and* 1 *are next to each other in the text and given the text* 43251, 425 *is not an occurrence because* 4 *and* 2 *are not next to each other (in the text) but* 435 *is an occurrence.*

Example 24. *Given the pattern* 213 *with the condition that the element corresponding to* 2 *is the leftmost element and in the text and given the text* 43251, 325 *is not an occurrence because* 3 *is not the leftmost element in the test but* 425 *is an occurrence.*

The same remark about the "compatibility" of a pattern and constraints also holds with the constraints on position.

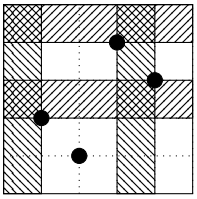
We give here the notation of BVP , this notation is useful because we can create a BVP from a "regular" pattern by adding constraints such that the constraints and the pattern are always compatible. We denote a BVP $\widetilde{\sigma}$ of size k in a two-line notation: the top row is $12 \dots k$ and the bottom row is a permutation $\sigma_1\sigma_2 \dots \sigma_k$. We have the following conditions for the top and bottom rows of σ (Definition 1.4.1 in [37]):

- If the bottom line of $\widetilde{\sigma}$ contains $\underline{\sigma_i\sigma_{i+1} \dots \sigma_j}$ then the elements corresponding to $\sigma_i\sigma_{i+1} \dots \sigma_j$ in an occurrence of $\widetilde{\sigma}$ in π must be adjacent, whereas there is no adjacency condition for non-underlined consecutive elements. Moreover if the bottom row of $\widetilde{\sigma}$ begins with ${}_{\llcorner}\sigma_1$ then any occurrence of $\widetilde{\sigma}$ in a permutation π must begin with the leftmost element of π, and if the bottom row of $\widetilde{\sigma}$ ends with $\sigma_{k_{\lrcorner}}$ then any occurrence of $\widetilde{\sigma}$ in a permutation π must end with the rightmost element of π.

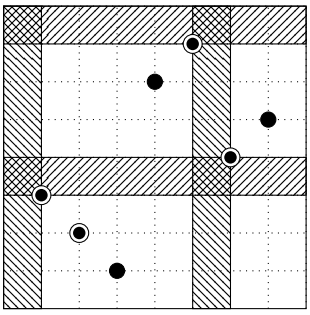
- If the top line of $\widetilde{\sigma}$ contains $\overline{i\, i+1 \dots j}$ then the elements corresponding to $i, i+1, \dots, j$ in an occurrence of $\widetilde{\sigma}$ in π must be adjacent in values, whereas there is no value adjacency restriction for non-overlined elements. Moreover, if the top row of $\widetilde{\sigma}$ begins with ${}^{\ulcorner}1$ then any occurrence of $\widetilde{\sigma}$ in a permutation π must contain the smallest element of π, and if top row of $\widetilde{\sigma}$ ends with $k^{\urcorner}$ then any occurrence of $\widetilde{\sigma}$ in a permutation π must contain the largest element of π.

For example, let $\widetilde{\sigma} = \overline{{}^{123}_{\llcorner 213 \lrcorner}}$. If $\pi_i\pi_j\pi_\ell$ is an occurrence of σ in $\pi \in S_n$, then $\pi_i\pi_j\pi_\ell = (x+1)x(x+2)$ for some $1 \leq x \leq n-2$, $i = 1$ and $\ell = n$. For example 316524 contains one occurrence of $\widetilde{\sigma}$ (the subsequence 324), whereas 42351 contains an occurrence of pattern σ but not the BVP $\widetilde{\sigma}$.

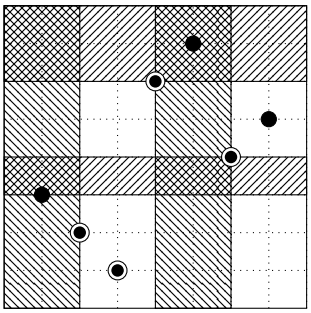
We represent BVP (as well as occurrences of BVP in permutations) by plots. Such plots consist of sets of points at coordinates $(i, \sigma[i])$ drawn in the plane together with forbidden regions that represent adjacency constraints. A vertical forbidden region between two points denotes the fact that the occurrences of these two points must be consecutive in positions. It can also represents the condition on the leftmost and rightmost element, in which case we draw a forbidden area between the leftmost element and the "left border" of the plot and the rightmost element and the "right border" of the plot, respectively. In a similar approach, a horizontal forbidden region between two points denotes the fact that the occurrences of these two points must be consecutive in values. It can also represents the condition on the bottommost and topmost element, in which case we draw a forbidden area between the bottommost element and the "bottom border" of the plot and the topmost element and the "top border" of the plot, respectively. Given a permutation π and a BVP $\widetilde{\sigma}$, the rationale for introducing these augmented plots stems from the fact that π contains an occurrence of a BVP $\widetilde{\sigma}$ if there exists a set of points in the plot of π that is order-isomorphic to σ and if the forbidden areas does not contain any points (see Figure 3.14).



(a) The BVP $\widetilde{\sigma} = {}^{1\overline{2}\overline{3}4\urcorner}_{\llcorner 21\underline{43}}$.



(b) An occurrence of $\widetilde{\sigma}$ in 3216745.



(c) An occurrence of σ in 3216745 but not $\widetilde{\sigma}$ because the points $(1,3)$ and $(5,7)$ are in the forbidden areas.

Figure 3.14

Problem 25. *Given a BVP permutation σ of size k and a permutation π of size n, the PPM problem asks whether σ occurs in π.*

Chapter 4

The Permutation Pattern Matching Problem Paradigm

4.1 General Case of the Problem

The first result for the PPM problem concerns the enumeration of sequences of integers that can be partitioned into two decreasing subsequences, which are the 123-avoiding permutations.s This result can be found in [8], which was written in 1915 by MacMahon. The problem of finding the longest increasing (or decreasing) subsequence, which can be used to determine whether an increasing permutation $12\dots n$ occurs in a permutation, was first studied in 1961 in [52] in a Schensted's article.

The seed of PPM is found in the notion of avoidance presented in [38], where Knuth show that stack-sortable permutations can be defined as permutations that avoid 213. This new way of characterisation provided a powerful tool to the combinatorics field. The combinatorics used the notion of avoidance to enumerate permutations and create new generating functions. This also led to the creation of new bijections between permutations of a certain class and other objects, some of which, are useful in the current study. In this thesis we are only interested in the algorithmic part of the PPM problem. A more combinatorial overview of PPM can be found in [55] written by Vatter whereas, a more general overview can be obtained from Kitaev's book [37].

The first "official" appearance of PPM occurred at the 1992 SIAM Discrete Mathematics meeting, where Wilf asked the community whether there exists an algorithm that solve the PPM problem in a non-exponential time. Bosen, Buss and Lubiw presented an answer in 1993 in [18], where they provided proof that the problem is NP-complete by reducing the boolean satisfiability problem to PPM. However, the problem is solvable by a brute force algorithm in polynomial time if the size of the pattern is fixed. This algorithm was studied by Albert et al. in [4], in which they provide a $O(n^k)$ running time algorithm where n is the size of the text and k is the size of the pattern. The best result up to this date is attributed to Ahal and Rabinovich in [1] where they give an algorithm running in $O(n^{0.47k+o(k)})$.

To solve the PPM problem, the algorithm using brute force checks all subsequences of size k in π, which results in a $O(\binom{n}{k}k)$-time algorithm. This is

because there are $\binom{n}{k}$ different subsequences. Moreover, we can check in linear time in k whether a subsequence is an occurrence of σ. This algorithm was improved by Bruner and Lackner in [23], which present an algorithm running in $O(1.52^n nk)$. One of the more important results is one found in [32] by Marx and Guillemot, who show that PPM is FPT by the size of σ by giving an $n.2^{O(k^2 \log k)}$-time algorithm. In the same veins as the last result PPM is FPT by the number of alternating runs in π (see [23]), especially an algorithm running in $O(1.79^{run(\pi)}nk)$-time exists.

All of the results at the time of writing are as follows:

- PPM is FPT by the size of σ, especially one can decide whether σ occurs in π running in $n.2^{O(k^2 \log k)}$ (See [32]).
- PPM is FPT by the number of runs in π, especially one can decide whether σ occurs in π in $O(1.79^{run(\pi)}.n.k)$ (See [23]).
- One can decide whether σ occurs in π in $O(n^{0.47k+o(k)})$ time for σ with a size smaller than k (See [1]).
- One can decide whether σ occurs in π in $O(1.52^n nk)$ time (See [23]).

4.2 Adding Constraints to the Permutation Pattern Matching Problem

The general case of the PPM problem is NP-complete, but all hope is not lost. Some special cases can be solved polynomially. A strategy is to limit the possible inputs. Because in the PPM problem, the only inputs are the permutation π and σ, we differentiate 3 class of constraints: if we constraint σ, if we constraint π and σ or if we constraint π. The restrictions usually consist of permutations that belong to certain classes or in a restriction on the size of the permutations. Another strategy consists of adding restrictions to an occurrences of σ; whereas these restrictions are added by considering a more general definition of a permutation pattern as a BVP or mesh pattern, we prefer to interpret them as a restriction on the output occurrence and not as a generalisation.

4.3 The Permutation Pattern Matching Problem with Specific Classes of Permutations

4.3.1 Separable Permutations

One of the most prominent classes of permutations studied in permutation is the class of $(2413, 3142)$-avoiding permutations, which are also called separable permutations. The first paper on PPM [18], which proved that the problem is NP-complete, presented a positive result for the special case in which σ is a separable permutation. The authors presented an $O(kn^6)$ time algorithm by exploiting the tree structure of a separable permutation. Ibarra latter improved this algorithm in [34] to a $O(kn^4)$-time and $O(kn^3)$-size algorithm. In this thesis we improve this algorithm to $O(n^3 \log k)$-space algorithm by a simple

observation. Due to the tree structure of separable permutations, the PPM problem on separable permutations can be reduced to a tree inclusion problem. Many results exist for the tree inclusion problem and special cases like a tree with few leaves or tree with small depth are studied. The best algorithm is by Bille and Gørtz [17] who gave a $O(n_T)$ space and

$$O\left(\min\left\{\begin{array}{l} l_{T'}\, n_T \\ l_{T'}\, l_T\, \log\log n_T + n_T \\ \frac{n_T\, n_{T'}}{\log n_T} + n_T \log n_T \end{array}\right\}\right)$$

time algorithm, where T denotes the tree of π, T' denotes the tree of σ n_T (resp. $n_{T'}$) denotes the number of nodes of T (resp. T') and l_T (resp. $l_{T'}$) denotes the number of leaves of T (resp. T'). Whereas these algorithms are specialised for trees and somehow difficult to implement, we offer, in this thesis (in Chapter 6), a simpler approach to the PPM problem when σ and π are separable. It should be noted that this algorithm can be used to solve the PPM problem whenever the pattern avoids a pattern of size 3, except for the pattern 321 and 123. Indeed, all other patterns of size 3 are a sub-class of separable permutations.

4.3.2 Increasing Permutations

Another class of PPM problems was solved early, even before it officially existed. The increasing permutations can be characterised as the 21-avoiding permutations and for a given size k there is a unique (canonised) increasing permutation, namely the permutation $12\ldots k$. As noted previously, the problem of finding the longest increasing subsequence can be used to test whether the permutation $\sigma = 12\ldots k$ occurs in π: compute the size of the longest increasing subsequence in π, obviously if this size is equal to or larger than k then σ occurs in π. See [15] for the best algorithm to date for computing the longest increasing subsequence in a permutation.

4.3.3 321-Avoiding Permutations

The first result related to the 321-avoiding permutation (or by symmetry the 123-avoiding permutations) is provided by Guillemot and Vialette in [33] where they give an algorithm running in $O(kn^{4\sqrt{k}+12})$ time for computing this problem. It is worth mentioning that the authors show that, the PPM problem with 321-avoiding coloured permutations is NP-complete, in the coloured version of PPM problem, each element is associated with a colour. The problem seeks to find an occurrence such that the i^{th} element of the occurrence and σ have the same colour. Another positive result is recorded in the same paper: An algorithm that solves the PPM problem in $O(k^2n^6)$ time when both σ and π are 321-avoiding. This result was improved in 2015 by Albert et al. in [3] which offers an $O(nm)$-time algorithm. The argument presented is that the set of increasing matchings that match an element of the first (resp. second) increasing subsequence of σ to an element of the first (resp. second) increasing subsequence of π forms a lattice. Then they use two properties of a lattice: The set is partially ordered and a unique minimal and maximal element exists. To find an occurrence, the algorithm creates an increasing sequence of matching, starting from the minimal

matching. The algorithm also assures that we choose a good matching, such that, if we are at the maximal matching and we did not find an occurrence then no occurrence exists. Finally, Jelínek and Kynčl in [35] prove that PPM problem is NP-complete when the pattern avoids 321 and when the text avoids 4321. They also attain a positive result by showing that any PPM problem can be solved in polynomial time if the pattern is in the proper subclass of 321-avoiding permutation.

4.3.4 Skew-Merged Permutations

In [3], Albert et al. also present an algorithm to decide whether σ occurs in π when both σ and π are Skew-merged permutations. Skew-merged permutations are the $(3412, 2143)$-avoiding permutations. These permutations can be understood as the permutations that can be partitioned into one increasing subsequence and one decreasing subsequence. Jelínek and Kynčl in [35] also provide a positive result over a proper subclass of skew-merged permutations, especially the $(3412, 2143, 3142)$-avoiding permutations. They prove that the PPM problem can be solved polynomially when the pattern is in a proper subclass of this class. It should be noted that this is the first positive PPM result for a class that is not a subclass of separable permutations.

4.3.5 The Case of Fixed Permutations

Finally, whenever a pattern's size is fixed, we can decide the PPM problem polynomially. We can improve this result even further for all permutations with four elements. Indeed, we obtain an $O(n^{1.88})$-time algorithm by applying the general algorithm. Albert et al. in [4] provide an algorithm to solve this problem in $O(n \log n)$ time. They improve this even further for the special cases of 4312, which can be solved in linear time.

4.3.6 Wedge Permutations

In this thesis (in Chapter 7) we focus on wedge permutations, that are the permutations that can be split into a increasing and a decreasing subsequence such that the elements of the decreasing sequence are above the elements of the increasing sequence. We study cases in which σ is a wedge permutation and σ and π are wedge permutations. We show that when only σ is a wedge permutation, we can decide whether a permutation σ of size k occurs in a permutation π of size n in $O(\max(kn^2, n^2 \log\log n))$-time and $O(n^3)$-space. We can also determine whether σ occurs in π in linear time when σ and π are wedge permutations.

4.4 Generalisation of Patterns

4.4.1 Complexity with Mesh or Bivincular Patterns

The PPM problem when considering permutation as pattern are a special case of the PPM problem when considering BVP . This can be generalized even more by considering mesh pattern. Mesh pattern generalises Bivincular pattern, whereas in bivincular pattern the forbidden areas are a entire columns or line of the plot,

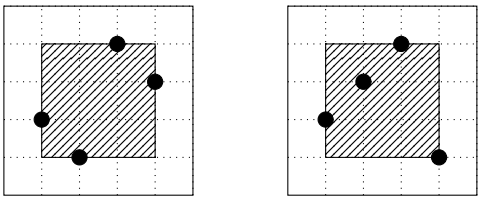


Figure 4.1 – The plots of the boxed mesh pattern 2143 and 2341.

forbidden areas in mesh pattern correspond to one square of the plot. For this reason, the PPM with BVP and mesh patterns is NP-complete. It should be noted that the problem is not FPT but is W[1]-hard (See [22] section 5 for the proof). In other words, we cannot hope for FPT algorithms.

4.4.2 Consecutive Pattern

Some positive results come with stronger constraints on the pattern. Consider the consecutive pattern, which is a special case of BVP where all the elements are underlined. In other words, we are looking for sequence (that is, a subsequence in which all elements are contiguous) in π that is isomorphic to σ. This problem can be solved in linear time, as shown by Kubica et.al. in [40].

4.4.3 Boxed Mesh Pattern

Another strong constraint is when we want to decide whether an occurrence exists and is contained in a rectangle and the inside of the rectangle contains only the occurrence's elements (see Figure 4.1 for an example). These patterns are the so-called boxed mesh patterns and were examined in [11] by Avgustinovich, Kitaev and Valyuzhenich.

The problem of detecting a boxed mesh permutation was successfully studied and an algorithm running in $O(n^3)$-time is shown in [22].

4.5 State of the Art

This section summarize the results of the PPM problem when considering certain classes for the text and/or the pattern in the tables 4.2, 4.3 and 4.4 and of the PPM problem when considering different type of pattern are summarize in the table 4.5.

4.6 Point of View of this Thesis

This thesis adopted the usual strategy of adding constraints to the input and examine what happens when constraints are added to the occurrences, especially when BVP are considered. This proved to be successful, as we found polynomial algorithms for some classes; the downside was an increasing time and space consumption compared to their counterpart with permutation pattern. We will show, in what follows, that

Condition on σ	Complexity[1]	Reference
Permutation of size 4	$O(n \log n)$-time	[6]
21-avoiding	$O(n \log \log n)$-time	[15]
321-avoiding	$O(kn^{4\sqrt{k}+12})$-time	[33]
Proper subclass of 321-avoiding	Polynomial algorithm	[35]
Proper subclass of $(2143, 3412, 3142)$-avoiding		
4312	Linear algorithm	[42]
Separable permutations	$O(kn^4)$-time $O(\log kn^3)$-space	Section 6.2
Wedge permutations	$O(\max(kn^2, n^2 \log \log n))$-time $O(n^3)$-space	Section 7.3

[1] n and k refer to the size of the text and the pattern, respectively.

Figure 4.2 – Results for the PPM problem when considering class for the pattern.

- One can decide in $O(kn^4)$ time and $O(kn^3)$ space whether a wedge BVP occurs in π, see Section 6.6.
- One can decide in $O(kn^6)$ time whether a separable BVP occurs in π, see Section 7.4.

Condition both on σ and π	Complexity[1]	Reference
Skew-merged permutations	$O(nk)$-time	[3]
Separable Permutations	$O(n_T)$-space $O\left(\min\left\{\begin{array}{l} l_{T'}\, n_T \\ l_{T'}\, l_T\, \log\log n_T + n_T \\ \frac{n_T\, n_{T'}}{\log n_T} + n_T \log n_T \end{array}\right\}\text{-time}^2\right)$	[16]
321-avoiding	$O(nk)$-time	[33]
Wedge permutations	Linear algorithm	Section 7.2

[1] n and k refer to the size of the text and the pattern, respectively.
[2] where T (resp. T') is the tree representing π (resp. σ), n_T (resp. $n_{T'}$) denotes the number of node of T (resp. T') and l_T (resp. $l_{T'}$) denotes the number of leaves of T (resp. T').

Figure 4.3 – Results for the PPM problem when considering the same class for the pattern and the text.

Condition on σ	Condition on π	Complexity	Reference
$(2143, 3412, 3142)$-avoiding	decomposed into two increasing subsequences and one decreasing subsequence	NP-complete	[35]

Figure 4.4 – Result for the PPM problem when considering different classes for the text and the pattern.

Type of pattern	Complexity	Reference
BVP	$W[1]$-Hard	[22]
Boxed mesh pattern	$O(n^3)$-time[1]	
Consecutive pattern	Linear algorithm	[40]

[1] n refers to the size of the text.

Figure 4.5 – Results for the PPM problem when considering different type of pattern.

Chapter 5

Related problems

As in the pattern matching for words, some other problems naturally arise in relation to PPM problem. This chapter presents a non-exhaustive list of related problems.

5.1 Longest Subsequence

The longest subsequence (noted LS) problem seeks to find the LS in a permutation belonging to a given set. The LS problem is a special case of the longest common subsequence (noted LCS) problem, which is explained in this thesis. The LS problem has mainly been studied to solve the LCS.

Problem 26. *Given a class of permutations $\mathcal{S}$, a permutation π and an integer m, the LS problem asks whether there exists an occurrence of a permutation σ in π of size m such that $\sigma \in \mathcal{S}$.*

5.1.1 Longest Increasing Subsequence

A similar problem exists in a sequence of integers, namely the longest increasing subsequence (noted LIS) problem. Equivalently the longest decreasing subsequence (noted LDS) problem exists. The LIS problem seeks to find in a sequence of integers the LIS. There exists many methods to compute a LIS of a sequence, to cite just a few:

- A method use the representation of a sequence of integers called Young tableau (See [29]). In a Young tableau, a sequence is represented in a two-dimensional array, the size of the LIS is the size of the first row (See [52]). It should be noted that the size of the LDS is the size of the first column.
- An algorithm use tractable strategy: For each suffix of the sequence the size of LIS is computed using the following relation: A LIS of a suffix corresponds to a LIS of the current suffix without its first letter concatenated to the first letter (if possible). This algorithm run in $O(n^2)$ where n is the size of the sequence.

- The best algorithm runs in $O(n \log n)$. Its principle uses patience sorting, the idea being to construct a sequence of decreasing subsequences. To do so one has to read the sequence of integers from left to right. For each number read, puts it at the end of the latest created decreasing subsequence if the subsequence is still decreasing; otherwise create a new subsequence. Once the sequence of decreasing subsequences is created, one can find an LIS by picking one element of each decreasing subsequence. To this end, each integer must know which element is on its left (in the original sequence), below it and on the decreasing previous subsequence (See [7]).

Remark 27. *One can easily adapt these algorithms to find the LDS.*

In terms of permutations, the LIS can be understood as the longest 21-avoiding subsequence. Moreover, the (canonised) permutations have one advantage over a sequence of integers: We know every element and are sure that each one appears only once. Bespamyatnikh and Segal use this knowledge in [15], where they provide an algorithm to compute the LIS of the longest 21-avoiding subsequence in $O(n \log \log n)$, where n is the size of the permutation. Moreover, in addition to having LIS, we have the LIS for each contiguous subpermutation.

5.1.2 Longest Alternating Subsequence

The class of permutations is also thoroughly studied in relation to the LAS. An alternating permutation is a permutation that features elements in an alternating order: Every contiguous element is of a different order than the previous and next pairs (if any). A (folkloric) simple algorithm is to construct two alternating subsequences by adding the first element of the permutation and every element that is a peak (an element $\pi[i]$ such that $\pi[i-1] < \pi[i] > \pi[i+1]$) or a valley (an element $\pi[i]$ such that $\pi[i-1] > \pi[i] < \pi[i+1]$) in an alternating manner, starting with peak. The second alternating subsequence follows the same scheme: The first element of the permutation and every elements that is a peak or a valley is added in an alternating manner, starting with a valley. The longest alternating subsequence is the LS between the two subsequences created.

Remark 28. *Alternating permutations are sometimes called zigzag or up/down permutations.*

5.1.3 State of the Art

The results about longest subsequence are summarize in the table 5.1

5.2 Longest Common Subsequence

The LCS problem is the general case of LS. Instead of looking for the LS in one permutation, we look for the LS that occurs in each permutation of a set of permutations. The LCS problem is studied over words and one result is useful to us. As such, we start by introducing the problem of LCS for words, where a word is a sequence on a finite set. This problem is sometimes referred to as the consensus problem.

Type of subsequence	Complexity[1]	Reference
Increasing subsequence	$O(n \log\log n)$	[15]
Alternating subsequence	$O(n)$	
Wedge Permutation	$O(n \log\log n)$	Section 7.5

[1] n refers to the size of the permutation.

Figure 5.1 – Results about computing the longest subsequence of any permutation.

5.2.1 Longest Common Subsequence for Words

Problem 29. *Given a set $\mathcal{S}$ of words, the LCS problem seeks to find the LS common to all words in $\mathcal{S}$.*

The LCS is known to be NP-complete (See [43]) when the size of the set is not bounded, even when the words are binary words. A polynomial time algorithm exists when the size of the set is fixed, running in $O(n^k)$ where k is the size of the set and n is the size of the longest word in the set (See [43]). This algorithm is still impracticable even when the set is small, as the size of the words plays an important role in the time consumption and the words are commonly large. For this reason, much part of the research is focused on the special case of a set of two words.

The problem is presented as follows: given a word v of size m and a word w of size n, compute the LCS of v and w. A well-known algorithm solves this problem in $O(nm)$ using a dynamic programming strategy: Each instance of the problem can be solved by solving simpler subproblems (usually instances that are smaller in size) that are themselves solvable with simpler subproblems and so on until the instances can be solved trivially. As, a subproblem can appear multiple times, to avoid recomputing the same instance more than once, we memorise the solution of each subproblem and retrieve it when the same subproblem recurs.

A LCS is computed as follows:

- If the first letter of the LCS is the first letter of w and v, the first letter is concatenated with the LCS of the words $w[1:]$ and $v[1:]$.
- If the first letter of the LCS is the first letter of w but not of v, the LCS of the words w and $v[1:]$ is computed.
- If the first letter of the LCS is the first letter of v but not w, the LCS of the words v and $w[1:]$ is computed.
- If the first letter of the LCS is the first letter of w and v,the LCS compute the LCS of the words $w[1:]$ and $v[1:]$ is computed.

We only need two indexes to indicate where we are in the words w and v. As, all subproblems are computed at most once and in constant time, the algorithm runs in $O(nm)$.

5.2.2 Longest Common Subsequence for Permutations.

Problem 30. *Given a set $\mathcal{S}$ of permutations, the LCS seeks to find the longest pattern common to all permutations in $\mathcal{S}$.*

As with strings, the problem of a set having only two permutations has been a focus of research. However, this condition is not sufficient for computing the LCS in polynomial time; other approaches are needed. A possible strategy for solving the problem polynomially is to restrict the set $\mathcal{S}$ to be in a certain class. It should be noted that by restraining $\mathcal{S}$ to be in a class $\mathcal{C}$, the LCS is also an element of $\mathcal{C}$.

Problem 31. *Given $\mathcal{C}$, a class of permutations, given a set of permutation $\mathcal{S} \subset \mathcal{C}$, the LCS problem seeks to find the LCS of all permutations in $\mathcal{S}$.*

5.2.3 Restricting the Set to the Set of Permutations that are in Bijection with Binary Words

The problem 31 is solvable whenever $\mathcal{C}$ is a class of permutations of size n that are in bijection with binary words of size n. Such classes of permutations exist; they include the wedge permutations (as see above), the unimodal permutations (which are the permutations that can be split in one increasing and one decreasing permutation) and the riffle shuffle permutations (which are the permutation that can be partitioned into two increasing subsequence, one above the other). To compute the LCS over such permutations, the permutations are transformed into binary words. The LCS is then computed over the two binary words, with the positions used in the LCS being tracked. The LCS for the permutations is the subsequence with the elements at same position as the LCS on the binary words. As shown in Section 5.2.1 the algorithm for finding the LCS on words runs in $O(mn)$, but the bijection from permutation to word depends on the class so the whole algorithm runs in $\max(O(mn), BC)$ time, where BC is the complexity of the bijection.

5.2.4 Restricting the Set to the Permutations Separable Permutations

A result also exists whenever at least one permutation is a separable permutation, as shown later in this thesis (See section 6.5).

5.2.5 Computing the Longest Wedge Subsequence

Instead of computing any LCS, another strategy for reducing the complexity is, to compute LCS that belongs to a certain class. As demonstrated later in this thesis (see Section 7.5), whenever we want to find a LCS that is a wedge permutation, there exists a $O(n^3m^3)$-time algorithm where n and m are the size of the permutations.

5.2.6 State of the Art

The results about the LCS of two permutations are summarize in table 5.2.

5.3 Superpattern

Definition 32. *Given a set $\mathcal{S}$ of permutations, a superpattern of $\mathcal{S}$ is a permutation that contains all of the permutations in $\mathcal{S}$.*

Problem	Complexity[1]	Reference
Computing the LCS of two permutations which are in a class in bijection with binary word	$\max(O(mn), BC)$ [2]	
Computing the LCS that is a wedge permutation of two permutations	$O(n^3m^3)$-time	Section 7.5

[1] n and m refer to the size of the two permutations.
[2] BC is the complexity of the bijection.

Figure 5.2 – Results about computing the longest common subsequence of two permutations.

Example 33. *The permutation* 4123 *contains both the permutations* 123 *and* 312*:* 4123 *is a superpattern of* $\{123, 312\}$*.*

We can specify the definition when $\mathcal{S}$ corresponds to all of the permutations of a given size.

Definition 34. *A k-superpattern is a permutation that contains all permutations of size k.*

Example 35. *The permutation 25314 is a 3-superpattern.*

5.3.1 Lower and Upper Bounds for the size of the Minimal Superpattern for all Permutations of the same Size

The problem of finding a k-superpattern is trivial, as one can find a k-superpattern by a direct sum with all of the permutations of size k (which results in a permutation of size $kk!$). The problem becomes more interesting in its minimisation: Given a size k, compute a k-superpattern with minimal size with respect to other k-superpatterns. This problem can be transformed into a decision problem in which the minimisation factor is replaced by the size of the superpattern and we can play with this value to find the minimal size required.

Problem 36. *Given two integers k and n, the k-superpattern minimisation asks whether a k-superpattern of size n exists.*

Example 37. *The permutation 25314 is a smallest 3-superpattern. Indeed, the occurrence of the permutation* 123 *and* 321 *can only share the element representing the 2, so we need two more elements to represent the* 1 *and* 3 *of* 123 *and two more to represent the* 3 *and* 1 *of* 321*.*

The problem was first introduced by Arratia in 1999 in [10]. This article presents a k-superpattern of size k^2 by exhibiting the grid permutation, also known as k-grid permutation (note that the notion of "grid classes of permutation" exists and both notion are different, See [2]). The grid permutation is constructed by iterating over all of the positions (starting at 1) moving to the next position by a step of k and putting the elements in increasing order. More formally, this corresponds to the permutation in which elements at position i have values $rk + 1 + q$ where r and q are the remainder and the quotient of the euclidean division of $i - 1$ by k. See Figure 5.3 for an illustration.

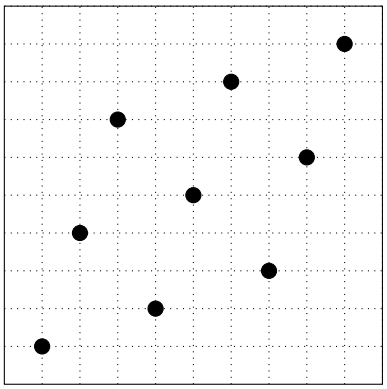


Figure 5.3 – The 3-grid permutation.

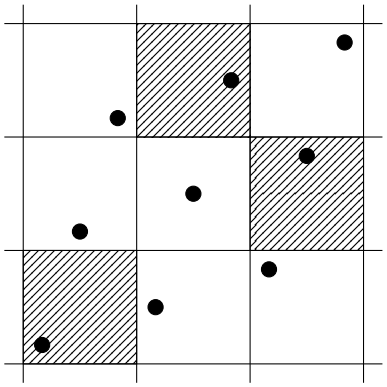


Figure 5.4 – The 3-grid permutation and the occurrence of 132.

It is easy to see that the k-grid permutation contains all permutations of size k. To find an occurrence of a given permutation σ, the grid permutation is split into k x k squares of the same size. Observing that each square contains one point, we define each square by its coordinates on the plan, such that the square in the bottom left corner is square $(0, 0)$, the one above it is $(0, 1)$, the one on its right is $(1, 0)$ and so on. From this construction, there exists an occurrence of σ such that each element $(i, \sigma[i])$ is represented by the point in the square at coordinate $(i, \sigma[i])$. See Figure 5.4 for the occurrence of the permutation 132 in the 3-grid permutation.

In the same paper, Arratia also provided the lower bound of $(k/e)^2$ for the size of a k-superpattern by an enumerative argument on the number of permutations of size k.

A more refined result concerning the minimal size of k-superpattern was later provided by Eriksson et al. who showed in 2007 in [28] that a k-superpattern of size $2k^2/3$ exists; moreover, if k tends to infinity, then a k-superpattern of size $k^2/4$ exists. They also demonstrated that there exists a permutation that contains every permutation of size k or its inverse of size $k^2/2$. In 2009, Miller stated in [45] that a k-superpattern of size $(k + 1)k/2$ exists.

The general problem of superpatterns (in permutation) that is, when the set

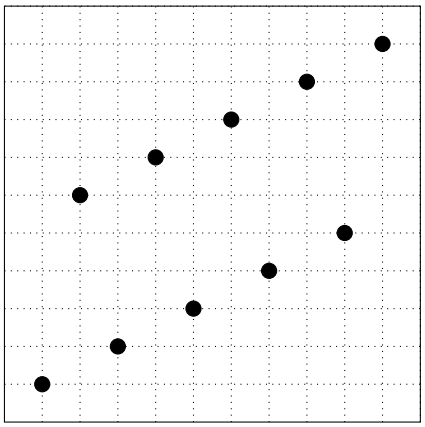


Figure 5.5 – The superpattern of the set of riffle shuffle permutations of size 5: this permutations contains any riffle shuffle permutation of size 5.

$\mathcal{S}$ can be anything has to our knowledge not been studied. Nevertheless, some results can be found when some constraints are added to the set, these results are presented in the following section.

5.3.2 Superpattern for Riffle-Shuffle Permutations

We have examples of refined results whenever constraints are put on the set. In 201 3, Wismath and Wolff exhibited in [14] a superpattern of size $2k$ for the riffle-shuffle permutations of size k. The first result was obtained by remarking that any riffle-shuffle permutation can be partitioned into two increasing subsequences such that the elements of increasing subsequence are below the element of the other subsequence. This leads to the construction of a permutation by pair of elements: Each time we add an element that can be part of the below increasing subsequence and an element that can be part of the above increasing subsequence. This gives us the permutation $1(k)2(k+1)3(k+2)\dots k(2k)$. See Figure 5.5 for an example.

It should be noted that the elements of a riffle-suffle permutation can be partitioned into two sets, thus, one can transform a riffle-shuffle permutation into a binary words; finding a superpattern for a riffle-shuffle permutation of size k is the same as finding a binary word that contains every binary words of size k. This formulation makes it clear that the word formed of 01 repeated k times contains all binary words of size k. This result can be extended to every class that is in bijection with binary words.

5.3.3 Superpattern for 321-Avoiding Permutations

In [14], Wismath et al. also present a superpattern for the 321-avoiding permutations of size $O(k^{3/2})$.

5.3.4 Superpattern for 213-Avoiding Permutations

In 2014, in [13] Wismath et al. also prove that there exists a superpattern of size $k^2/4 - \Theta(k)$ for all of the permutations avoiding 213 of size k and provide

a superpattern for some proper subclasses of the 213-avoiding permutation.

5.3.5 State of the Art

The results on the superpattern problem are summarize in the tables 5.6 and 5.7.

5.4 Shuffle

The (riffle) shuffle of a deck of cards involves starting with a sorted deck, cutting it into two parts and then reforming the deck by randomly selecting a card from the bottom of one of the two parts and putting this card on top of the newly formed deck. The mathematical definition only takes from the last step of the (riffle) shuffle. We provide the general definition of a shuffle and extend it to the permutation. We first explain one notation. Given a letter a and a set of words $\mathcal{S}$ we refer to $a\mathcal{S}$ as the set in which we concatenate the letter a to every word in $\mathcal{S}$.

Definition 38. *Given two words v and w, the shuffle of v and w, (noted $v \sqcup\!\sqcup w$) is the set $v[0](v[1:] \sqcup\!\sqcup w) \cup w[0](v \sqcup\!\sqcup w[1:])$. This correspond to the set of words that we can obtain from merging the words u and w, by choosing letters randomly from u or w.*

Example 39. $aa \sqcup\!\sqcup bb = \{aabb, abab, abba, baab, baba, bbaa\}$

5.4.1 Word Shuffles

Shuffle in relation to words was first introduced by Eilenberg and Mac Lane in [27]. Later on, Prodinger and Urbanek in [47] studied a special case of shuffle called the perfect shuffle (denoted $\square$ in the article), which corresponds to the unique word created by picking letters from two words in an alternating manner. In their article, the authors present some properties of a perfect shuffle. They first introduce the notion of square word (although they do not name it).

Definition 40. *A square word w is a word such that there exists a word v such that $w \in v \sqcup\!\sqcup v$.*

Some notable results surrounding the shuffle in words are that it can be decided, in polynomial time whether a word u is in the shuffle of v and w (see [44] and [53]), and that the problem of deciding whether a word w such that a word u is the shuffle of word w with itself exists . The latter result is more interesting to us given its similarities to permutation. The problem turns out to be NP-complete (see [24] and [50]).

For an overview of the problem, see [48] and [49].

5.4.2 Shuffle for Permutations

We extend the notion of shuffle for words to that of shuffle for permutations. This operation was introduced as the supershuffle by Vargas in [54], and denoted by $\sqcup\!\sqcup$. In this thesis, we use the notation suggested by Giraudo and Vialette, as these authors present the only result that we are aware of (See [30]).

Result[1]	Reference
the minimum size is bounded by $(k/e)^2$	[10]
one of size $k^2/4$ when k tends to infinity	
a permutation that contains every permutations of size k or its inverse of size $k^2/2$	[28]
one of size $(k+1)k/2$	[45]

[1] k refers to the size of the permutations.

Figure 5.6 – Results for the k-superpattern for set of permutations of size k.

Result[1]	for the class	Ref.
one of size $2k$	for classes which are in bijection with binary word	[14]
one of size $O(k^{3/2})$	321-avoiding	
one of size $k^2/4$	213-avoiding	
s one of size $k \log^{O(1)} k$	for every set of proper subclass of the set 213-avoiding	
the minimum size is bounded by $k \log k$	$\{213, 132\}$-avoidings	
the minimum size is bounded by $k \log k$	213-avoiding	[28]
the minimum size is bounded by $3k - 4$	$\{213, 132, 3412, 4231\}$-avoiding	
one of size $2k - 1$	for every set of proper subclass of the set $\{213, 312\}$-avoiding	
one of size $k \log_2 k + k$	for every set of proper subclass of the set $\{213, 132\}$-avoiding	
one of size $3k - 4$	for every set of proper subclass of the set $\{213, 3412\}$-avoidings	

[1] k refers to the size of the permutations.

Figure 5.7 – Results for the k-superpattern when restricting the set of permutations.

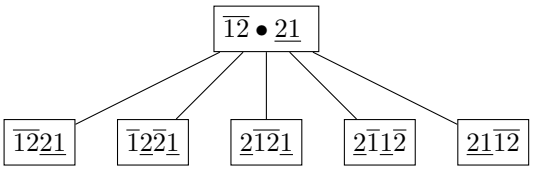


Figure 5.8 – The shuffle of 12 and 21 as a word, the elements are overlined and underlined to indicate which permutation the element comes from, and each leaf represent a set of permutations.

Definition 41. *Given σ and π, the shuffle of π and σ, (noted $\sigma \bullet \pi$), is the set of permutations obtained by applying the shuffle to σ and π as words, where each word represents a set of permutations. The set represented is the set of every different permutation obtained by the following rule: Relabel each letter by increasing integers by replacing either the lowest element of σ or π not yet replaced until all elements are replaced.*

An example is given in Figures 5.8 and 5.9.

The definition of shuffle gives rise the following decision problem:

Problem 42. *Given the permutations σ_1, σ_2 and π, the recognising shuffle for permutation problem asks whether $\pi \in \sigma_1 \bullet \sigma_2$.*

The recognising shuffle problem is sometimes referred as the unshuffling problem. Although the definition of shuffle for permutations is complicated, the recognizing shuffle for permutation problem can be formulated in another way to remove the use of shuffle for permutations.

Problem 43. *Given the permutations σ_1, σ_2 and π, the recognising shuffle for permutation problem asks whether there exists an occurrence of σ_1 and σ_2 in π that do not share any elements and use every element of π.*

While no result exists concerning the recognising shuffle problem, a result to the related problem of recognising a square shuffle can be found.

Problem 44. *Given the permutations σ and π, the recognising square shuffle for permutation problem asks whether $\pi \in \sigma \bullet \sigma$.*

This problem is of interest, as it is strongly related to another problem that naturally arises in the context of

PPM

. This problem is deciding whether the PPM problem for parameter $n - k$ is FPT. (recalling that the PPM problem is FPT for parameter k [32]).

In [30], Giraudo and Vialette showed that the recognising square shuffle for permutations is NP-complete. It is worth mentioning that they put in relation the recognising shuffle for binary words as similar to the recognising shuffle for permutation over the wedge permutation ($\{231, 213\}$-avoidance permutation); as such, any results for either problem can be applied to the other. It should be noted that [9] claims that recognising shuffle for binary words is NP-complete but without proof. Thanks to this relation, we can apply the algorithm to detect whether u is in the shuffle of v and w, to decide whether a wedge permutation π is in the shuffle of two wedge permutations σ_1 and σ_2.

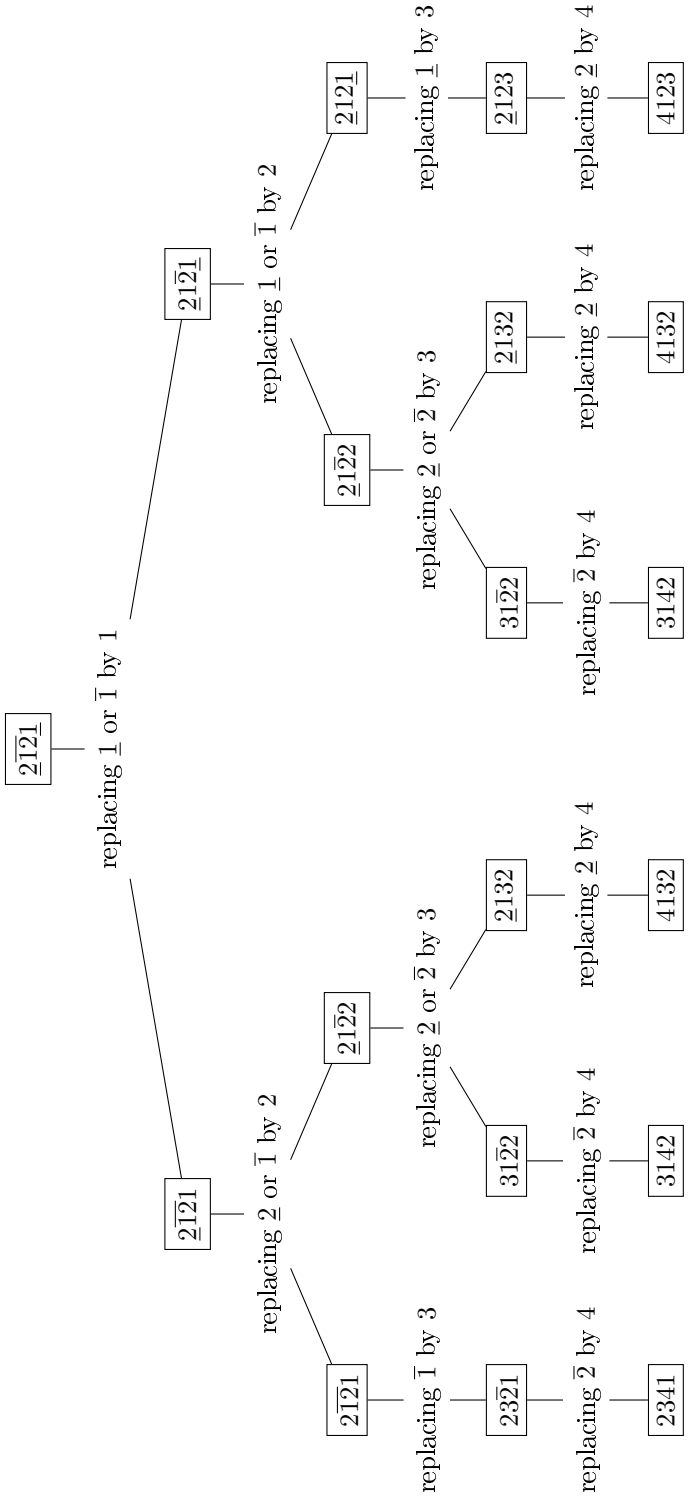


Figure 5.9 – The set of permutations obtained by the development of $\underline{2}\overline{1}\overline{2}\underline{1}$.

Result	Reference
Deciding whether a permutation is a square shuffle for permutation is NP-complete	[30]
Deciding whether a separable permutation π of size n is in the shuffle of two separable permutations σ_1 of size k and σ_2 of size l can be done in $O(nk^3l^2)$ time and $O(nk^2l^2)$ space	Section 6.4

Figure 5.10 – Results on deciding whether a permutation is a shuffle of two permutations.

5.4.3 Recognising the Shuffle of two Separable Permutations

We will see later, in this thesis (in Section 6.4), that an algorithm to decide whether a separable permutation π is in the shuffle of two separable permutations σ_1 and σ_2 also exists.

5.4.4 State of the Art

The results on deciding whether a permutation is a shuffle of two permutations are summarize in the table 5.10.

Chapter 6

Separable Permutations

This chapter is devoted to separable permutations, which is one of the first avoiding class studied. It appears in [18] which is the first paper that provides results on PPM.

This class is also a generalisation of the first avoiding class studied: the 213-avoiding permutations. A 213-avoiding permutation can be characterised by the fact that it can be sorted by a unique stack. Moreover, it appears that a separable permutation can be sorted by an arbitrary number of stacks (See [12]). We introduce the following definition of a separable permutation.

Definition 45. *The set of separable permutations is the set of* (2413, 3142)*-avoiding permutations.*

We start by presenting the structure of a separable permutation needed for this chapter. In Section 6.2, we revisit the polynomial-time algorithm of Ibarra [34] for the PPM problem and propose a simpler dynamic programming approach. In Section 6.3 we focus on the case in which both the pattern and the target permutation are separable. Section 6.4 is concerned with presenting an algorithm to test whether a separable permutation is the disjoint union of two given (necessarily separable) permutations. In Section 6.5, we revisit the classical problem of computing a longest common separable pattern as introduced by Rossin and Bouvel [51] and propose a slightly faster –yet still not practicable– algorithm. Finally, in Section 6.6, we show that the PPM problem is polynomial-time solvable for bivincular separable patterns. To the best of our knowledge, this is the first time the PPM problem is proven to be tractable for a generalisation of separable patterns. All the results in this chapter can also be found in [46].

6.1 The Structure of Separable Permutations

6.1.1 Substitution Decomposition

We present in this section the substitution decomposition. Any permutation is decomposable using the operation of substitutions. Especially, decomposition of a separable permutations has a remarkable structure. We begin by giving a definition of the operation of substitution.

Definition 46. *Given a permutation π of size n and n permutations π_1, π_2, ..., π_n, the substitution of π_1, π_2, ..., π_n in π, noted $\pi[\pi_1, \pi_2, \ldots, \pi_n]$, is the permutation that can be split into n different rectangles R_1, ..., R_n such that:*

- *Each element $\pi[i]$ is associated with the rectangle R_i.*
- *Every pair of rectangles is comparable, that is, one is on the left of and below the other one.*
- *For every rectangles R_i and R_j, $R_i \neq R_j$, R_i is on the left of R_j if and only if $i < j$.*
- *For every rectangles R_i and R_j, $R_i \neq R_j$, R_i is below R_j if and only if $\pi[i] < \pi[j]$.*
- *For every rectangle R_i, the elements contained in R_i are an occurrence of π_i.*

For example $132\ [123, 4321, 12]$ is the permutation 123987645. It should noted that a unique decomposition (called the canonical decomposition) for any permutation exists when considering substitution in simple permutations, which are permutation that contains no (contiguous) sequence such that the values of the sequence form an interval (See [5] for more details on substitutions). When considering the canonical substitution decomposition, a separable permutation can be define as follows.

Definition 47. *A separable permutation is an increasing permutation or a decreasing permutation or it is the canonical substitution of π_1, π_2, ... and π_n in π, where π is either the increasing or decreasing permutation and π_1, π_2, ... and π_n are separable permutations.*

It should be noted that direct an skew sum are special cases of substitution. Indeed $\pi_1 \oplus \pi_2$ is equivalent to $12[\pi_1, \pi_2]$ and $\pi_1 \ominus \pi_2$ is equivalent to $21[\pi_1, \pi_2]$. In the following, we focus in decomposition in direct and skew sum.

6.1.2 Separable Permutations Seen as a Direct or Skew Sum

Simple Decomposition in Sum

Proposition 48. *A separable permutation is either the trivial permutation of size 1 or obtained by a direct or skew sum of two separable permutations.*

Proof. We prove this claim by induction on the size of the permutation. A separable permutation π of size n can start by the topmost or bottommost element. In such cases, π can be written as $1 \ominus \pi[2:]$ or $1 \oplus \pi[2:]$, respectively.

If π does not start with the topmost or bottommost element, by hypothesis $\pi[2:] = \pi[2:i] \oplus \pi[i+1:]$ or $\pi[2:] = \pi[2:i] \ominus \pi[i+1:]$. We focus on the case in which $\pi[2:] = \pi[2:i] \oplus \pi[i+1:]$ and prove that π is decomposed into a direct sum of two separable permutations which is the other case in which $\pi[2:] = \pi[2:i] \ominus \pi[i+1:]$ can be dealt by following the same idea or using an argument based on the complement of a permutation.

We start by discussing the value of the leftmost element if $\pi[1]$ is below (resp. above) any element of $\pi[i+1:]$, then $\pi = \pi[:i] \oplus \pi[i+1:]$, where $\pi[:i]$ and $\pi[i+1:]$ are separable permutations which is the claim.

If it is not true, we show that there exists an index j such that every element of $\pi[:j]$ is below $\pi[1]$ and every element of $\pi[j+1:]$ is above $\pi[1]$. This supports our claim, as π would be written as $\pi = \pi[:j] \oplus \pi[j+1:]$. Suppose that such j does not exist; in other words, at least one element (namely $\pi[i_3]$) is above $\pi[1]$ and between two elements $\pi[i_2]$ and $\pi[i_4]$ which are below $\pi[1]$. This can be expressed as follows: $1 < i_2 < i_3 < i_4$, $\pi[i_2] < \pi[1] < \pi[i_3]$ and $\pi[i_4] < \pi[1] < \pi[i_3]$. Moreover, any element to the left of $\pi[i]$ is a candidate for i_2. Indeed, any element to the left of $\pi[i]$ is below $\pi[1]$. As $\pi[i_2]$ is to the left of $\pi[i]$ and $\pi[i_4]$ is to the left of $\pi[i]$ we have that $\pi[i_2] < \pi[i_4]$. As such, $1 < i_2 < i_3 < i_4$ and $\pi[i_2] < \pi[i_4] < \pi[1] < \pi[i_3]$, which implies that $\pi[i_1]\pi[i_2]\pi[i_3]\pi[i_4]$ is an occurrence of 2413 which is not possible. □

This is the main structure that we use for the separable permutation. This result yields a strong recursive definition of separable permutations that allows us to use an algorithm that treats the two parts separately. A separable permutation can be seen as two rectangles, one to the right and below/above the other. More than one decomposition may exists for the same separable permutation. For example, the permutation 214365 can be decomposed into $21 \oplus 4365$ and $2143 \oplus 21$.

Definition 49. *Given a separable permutation, the sign of the decomposition refers to whether the permutation is decomposed into direct or skew sum.*

Largest Decomposition in Sum

The direct and skew sums are associative operations, which implies that $\pi_1 \oplus (\pi_2 \oplus \pi_3)$ can be written as $\pi_1 \oplus \pi_2 \oplus \pi_3$. The consequence of this for separable permutations is that one separable permutation may have more than one decomposition (as said above). Moreover, some brackets are useless. To avoid theses situations, we consider a special decomposition that contain no useless bracket, that is when the operator inside an operand (if any) is the same as the operator of this operand. This has for consequence that the decomposition become unique. Moreover, this decomposition can be identified (from all the decompositions) as the one with the most operands, thus the name of largest decomposition. For example, the permutation 214365 is decomposed into $21 \oplus 21 \oplus 21$.

Remark 50. *Given π and its largest decomposition $\pi_1 \oplus \ldots \oplus \pi_i \oplus \ldots \oplus \pi_\ell$, all sub-permutations $\pi_1, \pi_2, \ldots, \pi_\ell$ are decomposed into a skew sum. Indeed, if that were not the case, a pair of bracket would be useless.*

Computing the Decomposition of a Separable Permutation

Computing the simple decomposition. This can be realised using the following algorithm. The first step is to decide whether the permutation is decomposed into a direct or a skew sum. To do so, the leftmost element and rightmost elements are compared. If the leftmost element is smaller than the rightmost, the permutation is decomposed into a direct sum; if not, it is decomposed into a skew sum. The next step depends on the decomposition. If the

permutation is decomposed into a direct sum, an index (they may be more than one) is sought such that the LRMax is bigger to the RLMin. The permutation is then split at the index, creating the left and the right part. Otherwise, an index (they may be more than one) is sought such that the LRMin is smaller than the RLMax, The permutation is then split at the index, creating the left and the right part. This can be archived by computing the LRMax and RLMin for each index and comparing the value.

Algorithm 1:

Data: A separable permutation π
Result: A decomposition of π
$LRMaxima$ = Compute all of the LRMax of π for each index ;
$RLMinima$ = Compute all of the RLMin of π for each index ;
if *the permutation is decomposed into a direct sum* **then**
  for *each index* i *in* π **do**
    if $LRMaxima[i] > RLMinima[i]$ **then**
      return that $\pi = \pi[: i-1] \oplus \pi[i :]$;
    else
      continue ;
else
  for *each index* i *in* π **do**
    if $LRMinima[i] < RLMaxima[i]$ **then**
      return that $\pi = \pi[: i-1] \ominus \pi[i :]$;
    else
      continue ;

It should be noted that the permutation is decomposed into a direct sum if and only if the leftmost element is below the rightmost element. This algorithm is based on the fact that if the permutation is decomposed into a direct sum, than the maximum element of the permutation on the left is smaller than the minimum of the permutation on the right (noting that the left rectangle is below the right rectangle).

Another algorithm computes an entire decomposition. We first introduce the notion of a break, which happens when two adjacent elements are not contiguous. For example, in the permutation 123654978, the break are 36, 49 and 97.

The algorithm creates a decomposition (treated as a word) while reading the permutation from left to right stopping at each break. One difficulty is when a break happens to decide whether what we are about to read is part of the same rectangle or a new rectangle. For example, in the permutation 321654987, 49 is a break, 321654 is a complete rectangle and 987 is a new rectangle. But in the permutation 654987321, 49 is a break, but 987 is in the same rectangle as 654. The idea is to create a new rectangle at each break, but at the next break, we have to check whether what we are currently reading is part of the previous rectangle; this is done by checking whether the current and previous rectangles form a contiguous set of elements (which is a characteristic of a rectangle). If the current rectangle is not part of the previous one, we need to keep it in a stack, and continue the procedure. If it is, we create a new rectangle (by merging the

previous and current one), so we also need to check whether the new rectangle is part of the previous one.

Algorithm 2:

Data: A separable permutation π
Result: A decomposition of π as a word
$R_c = \pi[1]$;
$stack = \emptyset$;
for *each index* i *of* π *starting with 2* **do**
  if $R_c \cup \{\pi[i]\}$ *is a contiguous set* **then**
    $R_c.append(\pi[i])$;
  else
    while *the last rectangle in the stack and* R_c *form a contiguous set* **do**
      pop R_p from the *stack* ;
      if R_c *and* R_p *form a contiguous set* **then**
        if *the elements of* R_p *are above the elements of* R_p **then**
          $R_c = (R_p \oplus R_c)$;
        else
          $R_c = (R_p \ominus R_c)$;
    push R_c in the *stack* ;

It should be noted that this algorithm returns a word representing a decomposition, in which each rectangle is encoded within a pair of parentheses.

Computing the largest decomposition from a simple decomposition This algorithm is quite natural: Whenever we have $\pi_1 \oplus (\pi_2 \oplus \pi_3)$, we want to write $\pi_1 \oplus \pi_2 \oplus \pi_3$. This correspond to remove any useless parenthesis. Remember that the parenthesis are useless the operator of the operand (where the parenthesis are from) is the same as the operator associated to this operand. More formally, the algorithm considers the decomposition as a word (as given in the previous algorithm). It should be noted that the word is composed of 3 parts: a prefix, then the sign (of the decomposition) and finally a suffix. The prefix and suffix start with an opening parenthesis and end with a closing parenthesis and they are well-balanced for parentheses, that is the word contain the number of opening parenthesis as closing parenthesis and every prefix of the word contain at least more opening parenthesis than closing parenthesis. Moreover they are decompositions of separable permutations. We first need to obtain the sign of the decomposition. To do so, we need to know when the prefix end, this can be done by computing the first well-balanced word (for parentheses) of the decomposition, the next letter of the word computed is the sign of the decomposition. Note that we obtain the 3 parts describe above: the word computed is the prefix, we have the sign and what is right to sign is the suffix. A decision needs to be made as to whether to remove the parentheses of the prefix and the suffix. We remove parentheses if and only if the sign of the prefix (or the suffix), is the same as the sign of the decomposition.

6.1.3 Binary Trees

This section present the representation of separable permutation as a tree. A tree consists of nodes (which are usually labelled to contain informations). A nodes is related to another one by the relation of being a child or parent, such that every nodes have one parent except for a specific node called the root and each node is a child or a parent to at least another node. Moreover the relations cannot create a circle, that is there exists no sequence of nodes such that each pair of nodes are related (with the same relation for all pairs) and the last node is related to the first node.

We represent a tree by drawing every node in a top-down representation, starting with the root. We represent the child relation with a edge between the two related nodes. It should be noted that edges can be understood as the child relation when reading the tree from top to bottom, or as the parent relation when reading the tree from bottom to top. See Figure 6.1 for an example of tree.

We say that a node C is a descendant of a node P if and only if there exists a sequence of node, such that P is the first element and C is the last element, and each node is the child of the node to its left. We can also say that P is an ancestor of C. Moreover we say that a node is a leaf if and only if it has no child. We refer the reader to [39] Chapter 2 Section 3 for more informations on tree.

We consider a special case of tree. Binary trees add the condition that each node has two children or is a leaf. A definition of a separating tree was given by Bose, Buss, and Lubiw in [18]: a rooted binary tree in which the elements of the permutation appear (in permutation order) at the leaves of the tree, and in which the descendants of each tree node form a contiguous subset of these elements. Each interior node of the tree is either a *positive* node in which all descendants of the left child are smaller than all descendants of the right node, or a *negative* node in which all descendants of the left node are greater than all descendants of the right node. See Figure 6.1 for an illustration. Each subtree of a separating tree may be interpreted as itself representing a smaller separable permutation, whose element values are determined by the shape and sign pattern of the subtree. Constructing a separating tree of a separable permutation is in linear time and space [18].

Remark 51. *The labels on the leaves do not matter, as only the structure of the tree matters. However, for the sake of comprehension, the leaves are labelled to their corresponding element in the permutation.*

6.1.4 A Separable Permutations Seen as a Tree

As the binary tree decomposition appears to be not well suited for our needs, we must use the notion of compact separating tree. Instead of considering only a binary tree, we consider a tree in which each node has the largest number of children possible. Informally, in a compact separating tree we strive for every node to have as many children as possible. Such a tree can be formally define by modifying the following definition: a rooted tree in which the elements of the permutation appear (in permutation order) at the leaves of the tree, in which the descendants of each tree node form a contiguous subset of these elements. Each interior node of the tree is either a *positive* node in which all

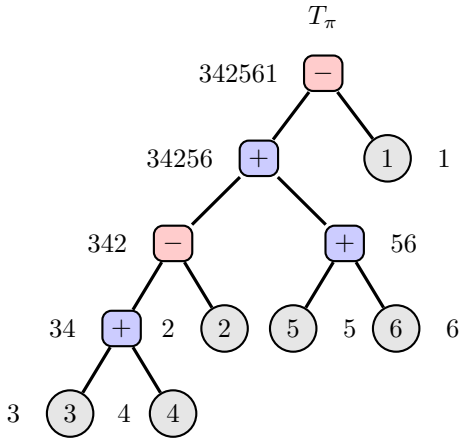


Figure 6.1 – A separating tree T_π for the permutation $\pi = 342561$ together with the corresponding sequences for each node.

descendants of a child are smaller than all descendants of another child to its right, or a *negative* node in which all descendants of a child are greater than all descendants of another child to its right. See Figure 6.1 for an example. We denote a separating tree representing π by T_π. Moreover, for every node v of T_σ, we let $\sigma(v)$ stand for the sequence of elements of σ stored at the leaves of the subtree rooted at v.

Remark 52. *In such a tree, two nodes (one being the child of the other) never share the same sign. Indeed, if two nodes are father and child, then the father can adopt the children of this particular child, which violates the definition.*

Remark 53. *In such a tree, two nodes (one being the child of the other) never share the same sign. Indeed, if two nodes are father and child, then the father can adopt the children of this particular child, which violates the definition.*

6.1.5 Computing the Tree of a Separable Permutation

Computing the compact separating tree from a separating tree A simple linear time post-processing can be used to produce the decomposition tree from a binary separating tree: As long as the separating tree contains a positive node whose father is also a positive or a negative node whose father is also negative, we simply suppress that node, and let all of its children be adopted by their grandfather in proper order.

6.1.6 Relations Between the Decomposition in Direct and Skew Sums and the Separable Tree

The decomposition in direct/skew sums exhibits a recursive structure. As a result, we can represent a separable permutation by a binary tree. The tree representing the separable permutations with a unique element is the tree a with a unique node. Any other separable permutations are represented by a binary tree. The root encodes what operation (direct or skew sum) the permutation is decomposed into, with the tree's left and right children representing

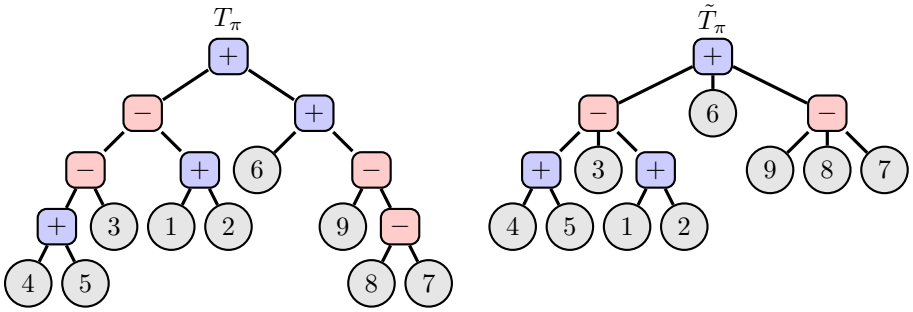


Figure 6.2 – A separating tree and the compact separating tree of the permutation $\pi = 453126987$.

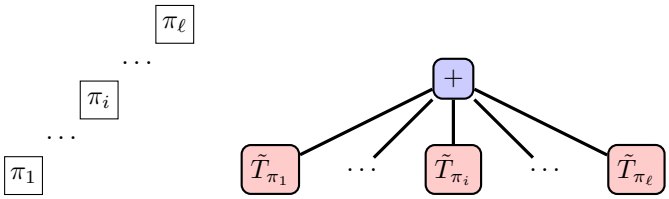


Figure 6.3 – On the left a rough plot of the permutation $\pi = \pi_1 \oplus \ldots \oplus \pi_i \oplus \ldots \oplus \pi_\ell$ and on the right its corresponding compact separating tree.

the operation's left and right permutation, respectively. This corresponds to the definition of a separable tree. The relation between the decomposition and the separating tree can be summarised as follows:

$\pi = e$	$\pi = \pi_1 \oplus \pi_2$	$\pi = \pi_1 \ominus \pi_2$
e	T_{π_1} T_{π_2}	T_{π_1} T_{π_2}

The same idea applies to the compact separating tree. Each node encodes which operation the permutation is decomposed into, and each child of this node represents a "subpermutation", in proper order, which means that the first child represent the first "subpermutation", the second child the second "subpermutation" and so on. More formally, if π has for the largest decomposition into direct sum $\pi = \pi_1 \oplus \ldots \oplus \pi_\ell$, then the (unique) compact separating tree of π is the tree with a positive root and with the compact separating tree of π_1 as first child, the compact separating tree of π_i as i^{th} child, and the compact separating tree of π_ℓ as ℓ^{th} child. The same is true if π is decomposed into a skew sum. See Figure 6.3 for an example. It should be noted that when π is decomposed into direct (resp. skew) sums it forms a stair up (resp. down) of rectangles.

6.2 Detecting a Separable Pattern

This section is devoted to examining of PPM problem whenever we add the condition that the pattern is a separable permutation. Our contribution is that we reduced the space consumption of the best algorithm so far to $O(n^3 logk)$ from $O(kn^3)$, where k is the size of the pattern and n is the size of the text.

Problem 54. *Given a separable permutation σ of size k and a permutation π of size n, we wish to determine whether σ occurs in π.*

In what follows, we present a polynomial time algorithm, in time and space, to solve this problem. The next subsections break the algorithm down. We first introduce the basic idea of the algorithm. Thereafter we show how Ibarra strengthened this idea. Finally, we demonstrate how we improved the algorithm of Ibarra.

6.2.1 Simple Algorithm

A simple algorithm arises from the decomposition of a separable permutations into direct/skew sums. The following property is the main idea of this algorithm.

Property 55. *Given a permutation such that $\sigma = \sigma_l \oplus \sigma_r$ (resp. $\sigma = \sigma_l \ominus \sigma_r$), if we are given an occurrence o_l of σ_l in a permutation text π and an occurrence o_r of σ_r in a permutation text π, such that every element of o_l is at the left and below (resp. left and above) of the elements o_r, then the subsequence formed by the concatenation of o_l and o_r is an occurrence of σ in π.*

Proof. We focus on the case in which $\sigma = \sigma_l \oplus \sigma_r$. Suppose that the concatenation is not an occurrence, which means that at least two elements are not "well" ordered. By hypothesis, these two elements cannot both be in o_l (as o_l is a "correct occurrence") or o_r (as o_r is a "correct occurrence"); as such, one element is in o_l and the second is in o_r. Remark that the correct order is that the two elements are increasing, as one represent an element in σ_l, the other an element in σ_r, and by the definition of a direct sum, every element in σ_l is below σ_r. This means that the two problematic elements are in decreasing order. However, as by hypothesis every element in o_l is above every element of o_r this is a contradiction., The case in which $\sigma = \sigma_l \ominus \sigma_r$ follows the same idea. □

An algorithm that follows this property computes occurrences for the left and for the right operands such that the two rectangles of the occurrences are comparable and in order, which depends on whether the permutation is decomposed into direct or skew sum. This implies a recursive algorithm: For every way to split the text into a left and right parts, we first decide whether the text's left part has an occurrence of the pattern's left part and whether the text's right part has an occurrence of the pattern's right part. This is done recursively, with each recursive call receiving a text and a pattern of a size smaller than the original problem. At some point, we have a trivial problem (which corresponds to the case in which $\sigma = 1$). The algorithm still needs to assure that the occurrences are comparable and in order. This is done while splitting the text: We always split the pattern into its left and right parts, but we split the text into two part such that the rectangle of the left part is at the left and below (right

and above depending of the sign) the rectangle of the right part. This gives us the following algorithm.

Algorithm 3:

Data: A permutation π
Data: A separable permutation σ
Result: Whether σ occurs in π
if π *is empty* **then**
  return False ;
if $\sigma = \sigma_l \oplus \sigma_r$ **then**
  for *every pair of rectangles* (r_l, r_r) *of* π *such that* r_l *is left and below* r_r **do**
    if σ_l *occurs in* r_l *AND* σ_r *occurs in* r_r **then**
      return True ;
else if $\sigma = \sigma_l \ominus \sigma_r$ **then**
  for *every pair of rectangles* (r_l, r_r) *of* π *such that* r_l *is left and above* r_r **do**
    if σ_l *occurs in* r_l *AND* σ_r *occurs in* r_r **then**
      return True ;
else if $\sigma = 1$ **then**
  return True ;
return False ;

The following remarks improve the current state of the algorithm or provide more insight for understanding the difference with the algorithm of Ibarra (Algorithm s3) :

Remark 56. *For practical reasons and to understand the improvement of Ibarra's algorithm, we need to delve deeper into the representation of the input. In particular, the algorithm does not receive a subsequence of* π*, but* π *and a rectangle.*

Remark 57. *One instance may be called multiple times. To avoid computing the same instance, the algorithm uses a dynamic programming strategy: Each time that it computes an instance, it remembers the value computed. Whenever it needs an instance, it first checks whether this instance has been already computed.*

Remark 58. *Since* σ *is a separable permutation, we can assume that we are given a separating tree* T_σ *for* σ*, or we construct it in linear time and space. Having a separating tree allows to decide whether* $\sigma = \sigma_l \oplus \sigma_r$ *or* $\sigma = \sigma_l \ominus \sigma_r$ *in constant time.*

Remark 59. *To iterate over all of the pair rectangles* (r_l, r_r) *such that* r_l *is left and below* r_r*, we only need for the top right corner of* r_l *to be to the left and below the bottom left corner of* r_r*. A similar remark applies to the pair of rectangles* (r_l, r_r) *such that* r_l *is left and above* r_r*: We only need the bottom right corner of* r_l *to be left and above the top left corner of* r_r*.*

Remark 60. *We actually do not need to iterate over all of the pairs of rectangles that are compatible. Indeed, if a smaller rectangle* r_s *is contained in a*

larger rectangle r_g and that r_s contains an occurrence of a pattern, then r_g also contains an occurrence of that pattern. This implies that the algorithm loses computational time by testing r_s and r_g. To be optimal, it should only test r_g. More practically, for the case in which $\sigma = \sigma_l \oplus \sigma_r$, this has the following consequence:

1. *If the pair of rectangles are not next to each other then we can find a rectangle containing the left rectangle (resp. right rectangle) such that this rectangle and the right rectangle (resp. and the left rectangle) are pairs of rectangle that are compatible. This means that the first pair of rectangle is not optimal. In other words, we always want a pair of rectangle that are next to each other. More formally, if the top right corner of the left rectangle is (x, y) then the bottom left corner of the right rectangle need to be $(x + 1, y + 1)$.*

2. *Given that we receive the rectangle $((i, \text{lb}), (j, \text{ub}))$ (See Remark 56), the left rectangle is always contained in (or equal to) the rectangle composed with bottom left corner (i, lb) and with the same top right corner as the left rectangle. This means that if the left rectangle is not equal to the rectangle latterly described, then the pair of rectangle is not optimal. In other words, we always want a left rectangle that has the same bottom left corner as the rectangle received. More formally, the bottom left corner of the left rectangle is (i, lb). In the same fashion, the top right corner of the right rectangle is (j, ub).*

Thank to these remarks, the iteration of pairs of rectangles is done over the following set: $\{((i, \text{lb}), (\ell, \pi[\ell])), ((\ell + 1, \pi[\ell] + 1), (j, \text{ub})) \mid i \leq \ell \leq j\}$

To understand the time and space complexity of this algorithm, it is important to note the following elements:

1. The set of all possible instances of this algorithm can be enumerated. This gives us a bound on the number of times that the algorithm is used and thus a time complexity for the algorithm. The algorithm takes as input a permutation π with a rectangle (See Remark 56) and a permutation σ. The number of different rectangles is $O(n^4)$ as any indexes $((*, *), (*, *)$ can have a value between 1 and n. For σ, the algorithm only receives a subpermutation of σ. The problem lies in enumerating all the different subpermutations σ. The subpermutations of σ are a left or right part of a decomposition or σ. We use the analogy between a subpermutation and a node of a tree to count the number of subpermutations. This number is related to the number of nodes of a binary tree. Remember that any subpermutation of a decomposition in a direct/skew sum of a separable permutation can be associated with a node on a separable tree. So there are at much subpermutation that there are nodes. In addition, each element is associated with a leaf, which means that there are k leaves. As such, the tree theory tells us that if a binary tree has k leaves then the tree has at most $k + k - 1$ nodes. The maximal number of subpermutations of σ is thus $O(k)$. So the algorithm has $O(kn^4)$ different instances.

2. Each instance iterates over every pair of rectangle. Based on the Remark 60, the iteration consists of an iteration of an index between the

values i and j. As i and j represent the index of π, the minimal value of i is 1 and the maximal value of j is n, in the worst case the index goes from 1 to n. Moreover, any other operations are done in constant time, which means, that an instance is computed in $O(n)$ time in the worst case.

As we must remember any instance computed (See Remark 57), we need $O(kn^4)$ space. Moreover, as $O(kn^4)$ different instances of the problem exist and each instance takes at most $O(n)$ time to compute, the algorithm runs in $O(kn^5)$ time.

6.2.2 Algorithm of Ibarra

This subsection is devoted to presenting the work of Ibarra in [34].

Proposition 61. *There exists an algorithm that decides whether σ occurs in π in $O(kn^4)$ time and $O(n^3k)$ space.*

The idea of the algorithm of Ibarra follows the same idea presented in the previous subsection. However, it uses another trick to reduce the number of different instances to compute, reducing the number of different instances from $O(kn^4)$ to $O(kn^3)$, which obliviously results in the complexity given in the latter proposition.

In the case where $\sigma = \sigma_l \oplus \sigma_r$, while splitting the rectangle of π into R_r and R_l, if a R_r is given σ_r, R_l need to be the largest possible to increases the chance of containing an occurrence of σ_l. Indeed if R_l is as large as possible, it contains all other rectangles; as such instead of checking whether each of these rectangles contains an occurrence of σ_l, it is faster to check whether the largest one contains an occurrence. Moreover, given an occurrence of σ_r, the occurrence's minimal element can be used as a bottom edge of a rectangle that contain this occurrence and it is the largest top edge possible for this occurrence. Choosing the largest top edge for the right rectangle, allows the left rectangle to be the largest possible. As such, given that $\sigma = \sigma_l \oplus \sigma_r$, we first split the rectangle vertically. we then focus on computing all of the occurrences of σ_r contained in the right rectangle. In all of these occurrences, we choose the one with the largest minimal element (for the reason explained above). This element can now be used as a bottom edge for the right rectangle.

To use a recursive algorithm (as before), we need the bottom edge's value, which can be done using the return value. The return value needs to encode the largest bottom edge's value. We use the following convention to determine whether an occurrence exists: If the value returned is 0, then no occurrence exists. if not, an occurrence exists and the value returned is the value of the largest bottom edge.

Algorithm 4:

```
Function PPM_Ibarra
    Data: A permutation π
    Data: A left edge left_edge
    Data: A top edge top_edge
    Data: A right edge right_edge
    Data: A separable permutation σ
    Result: 0 if no occurrence exists, the value of the higher bottom edge
            otherwise
    if π is empty then
        return 0 ;
    if σ = σ_l ⊕ σ_r then
        for each ℓ in [i, j] do
            higher_bottom_edge =
              PPM_Ibarra(π, ℓ + 1, top_edge, right_edge, σ_r) ;
            if higher_bottom_edge is 0 then
                return 0 ;
            else
                return
                  PPM_Ibarra(π, left_edge, higher_bottom_edge, ℓ, σ_l) ;
    else if σ = σ_l ⊖ σ_r then
        for each ℓ in [i, j] do
            higher_bottom_edge =
              PPM_Ibarra(π, left_edge, top_edge, ℓ, σ_l) ;
            if higher_bottom_edge is 0 then
                return 0 ;
            else
                return PPM_Ibarra(π, ℓ +
                  1, higher_bottom_edge, reight_edge, σ_r) ;
    else if σ = 1 then
        return the higher value of π[left_edge : right_edge] ;
    return 0 ;
```

6.2.3 Improved Version of Ibarra

In this section, we show that we can improve the algorithm of Ibarra to reduce the space consumption.

Proposition 62. *The memory consumption of the algorithm of Ibarra can be reduced to* $O(n^3 \log k)$.

Proof. It should first be observed that to compute all of the instances in which the pattern is the subpermutation σ (i.e. $PPM\_Ibarra(\pi, *, *, *, \sigma)$), we only need all of the instances in which the pattern is the left operand of σ in the decomposition, (that is, $PPM\_Ibarra(\pi, *, *, *, \sigma_l)$) and all of the instances in which the pattern is the right operand of σ (that is, $PPM\_Ibarra(\pi, *, *, *, \sigma_r)$). As such, some entries in the dynamic programming array become

useless, which means that we can "forget" them and release some space. It should be noted that, we use the notation of a node to represent σ, as it is better suited for the explanation. The policy for achieving the memory spearing is as follows:

- All problems for a same node v are solved together, and their solutions are maintained in the memory until the problems for the parent of v have also been solved. At that point, the memory used for node v is released.
- We use a modified depth-first search (noted as DFS) on T_σ: For every node v with two children, we first process its largest child (in terms of the number of nodes in the subtree rooted at that child), then the other child, and finally v itself.

We claim that the above procedure yields a $O(n^3 \log k)$ space algorithm and initially expand our DFS algorithm to what is known as the White-Gray-Black DFS [25]. First, we mark all vertices white. When we call $\textsc{dfs}(u)$, we mark u to be gray. Finally, when DFS(u) returns, we mark u to be black. Provided by this colour scheme, at each step of the modified DFS, we may partition T_σ into a white-gray subtree (all nodes are either white or gray) and a forest of maximal black subtrees (all nodes are black and the parent of the root, if it exists, is either white or gray). Our space complexity claim is now reduces to proving that, at any time of the algorithm, the forest contains at most $O(\log k)$ maximal black subtrees. Let h_σ be the height of T_σ, and consider any partition of T_σ into a white-gray subtree and an non-empty forest $\mathcal{T}^{\mathrm{b}}$ of maximal black subtrees. The following property easily follows from the (standard) DFS colour scheme.

Claim 63. *For every $1 \leq i \leq h_\sigma$, there exists at most two maximal black subtrees in $\mathcal{T}^b$ whose roots are at height i in T_σ. Furthermore, if there are two maximal black subtrees in $\mathcal{T}^b$ whose roots are at height i in T_σ (they must have the same parent), then $\mathcal{T}^b$ contains no maximal black subtree whose root is at height $j > i$ in T_σ.*

According to Claim 63, and aiming at maximising $|\mathcal{T}^{\mathrm{b}}|$, we may focus on the case in which $\mathcal{T}^{\mathrm{b}}$ contains one maximal black subtree whose root is at height i, $1 \leq i < h_\sigma$, in T_σ (if $\mathcal{T}^{\mathrm{b}}$ contains one maximal black subtree whose root is at height 0 in T_σ then $|\mathcal{T}^{\mathrm{b}}| = 1$), and $\mathcal{T}^{\mathrm{b}}$ contains two maximal black subtrees whose roots are at height h_σ in T_σ (these two maximal black subtrees reduce to size-1 subtrees). The claimed space complexity for the dynamic programming algorithm (*i.e.*, $|\mathcal{T}^{\mathrm{b}}| = \log(k)$) now follows from the fact that we are using a modified DFS algorithm where we branch the largest subtree first after having marked a vertex gray. Indeed, the maximal black subtree whose root is at height 1 in T_σ contains at least half of the nodes of T_σ. The same argument applies for subsequent maximal black subtrees in the forest $\mathcal{T}^{\mathrm{b}}$. □

6.3 Both π and σ are Separable Permutations

In this section, we focus on the case in which both the text and the pattern are separable permutations. Although our algorithm does not have better complexity in terms of time and space than the best algorithm so far, we contribute by providing an algorithm more specific to the separable permutations.

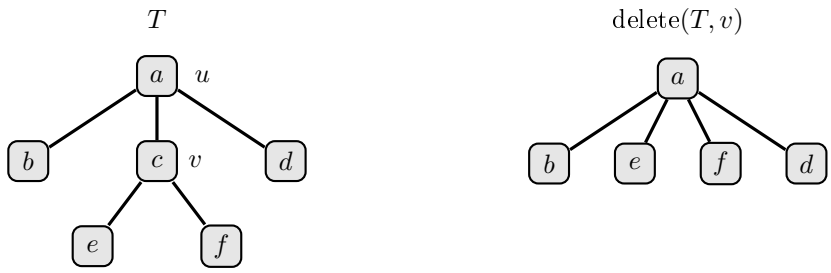


Figure 6.4 – The effect of removing a node from a tree.

Problem 64. *Given a separable permutation σ of size k and a separable permutations π of size n, we wish to determine whether σ occurs in π.*

In what follows, we will show that there exist a polynomial time algorithm, in time and space, to solve the Problem 64.

6.3.1 Best algorithm so far

We can strive for more efficient solutions when both π and σ are separable permutations since, we can construct in linear time the two separating trees T_π and T_σ. However, It turns out that the binary separating trees are not well suited to handle this task. We instead need compact separating trees. We adopt the convention that a compact separating tree of a separating tree T_π is denoted $\tilde{T}_\pi$.

To introduce the best algorithm so far, we need a definition of the *tree inclusion*. The problem is defined as follows: Given two ordered and labelled trees T and T', can T be obtain from T' by deleting nodes? (Deleting a node v corresponds to removing all edges incident to v and, if v has a parent u, replacing the edge from u to v by edges from u to the children of v. In other words, the father of the node deleted adopt the children of the node deleted, with respect to the original order of the children. See Figure 6.4.) This problem has recently been recognised as an important query primitive in XML databases. This is the rationale for considering compact separating trees stems from the following property. This problem is interesting to us because it can be used to solve the PPM when both the text and the pattern are separable permutations.

To do so we need two modifications:

1. We need to change the definition of the deletion: When we delete a node, we have the following rule for the adoption. The father of the deleted node adopts one child of the deleted node; moreover, if it adopts a node (remark that the father of the deleted node and the child of the deleted node share the same sign), then the father adopts all the children of this node. See Figure 6.5 for the different outcomes of a deletion. Intuitively, a subtree of a tree represents a subpermutation of the permutation represented by the tree, and the former definition of deletion can create a tree which do not represent a subpermutation. See Figure 6.6 an example.

2. All the leaves of the compact separable trees have to be labelled 1. It should be noted that this does not modify the corresponding separable

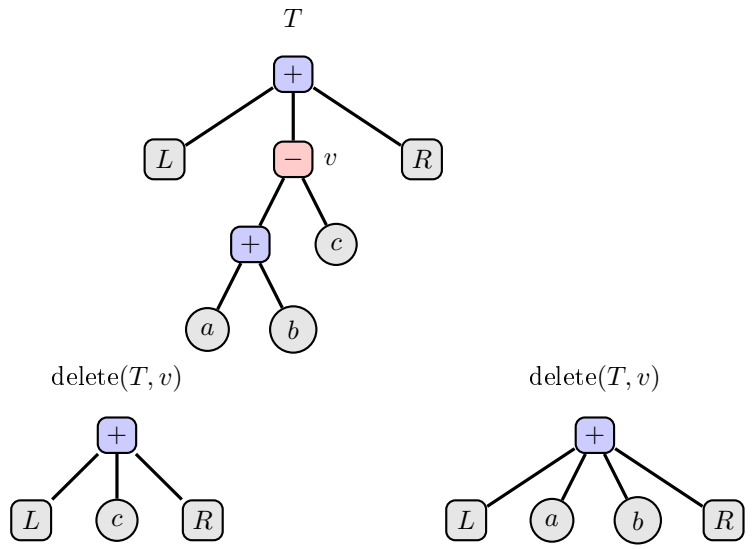


Figure 6.5 – The possible effects of removing a node. Note that the leaves are label with letters for the sake of comprehension.

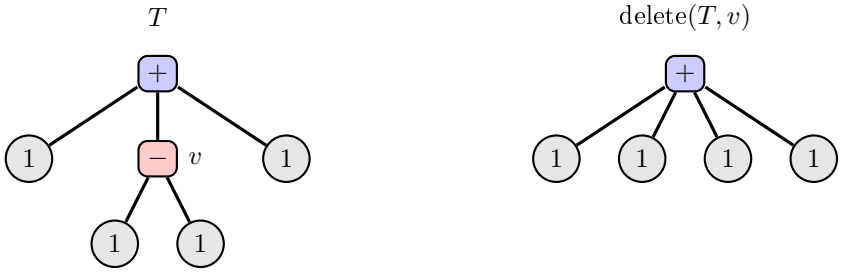


Figure 6.6 – The effect of a "normal" removing a negative node with a positive father, the permutation encoded in the left tree is 1324, whereas the permutation encoded in the right is 1234.

permutation, as the leaf's labels are included for the sake of readability (see Remark 51).

Remark 65. *The definition of deletion can be seen easily in a plot of a permutation. A rectangle should be first selected (which corresponds to choosing the node to delete). Another rectangle within this rectangle should then be selected and other rectangles removed.*

Property 66. *Deleting a node from a compact separable tree constructs another compact separable tree. Moreover, the permutation represented by the new compact separable tree is a subpermutation of the original permutation.*

Proof. First note that any child of a deleted node can be seen as a root of a compact separable tree. As such, adopting all children of a child a of deleted node implies that the children are already decomposed into the largest nodes

possible, which means that the deletion creates a compact separable tree. Now remark that deleting a node corresponds, in the permutation, to removing some elements represented by this node. As such, clearly yields a subpermutation. □

Property 67. *Let π and σ be two separable permutations. Then, σ occurs in π if and only if the compact separating tree $\tilde{T}_\sigma$ is included into the compact separating tree $\tilde{T}_\pi$.*

Proof. This is a direct consequence of the previous property. □

Kilpeläinen and Manilla [36] presented the first polynomial time algorithm using quadratic time and space for the tree inclusion problem. Since then, several improved results have been obtained for special cases in which T and T' have a small number of leaves or small depth. However, in the worst case, these algorithms still use quadratic time and space. The best algorithm is provided by Bille and Gørtz [16] who presented an $O(n_T)$ space and

$$O\left(\min\left\{\begin{array}{l} l_{T'}\, n_T \\ l_{T'}\, l_T \log\log n_T + n_T \\ \frac{n_T\, n_{T'}}{\log n_T} + n_T \log n_T \end{array}\right\}\right)$$

time algorithm, where T (resp. T') is the tree representing π (resp. σ), n_T (resp. $n_{T'}$) denotes the number of node of T (resp. T') and l_T (resp. $l_{T'}$) denotes the number of leaves of T (resp. T').

However, all efficient solutions developed so far for the tree inclusion problem result in very complicated and hard-to-implement algorithms. For example, the main idea of the algorithm presented in [16] is to construct a data structure on T supporting a small number of procedures, called the set procedures, on subsets of nodes of T.

6.3.2 Our Solution

We propose here a different solution that is less effective in time and space than the above algorithm.

Proposition 68. *An $O(n^2k)$ time and $O(nk)$ space algorithm to find an occurrence of a separable pattern of size k in a separable permutation of size n exists.*

We need to consider four cases in the algorithm, depending on the decompositions of σ and π. For these cases, we begin with a brute force algorithm that tries all possible subsequences, and reduce the number of subsequences to test.

The next paragraph deals with the case where σ and π are decomposed with the same sum.

Property 69. *Given the largest decomposition $\pi = \pi_1 \oplus \ldots \oplus \pi_{\ell_\pi}$ and the largest decomposition $\sigma_1 \oplus \ldots \oplus \sigma_{\ell_\sigma}$, no occurrence of σ_i can be split into two or more rectangles of π.*

Proof. If the occurrence of σ_i can be split into two or more rectangles of π then σ_i can be decomposed into a direct sum, which is not possible as σ_i is part of the largest decomposition of the direct sum. □

From this property, we know that, in an occurrence of σ in π, any rectangle π_i contains no, one or more than one rectangle of σ. Given an occurrence, taking all of the rectangles of π that contain at least one rectangle of σ, forms a subsequence of rectangles $\pi_{P_1}, \ldots, \pi_{P_m}$ of $\pi_1, \ldots, \pi_{\ell_\pi}$. For each of those rectangles, we associate the set of rectangles that this rectangle contains in the occurrence, and form a sequence of pair (π_i, P_i) in which the permutation represented by the set P_i occurs in π_i. It should be noted that the P_i forms an ordered partitioning of the rectangles of σ. Finding such a sequence is enough to prove that an occurrence of σ exists in π.

Lemma 70. *Given the largest decomposition $\pi = \pi_1 \oplus \ldots \oplus \pi_{\ell_\pi}$ and the largest decomposition $\sigma = \sigma_1 \oplus \ldots \oplus \sigma_{\ell_\sigma}$, if there exists a sequence (π_i, P_i) such that the permutation represented by the set P_i occurs in π_i and all the P_i forms an ordered partitioning of the rectangles of σ, then σ occurs in π.*

Proof. If such a sequence does not yield an occurrence, there exist at least two elements $\sigma[i]$ and $\sigma[j]$ such that their matchings do not have the same order as $\sigma[i]$ and $\sigma[j]$. Without a loss of generality, we can suppose that $i < j$. Suppose that $\sigma[i]$ and $\sigma[j]$ are contained in one rectangle/two rectangles that is/are contained in P_i, by definition of the sequence the permutations represented by P_i occur in π_i which means that every elements are well-ordered, so this cannot happen. We are left with the case in which $\sigma[i]$ and $\sigma[j]$ are contained in rectangles that are contained in P_i and P_j. As the elements are contained in different sets, it can be deduced that they are in different rectangles. Moreover, as σ is decomposed into a direct sum, P_i is on the left and below P_j; as such, $\sigma[i] < \sigma[j]$. The hypothesis indicates that the matchings of $\sigma[i]$ and $\sigma[j]$ do not have the same order, but their matchings are contained in π_i and π_j. Moreover, π_i is on the left and below π_j as π is decomposed into a direct sum. Their matching thus cannot have a different order, which shows that the hypothesis does not hold. Such sequences therefore contain an occurrence. □

Algorithm 5 presents the pseudo-code of an algorithm that constructs such a sequence, which can be used to decide whether σ occurs in π.

Algorithm 5:

Data: The sequence of rectangles $\pi_1, \ldots, \pi_{\ell_\pi}$ of the largest decomposition of π.
Data: The sequence of rectangles $\sigma_1, \ldots, \sigma_{\ell_\sigma}$ of the largest decomposition of σ.
Result: A sequence as describe above.
start with an empty ordered partition and an empty subsequence ;
Initialise s with $s = \sigma_1, \ldots, \sigma_{\ell_\sigma}$;
for *every rectangle* R *in* $\pi_1, \ldots, \pi_{\ell_\pi}$ **do**
    p = the largest sequence of rectangles of s starting from the first rectangle such that this sequence occurs in R ;
    if p *is not empty* **then**
        add p to the ordered partition ;
        add R to the subsequence ;
        remove p from the s ;
if s *is empty* **then**
    return the solution
else
    return that no solution exists

It should be noted that during the loop, the current rectangle is decomposed into a skew sum. We consider three cases: p is empty, p contains one rectangle R_σ or p contains several rectangles $\mathcal{S}$. In the second case, both R and R_σ are decomposed into skew sum, which means that we must decide whether a largest decomposition in skew sum occurs in another largest decomposition in skew sum, which is solved by the current algorithm, so this case is solved by a recursive call. In the third case, $\mathcal{S}$ represents largest decomposition in direct sum, which means that we must decide whether the largest decomposition in direct sum ($\mathcal{S}$) occurs in a largest decomposition in skew sum (the current rectangle). We show how to solve this later.

As seen above, the algorithm computes a sequence. As such, if it yields a solution, then σ occurs in π. To show that our algorithm finds a solution if and only if σ occurs in π, we still need to prove that no solution exists if the algorithm does not find a sequence. We prove the contraposition below.

Suppose that a sequence (π_i, P_i) solution exists and that our algorithm does not compute a solution. We prove that this assertion is not possible. Suppose the first difference between a sequence solution and the sequence computed by the algorithm appears for the rectangle π_i, that is that the set P_i is not the same as in the solution. The set can be different in three ways: The computed set is empty, the computed set is contained in P_i or the computed set contains P_i. The first case is not possible because it implies that nothing occurs in π_i; however, by hypothesis at least P_i occurs in π_i (because of the sequence solution). The second case implies that the algorithm does not compute the largest set of rectangles, which is also not possible. If the third case occurs, we can create a new solution that is closer to the sequence computed: As the set $P_{i'}$ computed by the algorithm contains the set P_i, we replace the pair (π_i, P_i) with the pair $(\pi_i, P_{i'})$ in the sequence of solution, to create a new sequence (that is also a solution). As $P_{i'}$ contains more rectangles than P_i, in the next sets P_j, we remove the rectangles that appear in $P_{i'}$ but not in P_i. We thus

obtain a sequence with either no difference between the sequence computed by the algorithm or where the difference starts after the pair corresponding to p_i. By iterating this process, we can create a solution that does not differ from the sequence computed by the algorithm; the algorithm thus computes a solution, which is a contradiction.

The next paragraph deals with the case where σ and π are decomposed with different sums. We focus on the case in which π is decomposed into a skew sum and σ is decomposed in direct sum.

Proposition 71. *Given the largest decomposition $\pi = \pi_1 \ominus \ldots \ominus \pi_{\ell_\pi}$ and the largest decomposition $\sigma_1 \oplus \ldots \oplus \sigma_{\ell_\sigma} = \sigma$, σ occurs in π if and only if there exists π_i such that σ occurs in π_i.*

This proposition claims that σ can only occur in a rectangle of π. This can easily be explained visually; σ forms a "stair up", if it is contained in more than one rectangle of π, as the rectangles of π form a "stair down", the "stair up" of σ would be cut.

Proof. For the backward implication, remark that π_i occurs in π, and σ occurs in π_i, as such, σ occurs in π. For the forward implication, suppose that the leftmost element of σ is matched to an element of π_α and that the rightmost element of σ is a matched to an element of π_β. The leftmost element of σ is below the rightmost element of σ, as σ is formed by an increasing sequence of rectangles; however every element of π_α is above every element of π_β, as π is formed by a decreasing sequence of rectangles. Such an occurrence is thus not possible. □

This proposition leads directly to the following algorithm for deciding whether σ occurs in π.

Algorithm 6:

Data: The sequence of rectangles $\pi = \pi_1, \ldots, \pi_{\ell_\pi}$ of the largest decomposition of π
Data: σ
Result: Whether σ occurs in π
for *every rectangle in* $\pi_1, \ldots, \pi_{\ell_\pi}$ **do**
  if *σ occurs in the current rectangle* **then**
    return that σ occurs π ;
return that σ does not occur in π ;

It should be noted that, that Algorithm 5 and 6 call each other.

We modify this algorithm for the purpose of computing the p required for the first algorithm: Instead of computing whether σ occurs in π, the algorithm computes the rightmost rectangle such that the $\sigma_1, \ldots, \sigma_i$ occurs in π. If this rectangle is σ_{ℓ_σ} then we can conclude that σ occurs in π.

Algorithm 7:

Data: The sequence of rectangles $\pi = \pi_1, \ldots, \pi_{\ell_\pi}$ of the largest decomposition of π
Data: The sequence of rectangles $\sigma_1, \ldots, \sigma_{\ell_\sigma}$ of the largest decomposition of σ
Result: The rightmost σ_i such that $\sigma_1, \ldots, \sigma_i$ occurs in π
initialising r with nothing ;
for *every rectangle in* $\pi_1, \ldots, \pi_{\ell_\pi}$ **do**
    tmp = The rightmost σ_i such that $\sigma_1, \ldots, \sigma_i$ occurs in the current rectangle ;
    if *tmp is at the right of* r **then**
        $r = tmp$;
return r ;

We also need to modify Algorithm 5 to compute *tmp*: Instead of returning the solution or that no solution exists, the algorithm returns the last rectangle added in p.

We have described two algorithms that can be used to compute every case possible; as a result we have an algorithm for deciding whether σ occurs in π. We still need to prove the complexity claim. To do so, we introduce a closed set of entries for the algorithms, and, use dynamic programming to ensure that each algorithm is only called at most once for each element of the closed set. This gives us a bound to the number of times that each algorithm is called. The first algorithm is called to compute *tmp* for the third algorithm or to decide whether σ occurs in π. In both cases, it only requires two rectangles: σ_i and π_j. The third algorithm is called to compute p for the first algorithm or to decide whether σ occurs in π. The second case requires two rectangles σ and π; however the first case requires a π_i and a sequence of rectangles, especially $s = \sigma_i, \ldots, \sigma_{\ell_\sigma}$. Nonetheless, the last rectangle of the sequence is always the rightmost rectangle of the decomposition. We use a well-suited structure to represent the rectangles so that we can represent the sequence with only one rectangle. A compact separable tree fills the requirement. Indeed, remember that, in the tree representation, a node represents a rectangle and the decomposition of a rectangle is represented by the children of the node. This means that the node representing $\sigma_i, \ldots, \sigma_{\ell_\sigma}$ has the same father. Thus from the node representing σ_i, the node representing σ_{ℓ_σ} is its rightmost brother. To summarise, each algorithm only needs to take one node from the compact tree of σ and one node from the compact tree of π. As the tree of π has $O(n)$ nodes and the tree of σ has $O(k)$ nodes, $O(nk)$ entries are possible for the algorithms. Finally, each algorithm is computed in $O(n)$ times, as they only involve doing an iteration over the rectangles of π. The algorithm thus runs in $O(kn^2)$ time and $O(kn)$ space.

6.4 Deciding the Union of a Separable Permutation

This subsection is devoted to shuffling permutations. Given three permutations (namely π, σ and τ), the problem is to decide whether π is the disjoint union

of two patterns that are order-isomorphic to σ and τ, respectively, in notation $\pi \in \sigma \sqcup\!\sqcup \tau$. For example 937654812 is the disjoint union of two patterns that are order-isomorphic to 2431 and 53241, as can be seen in the highlighted form **937654812**.

Proposition 72. *Given three separable permutations π of size n, σ of size k and τ of size ℓ, there exists an $O(nk^3\ell^2)$ time and $O(nk^2\ell^2)$ space algorithm for deciding whether π is the disjoint union of two patterns that are order-isomorphic to σ and τ, respectively.*

Proposition 72 become more interesting if we observe that the complexity of the problem is still open if we do not restrict the input permutations to be separable [31].

First, if π is a separable permutation and $\pi \in \sigma \sqcup\!\sqcup \tau$, then σ and τ are also separable. As such, we consider the non-trivial case in which σ and τ are separable.

Given that π is a separable permutation we denote by π_{ℓ_π} the rightmost rectangle of the permutation's largest decomposition (especially $\pi = \pi_1 \oplus \ldots \oplus \pi_{\ell_\pi}$). Moreover, we let $\pi(i,j)$ be $\pi_i \oplus \ldots \oplus \pi_j$.

Proof. Consider the following family of subproblems: Given a separable permutation π and a sequence of its largest decomposition $\pi(i_\pi, j_\pi)$, a separable permutation σ and a sequence of its largest decomposition $\sigma(i_\sigma, j_\sigma)$ and a separable permutation τ and a sequence of its largest decomposition $\tau(i_\tau, j_\tau)$, we want to check whether $\pi(i_\pi, j_\pi)$ is the shuffle of $\sigma(i_\sigma, j_\sigma)$ and $\tau(i_\tau, j_\tau)$. We notate this as follows:

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau)) = \begin{cases} True & \text{if } \pi(i_\pi, j_\pi) \in \sigma(i_\sigma, j_\sigma) \sqcup\!\sqcup \tau(i_\tau, j_\tau) \\ False & \text{otherwise.} \end{cases}$$

By definition $\pi \in \sigma \sqcup\!\sqcup \tau$ if and only if $\mathrm{S}(\pi(1, \ell_\pi), \sigma(1, \ell_\sigma), \tau(1, \ell_\tau))$ is true.

Base.

The base cases are when either $\sigma = 1$ or $\tau = 1$. Then, if π is also a leaf then the problem is true; otherwise the problem is false, as elements will be left unmatched in π.

- If π and σ are the trivial permutation of size 1 and τ has no element then $\mathrm{S}(\pi(1,1), \sigma(1,1), \emptyset) = True$.
- If π and τ are the trivial permutation of size 1 and σ has no element then $\mathrm{S}(\pi(1,1), \emptyset, \tau(1,1)) = True$.
- Otherwise the problem is $False$.

Reduction.

Two instances of the problem can represent the same problem. Especially when one of the arguments represents an unique node, we replace this node by all of its children. This happens only when $i_* = j_*$, where $* \in \{\pi, \sigma, \tau\}$.

- If $i_\pi = j_\pi$ then

$$\mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau)) = \mathrm{S}(\pi_{i_\pi}(1, \ell_{\pi_{i_\pi}}), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau))$$

- If $i_\sigma = j_\sigma$ then

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, i_\sigma), \tau(i_\tau, j_\tau)) = \mathrm{S}(\pi(i_\pi, j_\pi), \sigma_{i_\sigma}(1, \ell_{\sigma_{i_\sigma}}), \tau(i_\tau, j_\tau))$$

- If $i_\tau = j_\tau$ then

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, i_\tau)) = \mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau_{i_\tau}(1, \ell_{\tau_{i_\tau}}))$$

Recurrence

The idea of the recursion is to split $\sigma(i_\sigma, j_\sigma)$ and $\tau(i_\tau, j_\tau)$, in every possible way in $\pi(i_\pi, j_\pi)$. Each splitting yields two different instances of the problem. We differentiate two classes of instances: The first class can be characterised by the fact either σ or τ is empty. This is the best case, as it corresponds to deciding whether σ (or τ, depending on which one is empty) is an occurrence of π; we let the other class contain all instances that do not belong to the first one. In all cases, both instances are smaller in size, especially the size of the text permutation is reduced and the size of the first or second pattern is reduced. The recursion uses Property 71 and Lemma 70 to make a "smart" splitting.

- In cases in which $\pi(i_\pi, j_\pi)$ represents a direct sum decomposition, $\sigma(i_\sigma, j_\sigma)$ represents a direct sum decomposition and $\tau(i_\tau, j_\tau)$ represents a skew sum decomposition. Note that according to Property 71, if $\tau(i_\tau, j_\tau))$ occurs in $\pi(i_\pi, j_\pi)$ then it occurs in a unique rectangle, in particular, we can consider the first rectangle of $\pi(i_\pi, j_\pi)$ for the occurrence of $\tau(i_\tau, j_\tau)$ and handle the case accordingly. As such, π_{i_π} can contain
  - an occurrence of $\tau(i_\tau, j_\tau))$ and some part of an occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - an occurrence of $\tau(i_\tau, j_\tau))$ and no occurrence of $\sigma(i_\sigma, j_\sigma)$ or
  - nothing of $\tau(i_\tau, j_\tau))$ and some part of an occurrence of $\sigma(i_\sigma, j_\sigma)$.

  The case in whichs π_{i_π} contains nothing cannot occur, as we want every element of π to be used in an occurrence. In those three cases, what is left of σ and τ has to occur in $\pi(i_\pi + 1, j_\pi)$, which yields us the following solution:

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau))$$

$$=$$

$$\bigcup_{j'_\sigma < j_\sigma} (\mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j'_\sigma), \tau(i_\tau, j_\tau)) \wedge \mathrm{S}(\pi(i_\pi + 1, j_\pi), \sigma(j'_\sigma + 1, j_\sigma), \emptyset))$$

$$OR$$

$$(\mathrm{S}(\pi(i_\pi, i_\pi), \emptyset, \tau(i_\tau, j_\tau)) \wedge \mathrm{S}(\pi(i_\pi + 1, j_\pi), \sigma(i_\sigma, j_\sigma), \emptyset))$$

$$OR$$

$$\bigcup_{j'_\sigma < j_\sigma} (\mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j'_\sigma), \emptyset) \wedge \mathrm{S}(\pi(i_\pi + 1, j_\pi), \sigma(j'_\sigma + 1, j_\sigma), \tau(i_\tau, j_\tau)))$$

- In cases in which $\pi(i_\pi, j_\pi)$ represents a direct sum decomposition, both $\sigma(i_\sigma, j_\sigma)$ and $\tau(i_\tau, j_\tau)$ represent a skew sum decomposition. Based on Property 71, $\sigma(i_\sigma, j_\sigma)$ occurs in a unique child of $\pi(i_\pi, j_\pi)$ and $\tau(i_\tau, j_\tau)$ occurs in a unique child of $\pi(i_\pi, j_\pi)$. Moreover, as every element must be used in an occurrence, $\pi(i_\pi, j_\pi)$ cannot have more than two rectangles. In other words, if $\pi(i_\pi, j_\pi)$ has more than two rectangles, then this instance is false. In addition, if $\pi(i_\pi, j_\pi)$ has two rectangles $\sigma(i_\sigma, j_\sigma)$ occurs π_{i_π}, and $\tau(i_\tau, j_\tau)$ occurs π_{j_π}, or $\sigma(i_\sigma, j_\sigma)$ occurs π_{j_π} and $\tau(i_\tau, j_\tau)$ occurs π_{i_π}:

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau))$$

$$=$$

$$\mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j_\sigma), \emptyset) \wedge \mathrm{S}(\pi(j_\pi, j_\pi), \emptyset, \tau(i_\tau, j_\tau))$$

$$OR$$

$$\mathrm{S}(\pi(i_\pi, i_\pi), \emptyset, \tau(i_\tau, j_\tau)) \wedge \mathrm{S}(\pi(j_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \emptyset)$$

- If $\pi(i_\pi, j_\pi)$, $\sigma(i_\sigma, j_\sigma)$ and $\tau(i_\tau, j_\tau)$ represent direct sum decompositions then according to Lemma 70, π_{i_π} contains
  - some part of the occurrence of $\tau(i_\tau, i_\tau)$ and some part of the occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - some part of the occurrence of $\tau(i_\tau, i_\tau)$ and no element of an occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - no element of an occurrence of $\tau(i_\tau, i_\tau)$ and some part of the occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - the occurrence of $\tau(i_\tau, i_\tau)$ and some part of the occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - the occurrence of $\tau(i_\tau, i_\tau)$ and no element of an occurrence of $\sigma(i_\sigma, j_\sigma)$,
  - some part of the occurrence of $\tau(i_\tau, i_\tau)$ and the occurrence of $\sigma(i_\sigma, j_\sigma)$ or
  - no element of an occurrence of $\tau(i_\tau, i_\tau)$ and the occurrence of $\sigma(i_\sigma, j_\sigma)$.

  In all of these cases, what is left occurs in $\pi(i_\pi + 1, j_\pi)$.

$$\mathrm{S}(\pi(i_\pi, j_\pi), \sigma(i_\sigma, j_\sigma), \tau(i_\tau, j_\tau))$$

$$=$$

$$\bigcup_{\substack{j'_\sigma < j_\sigma \\ j'_\tau < j_\tau}} \mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j'_\sigma), \tau(i_\tau, j'_\tau)) \wedge \mathrm{S}(\pi(i_\pi{+}1, j_\pi), \sigma(j'_\sigma{+}1, j_\sigma), \tau(j'_\tau{+}1, j_\tau))$$

$$OR$$

$$\bigcup_{j'_\sigma < j_\sigma} \mathrm{S}(\pi(i_\pi, i_\pi), \sigma(i_\sigma, j'_\sigma), \emptyset) \wedge \mathrm{S}(\pi(i_\pi + 1, j_\pi), \sigma(j'_\sigma + 1, j_\sigma), \tau(i_\tau, j_\tau))$$

$$OR$$

$$\bigcup_{j'_\tau<j_\tau} \mathrm{S}(\pi(i_\pi,i_\pi),\emptyset,\tau(i_\tau,j'_\tau)) \wedge \mathrm{S}(\pi(i_\pi+1,j_\pi),\sigma(i_\sigma,j_\sigma),\tau(j'_\tau+1,j_\tau))$$

$$OR$$

$$\bigcup_{j'_\tau<j_\tau} \mathrm{S}(\pi(i_\pi,i_\pi),\sigma(i_\sigma,j_\sigma),\tau(i_\tau,j'_\tau)) \wedge \mathrm{S}(\pi(i_\pi+1,j_\pi),\emptyset,\tau(j'_\tau+1,j_\tau))$$

$$OR$$

$$\mathrm{S}(\pi(i_\pi,i_\pi),\sigma(i_\sigma,j_\sigma),\emptyset) \wedge \mathrm{S}(\pi(i_\pi+1,j_\pi),\emptyset,\tau(i_\tau,j_\tau))$$

$$OR$$

$$\bigcup_{j'_\sigma<j_\sigma} \mathrm{S}(\pi(i_\pi,i_\pi),\sigma(i_\sigma,j'_\sigma),\tau(i_\tau,j_\tau)) \wedge \mathrm{S}(\pi(i_\pi+1,j_\pi),\sigma(j'_\sigma+1,j_\sigma),\emptyset)$$

$$OR$$

$$\mathrm{S}(\pi(i_\pi,i_\pi),\emptyset,\tau(i_\tau,j_\tau)) \wedge \mathrm{S}(\pi(i_\pi+1,j_\pi),\sigma(i_\sigma,j_\sigma),\emptyset)$$

- All others cases can be affiliate with one of the above cases, such that it can be solved by applying the idea of that case.

We are left with proving the complexity claim. As before we use a dynamic programming strategy so that we only need to compute each instance of the problem once. As such, we only need to identify the number of different entries possible for the problem to determine its complexity. In this paragraph, let ℓ be the size of τ. Without a loss of generality, we can suppose that $\ell < k$. At first, it seems that the problem has $n^2k^2d\ell^2$ cases: For every permutation, we associate two rectangles of its decomposition.

However, one of the three pairs has redundant information. As, the size of $\pi(i_\pi, j_\pi)$ must be equal to the size of $\sigma(i_\sigma, j_\sigma)$ plus the size of $\tau(i_\tau, j_\tau)$, given $\sigma(i_\sigma, j_\sigma)$, $\tau(i_\tau, j_\tau)$, π and i_π, we can deduce j_π. It should be noted that j_π may not exist, as the size of $\sigma(i_\sigma, j_\sigma)$ plus the size of $\tau(i_\tau, j_\tau)$ may not exactly equal the size of a $\pi(i_\pi, j_\pi)$; however, in such a case, we can immediately say that π is not a shuffle. We thus have $O(nk^2\ell^2)$ different cases to compute. This strategy implies that for every permutation, we have to compute the size of every $\pi(i_\pi, j_\pi)$ possible, which can be pre-computed in $O(n^2)$ time and takes $O(n^2)$ space in the memory.

The worst case for this algorithm is when π contains elements of occurrences for both σ and τ. In this case, we must iterate every j'_σ, $i_\sigma \leq j'_\sigma \leq j_\sigma$ and every j'_τ, $i_\tau \leq j'_\tau \leq j_\tau$ possible. However, we can deduce j'_σ from $\pi(i_\pi, i_\pi)$, $\tau(i_\tau, j'_\tau)$, σ and i_σ (as above). As a result we only need to iterate every j'_τ, $i_\tau \leq j'_\tau \leq j_\tau$ possible. Finally, each recursive problem is solved in constant time by dynamic programming. Computing one case thus takes at most $O(\ell)$ time, which gives us an $O(nk^2\ell^3)$ time and $O(nk^2\ell^2)$ space algorithm. □

6.5 Finding a Maximum Size Separable Subpermutation

The longest common pattern problem for permutations is finding the largest permutation that occurs in each input permutation, which is intended to be the natural counterpart to the classical LCS problem. Rossin and Bouvel [51] provide an $O(n^8)$ time algorithm for computing the largest common separable pattern that occurs in two permutations of size (at most) n, one of these two permutations being separable. This problem was further generalised in [19], which demonstrates that the problem of computing the largest separable pattern that occurs in k permutations of size (at most) n is solvable in $O(n^{6k+1})$ time and $O(n^{4k+1})$ space. It should be noted that the latter problem is **NP**-complete for unbounded k, even if all input permutations are actually separable. The following proposition improves upon the algorithm of Rossin and Bouvel [51].

Proposition 73. *Given a permutation of size n and a separable permutation of size k, the largest common separable pattern that occurs in the two input permutations can be computed in $O(kn^6)$ time and $O(n^4 \log k)$ space.*

Proof. For clarity of exposition, we begin by considering the problem of computing the largest separable pattern that occurs in a single permutation π. We consider the following family of subproblems: For every two $i, j \in [n]$ with $i \leq j$ and every lower and upper bound $\mathrm{lb}, \mathrm{ub} \in [1, 2, \ldots, n]$ with $\mathrm{lb} \leq \mathrm{ub}$, we have the subproblem $P_{i,j,\mathrm{lb},\mathrm{ub}}$, where the semantic is as follows:

$$P_{i,j,\mathrm{lb},\mathrm{ub}} = \max\{|s| : s \text{ is a subsequence of } \pi[i,j],\ \mathit{reduction}(s) \text{ is separable, and every element in } s \text{ is in the interval } [\mathrm{lb},\mathrm{up}]\}.$$

We show that this family of problems is closed under induction.

- **Base**: We have two cases.
  - If $i = j$, then
$$P_{i,i,\mathrm{lb},\mathrm{ub}} = \begin{cases} 1 & \text{if } \mathrm{lb} \leq \pi[i] \leq \mathrm{ub}, \\ 0 & \text{otherwise.} \end{cases}$$
  - If $\mathrm{lb} = \mathrm{up}$, then
$$P_{i,j,\mathrm{b},\mathrm{b}} = \begin{cases} 1 & \text{if there exists } \iota \in [i,j] \text{ such that } \pi[\iota] = \mathrm{b}, \\ 0 & \text{otherwise.} \end{cases}$$
- **Step**: Here $i < j$, $\mathrm{lb} < \mathrm{ub}$, and we must decide whether the optimum $P_{i,i,\mathrm{lb},\mathrm{ub}}$ is achieved with a positive or negative root node. Thus
$$P_{i,j,\mathrm{lb},\mathrm{ub}} = \max\left\{P^+_{i,j,\mathrm{lb},\mathrm{ub}}, P^-_{i,j,\mathrm{lb},\mathrm{ub}}\right\},$$
  where
  - (Hypothesis of a positive node)
$$P^+_{i,j,\mathrm{lb},\mathrm{ub}} = \max\{P_{i,\iota-1,\mathrm{lb},\mathrm{b}-1} + P_{\iota,j,\mathrm{b},\mathrm{ub}} : i < \iota \leq j \text{ and } \mathrm{lb} < \mathrm{b} \leq \mathrm{ub}\}.$$

– (Hypothesis of a negative node)

$$P^{-}_{i,j,\mathrm{lb},\mathrm{ub}} = \max\{P_{i,\iota-1,\mathrm{b}-1,\mathrm{ub}} + P_{\iota,j,\mathrm{lb},\mathrm{b}} : i < \iota \leq j \text{ and } \mathrm{lb} < \mathrm{b} \leq \mathrm{ub}\}.$$

This implies an $O(n^6)$ time and $O(n^4)$ space algorithm for finding the largest separable pattern that occurs in a permutation of size n.

The next step is to consider the problem of finding a longest separable pattern that occurs in two given permutations (that may not be separable themselves) [51]. Let π be a permutation of size n. For every $i, j \in [n]$ with $i \leq j$ and every $\mathrm{ub}, \mathrm{lb} \in [n]$ with $\mathrm{lb}] \leq \mathrm{up}$, we let $\pi[i, j, \mathrm{lb}, \mathrm{up}]$ stand for the subsequence obtained from $\pi[i, j]$ by trimming away all elements above lb or below up. Then, let π_1 and π_2 be two permutations of S_n. We consider the following family of subproblems: For every $i_1, j_1, \mathrm{lb}_1, \mathrm{ub}_1 \in [n]$ with $i_1 \leq j_1$ and $\mathrm{lb}_1 \leq \mathrm{ub}_1$, and every $i_2, j_2, \mathrm{lb}_2, \mathrm{ub}_2 \in [n]$ with $i_2 \leq j_2$ and $\mathrm{lb}_2 \leq \mathrm{ub}_2$, we have the subproblem $P_{i_1,j_1,\mathrm{lb}_1,\mathrm{ub}_1,i_2,j_2,\mathrm{lb}_2,\mathrm{ub}_2}$ whith the following semantic:

$$P_{i_1,j_1,\mathrm{lb}_1,\mathrm{ub}_1,i_2,j_2,\mathrm{lb}_2,\mathrm{ub}_2} = \max\{|s| : s \text{ is a pattern occurring in both } \pi_1[i_1, j_1, \mathrm{lb}_1, \mathrm{ub}_1] \text{ and in } \pi_2[i_2, j_2, \mathrm{lb}_2, \mathrm{ub}_2]\}$$

It is easy to see that this family of subproblems is closed under induction and yields an $O(n^{12})$ time and $O(n^8)$ space algorithm for finding the size of largest separable pattern that occurs in two permutations of size (at most) n. This is a slight improvement compared to [19], which proposes an $O(n^{13})$ time and $O(n^9)$ space algorithm.

The outlined approach can be extended to a polynomial time algorithm for a fixed number of input permutations (as shown in [19]). However, in practice the complexity of this solution is already prohibitive for just two sequences. Therefore, rather than further extending this approach, we focus on underlining how it encompasses other natural problems. Indeed, following Bouvel and Rossin [51], we consider the problem of computing a longest separable pattern that occurs in two input permutations of size at most n, one of these two permutation being separable. For this precise problem, Bouvel and Rossin provide an $O(n^8)$ algorithm. We consider the following family of subproblems: Given σ a separable permutation and T_σ its separating tree. For every node v of T_σ, every two $i, j \in [n]$ with $i \leq j$, and every lower and upper bounds $\mathrm{lb}, \mathrm{ub} \in [n]$ with $\mathrm{lb} \leq \mathrm{up}$, we have the subproblem $P_{v,i,j,\mathrm{lb},\mathrm{ub}}$, with the following semantic:

$$P_{v,i,j,\mathrm{lb},\mathrm{ub}} = \max\{|s| : s \text{ is a common subsequence of } v \text{ and } \pi[i, j] \text{ with all values in the interval } [\mathrm{lb}, \mathrm{ub}].$$

We show that this family of problems is closed under induction.

- **Bases**: We have three base cases:
  - If $i = j]$ then

$$P_{v,i,i,\mathrm{lb},\mathrm{ub}} = \begin{cases} 1 & \text{if } lb \leq \pi[i] \leq \mathrm{ub}, \\ 0 & \text{otherwise.} \end{cases}$$

  - If $\mathrm{lb} = \mathrm{ub}]$ then

$$P_{v,i,j,\mathrm{b},\mathrm{b}} = \begin{cases} 1 & \text{if there exists } \iota \in [i, j] \text{ such that } \pi[\iota] = b, \\ 0 & \text{otherwise.} \end{cases}$$

– If v is a leaf then

$$P_{v,i,j,\mathrm{lb},\mathrm{ub}} = \begin{cases} 1 & \text{if there exists } \iota \in [i,j] \text{ such that } \mathrm{lb} \leq \pi[\iota] \leq \mathrm{ub}, \\ 0 & \text{otherwise.} \end{cases}$$

- **Step**: Here $i < j$, lb $<$ ub, and we let v_L and v_R stand for the left and right children of v, respectively.

  – If v is a positive node then

$$P_{v,i,j,lb,ub} = \max_{i<\iota\leq j} \max_{\mathrm{lb}<\mathrm{b}\leq\mathrm{ub}} P_{v_L,i,\iota-1,\mathrm{lb},\mathrm{b}-1} + P_{v_R,\iota,j,\mathrm{b},\mathrm{ub}}.$$

  – If v is a negative node then

$$P_{v,i,j,lb,ub} = \max_{i<\iota\leq j} \max_{\mathrm{lb}<\mathrm{b}\leq\mathrm{ub}} P_{v_L,i,\iota-1,\mathrm{b}-1,\mathrm{ub}} + P_{v_R,\iota,j,\mathrm{lb},\mathrm{b}}.$$

The above description implies an $O(kn^6)$ time $O(kn^4)$ space algorithm for computing the largest common separable pattern that occurs in two permutations of size (at most) n, one of these two permutations being separable, with thus improves on Rossin and Bouvel [51]. The memory can be reduced to $O(n^4 \log k)$ using the approach detailed Section 6.2.3. □

6.6 Vincular and Bivincular Separable Patterns

This subsection is devoted to vincular pattern and BVP. We prove that detecting a vincular or bivincular separable pattern in a permutation is polynomial time solvable. To the best of our knowledge, this is the first time the PPM problem is proven to be tractable for a generalisation of separable patterns. Since a vincular pattern is a BVP, we focus on BVP.

The algorithm of Section 6.2 cannot be used to find an occurrence of a BVP as we do not have any control over the position and the value of matching elements.

Proposition 74. *Given a permutation π of size n and a bivincular separable pattern σ of size k, there exists an $O(kn^6)$ time and $O(kn^4)$ space algorithm to decide whether σ occurs in π.*

Given a BVP with a separable permutation $\widetilde{\sigma}$, and a subsequence σ', we define the bivincular permutation $\widetilde{\sigma'}$ as the bivincular permutation with the element of σ', with the top line of $\widetilde{\sigma}$ where we remove the elements not in σ' and with bottom line the elements of the node where m and $m+1$ are underlined if and only if m and $m+1$ are element of σ' and m and $m+1$ are underlined in $\widetilde{\sigma}$. Moreover,

- if $\sigma[i]$ is the bottommost element of σ' and $\underline{\sigma[i-1]\sigma[i]}$ or $\llcorner\sigma[i]$ then we add the bottom left corner,
- if $\sigma[i]$ is the topmost element of σ' and $\underline{\sigma[i]\sigma[i+1]}$ or $\sigma[i]\lrcorner$ then we add the bottom right corner,
- if $\sigma[i]$ is the leftmost element of σ' and $\overline{\sigma[i-1]\sigma[i]}$ or $\ulcorner\sigma[i]$ then we add the top left corner, and

- if $\sigma[i]$ is the leftmost element of σ' and $\overline{\sigma[i-1]\sigma[i]}$ or $\sigma[i]^\urcorner$ then we add the top right corner.

In particular (remember that $\sigma(v)$ represents the subsequence embedded in the node v) we represent by $\widetilde{\sigma}(v)$ the BVP formed by the subsequence $\sigma(v)$.

The idea of the algorithm is to use the separable tree of σ to find an occurrence. To find an occurrence of $\widetilde{\sigma}$, we need to find an occurrence for the (permutation embedded in) its left child and the (permutation embedded in) its right child such that those two occurrences are "compatible" with each other. That is, the occurrences must be contained in rectangles such that the rectangle of the occurrence of the left child is to the left of the rectangle of the occurrence of the right child. In addition, one has to be above the other, depending whether σ is decomposed in a direct or skew sum. This is sufficient for finding an occurrence of σ, but not of $\widetilde{\sigma}$. Indeed, the constraints on values and positions are not respected.

Moreover, given that $\widetilde{\sigma} = \widetilde{\sigma_\alpha} \oplus \widetilde{\sigma_\beta}$, if the topmost, bottommost, leftmost, and rightmost elements of $\widetilde{\sigma_\alpha}$ (resp. $\widetilde{\sigma_\beta}$) are respectively on the top, bottom, left, and right edges of the rectangle of the occurrence of $\widetilde{\sigma_\alpha}$ (resp. $\widetilde{\sigma_\beta}$) and if the two rectangles are consecutive on the x-coordinate (with the first minimal rectangle ending at x and the second minimal rectangle starting at $x+1$), then the matchings of the rightmost element of $\widetilde{\sigma_\alpha}$ and the matching of the leftmost element of $\widetilde{\sigma_\beta}$ are consecutive in index. In the same fashion, if the rectangles are consecutive on the y-coordinate (with the first minimal rectangle ending at y and the second minimal rectangle starting at $y+1$), then the matching of the topmost element of $\widetilde{\sigma_\alpha}$ and the matching of the bottommost element of $\widetilde{\sigma_\beta}$ are consecutive in value.

The above remark allows us address the constraint value between the bottommost element of $\widetilde{\sigma_\beta}$ and the topmost element of $\widetilde{\sigma_\alpha}$ and the constraint position between the leftmost element of $\widetilde{\sigma_\beta}$ and the rightmost element of $\widetilde{\sigma_\alpha}$.

Moreover, an occurrence of $\widetilde{\sigma_\alpha}$ (resp. $\widetilde{\sigma_\beta}$) takes care of all constraints on the value and position of the elements in σ_α (resp.σ_β), except for its topmost and rightmost (resp. bottommost and leftmost) elements. The only constraints left are thus on those elements, but we can resolve them by positioning the rectangle correctly.

If u is a leaf such that $\sigma[i] = \sigma(u)$, then we write $\sigma(u+1)$ to represent $\sigma[i+1]$ and $\sigma(u-1)$ to represent $\sigma[i-1]$.

The pattern matching problem with bivincular permutation $\widetilde{\sigma}$ can be solved by the following family of subproblems: Given T_σ the compact separating tree of σ. For every node v of T_σ, every two $i, j \in [n]$ with $i \leq j$, every lower and upper bound $\mathrm{lb}, \mathrm{ub} \in [n]$ with $\mathrm{lb} \leq \mathrm{ub}$, that form a rectangle with left bottom corner (i, lb) and right top corner (j, ub), we wish to decide whether the rectangle contains an occurrence of $\widetilde{\sigma}(v)$. This gives us the following notation:

$$\mathrm{PMB}_{v,i,j,\mathrm{lb},\mathrm{ub}} = \begin{cases} \mathit{True} & \text{if there exists an occurrence of the BVP} \\ & \widetilde{\sigma}(v) \text{ in } \pi \text{ where all elements are contained} \\ & \text{in the rectangle } ((i, \mathrm{lb}), (j, \mathrm{ub})). \\ \mathit{False} & \text{otherwise} \end{cases}$$

By abuse of language, we say that $\widetilde{\sigma}(v)$ occurs in $((i, \mathrm{lb}), (j, \mathrm{ub}))$. This problem can be solved by the following induction:

Base: If v is a leaf, then :

$$\mathrm{PMB}_{v,i,j,\mathrm{lb},\mathrm{ub}} = \begin{cases} True & \text{If } \exists \iota \in [i,j] \text{ and } \pi[\iota] \in [lb, ub] \\ & \text{and if } \overline{\sigma(v)(\sigma(v)+1)}, \text{ then } \pi[\iota] = ub \\ & \text{and if } \sigma(v)\urcorner, \text{ then } \pi[\iota] = ub = n \\ & \text{and if } \overline{(\sigma(v)-1)\sigma(v)}, \text{ then } \pi[\iota] = lb \\ & \text{and if } \ulcorner\sigma(v), \text{ then } \pi[\iota] = lb = 1 \\ & \text{and if } \underline{\sigma(v)\sigma(v+1)}, \text{ then } \iota = j \\ & \text{and if } \sigma(v)\lrcorner, \text{ then } \iota = j = n \\ & \text{and if } \underline{\sigma(v-1)\sigma(v)}, \text{ then } \iota = i \\ & \text{and if } \llcorner\sigma(v), \text{ then } \iota = i = 1 \\ False & \text{otherwise} \end{cases}$$

A leaf occurs in $((i, \mathrm{lb}), (j, \mathrm{ub}))$ if and only if the rectangle is not empty, which is what the first condition tests. The next four conditions assure that the matched element is on an edge of $((i, \mathrm{lb}), (j, \mathrm{ub}))$. For example, if $\underline{\sigma(v)\sigma(v+1)}$ then the matched element must be on the right edge, and intuitively $\overline{\sigma(v+1)}$ will be on the left edge of the "next" rectangle.

Step. Here $i < j$ and $\mathrm{lb} < \mathrm{ub}$ and we let v_L and v_R stand for the left and right children of v respectively.

Suppose that $\sigma(v)$ occurs in $((i, \mathrm{lb}), (j, \mathrm{ub}))$, and that v is a positive node. As such, $\sigma(v) = \sigma(v_L) \oplus \sigma(v_R)$; in other words, $\sigma(v)$ forms a stair up of two rectangles as must the occurrence of $\sigma(v)$. An occurrence of $\sigma(v)$ is thus composed of a left rectangle that contains the occurrence of $\sigma(v_L)$ and a right rectangle that contains the occurrence of $\sigma(v_R)$.

To find whether $\sigma(v)$ occurs in $((i, \mathrm{lb}), (j, \mathrm{ub}))$, we just have to find whether such occurrence of $\sigma(v_L)$ and $\sigma(v_R)$ exist. We can do so by trying every pair of such rectangles. However, to reduce the number of pairs to test and control the position and value of the elements of the occurrence, we require that the left rectangle share the same bottom and left edges as the rectangle $((i, \mathrm{lb}), (j, \mathrm{ub}))$. That is the left rectangle is $((i, \mathrm{lb}), (*, *))$, the right rectangle shares the same top edge and the same right edge as rectangle $((i, \mathrm{lb}), (j, \mathrm{ub}))$ It others words, the right rectangle is $((*, *), (j, \mathrm{ub}))$ and the left rectangle is consecutive in x and y coordinates to the right rectangle. We hence try to find occurrences of $\sigma(v_L)$ and $\sigma(v_R)$ in every pair of rectangles $((i, \iota-1), (\mathrm{lb}, b-1)$ and $((\iota, j), (b, \mathrm{ub}))$ (see Figure 6.7). In notation, this give us the following:

$$\mathrm{PMB}_{v,i,j,\mathrm{lb},\mathrm{ub}} = \bigvee_{\substack{\iota \in (i,j] \\ \mathrm{b} \in (\mathrm{lb},\mathrm{ub}]}} \mathrm{PMB}_{v_L,i,\iota-1,\mathrm{lb},\mathrm{b}-1} \wedge \mathrm{PMB}_{v_R,\iota,j,\mathrm{b},\mathrm{ub}}$$

The case in which v is a negative node can be deduced following the same idea:

$$\mathrm{PMB}_{v,i,j,\mathrm{lb},\mathrm{ub}} = \bigvee_{\substack{\iota \in (i,j] \\ \mathrm{b} \in (\mathrm{lb},\mathrm{ub}]}} \mathrm{PMB}_{v_L,i,\iota-1,\mathrm{b}-1,\mathrm{ub}} \wedge \mathrm{PMB}_{v_R,\iota,j,\mathrm{lb},\mathrm{b}}$$

If no conditions on positions or values exist, this algorithm solves the pattern matching problem: If $\sigma = \sigma_\alpha \oplus \sigma_\beta$, then every element of σ_α is left of and below every element of σ_β. Moreover, if occurrences of σ_α and σ_β exist such that

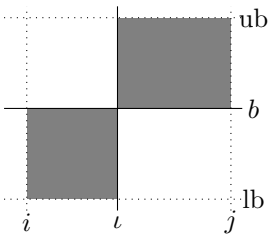


Figure 6.7 – The rectangle given in input is split into a pair of rectangles such that the left rectangle is below and at the left of the right rectangle. Moreover they are next to each other to be optimal.

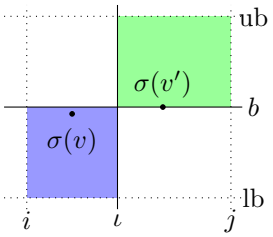


Figure 6.8 – The topmost element of the left rectangle $\sigma(v)$ and the bottommost element of the right rectangle $\sigma(v')$ are consecutive in value if and only if $\sigma(v)$ is matched to $b-1$ and $\sigma(v')$ is matched to b.

the elements of the occurrence of σ_α are left of and below the elements of the occurrence of σ_β, then there exists an occurrence of σ in π. More formally, we have to check every case possible to determine whether the elements really occur on the edges.

For the constraints on positions and values, intuitively, whenever we have $\overline{\sigma(v)\sigma(v')}$ and w as the deepest ancestors of v and v', $\sigma(v)$ is matched to the right edge of the left rectangle and $\sigma(v')$ is matched to the left edge of the right rectangle so that $\sigma(v)$ and $\sigma(v')$ are consecutive in index. Likewise, if $\underline{(\sigma(v))\sigma(v')}$, $\sigma(v)$ is matched to the top edge of the left rectangle and $\sigma(v')$ is matched to the bottom edge of the right rectangle so that $\sigma(v)$ and $\sigma(v')$ are consecutive in value. See Figure 6.9).

Position Constraints. Three types of position constraints are added by BVP .

- If $\llcorner\sigma[1]$, then the leftmost element of σ must be matched to the leftmost element of π ($\sigma[1]$ occurs in $\pi[1]$ in an occurrence of σ in π). Remark that the leaf v such that $\sigma(v) = \sigma[1]$ is the leftmost ancestor of r_σ. Note that the rectangle of a left child shares the same left edge as the rectangle of its father (by induction); plus, this is solved in the base case, which asks for the matched element to be on its left edge. Finally note that the main problem has $x = 1$ for its left edge.
- The condition $\sigma[n_\sigma]\lrcorner$ follows the same idea as the condition $\llcorner\sigma[1]$.

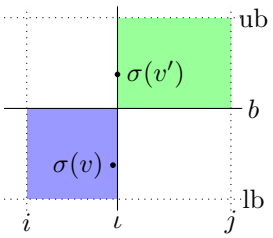


Figure 6.9 – The matching element of $\sigma(v)$ and $\sigma(v')$ are consecutive in position if and only if $\sigma(v)$ is matched to $\pi[\iota - 1]$ and $\sigma(v')$ is matched to $\pi[\iota]$.

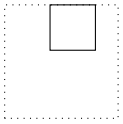


Figure 6.10 – If v occurs in the dotted rectangle then u occurs in the bold rectangle.

- If $\underline{\sigma(v)\sigma(v')}$ (and thus $\sigma(v+1) = \sigma(v')$ and $\sigma(v) = \sigma(v'-1)$), then the index of the occurrences of $\sigma(v)$ and $\sigma(v')$ must be consecutive. In other words, if $\sigma(v)$ occurs in $\pi[j]$ then $\sigma(v')$ must occur in $\pi[j+1]$. Let w be the first common ancestor of v and v' and w_L be the left child of w and w_R be the right child of w. First note that v is the rightmost ancestor of w_L. As such, the rectangle of v shares the same right edge as the rectangle of w_L, plus v' is the leftmost ancestor w_R and thus the rectangle of v' shares the same left edge as the rectangle of w_R. Moreover, these cases are solved as base cases, which ask for the element matching v to be on its right edge, and for the element matching v' to be on its left edge. Finally, remark that the pair of rectangles of two brothers is consecutive in the x-coordinate (by induction).

Before we delve into the value constraints, it is important to highlight a property.

Property 75. *Given a node v and a leaf u, such that $\sigma(u)$ is the maximal (resp. minimal) element of $\sigma(v)$, if v occurs in $((i, \mathrm{lb}), (j, \mathrm{ub}))$, then there exists $i \leq \iota \leq \iota' \leq j$, $\mathrm{lb} \leq b \leq \mathrm{ub}$ and u occurs in $((\iota, b), (\iota', \mathrm{ub}))$ (resp. $((\iota, \mathrm{lb}), (\iota', b))$).*

In other words, if v occurs in a rectangle R and u is the leaf with the maximal element of $\sigma(v)$, then there exists an occurrence of u included in a rectangle that shares the same top edge as R (see Figure 6.10). Especially if $\mathrm{PMB}_{v,i,j,\mathrm{lb},\mathrm{ub}}$ is true then $\mathrm{PMB}_{u,\iota,\iota',b,\mathrm{ub}}$ is true.

Proof. We focus on supporting the assertion in the case in which $\sigma(u)$ is the maximal element of $\sigma(v)$; the other case can be dealt by following the same idea or using a complement's argument. Suppose that the assertion is false. If the rectangle of u cannot have the same top edge as the rectangle of v a rectangle in between exists; in other words, there exists a value larger than $\sigma(u)$ which is not possible. □

Value Constraints. Three types of value constraints are added by BVP .

- If $\ulcorner\sigma(v)$ (and thus $\sigma(v) = 1$), then the minimal value of σ must occur in the minimal value of π. According to Property 75, the rectangle of $\sigma(v)$ has the same bottom edge as the rectangle of $\sigma(r_\sigma)$. This is also solved in the base case, which asks the matching element to be on the bottom edge. Finally, the rectangle of r_σ has $y = 1$ as its bottom edge.
- The $\sigma(v)^\urcorner$ follows the same idea as $\ulcorner\sigma(v)$.
- If $\overline{\sigma(v)\sigma(v')}$ (and thus $\sigma(v') = \sigma(v) + 1$), then the occurrences of $\sigma(v)$ and $\sigma(v')$ must be consecutive. In other words, if $\sigma(v)$ occurs in $\pi[j]$ then $\sigma(v')$ must occur in $\pi[j] + 1$. Let w be the first common ancestor of v and v'. w_L be the left child of w and w_R be the right child of w. if w is positive, then v is a child of w_L and v' is a child of w_R. First $\sigma(v)$ is the maximal element of $\sigma(w_L)$; otherwise, $\sigma(v)$ and $\sigma(v')$ would not be consecutive. According to Property 75, the rectangle of $\sigma(v)$ has the same top edge as the rectangle of $\sigma(w_L)$. In addition, $\sigma(v')$ is the minimal element of $\sigma(w_R)$; otherwise, $\sigma(v)$ and $\sigma(v')$ would not be consecutive. Property 75 therefore dictates that the rectangle of $\sigma(v)$ has the same bottom edge as the rectangle of $\sigma(w_R)$. Moreover, this is solved in the base case and the base case asks the element matching $\sigma(v)$ to be on its top edge and the element matching $\sigma(v')$ to be on its bottom edge. Finally,the pair of rectangles of two brothers is consecutive in the y-coordinate (by induction). We can prove the case in which if w is negative by following the same idea or a complement's argument.

As such, at the end of the algorithm we can decide whether $\widetilde{\sigma}$ occurs in π. Finally, the algorithm has kn^4 different cases, and each case tries every pair of rectangles in constant time (by dynamic programming), which means that each case takes $O(n^2)$ time to solve. This gives us a $O(kn^6)$ time and $O(kn^4)$ space algorithm.

Chapter 7

Wedge Permutations

This chapter focuses on wedge permutations, which are a sub-class of the separable permutations. Wedge permutations have more constraints on their structures than separable permutations, which allows us to reduce the computational time and space of some problem solutions. A formal definition of a wedge permutations is provided below.

Definition 76. *The set of wedge permutations is the set of* $(213, 231)$*-avoiding permutations.*

This chapter is organised as follows. We first describe the structure of a wedge permutation. Section 7.2 is then presents an online linear-time algorithm in the case that both permutations are wedge permutations, whereas Section 7.3 focuses on the case in which only the pattern is a wedge permutation. In Section 7.4, we offer a polynomial-time algorithm for a bivincular wedge permutation pattern. Section 7.5 then considers the problem of finding the longest wedge permutation pattern in permutations.

7.1 Structure of a Wedge Permutation

7.1.1 General Structure.

Property 77. *The first element of any wedge permutations must be either the bottommost or the topmost element.*

Proof. Any other initial element would serve as a '2' in either a 231 or 213 with 1 and n as the '1' and '3' respectively. □

Property 78. π *is a wedge permutation of size* n *if and only if for* $1 \leq i \leq n$, $\pi[i]$ *is an* RLMax *or* RLMin *element.*

Proof. It should be noted that every sub-permutation $\pi[i:]$ is a wedge permutation. As a wedge permutation and given property 77, $\pi[i]$ is either the topmost or the bottommost element of $\pi[i:]$, which is similar to saying that $\pi[i]$ is an RLMax or RLMin element. □

Property 79. *The sub-permutation formed with all of the* RLMax *elements of a wedge permutation is a decreasing subsequence. In the same fashion, the set of* RLMin *elements is an increasing subsequence.*

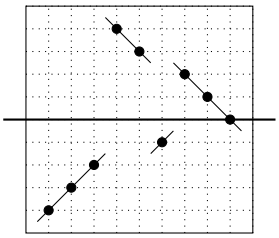


Figure 7.1 – The wedge permutation 123984765. Every point that is on a north-west to south-east line represents a descent element and every point that is on a south-west to north-east line represents an ascent element.

Proof. Indeed, each element that is to the left of and above the next RLMax element is by definition an RLMax element. See Figure 7.1. □

Property 80. *Any* RLMax *element is below any* RLMin *element in a wedge permutation.*

Proof. Suppose that this is not the case. By definition, any RLMax element is the topmost element of all the elements to its right. As such, any RLMin element above an RLMax element must be on its left. However, the definition of an RLMin element dictates that no element comes below, so we have a contradiction. □

Remark 81. *We can draw a horizontal line (passing thought the rightmost element) on a wedge permutation such that the upper half contains only a decreasing subsequence and the lower half contains only an increasing subsequence. This shapes the permutation as a >, which is why it is called a wedge permutation. See Figure 7.1.*

7.1.2 Bijection with Binary Words

As a consequence of Property 78, a wedge permutation can be partitioned into three sets: the set of RLMax elements, the set of RLMin elements and the rightmost element, which can be both a RLMax element and a RLMin element. The partition gives a bijection B between the set of wedge permutations of size n with elements $1, \ldots, n$ and the set of binary words of size $n-1$. The word w which corresponds to π is the word in which each letter at position i represents whether $\pi[i]$ is a RLMax or RLMin element. We also extend this transformation to the subsequence of wedge permutations. Given a subsequence s, the binary word $\mathrm{B}(s)$ is the word in which the letter at position i represents whether $s[i]$ is a RLMax or RLMin element. For any permutation, figuring out whether an element is a RLMax or RLMin element requires one to read the whole permutation from left to right, starting from the element's position.

Property 82. *Let π be a wedge permutation and $1 \leq i < n$. Then,*

1. *$\pi[i]$ is a* RLMin *element if and only if $\pi[i] < \pi[i+1]$ and*
2. *$\pi[i]$ is a* RLMax *element if and only if $\pi[i] > \pi[i+1]$.*

Proof. We focus on proving the case of an RLMin element, as the case of a RLMax element follows the same idea. The forward implication is trivial, as the element is an RLMin element and it is the bottommost element of all the elements to its right. In particular, an RLMin element is below the next element; as such, if $\pi[i]$ is an RLMin, then $\pi[i] < \pi[i+1]$. For the backward implication, if $\pi[i]$ is not an RLMin element, then there exists an element $\pi[j]$ such that $\pi[i] > \pi[j]$ and $i < j$. In other words, we have that $\pi[j] < \pi[i] < \pi[i+1]$, and $i < i+1 < j$ which is equivalent to saying that $\pi[i]\pi[i+1]\pi[j]$ is an occurrence of 231, which is impossible, as π is a wedge permutation. □

7.1.3 Decomposition of a Wedge Permutations into Factors

We need to consider a specific decomposition of σ into *factors*. We split the permutation into maximal sequences of consecutive RLMin and RLMax elements, which are respectively called an RLMin factor and an RLMax factor. This corresponds to splitting the permutation between every pair of RLMin-RLMax and RLMax-RLMin elements (see Figure 7.1). For the special case of a wedge permutation, this also corresponds to splitting the permutation into maximal sequences of elements consecutive in value. We label the factors from right to left. It should be noted that the rightmost element can be both an RLMin and RLMax element. We follow the convention that it is the same type as the element before it for the factorisation, so that the first factor (that is the rightmost factor) always contains at least two elements. For example, $\sigma = 123984765$ is split as $123 - 98 - 4 - 765$. Hence, $\sigma = \text{factor}(4)\ \text{factor}(3)\ \text{factor}(2)\ \text{factor}(1)$ with $\text{factor}(4) = 123$, $\text{factor}(3) = 98$, $\text{factor}(2) = 4$ and $\text{factor}(1) = 765$. See Figure 7.1.

7.2 Both π and σ are Wedge Permutations

This section presents a fast algorithm for deciding if σ occurs in π in the case that both π and σ are wedge permutations.

Proposition 83. *Let π and σ be two wedge permutations. It can be decided whether π has an occurrence of σ in linear time.*

We first need the following lemma.

Lemma 84. *Let π and σ be two wedge permutations. Given a bijection* B *from the wedge permutations of size n to the binary word of size $n-1$. Then, π contains an occurrence of σ if and only if there exists a subsequence t of π such that* $\mathrm{B}(t) = \mathrm{B}(\sigma)$.

Proof. The forward direction is obvious. We prove the backward direction by induction on the size of σ: If $\mathrm{B}(t) = \mathrm{B}(\sigma)$, then t is an occurrence of σ. The base case is a pattern of size 2. Suppose that $\sigma = 12$ and thus $\mathrm{B}(\sigma) = \text{RLMin}$. Let $t = \pi_{i_1}\pi_{i_2}$, $i_1 < i_2$, be a subsequence of π such that $\mathrm{B}(t) = \text{RLMin}$; this reduces to saying that $\pi_{i_1} < \pi_{i_2}$, and hence that t is an occurrence of $\sigma = 12$ in π. A similar argument shows that the lemma holds for $\sigma = 21$. Next, assume that the lemma is true for all patterns up to size $k \geq 2$. Let σ be a wedge permutation of size $k+1$ and let t be a subsequence of π of size $k+1$ such that

$\mathrm{B}(t) = \mathrm{B}(\sigma)$. As $\mathrm{B}(t)[2:] = \mathrm{B}(\sigma)[2:]$ according to the induction hypothesis, it follows that $t[2:]$ is an occurrence of $\sigma[2:]$. Moreover, as $\mathrm{B}(t)[1] = \mathrm{B}(\sigma)[1]$, $t[1]$ and $\sigma[1]$ are both either the bottommost or the topmost elements of their respective sequences. Therefore, t is an occurrence of σ in π. □

A greedy algorithm decides whether σ occurs in π. One needs to read both permutations from left to right stopping at the second to last element. If both elements are RLMax or RLMin, the element of π is added to the solution, and it goes to the next elements in both permutation; it otherwise goes to the next element in the text permutation. As soon as the iteration stops because σ contains no elements, there exists an occurrence; otherwise, no occurrence exists.

Moreover, according to Property 82, we do not need to compute the words $\mathrm{B}(\sigma)$ and $\mathrm{B}(\pi)$ before running the greedy algorithm. Deciding whether an element is RLMax or RLMin can be done locally. The computation can be done while the algorithm is being run, which thus yields an on-line algorithm.

Data: A wedge permutation π
Data: A wedge permutation σ
Result: An occurrence of σ in π, if any.
$S = \emptyset$;
$i = 1$;
$j = 1$;
while *i is smaller than the size of π AND j is smaller than the size of σ* **do**
  if *$\pi[i]$ and $\sigma[j]$ are both* RLMax *element* **then**
    add $\sigma[j]$ to S ;
    $i = i + 1$ $j = j + 1$
  else if *$\pi[i]$ and $\sigma[j]$ are both* RLMin *element* **then**
    add $\sigma[j]$ to S ;
    $i = i + 1$ $j = j + 1$
  else
    $i = i + 1$
if *j is larger than the size of σ* **then**
  add the rightmost element of σ to S ;
  return S ;
else
  return that no solution exists ;

7.3 Only σ is a Wedge Permutation

This section focusses on the permutation pattern matching problem in cases in which only the pattern σ is a wedge permutation. We first highlight some properties of the decomposition in factors of a wedge permutation.

Property 85. *In a wedge permutation, a factor is either an increasing or a decreasing sequence of contiguous elements.*

Proof. A factor is a sequence of RLMin or RLMax elements. It should be recalled that any sub-sequence of RLMin or RLMax forms an increasing or

decreasing subsequence (see Property 79), especially for a factor. For the contiguous part, we focus on RLMin elements, as the same idea can applied to RLMax elements. Suppose that two consecutive RLMin elements $\pi[i]$ and $\pi[i+1]$ are not contiguous. In that case, there exists an element $\pi[j]$ such that $\pi[i] < \pi[j] < \pi[i+1]$. As $\pi[i+1]$ is an RLMin element there is no elements to its right below which are below it, $\pi[j]$ is thus to the left of $\pi[i+1]$. Moreover, as $\pi[j] \neq \pi[i]$, $\pi[j]$ is left of $\pi[i]$. In other words, $j < i < i+1$ and $\pi[i] < \pi[i+1] < \pi[j]$, which means that $\pi[j]\pi[i]\pi[i+1]$ is an occurrence of 231, which is not possible, as π is a wedge permutation. □

Property 86. *A rectangle of π contains an occurrence of an* RLMin *(resp.* RLMax*) factor if and only if it contains an increasing (resp. decreasing) subsequence that is of the same size as or larger than the size of the factor.*

Proof. This is a direct consequence of Property 85 □

We provide a property of wedge permutations that allows us to decide whether a rectangle contains an occurrence of a wedge permutation.

Lemma 87. *There exists an occurrence of a wedge permutation starting with a leftmost* RLMin *(resp.* RLMax*) factor if and only if there are two occurrences o_1 and o_2, such that o_1 is to the left of and below (resp. to the left of and above) o_2, o_1 contains an occurrence of the leftmost factor, o_2 contains an occurrence of the rest and $o = o_1o_2$ is an occurrence of the wedge permutation.*

Proof. If this not true, there exists at least two elements $o[i]$ and $o[j]$ such that $o[i]$ and $o[j]$ are not in the same order as the elements they represent. Without a loss of generality, we can always assume that $i < j$. As o_1 and o_2 are occurrences, $o[i]$ and $o[j]$ cannot be both in o_1 or o_2, this means that $o[i]$ is in o_1 and that $o[j]$ is in o_2. As $o[i]$ represents an element of the leftmost factor, (which is RLMin), $o[i]$ must be below any elements that are on its right. By hypothesis, as $o[i]$ and $o[j]$ are not well ordered, $o[i] > o[j]$ however this is impossible because by hypothesis every element of o_1 is below every element of o_2. □

Remark 88. *Lemma 87 is based on the fact that a wedge permutation is a separable permutation, which means that, any wedge permutation can be decomposed into a direct/skew sum. In particular, for each wedge permutation π there exists a decomposition $\pi_1 \oplus \pi_2$ or $\pi_1 \ominus \pi_2$ in which π_1 is the leftmost factor and π_2 is the rest of the permutation. For example, for the permutation* 123984765*, which is split into factors as* $123 - 98 - 4 - 765$*, scan be written as* $123 \oplus 984765$.

7.3.1 A Simple Algorithm

We show how to compute permutation pattern matching using Lemma 87. The algorithm given in the proof is not the best we can do with respect to time and space complexity, but it helps to clarify the better version described in the proof of Proposition 98.

Lemma 89. *Let σ be a wedge permutation of size k and π a permutation of size n. It can be decided in polynomial time and space whether π contains an occurrence of σ.*

Proof. We solve the problem of deciding whether a rectangle $((j, lb), (j', ub))$ contains an occurrence of factor$(i) \dots$ factor(i'). More formally,

$$\mathrm{PM}^{\pi}_{\sigma}(i, ((j, lb), (j', ub))) = \begin{cases} True & \text{if and only if } ((j, lb), (j', ub)) \text{ contains} \\ & \text{an occurrence of factor}(i) \dots \text{factor}(1) \\ False & \text{Otherwise} \end{cases}$$

Based on Lemma 87, we know that this corresponds to computing an occurrence of factor(i) in a rectangle R_l and an occurrence of factor$(i-1) \dots$ factor(1) in a rectangle R_2 such that R_1 is to the left of and below R_2 if factor(i) is an RLMin factor; and R_1 is to the left of and above R_2 if factor(i) is an RLMax factor. The strategy for doing so is to try every pair of rectangles that follows this condition and determine whether they contain the occurrences. It should be noted that, for the rectangle R_1, we need to compute an occurrence for a unique factor, which we know how to solve as a result of Property 86. For the second R_2, we have the same problem with a different instance. As the new instance has a smaller size than the original one, each instance becomes easier until we have to compute a trivial instance.

Algorithm 8:

Data: A permutation π and a rectangle $((j, \mathrm{lb}), (j', \mathrm{ub})$
Data: A wedge permutations σ and its decomposition into factors $\sigma_\ell \dots \sigma_1$
Data: An index i
Result: Whether σ occurs in the rectangle.

```
if i is equal to 1 then
    if the rectangle ((j, lb), (j', ub) contains an occurrence of σ_1 then
        return True ;
    else
        return False ;
else
    if σ_i is a RLMax factor then
        for every pair of rectangles (r_l, r_r) of in ((j, lb), (j', ub) such that
          r_l is left and below r_r do
            if σ_i occurs in r_l AND σ_{i-1} ... σ_1 occurs in r_r then
                return True ;
            else
                return False ;
    else
        for every pair of rectangles (r_l, r_r) of in ((j, lb), (j', ub) such that
          r_l is left and above r_r do
            if σ_i occurs in r_l AND σ_{i-1} ... σ_1 occurs in r_r then
                return True ;
            else
                return False ;
```

Algorithm 8 is not complete, as we do not know how to compute whether a rectangle contains an occurrence of a factor. The following algorithm solves this issue.

Algorithm 9:

Data: A permutation π and a rectangle $((j, \text{lb}), (j', \text{ub})$
Data: A Factors σ_i
Result: Whether σ_i occurs in the rectangle.
if *σ_i is an* RLMax *factor* **then**
  if *the rectangle* $((j, \text{lb}), (j', \text{ub})$ *contains a decreasing sequence of size equal or larger than the size of* σ_i **then**
    return True ;
  else
    return False ;
if *σ_i is a* RLMin *factor* **then**
  if *the rectangle* $((j, \text{lb}), (j', \text{ub})$ *contains an increasing sequence of size equal or larger than the size of* σ_i **then**
    return True ;
  else
    return False ;

To decide whether a rectangle contains an increasing or decreasing subsequence of sufficient size, Algorithm 9 can compute the LIS or LDS of this rectangle. This can be done in polynomial time in number of elements of the rectangle, this number is bounded by the size of the text. Algorithm 9 thus run in polynomial time in the size of the text. Algorithm 8 have a rectangle as an input and iterate over every way of splitting this rectangle into two rectangles which are comparable. The number of splitting is polynomial bounded by the rectangle in input which is bounded by the size of the the text. Algorithm 8 thus run in polynomial time in the size of the text. □

The following remarks reduce the complexity of Algorithm 8.

Remark 90. *Remark 60 also applies here.*

Remark 91. *Deciding whether a rectangle contains a factor corresponds to finding its longest increasing or decreasing subsequence and comparing this subsequence's size with the factor's size. The best algorithm so far for computing the longest increasing/decreasing is provided by Bespamyatnikh and Segal in [15]. It runs in $O(n \log\log n)$ where n is the size of the permutation we want to compute.*

Remark 92. *Computing every solution of the longest increasing/decreasing subsequence before running the main algorithm offers two advantages:*

1. *We may want to find an occurrence in a rectangle more than one time, and computing the answer once is enough.*
2. *The algorithm in [15] is well suited for our case as it computes not only the longest increasing/decreasing subsequence between j and j' but also for every k and k' such that $j < k < k' < j'$.*

Remark 93. *The algorithm in [15] runs in $O(n \log\log n)$. However, as we add an upper bound and a lower bound to compute every different instance, in our algorithm the whole computation is done in $O(n^3 loglogn)$ time.*

Remark 94. *Given that the pattern starts with an* RLMin *(resp.* RLMax*) element, if the occurrence starts at the left edge of the rectangle (i.e.. $\pi[j]$ is part of the occurrence), then as the leftmost element of the pattern is an* RLMin *(resp* RLMax*) value, and every element of the occurrence is above (resp. below) it, $\pi[j]$ can be used as a lower (resp. upper) bound and thus as the bottom (resp. top) edge of the rectangle. In the algorithm, instead of finding any occurrence we find an occurrence that start at the left edge. This has two consequences on the algorithm:*

1. *We can remove one argument corresponding to the bottom or top edge, as it only depends on $\pi[i]$. We remove the top edge if σ_i is an* RLMin *factor and the bottom edge if σ_i is an* RLMax *factor. To do so, the algorithm takes the left and right edge of the rectangle and a bound. The bound represents the top edge of the rectangle if σ_i is an* RLMin *factor and the bottom edge, otherwise.*

2. *The definition of the problem also changes. Instead of looking for any occurrence on the rectangle, we want an occurrence that starts at the left edge of the rectangle. It should be noted that the occurrence must start at the left edge of the rectangle. We need to modify this part reflect in account the latter change (this corresponds to changing Algorithm 9). Thankfully, the algorithm in [15] takes this into account by It actually computing the longest increasing/decreasing subsequence starting at element $\pi[i]$. As a result, we do not need to modify our algorithm. Moreover, as in accordance with Remark 93 we only have one bound, the whole computation is thus done in $O(n^2 \log\log n)$ time.*

7.3.2 Improving the Simple Algorithm

We extend our research to reduce the number of pairs of rectangles that we have to test. The idea is that when deciding whether a rectangle contains an occurrence of a pattern, to split the rectangle, the algorithm first "cuts" it vertically, to produce two rectangles (with the first to the left of the second one). The algorithm then decides whether the second rectangle contains an occurrence of the "right part of the pattern". From all of the occurrences that the algorithm finds, it is in our best interest to select the one that is contained in the smallest rectangles possible; this ensures that the first rectangle is larger and thus contains more elements, so it has a better chance to contain an occurrence of the left part of the pattern. It should be noted that given that the leftmost factor of the pattern is an RLMin (resp. RLMax) factor, the left, right, and top (resp. bottom) edges of the second rectangle are fixed. Indeed, it should be recalled that in the algorithm, the second rectangle shares its top (resp. bottom) right corner with the original rectangle which provide the right and top (resp. bottom) edges; moreover the algorithm has already "cut" the original rectangle vertically, which gives the left edge. The "smallest rectangle" can thus only be obtained using the bottom (resp. top) edge, which should be the topmost (resp.

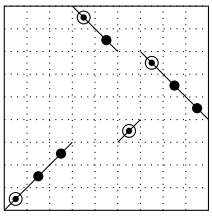


Figure 7.2 – The wedge permutation 123984765. Every line represents a factor, every circled point represents the leftmost element of each factor, and LMEp represent the indexes of those elements.

bottommost) possible. The next property indicates which element are topmost and bottommost, with ensuing lemma formalising what is said above.

However we first introduce the notation LMEp(s), which stands for the leftmost element position: Suppose that s is a subsequence of S, which means that LMEp(s) is the position of the leftmost element of s in S. For every factor, LMEp(factor(j)) thus stands for the position in σ of the leftmost element of factor(j), for example, LMEp(factor(4)) = 1, LMEp(factor(3)) = 4, LMEp(factor(2)) = 6, and LMEp(factor(1)) = 7. See Figure 7.2 for a representation of every LMEp of each factor.

We also provide the following property to know the position of the topmost and leftmost elements on a wedge permutation.

Property 95. *Given a wedge permutation σ if* factor(i) *is an* RLMin *(resp.* RLMax*) factor the topmost (resp. bottommost) element of* factor(i)...factor(1) *is the leftmost element of* factor($i-1$).

Simply put, the bottommost element is on the second factor from the left if the first factor from the left is an RLMax factor; otherwise, the topmost element is on the second factor from the left. See Figure 7.3.

Proof. This lemma states that, given a wedge permutation, if the permutation starts with an RLMin (resp. RLMax) element then the topmost (resp. bottommost) element of this permutation is the first RLMax (resp. RLMin) element (see Figure 7.3). This is easy to see from the shape of a wedge permutation. Formally, the RLMin elements are below the RLMax elements. The topmost element must thus be the first RLMax element, which is the leftmost element of factor($i-1$). □

We also define $\mathrm{S}^{\pi}_{\sigma}(i,j)$ as the set of all subsequences s of $\pi[j:]$ that start at $\pi[j]$ and are occurrences of factor(i)...factor(1).

Lemma 96. *Let σ be a wedge permutation and* factor(i) *be an* RLMin *(resp.* RLMax*) factor σ. Let π be a permutation and s a subsequence of π such that $s \in \mathrm{S}^{\pi}_{\sigma}(i,j)$, and s minimises (resp. maximises) the matching of the leftmost element of* factor($i-1$). *Let s' be a subsequence of π such that $s' \in \mathrm{S}^{\pi}_{\sigma}(i,j)$ and let $t = t's'$ be a subsequence of π that extends s' on the right. Assume that t is an occurrence of* factor($i+1$)...factor(1) *such that the leftmost element of* factor(i) *is matched to $\pi[j]$. The subsequence $t's$ is then also an occurrence of* factor($i+1$)...factor(1) *such that the leftmost element of* factor(i) *is matched to $\pi[j]$.*

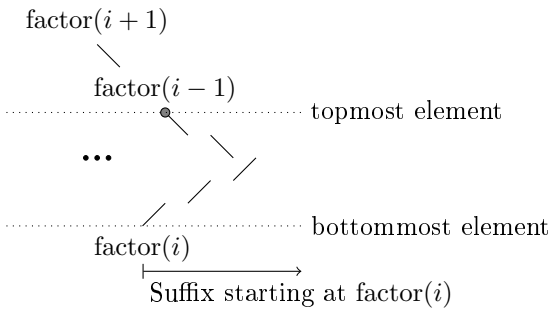


Figure 7.3 – The topmost element of the suffix starting at factor(i) is the leftmost element of factor($i-1$) (represented by the gray dot). $\mathrm{PM}^{\pi}_{\sigma}(i,j)$ is the smallest value of the matching of the topmost element (the gray dot) in all the occurrences of the suffix starting at LMEp(factor(i)) in $\pi[j:]$.

Informally, this lemma states that we can replace the rectangle of an occurrence in $\mathrm{S}^{\pi}_{\sigma}(i,j)$ by another rectangle of an occurrence in $\mathrm{S}^{\pi}_{\sigma}(i,j)$ such that the second rectangle shares the same edges as the first one, except for the top edge, which is lower. In particular, the second rectangle can be chosen as the one with the lowest top edge of all rectangles that share the right, bottoms and left edges. More formally, given any occurrence of factor($i+1$) factor(i) ... factor(1), in which factor(i) is an RLMin (resp. RLMax) factor, we can replace the part of the occurrence where factor(i) ... factor(1) occurs, by any occurrence that minimises (resp. maximises) the leftmost element of factor($i-1$). Indeed, the leftmost element of factor($i-1$) is the topmost (resp. bottommost) element of factor(i) ... factor(1) (see Figure 7.3).

Proof. Let us consider the case in which factor(i) is an RLMin factor. By definition s is an occurrence of σ[LMEp(factor(i)) :]. First of all, remark that t' is an occurrence of factor($i+1$). To prove that $t's$ is an occurrence of factor($i+1$)...factor(1), we thus need to prove that the elements of t' are above the elements of s. Since $t's'$ is an occurrence of factor($i+1$)...factor(1) it follows that the elements of t' are above the elements of s'. Moreover, the topmost element of s is below (or equal to) the topmost element of s'; as such, the elements of s are below the elements of t'. We use a symmetric argument if factor(i) is an RLMax factor. □

Lemma 97. *Let σ be a wedge permutation,* factor(i) *be an* RLMin *(resp.* RLMax*) factor and s be a subsequence of π such that $s \in \mathrm{S}^{\pi}_{\sigma}(i,j)$ and it minimises (resp. maximises) the matching of the leftmost element of* factor($i-1$)*. The following statements are equivalent:*

- *There exists an occurrence of σ in π in which the leftmost element of* factor(i) *is matched to $\pi[j]$, and*
- *There exists an occurrence t of* factor(ℓ)...factor($i+1$) *in $\pi[:j-1]$ such that ts is an occurrence of σ in π with the leftmost element of* factor(i) *matched to $\pi[j]$.*

Proof. The backward direction is trivial, and the forward direction follows from Lemma 96. □

This lemma goes a step further from the previous one, as it states that if there is no occurrence of σ in π in which the leftmost element of factor(i) is matched to $\pi[j]$ and such that the leftmost element of factor($i-1$) is minimised (resp. maximised), then no occurrence exists at all.

Proposition 98. *Let σ be a wedge permutation of size k and π be a permutation of size n. It can decided in $O(\max(kn^2, n^2\log(\log(n)))$ time and $O(n^3)$ space if π contains an occurrence of σ.*

Proof. This algorithm is similar to the previous one, as it takes a rectangle and decides whether it contains an occurrence of σ. However, instead of returning true or false, assuming that factor(i) is an RLMin (resp. RLMax) factor the algorithm returns the optimal value of the top (resp. bottom) edge of a rectangle that contain the pattern (or a special value that indicates that no occurrence exists). In the recursion, the computed value of the right rectangle is thus used as the top (resp. bottom) edge of the left rectangle.

We first introduce a set of values needed to solve the problems. Let $LIS_\pi(j, j'$ *bound*) (resp. $LDS_\pi(j, j', bound)$) be the longest increasing (resp. decreasing) subsequence in $\pi[j : j']$ starting at $\pi[j]$, with every element of this subsequence being smaller (resp. larger) than or equal to *bound*. LIS_π and LDS_π can be computed in $O(n^2\log(\log(n)))$ time (see [15]). $LIS_\pi(j, j', bound)$ (resp. $LDS_\pi(j, j', bound)$) allows us to decide the existence of an occurrence starting at $\pi[j]$ of any LRMin (resp. LRMax) factor in the rectangle ((j,$\pi[j]$), (j', *bound*)) (resp. ((j, *bound*) ,(j', $\pi[j]$)).

Given an RLMin (resp. RLMax) factor, namely factor(i), of σ and a position j in π, we want the value of the top (resp. bottom) edge of any rectangle with bottom left (resp. top left) corner $(j, \pi[j])$ containing an occurrence of factor(i)...factor(1) starting at $\pi[j]$ and minimises (resp. maximises) its top (resp. bottom) edge or a value, which indicates that no occurrence exists in this rectangle. This can be expressed more formally as follows:

- If factor(i) is an RLMin factor, then

$$\mathrm{PM}_\sigma^\pi(i, j) =$$

$$\begin{cases} \text{The top edge of any rectangle with bottom left corner } (j, \pi[j]) \\ \text{with right edge } n \text{ containing an occurrence of} \\ \text{factor}(i)\ldots\text{factor}(1) \text{ starting at } \pi[j] \text{ which minimises the top edge.} \\ \\ \text{Or } \infty \text{ if no occurrence exists.} \end{cases}$$

- If factor(i) is an RLMax factor, then

$$\mathrm{PM}_\sigma^\pi(i, j) =$$

$$\begin{cases} \text{The bottom edge of any rectangle with top left corner } (j, \pi[j]) \\ \text{with right edge } n \text{ containing an occurrence of} \\ \text{factor}(i)\ldots\text{factor}(1) \text{ starting at } \pi[j] \text{ which maximises its bottom edge.} \\ \\ \text{Or } 0 \text{ if no occurrence exists.} \end{cases}$$

By definition, there exists an occurrence of σ in π if and only if there exists a $j \in \{1, \ldots, n\}$ such that $\mathrm{PM}_\sigma^\pi(\ell, j) \neq 0$ and $\mathrm{PM}_\sigma^\pi(\ell, j) \neq \infty$ with ℓ the number of factors in σ.

We show how to compute, these values recursively.

- If the permutation is reduced to an RLMin (resp. RLMax) factor, then one has to compute the top (resp. bottom) edge of a rectangle that contains an increasing or decreasing subsequence that is the same size or larger than the size of the unique factor starting at $\pi[j]$. This case occurs when $i = 1$:
  - If factor(i) is an RLMin factor, then

    $$\mathrm{PM}_\sigma^\pi(1, j) = \min\{\pi[j'] \mid |factor(1)| \leq LIS_\pi(j, j', \pi[j'])\}_{j' \geq j} \cup \{\infty\}.$$

  - If factor(i) is an RLMax factor, then

    $$\mathrm{PM}_\sigma^\pi(1, j) = \max\{\pi[j'] \mid |factor(1)| \leq LDS_\pi(j, j', \pi[j'])\}_{j' \geq j} \cup \{0\}.$$

- Otherwise, if factor(i) is an RLMin (resp. RLMax) factor, we split the rectangle into two rectangles R_1 and R_2 such that R_1 is left of and below (resp. left of and above) R_2. R_1 contains an occurrence of factor(i) starting at j, and R_2 contains an occurrence of factor$(i-1)\ldots factor(1)$ starting at $j' > j$. As mentioned before, we only need to test the pair of rectangles in which the rectangle R_2 has for its left edge j' and has the highest bottom (resp. lowest top) edge that contains an occurrence. For R_2, we thus want a rectangle starting at j', which contains an occurrence of factor$(i-1)\ldots factor(i')$ starting at j' with maximise the bottom (resp. minimise the top) edge. Moreover, we also wish to know the value of the bottom (resp. top) edge to use it as a bound for R_1 so that R_1 is below R_2. In other words we want $\mathrm{PM}_\sigma^\pi(i-1, j')$. As before, R_1 has to be as large as possible. Moreover, to ensure that R_1 is to the left of and below (resp. to the left and above) R_2, R_1 must have $(j'-1, \mathrm{PM}_\sigma^\pi(i-1, j') - 1)$ as its top right corner (resp. $(j'-1, \mathrm{PM}_\sigma^\pi(i-1, j') + 1)$ as its bottom right corner). Finally, in all of the "correct" pairs of rectangles, we need the value of the top (resp. bottom) edge of a rectangle that minimises the top edge (resp. maximises the bottom edge) containing the pair of rectangles. It should be noted that the top (resp. bottom) edge has a value of $\pi[j']$ (the matching of LMEp(factor$(i-1)$)). Formally, this can be expressed as follows:

- If factor(i) is an RLMin factor, then

  $$\mathrm{PM}_\sigma^\pi(i, j) =$$

  $\min\{\infty\} \cup \{\pi[j'] \mid b = \mathrm{PM}_\sigma^\pi(i-1, j')$ is not 0 and $|factor(i)| \leq LIS_\pi(j, j'-1, b-1)\}_{j' < j}$

- If factor(i) is an RLMax factor, then

  $$\mathrm{PM}_\sigma^\pi(i, j) =$$

  $\max\{0\} \cup \{\pi[j'] \mid b = \mathrm{PM}_\sigma^\pi(i-1, j')$ is not ∞ and $|factor(i)| \leq LDS_\pi(j, j'-1, b+1)\}_{j' < j}$

The number of factors is bound by k. All instances of LIS_π and LDS_π can be computed in $O(n^2 \log(\log(n))$ time, which takes $O(n^3)$ space (see [15]). There are n base cases that can be computed in $O(n)$ time, which means that; computing all base cases takes $O(n^2)$ time. There are kn different instances of PM, each taking $O(n)$ time to compute; as such, computing every instance of PM takes $O(kn^2)$ time. Computing all of the values therefore takes $O(\max(kn^2, n^2 \log(\log(n)))$ time. As each value takes $O(1)$ space, the problem takes $O(kn^2)$ space, but LIS_π and LDS_π take $O(n^3)$ space. □

7.4 Bivincular Wedge Permutation Patterns

This section is devoted to the pattern matching problem with bivincular wedge permutation patterns. It should be recalled that a BVP generalises a permutation pattern by being able to force elements in an occurrence to be consecutive in values or/and positions. Intuitively, we cannot use the previous algorithm, as the restrictions on position and value are not managed.

Given a bivincular permutation pattern $\widetilde{\sigma}$, we let $\widetilde{\sigma}[i:]$ refer to the bivincular permutation pattern, which has for its top line the top line of $\widetilde{\sigma}$ in which we remove the elements before i. Moreover, $\widetilde{\sigma}[i:]$ has for bottom line the elements of $\sigma[i:]$ in which m and $m+1$ are underlined if and only if m and $m+1$ are in $\sigma[i:]$, and m and $m+1$ are underlined in $\widetilde{\sigma}$. For example, given $\widetilde{\sigma} = \begin{smallmatrix} 12\overline{34}\urcorner \\ \llcorner 21\underline{43} \end{smallmatrix}$, $\sigma = 2143$ and $\widetilde{\sigma}[2:] = \begin{smallmatrix} 2\overline{34}\urcorner \\ \underline{143} \end{smallmatrix}$.

Below, we describe other structure properties of wedge permutations needed to solve the problem.

Lemma 99. *Let σ be a wedge permutation.*

- *If $\sigma[i]$ is an* RLMin *element and not the rightmost element, then $\sigma[i]+1$ is to the right of $\sigma[i]$.*
- *If $\sigma[i]$ is an* RLMax *element and not the rightmost element, then $\sigma[i]-1$ is to the right of $\sigma[i]$.*

Proof. For the first point, by contradiction, $\sigma[i]+1$ would serve as a '2', $\sigma[i]$ would serve as a '1' and $\sigma[i+1]$ would serve as a '3' in an occurrence of 213. For the second point, by contradiction, $\sigma[i]-1$ would serve as a '2', $\sigma[i]$ would serve as a '3' and $\sigma[i+1]$ would serve as a '1' in an occurrence of 231.

□

Lemma 100. *Let σ be a wedge permutation.*

- *If m and $m+1$ are both* RLMin *elements, then any element between m and $m+1$ (if any) is an* RLMax *element.*
- *If m and $m-1$ are both RLmax elements, then any element between m and $m-1$ (if any) is an* RLMin *element.*

Proof. For the first point, by contradiction, let $\sigma[\alpha] = m$ and $\sigma[\beta] = m+1$ and suppose that there exists $\alpha < \gamma < \beta$, such that $\sigma[\gamma]$ is an RLMin element. As we know that RLMin elements are strictly increasing. $\sigma[\alpha] < \sigma[\gamma] < \sigma[\beta]$, which contradicts the fact that $\sigma[\beta] = \sigma[\alpha] + 1$.

□

Proposition 101. *Let $\widetilde{\sigma}$ be a bivincular wedge permutation pattern of size k and π be a permutation of size n. It can be decided in $O(kn^4)$ time and $O(kn^3)$ space if π contains an occurrence of $\widetilde{\sigma}$.*

Proof. Consider the following problem: Given a lower bound lb, an upper bound ub, a position i of σ and a position j of π, we want to know if there is an occurrence of $\widetilde{\sigma}[i:]$ in $\pi[j:]$ with every element of the occurrence in $[\text{lb}, \text{ub}]$ and starting at $\pi[j]$. In other words, we wish to determine if the rectangle with the bottom left corner (j, lb) and the top right corner (n, ub) contains an occurrence of $\widetilde{\sigma}[i:]$ starting at $\pi[j]$. More formally stated,

$$\text{PM}_\pi^{\widetilde{\sigma}}(\text{lb}, \text{ub}, i, j) =$$

$$\begin{cases} true & \text{if } \pi[j:] \text{ has an occurrence of the BVP } \widetilde{\sigma}[i:] \\ & \text{with every element of the occurrence in } [\text{lb}, \text{ub}] \\ & \text{and starting at } \pi[j] \\ & \text{and if } \overline{\sigma[i](\sigma[i]+1)} \text{ or } \sigma[i]^{\urcorner} \text{ appear in } \widetilde{\sigma} \text{ then } \pi[j] = \text{ub} \\ & \text{and if } \overline{(\sigma[i]-1)\sigma[i]} \text{ or } {}^{\ulcorner}\sigma[i] \text{ appear in } \widetilde{\sigma} \text{ then } \pi[j] = \text{lb} \\ false & \text{otherwise} \end{cases}$$

Clearly if ${}_{\llcorner}\sigma[1]$ does not appear in $\widetilde{\sigma}$ then π contains an occurrence of $\widetilde{\sigma}$ if and only if $\bigcup_{0<j} \text{PM}_\pi^{\widetilde{\sigma}}(1, n, 1, j)$ is true. If ${}_{\llcorner}\sigma[1]$ appears in σ then π contains an occurrence of the bivincular wedge permutation pattern $\widetilde{\sigma}$ if and only if $\text{PM}_\pi^{\widetilde{\sigma}}(1, n, 1, 1)$ is true.

We show how to compute those values recursively. Informally, given that $\sigma[i]$ is an RLMin element, finding an occurrence of $\widetilde{\sigma}[i:]$ in $\pi[j:]$ requires us to find a rectangle R in π such that R contains an occurrence of $\widetilde{\sigma}[i+1:]$ and R is to the right and above $\pi[j]$. Moreover

- If $\underline{\sigma[i]\sigma[i+1]}$, then we require that $\pi[j]$ and R be next to each other horizontally and that the occurrence in R starts at the left edge of R. In other words, R has $j+1$ for its left edge.
- If $\overline{\sigma[i](\sigma[i]+1)}$ then we require that $\pi[j]$ and R be next to each other vertically and that the minimal element in the occurrence is on the bottom edge of R. In other words, R has for its bottom edge $\pi[j]+1$.

Note that the parameters lb, ub and j correspond to the occurrence of $\widetilde{\sigma}[i+1:]$ and are respectively, the bottom, the top and the left edges of the rectangle R. The case in which $\sigma[i]$ is an RLMax element can be dealt with symmetrically. More formally stated:

BASE:

$$\text{PM}_\pi^{\widetilde{\sigma}}(\text{lb}, \text{ub}, k, j) = \begin{cases} true & \text{if } \pi[j] \in [\text{lb}, \text{ub}] \\ & \text{and if } \sigma[k]_{\lrcorner} \text{ appears in } \widetilde{\sigma} \text{ then } j = n \\ & \text{and if } {}_{\llcorner}\sigma[k] \text{ appears in } \widetilde{\sigma} \text{ then } j = 1 \\ & \text{and if } \sigma[k]^{\urcorner} \text{ appears in } \widetilde{\sigma} \text{ then } \pi[j] = \text{ub} = n \\ & \text{and if } {}^{\ulcorner}\sigma[k] \text{ appears in } \widetilde{\sigma} \text{ then } \pi[j] = \text{lb} = 1 \\ & \text{and if } \overline{(\sigma[k]-1)\sigma[k]} \text{ appears in } \widetilde{\sigma} \text{ then } \pi[j] = \text{lb} \\ & \text{and if } \overline{(\sigma[k]\sigma[k]+1)} \text{ appears in } \widetilde{\sigma} \text{ then } \pi[j] = \text{ub} \\ false & \text{otherwise} \end{cases}$$

The base case finds an occurrence of the rightmost element of the pattern. If the rightmost element does not have any restriction on positions and values, then $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \mathrm{ub}, k, j)$ is true if and only if $\sigma[k]$ is matched to $\pi[j]$. This is true if $\pi[j] \in [\mathrm{lb}, \mathrm{ub}]$. If $\sigma[k]_{\lrcorner}$ appears in $\widetilde{\sigma}$ then $\sigma[k]$ must be matched to the rightmost element of π; thus j must be n. If ${}_{\llcorner}\sigma[k]$ appears in $\widetilde{\sigma}$ then $\sigma[k]$ must be matched to the leftmost element; thus j must be 1. If $\sigma[k]^{\urcorner}$ appears in $\widetilde{\sigma}$ then $\sigma[k]$ must be matched to the topmost element, which is n. If ${}^{\ulcorner}\sigma[k]$ appears in $\widetilde{\sigma}$ then $\sigma[k]$ must be matched to the bottommost element, which is 1. If $\overline{(\sigma[k]-1)\sigma[k]}$ appears in $\widetilde{\sigma}$, then the matching elements of $\sigma[k]$ and $\sigma[k]-1$ must be consecutive in value, by recursion the value of the element matching $\sigma[k]-1$ plus 1 is recorded in lb. $\sigma[k]$ must thus be matched to lb. If $\overline{(\sigma[k]\sigma[k]+1)}$ appears in $\widetilde{\sigma}$, then the element matching $\sigma[k]$ and $\sigma[k]+1$ must be consecutive in value, by recursion the value of the element matching $\sigma[k]+1$ minus 1 is recorded in ub. $\sigma[k]$ must thus be matched to ub.

STEP:

We consider three cases for the problem $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \mathrm{ub}, i, j)$:

- If $\pi[j] \notin [\mathrm{lb}, \mathrm{ub}]$, then:

$$\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \mathrm{ub}, i, j) = false$$

  which is immediate from the definition.

- If $\pi[j] \in [\mathrm{lb}, \mathrm{ub}]$ and $\sigma[i]$ is an RLMin element, then:

$$\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \mathrm{ub}, i, j) =$$

$$\begin{cases} \bigvee_{\ell>j} \mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, \ell) & \text{if } \sigma[i] \text{ is not underlined} \\ & \text{and } \sigma[i] \text{ is not overlined} \\ \bigvee_{\ell>j} \mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, \ell) & \text{if } \sigma[i] \text{ is not underlined} \\ & \text{and } \overline{(\sigma[i]-1)\sigma[i]} \text{ or } {}^{\ulcorner}\sigma[i] \\ & \text{appears in } \widetilde{\sigma} \\ & \text{and } \pi[j] = \mathrm{lb} \\ \mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, j+1) & \text{if } \underline{\sigma[i]\sigma[i+1]} \\ & \text{appears in } \widetilde{\sigma} \\ & \text{and } \sigma[i] \text{ is not overlined} \\ \mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, j+1) & \text{if } \underline{\sigma[i]\sigma[i+1]} \\ & \text{and } \overline{(\sigma[i]-1)\sigma[i]} \text{ or } {}^{\ulcorner}\sigma[i] \\ & \text{appear in } \widetilde{\sigma} \\ & \text{and } \pi[j] = \mathrm{lb} \\ false & \text{otherwise} \end{cases}$$

  It should be noted that $\sigma[i]$ can be matched to $\pi[j]$ because $\pi[j] \in [\mathrm{lb}, \mathrm{ub}]$. As such, if $\pi[j+1:]$ has an occurrence of $\widetilde{\sigma}[i+1:]$ with every element of the occurrence in $[\pi[j]+1, \mathrm{ub}]$, if the condition on position between $\sigma[i]$ and if $\sigma[i+1]$ (if any) is respected and the condition on value between $\sigma[i]$ and $\sigma[i]+1$ (if any) is respected, then $\pi[j:]$ has an occurrence of $\widetilde{\sigma}[i:]$. To decide the latter condition, by recursion, it is sufficient to know if there exists ℓ, $j < \ell$ such that $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, \ell)$ is true. The first case

corresponds to an occurrence without restrictions on position and on value. It is enough to know if there exists $\ell > j$ such that $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\pi[j]+1, \mathrm{ub}, i+1, \ell)$ is true. The second case asks for the matching of $\sigma[i]-1$ and $\sigma[i]$ to be consecutive in value, but the matching of $\sigma[i]-1$ is $\mathrm{lb}-1$; as such, we want $\pi[j] = \mathrm{lb}$. As the third case asks for the matching of $\sigma[i]$ and $\sigma[i+1]$ to be consecutive in positions, the matching of $\sigma[i+1]$ must be $\pi[j+1]$. The fourth case is a union of the second and third cases.

- If $\pi[j] \in [\mathrm{lb}, \mathrm{ub}]$ and $\sigma[i]$ is an RLMax element, then:

$$\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \mathrm{ub}, i, j) =$$

$$\begin{cases} \bigvee_{\ell > j} \mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \pi[j]-1, i+1, \ell) & \text{if } \sigma[i] \text{ is not underlined} \\ & \text{and } \sigma[i] \text{ is not overlined} \\ \bigvee_{\ell > j} \mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \pi[j]-1, i+1, \ell) & \text{if } \sigma[i] \text{ is not underlined} \\ & \text{and } \overline{\sigma[i](\sigma[i]+1)} \text{ or } \sigma[i]^{\urcorner} \\ & \text{appear in } \widetilde{\sigma} \\ & \text{and } \pi[j] = \mathrm{ub} \\ \mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \pi[j]-1, i+1, j+1) & \text{if } \underline{\sigma[i]\sigma[i+1]} \\ & \text{appears in } \widetilde{\sigma} \\ & \text{and } \sigma[i] \text{ is not overlined} \\ \mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, \pi[j]-1, i+1, j+1) & \text{if } \underline{\sigma[i]\sigma[i+1]} \\ & \text{and } \overline{\sigma[i](\sigma[i]+1)} \text{ or } \sigma[i]^{\urcorner} \\ & \text{appear in } \widetilde{\sigma} \\ & \text{and } \pi[j] = \mathrm{ub} \\ false & \text{otherwise} \end{cases}$$

The same remark as in the previous case holds.

Some constraints do not appear when we compute $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$. In particular, in the case in which $\sigma[i]$ is an RLMin, the conditions of $\underline{\sigma[i-1]\sigma[i]}$ or/and $\overline{\sigma[i](\sigma[i]+1)}$ are not considered in the recursion. Moreover, $\underline{\sigma[i-1]\sigma[i]}$ is not considered because it is up to $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i-1, *)$ to ensure that the elements corresponding to elements at position $\underline{i-1 \text{ and } i}$ in an occurrence are next to each other in position. In addition $\overline{\sigma[i](\sigma[i]+1)}$ is not considered because $\sigma[i]+1$ is to the right of $\sigma[i]$ (see Lemma 99) and addressed during the calls of $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i', *)$ for some $i' > i$. When $\sigma[i]$ is an RLMax $\underline{\sigma[i-1]\sigma[i]}$ does not appear in the conditions for the exact same reason, and $\overline{\overline{(\sigma[i]-1)\sigma[i]}}$ because $\sigma[i]-1$ is to the right of $\sigma[i]$.

The procedure clearly returns true if π contains an occurrence of $\widetilde{\sigma}$ if constraints on position or value exist. We next discuss how the position and value constraints are taken into account so that the algorithm returns true if and only if π has an occurrence of $\widetilde{\sigma}$.

Position Constraints. Three types of position constraints can be added by the underlined elements.

- If $\llcorner\sigma[1]$ appears in $\widetilde{\sigma}$, then the leftmost element of σ must be matched to the leftmost element of π ($\sigma[1]$ is matched to $\pi[1]$ in an occurrence of $\widetilde{\sigma}$ in π). This constraint is satisfied by requiring that the occurrence starts at the leftmost element of π: if $\mathrm{PM}_\pi^{\widetilde{\sigma}}(1, n, 1, 1)$ is true.

- If $\sigma[k]\lrcorner$ appears in $\widetilde{\sigma}$, then the rightmost element σ must be matched to the rightmost element of π ($\sigma[k]$ is matched to $\pi[n]$ in a occurrence of $\widetilde{\sigma}$ in π). This constraint is checked in the base case.

- If $\sigma[i]\sigma[i+1]$ appears in $\widetilde{\sigma}$, then the positions of the element matching $\sigma[i]$ and $\sigma[i+1]$ must be consecutive. In other words, if $\sigma[i]$ is matched to $\pi[j]$ then $\sigma[i+1]$ must be matched to $\pi[j+1]$. We ensure this restriction, by recursion, by requiring that the matching of $\sigma[i+1:]$ starts at position $j+1$.

Value Constraints. Three types of value constraints can be added by the overlined elements.

- If $\ulcorner\sigma[i]$ appears in $\widetilde{\sigma}$ (and thus $\sigma[i] = 1$) then the bottommost element of σ must be matched to the bottommost element of π.
  - If $\sigma[i]$ is an RLMin element, then:
    - Every problem $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, *, i, *)$ is true only if $\sigma[i]$ is matched to the element with a value lb (by recursion). Tt is thus enough to require that $\mathrm{lb} = 1$.
    - The recursive calls for $\sigma[1]$, ..., $\sigma[i-1]$ do not change the lower bound, as $\sigma[i]$ is the leftmost RLMin element. Indeed the element 1 is the leftmost RLMin for any permutation. Finally, the recursive calls for RLMax elements do not change the lower bound.
    - The first call of the problem has $\mathrm{lb} = 1$ for parameter.

    As a result, $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$ returns true only if $\sigma[i]$ is matched to 1.
  - If $\sigma[i]$ is an RLMax element, then $i = k$ ($\sigma[i]$ is the rightmost element). Every $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i\ , *)$ is thus a base case and true only if $\sigma[i]$ is matched to lb. Moreover, recursive calls for RLMax elements do not change the lower bound. As such, $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$ returns true only if $\sigma[i]$ is matched to $\mathrm{lb} = 1$.

- If $\sigma[i]\urcorner$ appears in $\widetilde{\sigma}$ (and thus $\sigma[i] = k$), then the topmost element of σ must be matched to the topmost element of π.
  - If $\sigma[i]$ is an RLMax element, then:
    - Every problem $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, \mathrm{ub}, i, *)$ is true only if $\sigma[i]$ is matched to the element with value ub (by recursion). It is thus sufficient to require that $\mathrm{ub} = n$.
    - The recursive calls for $\sigma[1]$, ..., $\sigma[i-1]$ do not change the upper bound, as $\sigma[i]$ is the leftmost RLMax element. Indeed the element k is the leftmost RLMax for any permutation. Finally, the recursive calls for RLMin elements do not change the upper bound.

* The first call of the problem has ub $= n$ for parameter.

As such, $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$ returns true only if $\sigma[i]$ is matched to n.

– If $\sigma[i]$ is an RLMin element, then $i = k$ ($\sigma[i]$ is the rightmost element). Every $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i\ , *)$ is thus a base case and true if $\sigma[i]$ is matched to ub. Moreover, recursive calls for RLMin elements do not change the upper bound. As a result, $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$ returns true only if $\sigma[i]$ is matched to ub $= n$.

- If $\overline{\sigma[i]\sigma[i']}$ appears in $\widetilde{\sigma}$, (which implies that $\sigma[i'] = \sigma[i] + 1$) and $i < i'$ (the case in which $i > i'$ can be dealt following the same idea), then $\sigma[i]$ is matched to $\pi[j]$ and $\sigma[i']$ is matched to $\pi[j] + 1$.

  – In the case where $\sigma[i]$ is an RLMax element is impossible, since $\sigma[i']$ is to the right of and above $\sigma[i]$,

  – in the $\sigma[i]$ is an RLMin element, and $\sigma[i']$ is an RLMax element, then remark that

    * $i' = k$; indeed, if $\sigma[i']$ is not the rightmost element, there would exist an element between $\sigma[i]$ and $\sigma[i']$. Every recursive call $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, *, i', *)$ is thus solved as a base case. As such, $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, *, i', *)$ is true only if $\sigma[i']$ is matched to the element lb.
    * The recursive calls for $\sigma[i + 1]$, ..., $\sigma[i' - 1]$ do not change the lower bound. Indeed, $\sigma[i]$ is the rightmost RLMin. As a result $\sigma[i + 1]$, $\sigma[i + 2]$, ..., $\sigma[i' - 1]$ are RLMax elements. Moreover, recursive calls for an RLMax do not change the lower bound.
    * $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, *, i, *)$ sets the lb to $\pi[j] + 1$ and matches $\sigma[i]$ to $\pi[j]$.

    As such, $\sigma[i]$ is matched to $\pi[j]$, and $\sigma[i']$ is matched to $\pi[j] + 1$.

  – If $\sigma[i]$ is an RLMin element and $\sigma[i']$ is a RLMin, element then:

    * Every recursive call $\mathrm{PM}_\pi^{\widetilde{\sigma}}(\mathrm{lb}, *, i', *)$ is true only if $\sigma[i']$ is matched to the element with the value lb.
    * The recursive calls for $\sigma[i + 1]$, ..., $\sigma[i' - 1]$ do not change the lower bound. Indeed, based on Lemma 100, $\sigma[i+1]$, ..., $\sigma[i'-1]$ are RLMax elements. Finally, the recursive calls for RLMax elements do not change the lower bound.
    * $\mathrm{PM}_\pi^{\widetilde{\sigma}}(*, *, i, *)$ sets lb to $\pi[j] + 1$ and matches $\sigma[i]$ to $\pi[j]$.

    As such $\sigma[i]$ is matched to $\pi[j]$ and $\sigma[i']$ is matched to $\pi[j] + 1$.

There are n^3 base cases that can be computed in constant time and kn^3 different cases. Each case takes up to $O(n)$ time to compute, which means that computing all cases takes $O(kn^4)$ time. As each case takes $O(1)$ space, we need $O(kn^3)$ space.

□

7.5 Computing the Longest Wedge Permutation Pattern

This section focuses on a problem related to the PPM problem, continuing the work undertaken in [20] and in [46].

Given a set of permutations, the longest permutation that occurs in each permutation of the set. This problem is known to be NP-Hard for an arbitrary size of the set even when all of its permutations are separable (see [20]). We demonstrate how to compute the longest wedge permutation occurring in a set, although we do not hope that the problem is solvable in polynomial time if the size of the set is not fixed. Indeed, the size of the set appears in the exponent in the complexity of the algorithm. We thus focus on cases in which sets have only one or two permutations.

For convenience, we say that a subsequence is a wedge subsequence if and only if the permutation it represents is a wedge permutation.

We start with the easiest case in which only one input permutation is at play. We need the sets of RLMax and RLMin elements:
$A(\pi) = \{i|\pi[i] \text{ is an RLMin element}\} \cup \{n\}$ and
$D(\pi) = \{i|\pi[i] \text{ is an RLMax element}\} \cup \{n\}$.

Proposition 102. *If s_i is the LIS with the last element at position f in π and s_d is the LDS with the last element at position f in π then $s_i \cup s_d$ is a longest wedge subsequence with the last element at position f in π.*

Proof. Let us first prove that we have a wedge subsequence. Indeed, s_i is an increasing subsequence with values below or equal to $\pi[f]$, and s_d is a decreasing subsequence with values above or equal to $\pi[f]$; as such, $s_i \cup s_d$ is a wedge subsequence. Let us prove that this is a longest one. Let s be a wedge subsequence with its rightmost element at position f in π such that $|s| > |s_i \cup s_d|$. It should first be noted that $A(s)$ is also an increasing subsequence with its rightmost element at position f in π and that $D(s)$ is also a decreasing subsequence with its rightmost element at position f in π. As $|s| > |s_i \cup s_d|$, then either $|A(s)| > |s_i|$ or $|D(s)| > |s_d|$, which contradicts with the definitions of s_i and s_d. □

Proposition 103. *Let π be a permutation. The longest wedge subsequence that can occur in π can be computed in $O(n \log(\log(n)))$ time and $O(n)$ space.*

Proof. Proposition 102 leads to an algorithm in which one computes the LIS and LDS ending at every possible position and then finds the maximum sum of the LIS and LDS. LIS and LDS can be computed in $O(n \log(\log(n)))$ time and $O(n)$ space (see [15]); the maximums can then be found done in linear time.
□

We next consider a case in which the input is composed of two permutations.

Proposition 104. *Given two permutations π_1 of size n_1 and π_2 of size n_2, the longest common wedge subsequence can be computed in $O(n_1^3 n_2^3)$ time and space.*

Proof. Consider the following problem that computes the longest wedge subsequence common to π_1 and π_2: Given two permutations π_1 and π_2, we define
$LCS_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1,\pi_2,\mathrm{lb}_2,\mathrm{ub}_2,i_1,i_2}$
$= \max \{|s| \mid s \text{ occurs } \pi_1[i_1:] \text{ with every element of the occurrence in } [\mathrm{lb}_1, \mathrm{ub}_1],$ and s occurs $\pi_2[i_2:]$ with every element of the occurrence in $[\mathrm{lb}_2, \mathrm{ub}_2]$, and s is a wedge subsequence $\}$

We show how to solve this problem by dynamic programming.

BASE:

$$LCS_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1,\pi_2,\mathrm{lb}_2,\mathrm{ub}_2,n_1,n_2} = \begin{cases} 1 & \text{if } \mathrm{lb}_1 \leq \pi_1[n_1] \leq \mathrm{ub}_1 \ 7 \\ & \text{and } \mathrm{lb}_2 \leq \pi_2[n_2] \leq \mathrm{ub}_2 \\ 0 & \text{otherwise} \end{cases}$$

STEP:

$$LCS_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1,\pi_2,\mathrm{lb}_2,\mathrm{ub}_2,i_1,i_2} = \max \begin{cases} LCS_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1,\pi_2,\mathrm{lb}_2,\mathrm{ub}_2,i_1,i_2+1} \\ \\ LCS_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1,\pi_2,\mathrm{lb}_2,\mathrm{ub}_2,i_1+1,i_2} \\ \\ \mathrm{M}^{\pi_2,\mathrm{lb}_2,\mathrm{ub}_2}_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1}(i_1,i_2) \end{cases}$$

with

$$\mathrm{M}^{\pi_2,\mathrm{lb}_2,\mathrm{ub}_2}_{\pi_1,\mathrm{lb}_1,\mathrm{ub}_1}(i_1,i_2) =$$

$$\begin{cases} 1 + LCS_{\pi_1,\pi_1[i_1]+1,\mathrm{ub}_1,\pi_2,\pi_2[i_2]+1,\mathrm{ub}_2,i_1+1,i_2+1} & \pi_1[i_1] < \mathrm{lb}_1 \\ & \text{and } \pi_2[i_2] < \mathrm{lb}_2 \\ \\ 1 + LCS_{\pi_1,\mathrm{lb}_1,\pi_1[i_1]-1,\pi_2,\mathrm{lb}_2,\pi_2[i_2]-1,i_1+1,i_2+1} & \pi_1[i_1] > \mathrm{ub}_1 \\ & \text{and } \pi_2[i_2] > \mathrm{ub}_2 \\ \\ 0 & \text{otherwise} \end{cases}$$

The solution to this problem relies on the fact that the longest wedge subsequence is found by considering the problem with $\pi_1[i_1 :]$ and $\pi_2[i_2 + 1 :]$ or $\pi_1[i_1 + 1 :]$ and $\pi_2[i_2 :]$ or by matching $\pi_1[i_1]$ and $\pi_2[i_2]$ and adding to the solution the LCS for $\pi_1[i_1 + 1 :]$ and $\pi_2[i_2 + 1 :]$ which is compatible, meaning that, if we consider the current elements to correspond to an RLMin (resp. RLMax) element in the longest wedge subsequence, then we consider only the solution with elements above (below) $\pi_1[i_1]$ for the occurrence in $\pi_1[i_1 + 1 :]$ and $\pi_2[i_2]$ for the occurrence in $\pi_2[i_2 + 1 :]$.

These relations lead to an $O(|\pi_1|^3|\pi_2|^3)$ time and $O(|\pi_1|^3|\pi_2|^3)$ space algorithm. Indeed, $|\pi_1|^3|\pi_2|^3$ possible cases exists for the problem, each of which is solved in constant time. □

Chapter 8

Conclusion

In this thesis we presented polynomial algorithm for the problems of PPM when constraints are added on text and/or the pattern. We also explored the PPM problem with BVP which is the first positive result on theses pattern. We end this thesis by proposing some interesting questions in relation to PPM.

This thesis uses the classification of permutations by avoiding classes. However, other class types exist, especially, the class of grid permutations (See [2]). The grid class is given by a matrix $\mathcal{M}$ of -1, 0 or 1. Informally, in a plot of a permutation, there exists a grid with the same dimensions as the matrix. Intuitively, each brick of this grid is associated with the value in the matrix at the same position as the grid. As such, if the value in the matrix is 1, then the brick contains an increasing sequence, if the value is 0, then the brick contains no element and if the value is -1, then the brick contains a decreasing sequence. See Figure8.1 for an example.

It should be noted that [2] demonstrates that any grid class can be define as an avoiding class. For example any wedge permutations is in the grid class (but not the only grid class) with matrix $\begin{pmatrix} -1 \\ 1 \end{pmatrix}$ because every wedge permutation can be split horizontally such that the top part contains a decreasing subsequence and the bottom part contains an increasing subsequence. Another example is that a skew-merged permutation is in the grid class permutation with matrix $\begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$. Some interesting points about the grid permutation are as follows:

- A more relevant example of an avoiding class that can be define as a class of grid classes is that any separable permutation is in a grid class with a matrix of which each row and column have at most one element different from 0. This definition of separable permutation exhibits a class of matrix. A question that naturally arises is whether a class of matrices exists such that the PPM is solvable in polynomial time for the corresponding class of grid permutations. As a lead, we believe that whenever a pattern belong to the grid class with the matrix belonging in the class $\begin{pmatrix} 1 & \dots & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ \dots \\ 1 \end{pmatrix}$ of matrices, then PPM can be solved polynomially if the matrices have a fixed size. Another class of matrices of fixed size, that we believe can be solved polynomially, is defined as follow: for each element of the matrix different from 0, if its row contains another element different from 0 then its column is full of 0 and if its column contains another element different from 0 then its row is full of 0. The basics of an algorithm would be to

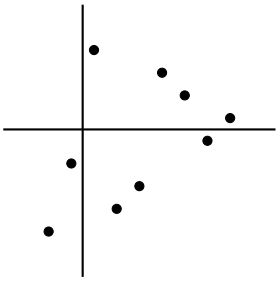


Figure 8.1 – The permutation 149238756 is in the grid class permutations with matrix $\left(\begin{smallmatrix} 0 & -1 \\ 1 & 1 \end{smallmatrix}\right)$.

use an algorithm solving the case above and to combine it with a splitting strategy.

- The following question has been formulated in converse to the above lead: Does there exists a class of matrices such that the PPM over the associated grid classes is NP-complete? An example of such result is in the 321-avoiding permutations. Indeed, every 321-avoiding permutation is known to be split-table into two increasing subsequences, which means that every 321-avoiding permutations can be understood as a permutation in the grid class with matrices filled with 0 except for two diagonals south-west north-east which are filled with 1.

- Our first intuition is that PPM can always be solved polynomially when the pattern is in a grid class of a given matrix. Indeed, an algorithm could try every possible grid in the text, and test whether each brick contains an increasing or a decreasing subsequence of sufficient size, and try the compatibility between each brick. This would result in a polynomial algorithm in the size of the matrix, but does it yield a correct algorithm?

- Another interesting question is whether the PPM problem can be solved by a FTP algorithm by a value relevant to the size of the matrix of a grid class permutations. The algorithm of Marx and Guillemot [32] is exponential by the size of the pattern. In practice, the pattern can have up to 10000 elements, but can be in grid class with a small matrix.

Bibliography

[1] S. Ahal and Y. Rabinovich, *On Complexity of the Subpattern Problem*, SIAM Journal on Discrete Mathematics **22** (2008), no. 2, 629–649.

[2] M. H. Albert, M. D. Atkinson, M. Bouvel, N. Ruškuc, and V. Vatter, *Geometric grid classes of permutations*, ArXiv e-prints (2011).

[3] M. H. Albert, M.-L. Lackner, M. Lackner, and V. Vatter, *The Complexity of Pattern Matching for* 321*-Avoiding and Skew-Merged Permutations*, ArXiv e-prints (2015).

[4] M.H. Albert, R.E.L. Aldred, M.D. Atkinson, and D.A. Holton, *Algorithms for pattern involvement in permutations*, Proc. International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Computer Science, vol. 2223, 2001, pp. 355–366.

[5] M.H. Albert and M.D. Atkinson, *Simple permutations and pattern restricted permutations*, Discrete Mathematics **300** (2005), no. 1, 1 – 15.

[6] Michael H. Albert, Robert E. L. Aldred, Mike D. Atkinson, and Derek A. Holton, *Algorithms for pattern involvement in permutations*, pp. 355–367, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[7] David Aldous and Persi Diaconis, *Longest increasing subsequences: From patience sorting to the baik-deift-johansson theorem*, Bull. Amer. Math. Soc **36** (1999), 413–432.

[8] MacMahon Percy Alexander, *Combinatory Analysis*, (1915).

[9] H. Aoki, R. Uehara, and K. Yamazaki, *Expected length of longest common subsequences of two biased random strings and its application*, Tech. Report 1185, RIMS Kokyuroku, 2001.

[10] Richard Arratia, *On the Stanley-Wilf conjecture for the number of permutations avoiding a given pattern*, (1999).

[11] Sergey Avgustinovich, Sergey Kitaev, and Alexandr Valyuzhenich, *Avoidance of boxed mesh patterns on permutations*, Discrete Applied Mathematics **161** (2013), no. 1–2, 43 – 51.

[12] D. Avis and M. Newborn, *On pop-stacks in series*, Utilitas Mathematica **19** (1981), 129–140.

[13] M. J. Bannister, Z. Cheng, W. E. Devanny, and D. Eppstein, *Superpatterns and Universal Point Sets*, ArXiv e-prints (2013).

[14] M. J. Bannister, W. E. Devanny, and D. Eppstein, *Small Superpatterns for Dominance Drawing*, ArXiv e-prints (2013).

[15] Sergei Bespamyatnikh and Michael Segal, *Enumerating longest increasing subsequences and patience sorting*, 2000.

[16] P. Bille and I.L. Gørtz, *The tree inclusion problem: In linear space and faster*, ACM Trans. Algorithms **7** (2011), no. 3, 38.

[17] Philip Bille and Inge Li Gørtz, *The tree inclusion problem: In linear space and faster*, CoRR **abs/cs/0608124** (2006).

[18] P. Bose, J.F.Buss, and A. Lubiw, *Pattern matching for permutations*, Information Processing Letters **65** (1998), no. 5, 277–283.

[19] M. Bouvel, D. Rossin, and S. Vialette, *Longest common separable pattern between permutations*, Proc. Symposium on Combinatorial Pattern Matching (CPM), London, Ontario, Canada (B. Ma and K. Zhang, eds.), Lecture Notes in Computer Science, vol. 4580, 2007, pp. 316–327.

[20] M. Bouvel, D. Rossin, and S. Vialette, *Longest Common Separable Pattern between Permutations*, Symposium on Combinatorial Pattern Matching (CPM'07) (London, Ontario, Canada, Canada) (Ma Bin and Zhang Kaizhong, eds.), LNCS, vol. 4580, Springer, 2007, pp. 316–327.

[21] Daniel Pierre Bovet and Pierluigi Crescenzi, *Introduction to the theory of complexity*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[22] M.-L. Bruner and M. Lackner, *The computational landscape of permutation patterns*, ArXiv e-prints (2013).

[23] M.-L. Bruner and M.Lackner, *A fast algorithm for permutation pattern matching based on alternating runs*, 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), Helsinki, Finland (F.V. Fomin and P. Kaski, eds.), Springer, 2012, pp. 261–270.

[24] S. Buss and M. Soltys, *Unshuffling a square is NP-hard*, Journal of Computer and System Sciences **80** (2014), no. 4, 766–776.

[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms*, third ed., MIT Press, Cambridge, 2009.

[26] M. Crochemore and E. Porat, *Fast computation of a longest increasing subsequence and application*, Information and Compututation **208** (2010), no. 9, 1054–1059.

[27] Samuel Eilenberg and Saunders Mac Lane, *On the groups $h(\pi, n), i$*, Annals of Mathematics **58** (1953), no. 1, 55–106.

[28] Henrik Eriksson, Kimmo Eriksson, Svante Linusson, and Johan Wästlund, *Dense packing of patterns in a permutation*, Annals of Combinatorics **11** (2007), no. 3, 459–470.

[29] William Fulton, *Young tableaux : with applications to representation theory and geometry*, London Mathematical Society student texts, Cambridge University Press, Cambridge, New York, 1997, Autres tirages : 1999.

[30] S. Giraudo and S. Vialette, *Unshuffling Permutations*, ArXiv e-prints (2016).

[31] S. Giraudo and S. Vialette, *Unshuffling permutations*, 12th Latin American Theoretical Informatics Symposium (LATIN), Lecture Notes in Computer Science, no. 9644, Springer, 2016, pp. 509–521.

[32] S. Guillemot and D. Marx, *Finding small patterns in permutations in linear time*, Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Portland, Oregon, USA (C. Chekuri, ed.), SIAM, 2014, pp. 82–101.

[33] S. Guillemot and S. Vialette, *Pattern matching for* 321*-avoiding permutations*, Proc. 20-th International Symposium on Algorithms and Computation (ISAAC), Hawaii, USA (Y. Dong, D.-Z. Du, and O. Ibarra, eds.), Lecture Notes in Computer Science, vol. 5878, 2009, pp. 1064–1073.

[34] L. Ibarra, *Finding pattern matchings for permutations*, Information Processing Letters **61** (1997), no. 6, 293–295.

[35] V. Jelínek and J. Kynčl, *Hardness of Permutation Pattern Matching*, ArXiv e-prints (2016).

[36] P. Kilpeläinen and H. Manilla, *Ordered and unordered tree inclusion*, SIAM J. on Comput. **24** (1995), no. 2, 340–356.

[37] S. Kitaev, *Patterns in permutations and words*, Springer-Verlag, 2013.

[38] D.E. Knuth, *Fundamental algorithms*, 3rd ed., The Art of Computer Programming, vol. 1, Addison-Wesley, Reading MA, 1973.

[39] Donald E. Knuth, *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[40] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń, *A linear time algorithm for consecutive permutation pattern matching*, Information Processing Letters **113** (2013), no. 12, 430 – 433.

[41] L. Mach, *Parameterized complexity : permutation patterns, graph arrangements, and matroid parameters*, Ph.D. thesis, University of Warwick, UK, 2015.

[42] H. Magnusson and H. Ulfarsson, *Algorithms for discovering and proving theorems about permutation patterns*, ArXiv e-prints (2012).

[43] David Maier, *The complexity of some problems on subsequences and supersequences*, J. ACM **25** (1978), no. 2, 322–336.

[44] A. Mansfield, *On the computational complexity of a merge recognition problem*, Discrete Applied Mathematics **5** (1983), 119–122.

[45] Alison Miller, *Asymptotic bounds for permutations containing many different patterns*, Journal of Combinatorial Theory, Series A **116** (2009), no. 1, 92 – 108.

[46] B. E. Neou, R Rizzi, and S. Vialette, *Pattern matching for separable permutations*, pp. 260–272, Springer International Publishing, Cham, 2016.

[47] Helmut Prodinger and Friedrich J. Urbanek, *Infinite 0–1-sequences without long adjacent identical blocks*, Discrete Mathematics **28** (1979), no. 3, 277 – 289.

[48] D. Henshall N. Rampersad and J. Shallit, *Shuffling and unshuffling*, http://arxiv.org/abs/1106.5767, 2011.

[49] Antonio Restivo, *The Shuffle Product: New Research Directions*, Proceedings of the Ninth International Conference on Language and Automata Theory and Applications, Lecture Notes in Computer Science, vol. 8977, Springer International Publishing, 2015, pp. 70–81.

[50] Romeo Rizzi and Stéphane Vialette, *On recognizing words that are squares for the shuffle product*, The 8th International Computer Science Symposium in Russia (Ekaterinburg, Russia) (Arseny M. Shur Andrei A. Bulatov, ed.), Lecture Notes in Computer Science, vol. 7913, Springer, June 2013, pp. 235–245.

[51] D. Rossin and M. Bouvel, *The longest common pattern problem for two permutations*, Pure Mathematics and Applications **17** (2006), 55–69.

[52] C. Schensted, *Longest increasing and decreasing subsequences*, (1961).

[53] J. van Leeuwen and M. Nivat, *Efficient recognition of rational relations*, Information Processing Letters **14** (1982), no. 1, 34–38.

[54] Yannic Vargas, *Hopf algebra of permutation pattern functions*, 26th International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC 2014) (Chicago, United States) (Louis J. Billera and Isabella Novik, eds.), DMTCS Proceedings, vol. AT, Discrete Mathematics and Theoretical Computer Science, 2014, pp. 839–850.

[55] V. Vatter, *Permutation classes*, Handbook of Enumerative Combinatorics (M. Bóna, ed.), Chapman and Hall/CRC, 2015, pp. 753–818.