ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Hamiltonicity of minimal jump graphs

Bachelor Thesis

Manuel Nowack

July 10, 2020

Advisors: Prof. Dr. B. Gärtner, Prof. Dr. T. Mütze, H. Hoang
Department of Computer Science, ETH Zürich

Abstract

For permutations a combinatorial Gray code, i.e. exhaustive generation with small changes between consecutive objects, can be given by minimal jumps. Exhaustive generation of a set of pattern-avoiding permutations with minimal jumps can be formulated as a Hamiltonian cycle/path problem. Such a minimal jump graph is known to have a Hamiltonian path if none of the avoiding patterns have the largest entry at the leftmost or rightmost position.

We show that reversal is an isomorphism on minimal jump graphs and thereby reduce the number of graphs that have to be considered by half. Then we compute small minimal jump graphs avoiding patterns up to length 4 that have the largest entry at the leftmost or rightmost position and test them for Hamiltonicity. In doing so we present simple reasons why many sufficiently large minimal jump graphs have no Hamiltonian cycle or path. We also examine unbalanced bipartitions in the minimal jump graph when avoiding the increasing pattern because they can be used to prove that there is no Hamiltonian path.

Contents

Chapter 1

Introduction

In mathematics and computer science we frequently encounter different kinds of combinatorial objects, such as permutations, binary strings, binary trees, set partitions, spanning trees of a graph, and so forth. A fundamental algorithmic task we want to perform with such objects is exhaustive generation. Of particular interest is exhaustive generation with small changes between consecutive objects, i.e. we obtain combinatorial Gray codes. Many fundamental classes of combinatorial objects can be encoded as pattern-avoiding permutations.

1.1 Basic concepts and definitions

1.1.1 Permutation

A permutation is a bijection from $\{1, 2, \ldots, n\}$ to itself and we say such a permutation has length n. We can also write a permutation of length n as $\pi = \pi(1)\pi(2)\ldots\pi(n)$ where $\pi(i)$ refers to the i-th entry (number) in the permutation. S_n denotes the set of all permutations of length n. For example $S_3 = \{123, 132, 213, 231, 312, 321\}$.

1.1.2 Pattern-avoiding permutation

Given two permutations $\pi \in S_n$ and $\tau \in S_k$, we say that π contains the pattern τ, if π contains a subsequence formed by (not necessarily consecutive) entries that appear in the same relative order as in τ; otherwise we say that π avoids τ. For example $\pi = 53412$ contains the pattern $\tau = 231$, as the entries $3, 4, 1$ (or alternatively $3, 4, 2$) form a match of τ in π. On the other hand, $\pi = 53412$ avoids the pattern $\tau = 123$.

$S_n(\tau)$ denotes the set of all permutations of length n that avoid the pattern τ. Sometimes we are interested in avoiding more than just one pattern. In

this case we write $S_n(F)$ where F is a propositional formula made of logical ANDs $\wedge$, ORs $\vee$ and patterns as variables. Here AND means that each of the patterns is avoided and OR means that at least one of the patterns is avoided. For example $S_3(123 \wedge 312) = \{132, 213, 231, 321\}$.

We say a pattern is *tame* if the largest entry is not at the leftmost or rightmost position. Otherwise we say a pattern is non-tame. For example 1423 is tame, but 4123 is not.

1.1.3 Elementary transformations

We consider three elementary transformations of permutations that are important in the context of pattern-avoidance as we shall see in Lemma 1.2.

The *reversal* of a permutation π changes each entry $\pi(i)$ to $\pi(n+1-i)$. For example $rev(1423) = 3241$.

The *complementation* of a permutation π changes each entry $\pi(i)$ to $n+1-\pi(i)$. For example $cpl(1423) = 4132$.

The *inversion* of a permutation π changes each entry $\pi(i)$ to the position of entry i. For example $inv(1423) = 1342$.

Notice that these three transformations are involutions, i.e. $cpl(cpl(\pi)) = \pi$, $inv(inv(\pi)) = \pi$ and $rev(rev(\pi)) = \pi$.

1.1.4 Ascent and descent

An ascent in a permutation π is a position i such that $\pi(i) < \pi(i+1)$. Similarly, a descent is a position i such that $\pi(i) > \pi(i+1)$.

1.1.5 Inversion number

The inversion number of a permutation π is defined as the number of pairs of entries in π such that $i < j$ and $\pi(i) > \pi(j)$. We say a permutation is even (odd) if the inversion number is even (odd). For example 3142 is an odd permutation and its inversion number is 3 because the pairs $(3,1), (3,2), (4,2)$ are out of their natural order.

1.1.6 Jump

A jump is a substring rotation of an entry in a permutation and neighbouring smaller entries. For example, a right jump of 4 in the permutation 53412 by 2 steps yields 53124 and a left jump of 4 in the permutation 53412 by 1 steps yields 54312.

A minimal jump is defined with respect to a set of permutations $L_n \subseteq S_n$. A jump in a permutation $\pi \in L_n$ is minimal if any jump of the same entry in

the same direction with fewer steps will result in a permutation not in L_n. Let's consider the permutation $321 \in S_3(231)$. A left jump of 3 obviously does not exist. A right jump of 3 to the right by 1 step results in a permutation not in $S_3(231)$. A right jump of 3 to the right by 2 steps results in 213 and hence the jump is minimal. A left jump of 2 does not exist because moving over a larger entry is not permitted. The minimal right jump of 2 is 312. A minimal jump of 1 never exists because all other entries are larger. Thus minimal jumps of entries in 321 with respect to $S_3(231)$ yield 231 and 312.

1.1.7 Minimal jump graph

We recap some basic graph theory. A graph $G = (V, E)$ consists of the vertex set V and the edge set E. Each edge $(u, v) \in E$ where $u, v \in V$ connects two vertices. The degree of a vertex is the number of edges adjacent to it. In this thesis we consider only undirected, unweighted, simple graphs.

- A graph is undirected when the two vertices of each edge are not distinguished from each other, i.e. $(u, v) = (v, u)$.
- A graph is unweighted when its edges have not been assigned weights.
- A graph is simple when it has no loops and no parallel edges. That is, each edge connects two distinct vertices and no two edges have the same vertices.

A path in a graph is a sequence of distinct edges which joins a sequence of distinct vertices. A cycle is a path that starts and ends in the same vertex.

A Hamiltonian cycle is a cycle that visits every vertex exactly once. A Hamiltonian path is a path that visits every vertex exactly once. Hamiltonicity is an umbrella term for whether a graph has a Hamiltonian cycle or path.

A graph is bipartite when its vertices can be partitioned into two disjoint sets such that the vertices in one set are not connected to each other, but may be connected to vertices in the other set.

A *minimal jump graph* $G_n(F)$ has vertex set $S_n(F)$ and an edge for every minimal jump in $S_n(F)$. For example Figure 1.1 displays a minimal jump graph.

A Hamiltonian path in a minimal jump graph corresponds to a Gray code for $S_n(F)$. If a minimal jump graph has no Hamiltonian path, $S_n(F)$ cannot be exhaustively generated with minimal jumps.

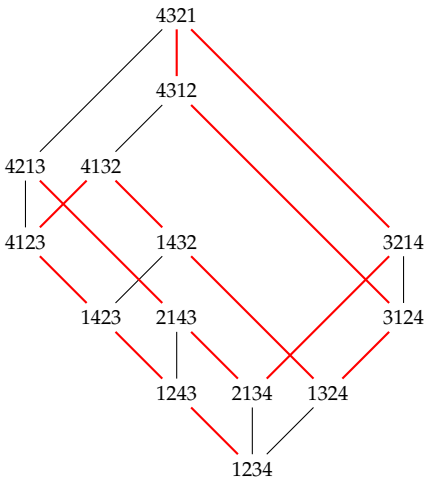

Figure 1.1: $G_4(231)$ with a Hamiltonian path highlighted in red

1.2 Previous work and open questions

Hartung et al. [4] presented a general and versatile algorithmic framework for exhaustively generating a set of permutations $L_n \in S_n$. Their main tool is Algorithm J.

Algorithm J. This algorithm attempts to greedily generate a set of permutations $L_n \subseteq S_n$ using minimal jumps starting from an initial permutation $\pi_0 \in L_n$.

Initialize Visit the initial permutation π_0.

Jump Generate an unvisited permutation from L_n by performing a minimal jump of the largest possible value in the most recently visited permutation. If no such jump exists, or the jump direction is ambiguous, then terminate. Otherwise visit this permutation and repeat Jump.

Theorem 1.1 *Let F be an arbitrary propositional formula consisting of logical ANDs $\wedge$, ORs $\vee$, and tame patterns as variables, then $S_n(F)$, $n \geq 0$, can be generated by Algorithm J.*

This corresponds to Theorem 8 in [4].

The open question is exhaustive generation of $S_n(F)$ when F contains non-tame patterns. There are two directions to consider depending on whether we use a Gray code only for efficient generation or to explicitly have minimal changes between consecutive permutations.

Goal 1: efficient exhaustive generation with minimal jumps between consecutive permutations

When generating $S_n(F)$ for a F containing only tame patterns, Algorithm J walks a Hamiltonian path on $G_n(F)$. We want to know whether $G_n(F)$ has a Hamiltonian path when F contains non-tame patterns. If $G_n(F)$ has no Hamiltonian path, no Gray code with minimal jumps exists for F. If $G_n(F)$ has a Hamiltonian path, we want to find an algorithm that walks the path without maintaining the graph.

Goal 2: only efficient exhaustive generation

We want to generate $S_n(F)$ for a F containing non-tame patterns, but we do not require that $G_n(F)$ has a Hamiltonian path. This means we find a bijection $f : S_n(G) \to S_n(H)$ where G contains only tame patterns and H contains non-tame patterns. Then we use Algorithm J to generate $S_n(G)$ and apply the bijection f to get $S_n(H)$.

For the second direction Hartung et al. [4] gave the following lemma that is very useful for the purpose of exhaustive generation.

Lemma 1.2 *Given any composition h of the elementary transformations reversal, complementation and inversion, and any propositional formula F consisting of logical ANDs $\wedge$, ORs $\vee$, and patterns $\pi_1, \ldots, \pi_l$ as variables, then the sets of pattern-avoiding permutations $S_n(F)$ and $S_n(h(F))$ are in bijection under h for all $n \geq 1$, where the formula $h(F)$ is obtained from F by replacing every pattern τ_i by $h(\tau_i)$ for all $i = 1, \ldots, l$.*

This sometimes allows us to generate $S_n(F)$ for a F containing non-tame patterns because even if τ_i is non-tame, then maybe $h(\tau_i)$ is tame.

Another useful bijection was given by West [7].

Theorem 1.3 *$S_n(1 \ldots k(k-1))$ and $S_n(1 \ldots k)$ are in bijection for $k \geq 3$.*

We do not present this bijection here because it requires the introduction of unrelated concepts. Instead we refer the curious reader to Theorem 3.1.10 and Algorithm 3.2.6 in [7].

With these bijections all single non-tame patterns up to length 4 except 1324 (and its elementary transformation 4231) have a bijection to a tame pattern. A bijection from 1324 to a tame pattern does not exist because $S_n(1324)$ is not equinumerous to another $S_n(\tau)$ where τ is a tame pattern of length 4.

1.3 Outline of this thesis

This thesis focuses on the first goal above, i.e. exhaustive generation with minimal jumps between consecutive permutations.

The theoretical results of the thesis are divided into three chapters. In Chapter 2 we show that complementation and inverse are not valid isomorphisms on minimal jump graphs, but we prove that reversal is one. This reduces the number of graphs we have to consider by half.

In Chapter 3 we present simple reasons why many $G_n(F)$ for F containing patterns up to length 4 have no Hamiltonian cycle or path for sufficiently large n.

In Chapter 4 we examine the minimal jump graph when avoiding the increasing pattern $1 \dots k$. We prove that $G_n(1 \dots k)$ is always bipartite and that the partitions are the sets of even and odd permutations in $S_n(1 \dots k)$. Furthermore, we prove that the size difference of the even and odd partition of $S_n(123)$ is zero for even n and corresponds to the Catalan numbers for odd n. This gives an alternative proof, besides the proof in chapter 3, that $G_n(123)$ has no Hamiltonian path for odd $n \geq 5$.

In order to gain some intuition about the properties of minimal jump graphs that avoid a non-tame pattern we have performed several computational experiments. The experiments and their results are detailed in Chapter 5. Notably, we have written a computer program to generate the minimal jump graph and check if it has a Hamiltonian cycle or path when avoiding one of the patterns 123, 213, 1234, 1324, 2134, 2314, 3124, 3214 and when avoiding one of the aforementioned patterns AND a pattern of the same length. This covers all non-tame patterns up to length 4, taking into account reversal.

Finally, in Chapter 6 we summarize the main points of the thesis and discuss some open problems.

Chapter 2

Isomorphisms on minimal jump graphs

Reversal, complementation and inversion are known elementary transformations on $S_n(F)$. Naturally we would like to know whether they can also be applied to minimal jump graphs because this will greatly reduce the number of graphs we have to consider.

Technically reversal, complementation and inversion can immediately be applied to minimal jump graphs because the vertices are simply $S_n(F)$ and the edges can be represented as (π, π') for two permutations in $S_n(F)$. However, complementation and inversion yield invalid minimal jump graphs in the general case. The problem is that they shuffle the order of the entries in a permutation and thus some morphed edges represent jumps that are no longer minimal or valid at all. In particular Hamiltonicity is not preserved.

Example 2.1 *The edge* $(1243, 4123)$ *exists in* $G_4(1423)$*, but its complementation* $(4312, 1432)$ *is not a jump and thus cannot exist in* $G_4(cpl(1423)) = G_4(4132)$*.*

Example 2.2 *The edge* $(1243, 4123)$ *exists in* $G_4(1423)$*, but its inverse* $(1243, 2341)$ *is not a jump and thus cannot exist in* $G_4(inv(1423)) = G_4(1342)$*.*

At least reversal is an isomorphism on minimal jump graphs. The intuition is that all permutations including their minimal jumps are simply mirrored.

Lemma 2.3 *Reversing all permutations in* $G_n(F)$ *is a graph isomorphism to* $G_n(rev(F))$*.*

Proof The vertices of $G_n(F)$ and $G_n(rev(F))$ are $S_n(F)$ and $S_n(rev(F))$. By Lemma 1.2 reversal is a bijection from $S_n(F)$ to $S_n(rev(F))$. It remains to show that reversal is a bijection between the two edge sets. Since $rev(rev(F)) = F$ it suffices to show an injection from one edge set to the other.

Let (π, π') be an arbitrary edge in $G_n(F)$. We want to show that $(rev(\pi), rev(\pi'))$ is an edge in $G_n(rev(F))$. Let m denote the entry in π that is moved by k steps to get π'. Then we can move the same entry m in $rev(\pi)$ by k steps in the opposite direction to get $rev(\pi')$. Clearly m is moved over the same

entries than before. Thus the jump is valid because m does not move over an entry larger than itself. The jump is minimal because a jump of m in $rev(\pi')$ with less steps than k steps in any direction implies the reverse jump is possible in π, contradicting the assumption that (π, π') is a minimal jump. $\square$

Chapter 3

Minimal jump graphs with a simple reason for non-Hamiltonicity

The computational experiments in section 5.1 show that for small n many minimal jump graphs have a simple reason for non-Hamiltonicity. We generalize some of these observations and prove that they hold for all sufficiently large n.

In this chapter, we denote by $a \dots b$ the sequence $a, a+1, \dots, b-1, b$ if $a < b$ and $a, a-1, \dots, b+1, b$ otherwise. For example $n132(n-1)\dots4$ equals 81327654 for $n = 8$.

3.1 Degree-1 vertices

Some graphs $G_n(F)$ have at least three degree-1 vertices for all $n \geq n_0$ and some n_0. Since all but two vertices in a Hamiltonian path must have two neighbours, a Hamiltonian path does not exist in these minimal jump graphs. Table 3.1 lists all such F where F contains one non-tame pattern up to length 4 AND another pattern of the same length.

Some other graphs $G_n(F)$ have at least one degree-1 vertex for all $n \geq n_0$ and some n_0. Since all vertices in a Hamiltonian cycle must have two neighbours, a Hamiltonian cycle does not exist in these minimal jump graphs. Table 3.2 lists all such F where F contains one non-tame pattern up to length 4 AND another pattern of the same length. In the computational experiments we have seen that at least for small n these graphs still have a Hamiltonian path.

F	n_0	degree-1 vertices		
123	4	$1n \dots 32$	$21n \dots 3$	$31n \dots 42$
$123 \wedge 231$	4	$1n \dots 32$	$21n \dots 3$	$321n \dots 4$
213	4	$1 \dots n$	$134 \dots n2$	$2 \dots n1$
$213 \wedge 312$	4	$1 \dots n$	$124 \dots n3$	$13 \dots n2$
$1234 \wedge 3124$	5	$132n \dots 4$	$142n \dots 53$	$1432n \dots 5$
$1234 \wedge 4123$	6	$142n \dots 53$	$152n \dots 643$	$1532n \dots 64$
$1234 \wedge 4132$	6	$12n \dots 3$	$132n \dots 4$	$142n \dots 53$
$1234 \wedge 4231$	7	$243n \dots 51$	$253n \dots 614$	$2543n \dots 61$
$1234 \wedge 4312$	7	$n132(n-1) \dots 4$	$n13(n-1) \dots 42$	$n1432(n-1) \dots 5$
$1324 \wedge 3142$	7	$13 \dots n2$	$1346 \dots n52$	$135 \dots n42$
$1324 \wedge 4312$	8	$3517 \dots n264$	$357 \dots n1642$	$468 \dots n13752$
$1324 \wedge 4321$	7	$15 \dots n342$	$16 \dots n2453$	$16 \dots n4523$
$2134 \wedge 1243$	6	$21n \dots 3$	$321n \dots 4$	$4321n \dots 5$
$2134 \wedge 3214$	5	$1342n \dots 5$	$231n \dots 4$	$2341n \dots 5$
$2134 \wedge 3241$	7	$13562n \dots 74$	$23561n \dots 74$	$2451n \dots 63$
$2134 \wedge 4132$	6	$21n \dots 3$	$231n \dots 4$	$241n \dots 53$
$2134 \wedge 4213$	6	$1352n \dots 64$	$2351n \dots 64$	$241n \dots 53$
$2134 \wedge 4231$	8	$13572n \dots 864$	$2461n \dots 753$	$462n \dots 7153$
$2134 \wedge 4312$	8	$2451n \dots 63$	$24561n \dots 73$	$36 \dots n1542$
$2134 \wedge 4321$	7	$4 \dots (n-1)231n$	$24 \dots (n-1)31n$	$5 \dots (n-1)2341n$
$2314 \wedge 3241$	7	$23 \dots n1$	$2346 \dots n51$	$2356 \dots n41$
$2314 \wedge 4312$	6	$34 \dots n12$	$35 \dots n142$	$45 \dots n132$
$2314 \wedge 4321$	6	$3 \dots n21$	$35 \dots n241$	$4 \dots n231$
$3124 \wedge 1342$	6	$31n \dots 42$	$421n \dots 53$	$531n \dots 642$
$3124 \wedge 2341$	6	$1542n \dots 63$	$2541n \dots 63$	$431n \dots 52$
$3124 \wedge 4312$	6	$146 \dots n253$	$346 \dots n152$	$35 \dots n142$
$3214 \wedge 2134$	5	$1342n \dots 5$	$231n \dots 4$	$2341n \dots 5$
$3214 \wedge 4321$	6	$346 \dots n251$	$35 \dots n241$	$36 \dots n2451$

Table 3.1: Three permutations that correspond to degree-1 vertices in $G_n(F)$ for $n \geq n_0$.

F	n_0	degree-1 vertex
$123 \wedge 132$	4	$(n-1)\dots 1n$
$123 \wedge 213$	4	$1n\dots 2$
$123 \wedge 312$	4	$n\dots 1$
$213 \wedge 123$	4	$1n\dots 2$
$213 \wedge 132$	4	$1\dots n$
$213 \wedge 231$	4	$1\dots n$
$213 \wedge 321$	4	$1\dots n$
$1234 \wedge 2143$	4	$12n\dots 3$
$3214 \wedge 2341$	4	$32n\dots 41$

Table 3.2: One permutation that corresponds to a degree-1 vertex in $G_n(F)$ for $n \geq n_0$.

3.2 Degree-2 vertices

In a Hamiltonian cycle every vertex has two adjacent edges. Thus both edges adjacent to a degree-2 vertex must be part of the Hamiltonian cycle. Consequently, if another vertex is adjacent to two degree-2 vertices, we know the edges to these two vertices are part of the Hamiltonian cycle and all other edges from this vertex cannot be part of the Hamiltonian cycle. Thus we can remove such edges from the graph without affecting the existence of a Hamiltonian cycle. Figure 3.1 illustrates such a constellation.

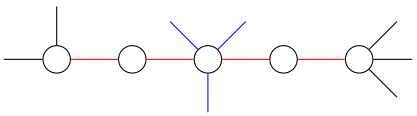

Figure 3.1: A component of a graph where two degree-2 vertices exclude the blue edges from a Hamiltonian cycle because the red edges must be part of it.

The removal of edges can reduce the degree of a vertex to one which means no Hamiltonian cycle exists in the graph. This can be used to prove that $G_n(1234 \wedge 2314)$ has no Hamiltonian cycle for $n \geq 5$. In the computational experiments we have seen that at least for some n this graph still has a Hamiltonian path.

Lemma 3.1 *$G_n(1234 \wedge 2314)$ has no Hamiltonian cycle for $n \geq 5$.*

Proof Figure 3.2 displays a component of $G_n(1234 \wedge 2314)$ that exists for all $n \geq 5$.

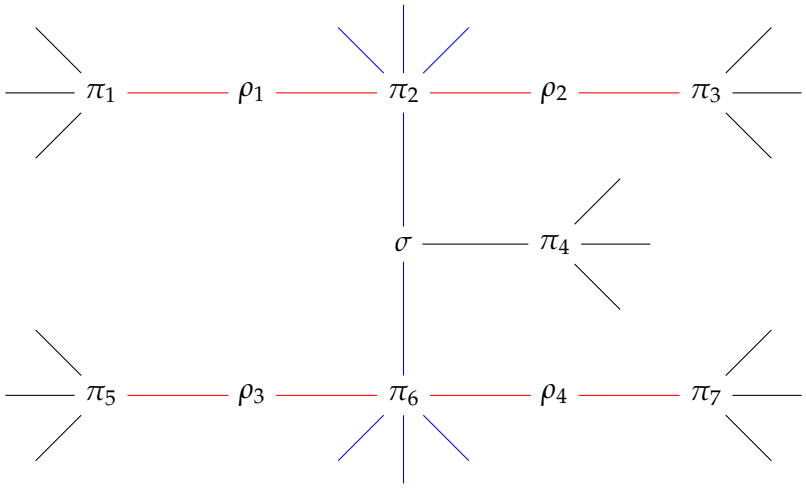

Figure 3.2: A component of $G_n(1234 \wedge 2314)$ that exists for all $n \geq 5$.

The vertices in Figure 3.2 correspond to the following permutations.

$$\begin{aligned}
\rho_1 &= (n-3)\ldots1(n-2)n(n-1)\\
\rho_2 &= (n-2)\ldots1(n-1)n\\
\rho_3 &= (n-2)\ldots2(n-1)n1\\
\rho_4 &= (n-3)\ldots2(n-2)n(n-1)1\\
\pi_1 &= (n-3)\ldots1n(n-2)(n-1)\\
\pi_2 &= (n-2)\ldots1n(n-1)\\
\pi_3 &= (n-1)\ldots1n\\
\pi_4 &= (n-2)\ldots3n21(n-1)\\
\pi_5 &= (n-1)\ldots2n1\\
\pi_6 &= (n-2)\ldots2n(n-1)1\\
\pi_7 &= (n-3)\ldots2n(n-2)(n-1)1\\
\sigma &= (n-2)\ldots2n1(n-1)
\end{aligned}$$

The vertices ρ_i always have exactly two neighbours and the vertex σ always has exactly three neighbours. The vertices π_i may have arbitrarily many edges to other vertices in the graph, subject to n.

The edges colored in red must be part of the Hamiltonian cycle because they are adjacent to a degree-2 vertex. Consequently, the edges colored in blue cannot be part of the Hamiltonian cycle. Thus vertex σ has only one remaining neighbour in the Hamiltonian cycle. Since every vertex in a

Hamiltonian cycle must have two neighbours, a Hamiltonian cycle does not exist. □

3.3 Empty minimal jump graphs

In the computational experiments we noticed that $G_n(123 \wedge 321)$ and $G_n(1234 \wedge 4321)$ are empty for large enough n. This observation can be proven in the general case $G_n(1 \dots k \wedge k \dots 1)$.

Lemma 3.2 *$G_n(1 \dots k \wedge k \dots 1)$ is empty for $n \geq (k-1)^2 + 1$.*

Proof The Erdős-Szekeres theorem [3] asserts that any sequence of length larger than $(k-1)^2$ has either an increasing or decreasing subsequence of length n. Thus a permutation of length at least $(k-1)^2 + 1$ cannot avoid the pattern $1 \dots k \wedge k \dots 1$. The lemma follows. □

Chapter 4

Minimal jump graph when avoiding an increasing sequence

The computational experiments on $G_n(123)$ and $G_n(1234)$ in section 5.1 indicate that unbalanced bipartitions can be used to prove non-Hamiltonicity for $G_n(1\dots k)$. First, we show why $G_n(1\dots k)$ is always bipartite.

Lemma 4.1 *All minimal jumps in a permutation $\pi \in S_n(\tau)$ are by 1 step for $\tau = 1\dots k$.*

Proof We proceed by case distinction on the direction of the jump.

Case: Left Jump We notice that moving an entry over smaller neighbouring entries to the left can never introduce a longer increasing subsequence. Thus for all entries $\pi(i)$ a left jump by 1 step that avoids τ exists if $\pi(i) > \pi(i-1)$. If $\pi(i) < \pi(i-1)$ no left jump exists for $\pi(i)$ because moving over larger entries is not permitted.

Case: Right Jump If we can make a right jump by m steps and obtain a τ-avoiding permutation, a right jump by 1 step will also avoid τ because moving an entry over smaller neighbouring entries to the left can never introduce a longer increasing subsequence. Thus if a minimal right jump in a permutation $\pi \in S_n(\tau)$ exists, it is by 1 step. □

Corollary 4.2 *$G_n(1\dots k)$ is bipartite and the partitions are the sets of even and odd permutations in $S_n(1\dots k)$.*

Proof From Lemma 4.1 it follows that each edge in $G_n(1\dots k)$ is between a vertex with an even inversion number and a vertex with an odd inversion number because swapping two adjacent entries changes the inversion number by one. Thus $G_n(1\dots k)$ is bipartite and the partitions are the vertices that correspond to permutations with odd respectively even inversion numbers. □

With that in mind, we have written a program to compute the number of even and odd permutations in $S_n(123)$, $S_n(1234)$, $S_n(12345)$ and $S_n(123456)$ for small n. The results are in section 5.2. In particular we noticed that when avoiding 123, the partitions have identical size for even n and for odd n the size differences are the Catalan numbers. We will now prove this observation. Note that this will not produce immediate new results because in section 3.1 we have already proven that $G_n(123)$ has no Hamiltonian path for $n \geq 4$. Rather we hope to gain insights into the size difference that can be extended to $G_n(1234)$ or even generalized to all $G_n(1 \dots k)$.

For $n \geq 0$, the Catalan numbers C_n are given by

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

In this section we will also see an interpretation of the k-fold Catalan convolutions $C_{n,k}$ given by

$$C_{n,k} = \frac{k}{2n-k}\binom{2n-k}{n}$$

They have recurrence

$$C_{n,k} = C_{n+1,k+1} - C_{n+1,k+2}$$

and the base cases are $C_{n,0} = 0$ and $C_{n,n} = 1$ [2].

Summing up the k-fold Catalan convolutions yields the Catalan numbers again.

$$\sum_{k=1}^{n} C_{n,k} = C_n$$

4.1 Counting even and odd permutations when avoiding 123

To count the size difference of the even and odd partition we will present a recursive formula for the number of even and odd 123-avoiding permutations based on the position of the first ascent in the permutation. We make this detour through first ascents because it allows us to easily keep track of a permutation's parity. The idea behind this proof has been previously used to show that the number of 123-avoiding permutations of length n with first ascent at position k is given by the k-fold Catalan convolutions $C_{n,k}$ [2].

Any 123-avoiding permutation of length n becomes a 123-avoiding permutation of length $n-1$ when the entry n is removed from it, so we can view each 123-avoiding permutation of length n as being grown uniquely by taking a 123-avoiding permutation of length $n-1$ and inserting n into certain

positions. The eligible positions are tightly connected with the position of the first ascent. Recall that an ascent in a permutation π is a position i such that $\pi(i) < \pi(i+1)$. The peak of an ascent is the position $i+1$. If a permutation of length n has no ascents at all (i.e. it is the decreasing permutation $n \dots 1$), we define it as having the first ascent at position n.

Suppose we have a 123-avoiding permutation of length $n-1$ with first ascent at position k. How can this be grown into a longer 123-avoiding permutation by inserting n? We consider all possible positions i for $1 \leq i \leq n$ and how an insertion at this position affects the first ascent at the new permutation.

$i = 1$
: If n is inserted at the very front of the permutation, the resulting permutation is still 123-avoiding with the original first ascent being "pushed forward" one position due to the presence of n. Thus the new first ascent is at position $k+1$.

$1 < i \leq k+1$
: If n is inserted anywhere between the front of the permutation and the peak of the original first ascent, then n becomes the peak of the new first ascent, which therefore has position 1 less than the position of n. Furthermore, as there are no ascents before n, and as n cannot be involved in any subsequent ascents, the new permutation is still 123-avoiding. Thus the new first ascent is at position $i-1$.

$i > k+1$
: If n is inserted after the peak of the original first ascent, then the resulting permutation is no longer 123-avoiding.

Note that all of the above hold exactly even if the original permutation is the descending permutation with first ascent defined to be at position $n-1$. To summarize, a 123-avoiding permutation of length $n-1$ with first ascent at position k gives rise to exactly one 123-avoiding permutation of length n with first ascent at position i for each $1 \leq i \leq k+1$, and each 123-avoiding permutation of length n must be grown in such a way.

However, what we want to show is that the size difference of the even and odd partition is given by the Catalan numbers. For this purpose we additionally have to consider how inserting n affects the parity of a permutation. Suppose we have inserted n into a 123-avoiding permutation of length $n-1$ and the new first ascent is at position k. Where was n inserted and how did it affect the parity of the permutation? n was either inserted at position 1, pushing the original first ascent forward by one position, or n was inserted at position $k+1$, creating a new first ascent at position k, while the original ascent was at or after position k. Inserting n at position i of a permutation of length $n-1$ increases the inversion number by $n-i$ because entry n and all successive entries are out of their natural order. If n was inserted at position

1, the parity of the permutation changes if n is even because the inversion number increases by $n-1$. If n was inserted at position $k+1$, the parity of the permutation changes if n and k have the same parity because the inversion number increases by $n-k-1$. We define $E_{n,k}$ to be the number of even 123-avoiding permutations of length n that have the first ascent at position k and define $O_{n,k}$ analogously for odd permutations. $E_{n,k}$ and $O_{n,k}$ can be expressed by recursion, as defined below. The first summand covers insertion at position 1 and the second summand covers insertion at position $k+1$.

$$E_{n,k} = \begin{cases} O_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} & \text{if } n \text{ is even and } k \text{ is even} \\ O_{n-1,k-1} + \sum_{i=k}^{n-1} E_{n-1,i} & \text{if } n \text{ is even and } k \text{ is odd} \\ E_{n-1,k-1} + \sum_{i=k}^{n-1} E_{n-1,i} & \text{if } n \text{ is odd and } k \text{ is even} \\ E_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} & \text{if } n \text{ is odd and } k \text{ is odd} \end{cases}$$

$$O_{n,k} = \begin{cases} E_{n-1,k-1} + \sum_{i=k}^{n-1} E_{n-1,i} & \text{if } n \text{ is even and } k \text{ is even} \\ E_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} & \text{if } n \text{ is even and } k \text{ is odd} \\ O_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} & \text{if } n \text{ is odd and } k \text{ is even} \\ O_{n-1,k-1} + \sum_{i=k}^{n-1} E_{n-1,i} & \text{if } n \text{ is odd and } k \text{ is odd} \end{cases}$$

The base cases are

$$E_{1,1} = 1$$
$$O_{1,1} = 0$$

and the boundary conditions are

$$E_{n,0} = O_{n,0} = 0$$
$$E_{0,k} = O_{0,k} = 0$$
$$E_{n,k} = O_{n,k} = 0 \text{ if } n < k$$

Because our interest lies in the difference, we introduce an additional variable $D_{n,k}$ for that.

$$D_{n,k} = E_{n,k} - O_{n,k}$$

This gives us a formula for the size difference of the even and odd partition of $S_n(123)$. Note that the value can be negative depending on which partition is larger.

$$\sum_{k=1}^{n} D_{n,k}$$

Now we have all the tools ready to count the size difference of the even and odd partition of $S_n(123)$.

Lemma 4.3 *For odd n and even k,*

$$D_{n,k} = 0$$

and for even n and even k,

$$D_{n,k} + D_{n,k-1} = 0$$

Proof We prove this claim by induction on n. For simplicity we use $n = 1, 2$ as base case, but the claim holds for the more relevant base case $n = 3, 4$ too. Since $D_{n,k} = 0$ for $k > n$, we only have to consider $k \leq n$.

Base Case: $S_1(123) = \{1\}$ contains no permutations with first ascent at an even position. Thus $D_{1,k} = 0$ trivially holds for all even k.

$S_2(123) = \{12, 21\}$ contains one even permutation with first ascent at position 1 and one odd permutation with first ascent at position 2. Thus we have $D_{2,2} + D_{2,1} = -1 + 1 = 0$ as required.

Induction step: We maintain two induction hypotheses.

I.H. 1: $D_{n-1,k} = 0$ holds for some odd $n-1$ and all even k.

I.H. 2: $D_{n-1,k} + D_{n-1,k-1} = 0$ holds for some even $n-1$ and all even k.

Then we show that $D_{n,k} = 0$ holds for odd n and all even k and $D_{n,k} + D_{n,k-1} = 0$ holds for even n and all even k.

For odd n and even k

$$\begin{aligned}
D_{n,k} &= E_{n,k} - O_{n,k} \\
&= E_{n-1,k-1} + \sum_{i=k}^{n-1} E_{n-1,i} - O_{n-1,k-1} - \sum_{i=k}^{n-1} O_{n-1,i} \\
&= D_{n-1,k-1} + \sum_{i=k}^{n-1} D_{n-1,i} \\
&= \sum_{i=k-1}^{n-1} D_{n-1,i} \\
&\overset{\text{I.H. 2}}{=} 0
\end{aligned}$$

For even n and even k

$$\begin{aligned}
D_{n,k} + D_{n,k-1} &= E_{n,k} - O_{n,k} + E_{n,k-1} - O_{n,k-1} \\
&= O_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} - E_{n-1,k-1} - \sum_{i=k}^{n-1} E_{n-1,i} \\
&\quad + O_{n-1,k-2} + \sum_{i=k-1}^{n-1} E_{n-1,i} - E_{n-1,k-2} - \sum_{i=k-1}^{n-1} O_{n-1,i} \\
&= -D_{n-1,k-1} - \sum_{i=k}^{n-1} D_{n-1,i} - D_{n-1,k-2} + \sum_{i=k-1}^{n-1} D_{n-1,i} \\
&= -D_{n-1,k-2} \\
&\overset{\text{I.H. } 1}{=} 0
\end{aligned}$$

□

Corollary 4.4 *For even n the sets of even and odd permutations in $S_n(123)$ have identical size.*

Proof With Lemma 4.3 we have

$$\sum_{k=1}^{n} D_{n,k} = 0$$

as required. □

The following lemma is just a helper for Lemma 4.6.

Lemma 4.5 *For odd n and odd k,*

$$D_{n,k} = D_{n-1,k-1}$$

Proof We simply apply the definitions and use Lemma 4.3 in the last step.

$$\begin{aligned}
D_{n,k} &= E_{n,k} - O_{n,k} \\
&= E_{n-1,k-1} + \sum_{i=k}^{n-1} O_{n-1,i} - O_{n-1,k-1} - \sum_{i=k}^{n-1} E_{n-1,i} \\
&= D_{n-1,k-1} - \sum_{i=k}^{n-1} D_{n-1,i} \\
&= D_{n-1,k-1}
\end{aligned}$$

□

Recall that the k-fold Catalan convolutions $C_{n,k}$ are given by

$$C_{n,k} = \frac{k}{2n-k}\binom{2n-k}{n}$$

They have recurrence

$$C_{n,k} = C_{n+1,k+1} - C_{n+1,k+2}$$

and the base cases are $C_{n,0} = 0$ and $C_{n,n} = 1$.

Lemma 4.6 *For $n = 1, 5, 9, \ldots$ and odd k,*

$$D_{n,k} = C_{\frac{n-1}{2},\frac{k-1}{2}}$$

and for $n = 3, 7, 11, \ldots$ and odd k,

$$D_{n,k} = -C_{\frac{n-1}{2},\frac{k-1}{2}}$$

Proof We present a recurrence for odd n and odd k and then show that the k-fold Catalan convolution obeys the same recurrence relation and has the same base cases. First we use Lemma 4.5

$$D_{n+2,k+2} - D_{n+2,k+4} = D_{n+1,k+1} - D_{n+1,k+3}$$

and then we repeatedly apply definitions to get

$$\begin{aligned}
D_{n+1,k+1} - D_{n+1,k+3} &= E_{n+1,k+1} - O_{n+1,k+1} - E_{n+1,k+3} + O_{n+1,k+3} \\
&= O_{n,k} + \sum_{i=k+1}^{n} O_{n,i} - E_{n,k} - \sum_{i=k+1}^{n} E_{n,i} \\
&\quad - O_{n,k+2} - \sum_{i=k+3}^{n} O_{n,i} + E_{n,k+2} + \sum_{i=k+3}^{n} E_{n,i} \\
&= -D_{n,k} - \sum_{i=k+1}^{n} D_{n,i} + D_{n,k+2} + \sum_{i=k+3}^{n} D_{n,i} \\
&= -D_{n,k} - D_{n,k+1}
\end{aligned}$$

By Lemma 4.3 we have $D_{n,k+1} = 0$ and finally arrive at

$$D_{n+2,k+2} - D_{n+2,k+4} = -D_{n,k}$$

We introduce a new variable $A_{n,k}$ to account for the fact we are only interested in $D_{n,k}$ for odd n and odd k. Note that the variable $A_{n,k}$ runs over all n and k, not just odd ones.

$$A_{n,k} = D_{2n+1,2k+1} \cdot (-1)^n$$

Putting this together we have $A_{n,k} = A_{n+1,k+1} - A_{n+1,k+2}$ which is the same recurrence as the k-fold Catalan convolution.

The first base case is $A_{n,0}$ and it is given by Lemma 4.5

$$A_{n,0} = (-1)^n \cdot D_{2n+1,1} = (-1)^n \cdot D_{2n,0} = 0$$

The second base case is $A_{n,n}$. In order to get $A_{n,n}$ we first show $D_{2n+3,2n+3} = -D_{2n+1,2n+1}$. Again we use Lemma 4.5 and then apply the definitions twice.

$$\begin{aligned} D_{2n+3,2n+3} &= D_{2n+2,2n+2} \\ &= E_{2n+2,2n+2} - O_{2n+2,2n+2} \\ &= O_{2n+1,2n+1} - E_{2n+1,2n+1} \\ &= -D_{2n+1,2n+1} \end{aligned}$$

Since $D_{1,1} = 1$ we have $D_{2n+1,2n+1} \cdot (-1)^n = 1$. $A_{n,n} = 1$ immediately follows.

Since $C_{n,k}$ obeys the same recurrence relation as $A_{n,k}$ and they have the same base cases, we find that $A_{n,k}$ equals $C_{n,k}$ everywhere. The lemma then follows. □

Corollary 4.7 *For odd n the size difference of the sets of even and odd permutations in $S_n(123)$ correspond to the Catalan numbers.*

Proof With Lemma 4.3 and Lemma 4.6 we have

$$|\sum_{k=1}^{n} D_{n,k}| = |\sum_{k=1,3,\cdots}^{n} C_{\frac{n-1}{2},\frac{k-1}{2}} \cdot (-1)^{\frac{n-1}{2}}| = \sum_{k=1}^{\frac{n-1}{2}} C_{\frac{n-1}{2},k}$$

Then we use that summing up the k-fold Catalan convolutions yields the Catalan numbers and arrive at

$$\sum_{k=1}^{\frac{n-1}{2}} C_{\frac{n-1}{2},k} = C_{\frac{n-1}{2}}$$

as required. □

Initially we tried to prove that the even and odd partition have equal size for even n by a direct proof by induction. Unfortunately the proof failed without Lemma 4.3. Once the Lemma was proven, the proof was superseded by the much simpler Corollary 4.4.

4.2 Counting even and odd permutations when avoiding any increasing sequence

The formula for 123-avoiding permutations in section 4.1 has three arguments: parity, length of the permutation and position of the first ascent. The idea behind this can be generalized to any increasing pattern $1 \dots k$. Instead of keeping track of the first ascent, we keep track of the first $1 \dots m$-containing subsequences for $m = 2 \dots k-1$. In case of 123 the first 12-containing subsequence is the first ascent.

We will now show the formula for 1234. Recall that any 1234-avoiding permutation of length n becomes a 1234-avoiding permutation of length $n-1$ when the entry n is removed from it, so we can view each 1234-avoiding permutation of length n as being grown uniquely by taking a 1234-avoiding permutation of length $n-1$ and inserting n into certain positions. Suppose we have a 1234-avoiding permutation of length $n-1$ with the end of the first 12-containing subsequence at position x and the end of first 123-containing subsequence at position y. If the permutation has no such subsequence, we define it as having the subsequence at position n. How can this be grown into a longer 1234-avoiding permutation by inserting n? We consider all possible positions i for $1 \leq i \leq n$ and how an insertion at this position affects the first 12-containing and 123-containing subsequences in the new permutation. Additionally, we consider changes to the parity of the permutation. Inserting n at position i of a permutation of length $n-1$ increases the inversion number by $n-i$ because entry n and all successive entries are out of their natural order.

$i = 1$
: If n is inserted at the very front of the permutation, the resulting permutation is still 1234-avoiding with the original first 12-containing and 123-containing subsequences being "pushed forward" one position due to the presence of n. Thus the end of the new first 12-containing subsequence is at position $x+1$ and the end of the new first 12-containing subsequence is at position $x+1$. The parity of the permutation changes if n is even because the inversion number increases by $n-1$.

$1 < i \leq x$
: If n is inserted anywhere between the front of the permutation and the end of the original first 12-containing subsequence, then n becomes the end of the new first 12-containing subsequence, which therefore has position i. Furthermore, as there are no 123-containing subsequences before n, and as n cannot be involved in any subsequent increasing subsequences, the new permutation is still 1234-avoiding. Thus the end of the new first 12-containing subsequence is at position i and

the end of the first 123-containing subsequence is still at position y. The parity of the permutation changes if n and x have different parity because the inversion number increases by $n - x$.

$x < i \leq y$
: If n is inserted anywhere between the end of the original first 12-containing subsequence and the end of the original first 123-containing subsequence, then n becomes the end of the new first 123-containing subsequence, which therefore has position i. Furthermore, as there are no 123-containing subsequences before n, and as n cannot be involved in any subsequent increasing subsequences, the new permutation is still 1234-avoiding. Thus the end of the first 12-containing subsequence is still at position x and the end of the new first 123-containing subsequence is at position i. The parity of the permutation changes if n and y have different parity because the inversion number increases by $n - y$.

$y < i \leq n$
: If n is inserted after the original first 123-containing subsequence, then the resulting permutation is no longer 1234-avoiding.

Note that all of the above hold exactly even if the original permutation has a subsequence defined to be at position n. Now suppose we have inserted n into a 1234-avoiding permutation of length $n - 1$ and the end of the new first 12-containing subsequence is at position x and the end of the new first 123-containing subsequence is at position y. Where was n inserted and how did it affect the parity of the permutation? From the cases above it follows that n was either inserted at position 1, x or y. We define $E_{n,x,y}$ to be the number of even 1234-avoiding permutations of length n that have the end of the first 12-containing subsequence at position x and the end of first 123-containing subsequence at position y and define $O_{n,x,y}$ analogously for odd permutations. $E_{n,x,y}$ and $O_{n,x,y}$ can be expressed by recursion, as defined below. The first summand covers insertion at position 1, the second summand covers insertion at position x and the third summand covers insertion at position y.

$$E_{n,x,y} = \begin{cases} O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is even, } y \text{ is even} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is even, } y \text{ is odd} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is odd, } y \text{ is even} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is odd, } y \text{ is odd} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is even, } y \text{ is even} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is even, } y \text{ is odd} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is odd, } y \text{ is even} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is odd, } y \text{ is odd} \end{cases}$$

$$O_{n,x,y} = \begin{cases} E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is even, } y \text{ is even} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is even, } y \text{ is odd} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is odd, } y \text{ is even} \\ E_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is even, } x \text{ is odd, } y \text{ is odd} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is even, } y \text{ is even} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} O_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is even, } y \text{ is odd} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} E_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is odd, } y \text{ is even} \\ O_{n-1,x-1,y-1} + \sum_{i=x}^{n}\sum_{j=x+1}^{n} E_{n-1,i,j} + \sum_{i=y}^{n} O_{n-1,x,i} & \text{if } n \text{ is odd, } x \text{ is odd, } y \text{ is odd} \end{cases}$$

It becomes apparent that the general formula requires k parameters when avoiding the pattern $1 \dots k$ and the recursion has 2^k case distinctions. This renders finding a proof for the general case, or even 1234, highly impractical.

Chapter 5

Computational Experiments

In order to gain some intuition about the properties of minimal jump graphs we have performed several computational experiments. This section details the insights gained from the experiments and sketches the programs. The code of all computer programs written as part of this thesis is given in the Appendix.

The computational experiments are executed on Ubuntu 18.04.4 with a i5-6200U CPU and 8GB of RAM. All code is written in C++.

5.1 Hamiltonicity of minimal jump graph

To have an overview of the Hamiltonicity of minimal jump graphs, we have written the program Listing A.10 to generate $G_n(F)$ for various F and test them for Hamiltonicity.

Outline of the program

The program first generates all permutations that avoid the pattern F. For this we use an efficient algorithm discovered and implemented by Kuszmaul [6]. While the algorithm is incredibly fast, the exponentially increasing size of the output is an insurmountable obstacle. We generate $S_n(F)$ only up to $n = 12$ to avoid running out of memory for larger values.

Next we naively compute the minimal jumps for all pattern-avoiding permutations and build the minimal jump graph stored as an adjacency list. There are probably more sophisticated methods, but the bottleneck of the program lies elsewhere.

Now we are ready to examine the Hamiltonicity of the graph. First we check some simple properties that have linear computational complexity but are sufficient to show that the graph has no Hamiltonian path.

- Is the graph empty?
- Has the graph at least three degree-1 vertices?
- Is the graph disconnected?
- Is the graph bipartite and have the two partitions size difference larger than one? If the graph is disconnected, we do not check this property because the partitions are ambiguous.

If one or several of these properties hold, the program terminates.

Then we transform the Hamiltonian cycle problem into an equivalent travelling salesman problem (TSP) and use the Concorde TSP solver [1] to find a Hamiltonian cycle. If a cycle is found, the program terminates. If no cycle is found, we use the Concorde TSP solver to find a Hamiltonian path. If no path is found either, the program terminates. While Concorde is state of the art, it cannot keep pace with the exponentially growing graph size and the exponential running time of the TSP problem. We have chosen not to invoke Concorde if the graph has more than 7000 vertices because it did not return a solution within reasonable time for such graphs. In this case the program terminates without deciding Hamiltonicity.

If a path but no cycle is found, we additionally try to find a reason why there is no cycle. To this end we check some simple properties that have linear computational complexity but are sufficient to show that the graph has no Hamiltonian cycle.

- Has the graph a degree-1 vertex?
- Is the graph bipartite and is the size difference of the two partitions non-zero?

If one of these properties hold, the program terminates. Otherwise we perform one more check, making use of degree-2 vertices. In a Hamiltonian cycle every vertex has two adjacent edges. Thus both edges adjacent to a degree-2 vertex must be part of the Hamiltonian cycle. Consequently, if another vertex is adjacent to two degree-2 vertices, we know the edges to these two vertices are part of the Hamiltonian cycle and all other edges from this vertex cannot be part of the Hamiltonian cycle. Figure 5.1 illustrates such a constellation.

The program repeatedly deletes such edges that cannot be part of the Hamiltonian cycle from the graph until no more edges can be deleted. In this pruned graph we again check some simple properties that have linear computational complexity but are sufficient to show that the graph has no Hamiltonian cycle.

- Has the graph a degree-1 vertex?

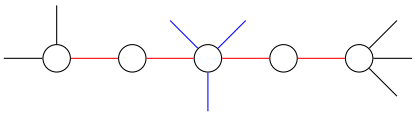

Figure 5.1: A component of a graph where two degree-2 vertices exclude the blue edges from a Hamiltonian cycle because the red edges must be part of it.

- Is the graph disconnected?
- Is the graph bipartite and have the two partitions size difference larger than one? If the graph is disconnected, we do not check this property because the partitions are ambiguous.

To summarize, the program tests the following properties of a minimal jump graph.

C The graph has a Hamiltonian cycle.
P The graph has a Hamiltonian path. It cannot have a Hamiltonian cycle because
  b the graph is bipartite and the two partitions have size difference larger than zero.
  c the graph is disconnected.
  d the graph has at least one degree-1 vertex.
  2 pruning degree-2 vertices reveals one or several of the three reasons above apply.
  ? there is no obvious reason.
X The graph has no Hamiltonian path because
  b the graph is bipartite and the two partitions have size difference larger than one.
  c the graph is disconnected.
  d the graph has at least three degree-1 vertices.
  e the graph is empty.
  ? there is no obvious reason.

This notation also serves as legend for the output of the program in the ensuing pages. If a graph has no simple reason for non-Hamiltonicity and finding an exact solution is computationally infeasible, the corresponding cell in the table is left blank.

5.1.1 Avoiding 123 AND a pattern of length 3

F	$n = 3$	4	5	6	7	8	9	10	11	12
123	Pbd	Xd	Xbd	Xd	Xbd	Xd	Xbd	Xd	Xbd	Xd
$123 \wedge 132$	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd
$123 \wedge 213$	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd
$123 \wedge 231$	Pd	Xd	Xd	Xbd	Xbd	Xbd	Xbd	Xbd	Xbd	Xbd
$123 \wedge 312$	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc
$123 \wedge 321$	Xcd	Xcd	Xe	Xe	Xe	Xe	Xe	Xe	Xe	Xe

5.1.2 Avoiding 213 AND a pattern of length 3

F	$n = 3$	4	5	6	7	8	9	10	11	12
213	Pbd	Xd	Xbd	Xd	Xbd	Xd	Xbd	Xd	Xbd	Xd
$213 \wedge 123$ [1]	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd
$213 \wedge 132$	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd	Pd
$213 \wedge 231$	Pd	X?	X?	X?	Pd	X?	X?	Pd	Pd	X?
$213 \wedge 312$	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
$213 \wedge 321$ [2]	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc	Xc

[1] identical to $123 \wedge 213$ in section 5.1.1
[2] equivalent to $123 \wedge 312$ in section 5.1.1

5.1.3 Avoiding 1234 AND a pattern of length 4

F	$n = 4$	5	6	7	8	9	10	11	12
1234	Pb	Pb	Xb	Xb	Xb	Xb	Xb	Xb	Xb
$1234 \wedge 1243$	C	C	C	C					
$1234 \wedge 1324$	C	C	C	Pb		Xb		Xb	
$1234 \wedge 1342$	P2c	C	C	C					
$1234 \wedge 1423$	P2c	C	C	C					
$1234 \wedge 1432$	C	C	C	C	C				
$1234 \wedge 2134$	C	C	C	C					
$1234 \wedge 2143$	Pd	Pd	Pd	Pd	Pd				
$1234 \wedge 2314$	P2c	X?	P2d	P2d					
$1234 \wedge 2341$	C	C	C	C	C				
$1234 \wedge 2413$	C	C	C	C	C				
$1234 \wedge 2431$	P2d	C	C	C	C				
$1234 \wedge 3124$	Xb	Xd	Xbd	Xd	Xbd	Xbd	Xbd	Xbd	Xbd
$1234 \wedge 3142$	C	C	C	C	C				
$1234 \wedge 3214$	C	C	C	C	C				
$1234 \wedge 3241$	P2d	C	C	C	C				
$1234 \wedge 3412$	P?	C	C	C	C				
$1234 \wedge 3421$	C	C	C	C	C				
$1234 \wedge 4123$	C	X?	Xd	Xbd	Xd	Xbd	Xbd	Xbd	Xbd
$1234 \wedge 4132$	P2d	Pd	Xd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
$1234 \wedge 4213$	Xb	Xb	Xb	Xb	Xb	Xb	Xb	Xb	Xb
$1234 \wedge 4231$	C	C	Pd	Xd	Xd	Xd	Xd	Xd	Xd
$1234 \wedge 4312$	C	C	X?	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
$1234 \wedge 4321$	Xb	Xb	X?	Xc	Xc	Xc	Xe	Xe	Xe

5.1.4 Avoiding 1324 AND a pattern of length 4

F	$n=4$	5	6	7	8	9	10	11	12
1324	C	C	C	C					
1324 ∧ 1234 [1]	C	C	C	Pb		Xb		Xb	
1324 ∧ 1243	C	C	C	C					
1324 ∧ 1342	C	C	C	C					
1324 ∧ 1423	C	C	C	C					
1324 ∧ 1432	C	C	C	C					
1324 ∧ 2134	P2c	C	C	C					
1324 ∧ 2143	C	C	C	C	C				
1324 ∧ 2314	C	C	C	C					
1324 ∧ 2341	C	C	C	C	C				
1324 ∧ 2413	C	C	C	C	C				
1324 ∧ 2431	C	C	C	C	C				
1324 ∧ 3124	C	C	C	C					
1324 ∧ 3142	Pd	Pd	Pd	Xd	Xd	Xd	Xd	Xd	Xd
1324 ∧ 3214	P2c	C	C	C					
1324 ∧ 3241	C	C	C	C	C				
1324 ∧ 3412	C	C	C	C	C				
1324 ∧ 3421	C	C	C	C	C				
1324 ∧ 4123	P2d	C	C	C	C				
1324 ∧ 4132	C	C	C	C	C				
1324 ∧ 4213	C	C	C	C	C				
1324 ∧ 4231	C	C	C	C	C				
1324 ∧ 4312	P2d	C	P2cd	X?	Xd	Xd	Xd	Xd	Xd
1324 ∧ 4321 [2]	C	C	Pd	Xd	Xd	Xd	Xd	Xd	Xd

[1] identical to 1234 ∧ 1324 in section 5.1.3
[2] equivalent to 1234 ∧ 4231 in section 5.1.3

5.1.5 Avoiding 2134 AND a pattern of length 4

F	$n=4$	5	6	7	8	9	10	11	12
2134	Pb	Pb	Pb	Pb	Xb	Xb	Xb	Xb	Xb
2134 ∧ 1234 [1]	C	C	C	C					
2134 ∧ 1243	Pd	Pd	Xd	Xd	Xd	Xd	Xd	Xd	Xd
2134 ∧ 1324 [2]	P2c	C	C	C					
2134 ∧ 1342	C	C	C	C	C				
2134 ∧ 1423	C	C	C	C	C				
2134 ∧ 1432	P2d	C	C	C	C				
2134 ∧ 2143	C	C	C	C					
2134 ∧ 2314	C	C	C	C					
2134 ∧ 2341	P2c	C	C	C	C				
2134 ∧ 2413	P2c	C	C	C	C				
2134 ∧ 2431	C	C	C	C	C				
2134 ∧ 3124	C	C	C	C					
2134 ∧ 3142	P2d	C	C	C	C				
2134 ∧ 3214	Xb	Xbd	Xd	Xd	Xbd	Xd	Xbd	Xbd	Xbd
2134 ∧ 3241	C	C	Pd	Xd	Xd	Xd	Xd	Xd	Xd
2134 ∧ 3412	C	C	C	C	C				
2134 ∧ 3421	P?	C	C	C	C				
2134 ∧ 4123	Xb	Xb	X?	P?	Xb	Xb		Xb	Xb
2134 ∧ 4132	C	Pd	Xd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
2134 ∧ 4213	C	Pd	Xd	Xd	Xbd	Xbd	Xbd	Xbd	Xbd
2134 ∧ 4231 [3]	P2d	C	P2d	X?	Xd	Xd	Xd	Xd	Xd
2134 ∧ 4312	Xb	Xb	Xc	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
2134 ∧ 4321 [4]	C	C	X?	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd

[1] identical to 1234 ∧ 2134 in section 5.1.3
[2] identical to 1324 ∧ 2134 in section 5.1.4
[3] equivalent to 1324 ∧ 4312 in section 5.1.4
[4] equivalent to 1234 ∧ 4312 in section 5.1.3

5.1.6 Avoiding 2314 AND a pattern of length 4

F	$n = 4$	5	6	7	8	9	10	11	12
2314	C	C	C	C					
2314 ∧ 1234 [1]	P2c	X?	P2d	P2d					
2314 ∧ 1243	C	C	C	C					
2314 ∧ 1324 [2]	C	C	C	C					
2314 ∧ 1342	C	C	C	C					
2314 ∧ 1423	C	C	C	C					
2314 ∧ 1432	C	C	C	C	C				
2314 ∧ 2134 [3]	C	C	C	C					
2314 ∧ 2143	C	C	C	C					
2314 ∧ 2341	C	C	C	C					
2314 ∧ 2413	C	C	C	C					
2314 ∧ 2431	C	C	C	C					
2314 ∧ 3124	P2c	P?	C	C					
2314 ∧ 3142	C	C	C	C					
2314 ∧ 3214	C	C	C	C					
2314 ∧ 3241	Pd	Pd	Pd	Xd	Xd	Xd	Xd	Xd	Xd
2314 ∧ 3412	C	C	C	C					
2314 ∧ 3421	C	C	C	C					
2314 ∧ 4123	C	C	C	C					
2314 ∧ 4132	C	C	C	C					
2314 ∧ 4213	P2d	C	C	C					
2314 ∧ 4231 [4]	C	C	C	C					
2314 ∧ 4312 [5]	C	Pd	Xd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd
2314 ∧ 4321 [6]	P2d	Pd	Xd	Xcd	Xcd	Xcd	Xcd	Xcd	Xcd

[1] identical to 1234 ∧ 2314 in section 5.1.3
[2] identical to 1324 ∧ 2314 in section 5.1.4
[3] identical to 2134 ∧ 2314 in section 5.1.5
[4] equivalent to 1324 ∧ 4132 in section 5.1.4
[5] equivalent to 2134 ∧ 4132 in section 5.1.5
[6] equivalent to 1234 ∧ 4132 in section 5.1.3

5.1.7 Avoiding 3124 AND a pattern of length 4

F	$n=4$	5	6	7	8	9	10	11	12
3124	Pb	Pb	Xb	Xb			Xb	Xb	Xb
3124 $\wedge$ 1234 [1]	Xb	Xd	Xbd	Xd	Xbd	Xbd	Xbd	Xbd	Xbd
3124 $\wedge$ 1243	C	C	C	C					
3124 $\wedge$ 1324 [2]	C	C	C	C					
3124 $\wedge$ 1342	Pd	Pd	Xd	Xd	Xd	Xd	Xd	Xd	Xd
3124 $\wedge$ 1423	P2d	C	C	C					
3124 $\wedge$ 1432	C	C	C	C	C				
3124 $\wedge$ 2134 [3]	C	C	C	C					
3124 $\wedge$ 2143	P2d	C	C	C					
3124 $\wedge$ 2314 [4]	P2c	P?	C	C					
3124 $\wedge$ 2341	C	Pd	Xd	Xd	Xd	Xd	Xd	Xd	Xd
3124 $\wedge$ 2413	C	C	C	C					
3124 $\wedge$ 2431	P?	C	C	C					
3124 $\wedge$ 3142	C	C	C	C					
3124 $\wedge$ 3214	C	C	C	C					
3124 $\wedge$ 3241	P2c	C	C	C					
3124 $\wedge$ 3412	P2c	C	C	C					
3124 $\wedge$ 3421	C	C	C	C	C				
3124 $\wedge$ 4123	C	C	C	C					
3124 $\wedge$ 4132 [5]	P2d	C	C	C					
3124 $\wedge$ 4213	Xb	C	C	C	Xb	Xb			Xb
3124 $\wedge$ 4231 [6]	C	C	C	C					
3124 $\wedge$ 4312 [7]	C	Pd	Xd	Xd	Xbd	Xbd	Xbd	Xbd	Xbd
3124 $\wedge$ 4321 [8]	Xb	Xb	Xb	Xb	Xb	Xb	Xb	Xb	Xb

[1] identical to 1234 $\wedge$ 1234 in section 5.1.3
[2] identical to 1324 $\wedge$ 1234 in section 5.1.4
[3] identical to 2134 $\wedge$ 1234 in section 5.1.5
[4] identical to 2314 $\wedge$ 1234 in section 5.1.6
[5] equivalent to 2314 $\wedge$ 4213 in section 5.1.6
[6] equivalent to 1324 $\wedge$ 4213 in section 5.1.4
[7] equivalent to 2134 $\wedge$ 4213 in section 5.1.5
[8] equivalent to 1234 $\wedge$ 4213 in section 5.1.3

5.1.8 Avoiding 3214 AND a pattern of length 4

F	$n = 4$	5	6	7	8	9	10	11	12
3214	Pb	Pb	Pb	Pb	Xb	Xb	Xb	Xb	Xb
3214 $\wedge$ 1234 [1]	C	C	C	C					
3214 $\wedge$ 1243	P2d	C	C	C					
3214 $\wedge$ 1324 [2]	P2c	C	C	C					
3214 $\wedge$ 1342	C	C	C	C	C				
3214 $\wedge$ 1423	C	C	C	C	C				
3214 $\wedge$ 1432	P?	C	C	C					
3214 $\wedge$ 2134 [3]	Xb	Xbd	Xd	Xd	Xbd	Xd	Xbd	Xbd	Xbd
3214 $\wedge$ 2143	C	C	C	C					
3214 $\wedge$ 2314 [4]	C	C	C	C					
3214 $\wedge$ 2341	Pd	Pd	Pd	Pd					
3214 $\wedge$ 2413	P2d	C	C	C					
3214 $\wedge$ 2431	C	C	C	C					
3214 $\wedge$ 3124 [5]	C	C	C	C					
3214 $\wedge$ 3142	P2c	C	C	C					
3214 $\wedge$ 3241	C	C	C	C					
3214 $\wedge$ 3412	C	C	C	C					
3214 $\wedge$ 3421	P2c	C	C	C					
3214 $\wedge$ 4123	Xb	Xb	C	C	Xb	Xb			Xb
3214 $\wedge$ 4132 [6]	C	C	C	C					
3214 $\wedge$ 4213 [7]	C	C	C	C					
3214 $\wedge$ 4231 [8]	P2d	C	C	C					
3214 $\wedge$ 4312 [9]	Xb	Xb	X?	P?	Xb	Xb		Xb	Xb
3214 $\wedge$ 4321 [10]	C	X?	Xd	Xbd	Xd	Xbd	Xbd	Xbd	Xbd

[1] identical to 1234 $\wedge$ 3214 in section 5.1.3
[2] identical to 1324 $\wedge$ 3214 in section 5.1.4
[3] identical to 2134 $\wedge$ 3214 in section 5.1.5
[4] identical to 2314 $\wedge$ 3214 in section 5.1.6
[5] identical to 3124 $\wedge$ 3214 in section 5.1.7
[6] equivalent to 2314 $\wedge$ 4123 in section 5.1.6
[7] equivalent to 3124 $\wedge$ 4123 in section 5.1.7
[8] equivalent to 1324 $\wedge$ 4123 in section 5.1.4
[9] equivalent to 2134 $\wedge$ 4123 in section 5.1.5
[10] equivalent to 1234 $\wedge$ 4123 in section 5.1.3

5.2 Bipartition by parity when avoiding an increasing sequence

According to Corollary 4.2 $G_n(1\dots k)$ is always bipartite and the partitions are the sets of even and odd permutations in $S_n(1\dots k)$. We are now interested in the size difference between the two partitions because a difference of more than one proves the minimal jump graph has no Hamiltonian path.

We wrote program Listing A.8 to count the size difference. The program iteratively generates all permutations without storing them, tests if they avoid the pattern $1\dots k$ and then computes their parity. This naive approach is preferable over using the efficient algorithm for generating the pattern-avoiding permutations discovered and implemented by Kuszmaul [6] because the algorithm requires storing all permutations at the same time and memory becomes a bottleneck before execution time.

5.2.1 Avoiding 123

For even n the partitions have identical size.

For odd n the size differences are the Catalan numbers. The larger partition is alternating.

These observations are proven in Corollary 4.4 and 4.7.

n	even partition	odd partition	size difference	larger partition
3	2	3	1	odd
4	7	7	0	-
5	22	20	2	even
6	66	66	0	-
7	212	217	5	odd
8	715	715	0	-
9	2438	2424	14	even
10	8398	8398	0	-
11	29372	29414	42	odd
12	104006	104006	0	-
13	371516	371384	132	even
14	1337220	1337220	0	-
15	4847208	4847637	429	odd
16	17678835	17678835	0	-

5.2.2 Avoiding 1234

For even n and odd $n+1$ the size differences appear to be identical. The larger partition is alternating.

The size differences match sequence A246138 [5] in the Online Encyclopedia of Integer Sequences to the extent of the experiments. However it is not clear if this holds for large n too.

n	even partition	odd partition	size difference	larger partition
4	11	12	1	odd
5	51	52	1	odd
6	258	255	3	even
7	1382	1379	3	even
8	7879	7888	9	odd
9	47175	47184	9	odd
10	293311	293279	32	even
11	1881661	1881629	32	even
12	12396285	12396420	135	odd
13	83539221	83539356	135	odd
14	574104369	574103721	648	even

5.2.3 Avoiding 12345

For even n the partitions have identical size.

For odd n the size differences have multiple matches in the Online Encyclopedia of Integer Sequences to the extent of the experiments. However it is not clear if this holds for large n too. The larger partition is alternating.

n	even partition	odd partition	size difference	larger partition
5	59	60	1	odd
6	347	347	0	-
7	2293	2289	4	even
8	16662	16662	0	-
9	130897	130911	14	odd
10	1095344	1095344	0	-
11	9659368	9659320	48	even
12	89054352	89054352	0	-
13	852992859	852993024	165	odd
14	8445810583	8445810583	0	-

5.2.4 Avoiding 123456

For even n and odd $n+1$ the size differences appear to be identical. The larger partition is alternating.

The size differences have multiple matches in the Online Encyclopedia of Integer Sequences to the extent of the experiments. However it is not clear if this holds for large n too. The larger partition is alternating.

n	even partition	odd partition	size difference	larger partition
6	359	360	1	odd
7	2501	2502	1	odd
8	19717	19712	5	even
9	172421	172416	5	even
10	1645785	1645805	20	odd
11	16917547	16917567	20	odd
12	185265879	185265804	75	even
13	2142855807	2142855732	75	even

5.3 Algorithm J and non-tame patterns

It has been proven that Algorithm J generates $G_n(F)$ if F contains only tame patterns. We wonder if this is a necessary or only a sufficient condition. We wrote a simple program Listing A.5 that runs Algorithm J on all non-tame patterns of length 4.

The program found that if F contains only a single non-tame pattern, Algorithm J fails with all possible initial values for $n = 5, 6, 7, 8$. If F contains one non-tame pattern AND any other pattern, Algorithm J fails with all possible initial values for $n = 6, 7, 8$. Testing all possible initial values for $n \geq 9$ takes too long.

We also tried relaxing the terminating condition of Algorithm J to always chose left or right if the jump direction is ambiguous, but Algorithm J still fails with all possible initial values for large enough n.

Chapter 6

Conclusions

We summarize the main points of the thesis and discuss some open problems.

In Chapter 2 we showed that reversal is an isomorphism on minimal jump graphs, which reduces the number of graphs we have to consider by half. Unfortunately, complementation and inverse, that are elementary transformations on a set of pattern-avoiding permutations, are not valid isomorphisms on minimal jump graphs. From the computational experiments about Hamiltonicity of minimal jump graphs in section 5.1 we can conclude that no further graph isomorphisms exist.

In Chapter 3 we gave simple reasons why many $G_n(F)$ for F containing patterns up to length 4 have no Hamiltonian cycle or path for sufficiently large n. They explain almost all cases of non-Hamiltonicity found in the computational experiments. The remaining cases are not of high interest because they are singular outliers that do not occur for smaller or larger n.

In Chapter 4 we proved that $G_n(1\dots k)$ is always bipartite and that the partitions are the sets of even and odd permutations in $S_n(1\dots k)$. Based on the computational experiments we establish the following conjectures for $S_n(1\dots k)$.

- For odd k and odd n the size differences of the even and odd partitions are strictly growing and the larger partition is alternating.
- For odd k and even n the even and odd partitions have identical size.
- For even k and even n the size differences of the even and odd partitions are strictly growing and the larger partition is alternating. Furthermore, for n and $n+1$ the size difference is identical.

The unbalanced bipartitions can be used to prove non-Hamiltonicity in the corresponding minimal jump graph. For $S_n(123)$ we gave a recursive formula for the number of even and odd permutations and used it to prove

that the size difference is zero for even n and corresponds to the Catalan numbers for odd n. This gives an alternative proof, besides the proof in Chapter 3, that $G_n(123)$ has no Hamiltonian path for odd $n \geq 5$. We also demonstrated how the idea behind the formula for 123 can be extended to any increasing pattern. However, the general formula requires k parameters when avoiding the pattern $1 \dots k$ and the recursion has 2^k case distinctions. This renders finding a proof for the general case, or even 1234, highly impractical.

In Chapter 5 we presented the results of the computational experiments. Notably, we tested Hamiltonicity of the minimal jump graph when avoiding one of the patterns 123, 213, 1234, 1324, 2134, 2314, 3124, 3214 and when avoiding one of the aforementioned patterns AND a pattern of the same length. This covers all non-tame patterns up to length 4, taking into account reversal. These results provide a nice overview of minimal jump graphs that avoid a non-tame pattern and serves as basis for conjectures to be proven.

6.1 Open questions

In the computational experiments we saw that some graphs have a Hamiltonian path to the extent of the experiments and some do not. The open question is if some of these graphs have a Hamiltonian path for all n or if for sufficiently large n eventually all graphs become non-Hamiltonian.

Another open question is if the existence of unbalanced bipartitions we have seen in some graphs can be shown for all n and thus prove that no Hamiltonian path exists. In this thesis we looked at $G_n(1 \dots k)$, but there are several other graphs with unbalanced bipartitions.

Appendix A

Appendix

A.1 Code

Listing A.1: minimal_jump_graph.h

```
#ifndef _MINIMAL_JUMP_GRAPH_H
#define _MINIMAL_JUMP_GRAPH_H

#include <cstdint>
#include <list>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include "pattern_avoidance.h"
using namespace std;  // pattern_avoidance.h already
   enforces namespace std...


namespace minimal_jump {

typedef struct min_jump_t {
    perm_t from;
    perm_t to;
    bool right;
    int steps;
    uint64_t digit;
    min_jump_t(perm_t from, perm_t to, bool right,
       int steps, uint64_t digit)
        : from(from)
        , to(to)
        , right(right)
```

```
        , steps(steps)
        , digit(digit)
    { }
} min_jump_t;


/** Returns all minimal jumps for a permutation of
   length n with respect to a
    set of permutations.
*/
vector<perm_t> compute_min_jumps(const perm_t perm,
   const int n, const unordered_set<perm_t>& S);


/** Returns all minimal jumps for a permutation of
   length n with respect to a
    set of permutations. The minimal jumps are
       returned as min_jump_t structs.
*/
vector<min_jump_t> compute_min_jumps_verbose(const
   perm_t perm, const int n, const unordered_set<
   perm_t>& S);


/** Returns the minimal jump graph for permutations
   of length n.

    The minimal jump graph is stored as an adjacency
       list. The vertex labels
    are given by perm_to_vertex_mapping<T>(S). The
       only supported typename is
    int64_t.
*/
template<typename T>
vector<vector<T>> compute_adjacency_list(const vector
   <perm_t>& S, const int n);


/** Returns the size difference of the bipartition of
    a graph. If the graph
    is not bipartite, 0 is returned. Therefore this
       function cannot distinguish
    a graph with balanced bipartition from a graph
       with no bipartition. However
```

```
    this distinction is not needed to decide
       Hamiltonicity of a graph.
*/
template<typename T>
T get_bipartition_size_difference(const vector<vector
   <T>>& G);


/** Returns the number of vertices in a graph that
   have a particular degree.
*/
template<typename T>
T get_count_of_degree(const vector<vector<T>>& G,
   const T degree);


/** Returns the vertices of a graph that have a
   particular degree.
*/
template<typename T>
vector<T> get_vertices_of_degree(const vector<vector<
   T>>& G, const T degree);


/** Returns true if a graph contains a particular
   cycle. The cycle is given as
    a vector of the visited vertices. The final
       vertex is omitted.
*/
template<typename T>
bool has_cycle(const vector<vector<T>>& G, const
   vector<T>& cycle);


/** Returns true if a graph contains a particular
   path. The path is given as
    a vector of the visited vertices.
*/
template<typename T>
bool has_path(const vector<vector<T>>& G, const
   vector<T>& path);


/** Returns true if and only if a graph has
```

```
   disconnected components.
*/
template<typename T>
bool is_disconnected(const vector<vector<T>>& G);


/** Returns true if and only if a graph is empty, i.e
   . has no vertices or
    edges.
*/
template<typename T>
bool is_empty(const vector<vector<T>>& G);


/** Returns a mapping from a vector S of permutations
    to [0, 1, ..., |S|-1]
    corresponding to the order of the permutations in
        S.

    Asserts that the arithmetic type T can store
       values up to |S|.
*/
template<typename T>
unordered_map<perm_t, T> perm_to_vertex_mapping(const
    vector<perm_t>& S);


/** Deletes edges that are adjacent to a vertex that
   has two degree two
    neighbours. Such edges cannot be part of a
       Hamiltonian cycle.

    Returns true if and only if edges have been
       removed.
*/
template<typename T>
bool prune_degree_two_vertices(vector<vector<T>>& G);


/** Reads a .sol file computed by the Concorde TSP
   solver and stores the cycle
    in the parameter solution.

    Returns true if and only the .sol file was
```

```
       successfully read.
*/
template<typename T>
bool read_sol_file(const string in_file, vector<T>&
   solution);


/** Writes a file in the TSPLIB format containing a
   TSP problem equivalent to
     the Hamiltonian cycle/path problem (determined by
         parameter path_problem)
     of a minimal jump graph avoiding a pattern of
        length n.

     Returns true if and only if the file was
        successfully written at the path
     given by parameter out_file.
*/
template<typename T>
bool write_tsp_file(const vector<vector<T>>& G, const
    string pattern, const int n, const bool
   path_problem, const string out_file);

} // namespace minimal_jump

#endif
```

Listing A.2: minimal_jump_graph.cpp

```
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <cstdlib>
#include <limits>
#include <list>
#include <fstream>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include "minimal_jump_graph.h"
#include "pattern_avoidance.h"
using namespace std;
using namespace pattern_avoidance;
```

```
namespace minimal_jump {

vector<perm_t> compute_min_jumps(const perm_t perm,
   const int n, const unordered_set<perm_t>& S)
{
    vector<perm_t> min_jumps;
    for (int i = 0; i < n; i++)
    {
        const uint64_t entry_i = getdigit(perm, i);
        perm_t right_jump = perm;
        for (int j = i + 1; j < n; j++)
        {
            const uint64_t entry_j = getdigit(perm, j
               );
            if (entry_i < entry_j)
            {
                break;
            }
            right_jump = setdigit(right_jump, j,
               entry_i);
            right_jump = setdigit(right_jump, j - 1,
               entry_j);
            if (S.count(right_jump) == 1)
            {
                min_jumps.push_back(right_jump);
                break;
            }
        }
        perm_t left_jump = perm;
        for (int j = i - 1; j >= 0; --j)
        {
            const uint64_t entry_j = getdigit(perm, j
               );
            if (entry_i < entry_j)
            {
                break;
            }
            left_jump = setdigit(left_jump, j,
               entry_i);
            left_jump = setdigit(left_jump, j + 1,
               entry_j);
            if (S.count(left_jump) == 1)
            {
                min_jumps.push_back(left_jump);
```

```
                break;
            }
        }
    }
    return min_jumps;
}


vector<min_jump_t> compute_min_jumps_verbose(const
   perm_t perm, const int n, const unordered_set<
   perm_t>& S)
{
    vector<min_jump_t> min_jumps;
    for (int i = 0; i < n; i++)
    {
        const uint64_t entry_i = getdigit(perm, i);
        perm_t right_jump = perm;
        for (int j = i + 1; j < n; j++)
        {
            const uint64_t entry_j = getdigit(perm, j
               );
            if (entry_i < entry_j)
            {
                break;
            }
            right_jump = setdigit(right_jump, j,
               entry_i);
            right_jump = setdigit(right_jump, j - 1,
               entry_j);
            if (S.count(right_jump) == 1)
            {
                min_jumps.push_back(min_jump_t(perm,
                   right_jump, true, j - i, entry_i))
                   ;
                break;
            }
        }
        perm_t left_jump = perm;
        for (int j = i - 1; j >= 0; --j)
        {
            const uint64_t entry_j = getdigit(perm, j
               );
            if (entry_i < entry_j)
            {
```

```
                    break;
                }
                left_jump = setdigit(left_jump, j,
                    entry_i);
                left_jump = setdigit(left_jump, j + 1,
                    entry_j);
                if (S.count(left_jump) == 1)
                {
                    min_jumps.push_back(min_jump_t(perm,
                        left_jump, false, i - j, entry_i))
                        ;
                    break;
                }
            }
        }
        return min_jumps;
    }


    template<typename T>
    vector<vector<T>> compute_adjacency_list(const vector
        <perm_t>& S, const int n)
    {
        const unordered_map<perm_t, T> vertex_map =
            perm_to_vertex_mapping<T>(S);
        const unordered_set<perm_t> S_hashtable(S.begin()
            , S.end());
        vector<vector<T>> G(S.size());
        for (const perm_t& perm : S)
        {
            const T u = vertex_map.at(perm);
            vector<perm_t> min_jumps = compute_min_jumps(
                perm, n, S_hashtable);
            G[u].reserve(min_jumps.size());
            for (const perm_t& jump : min_jumps)
            {
                const T v = vertex_map.at(jump);
                G[u].push_back(v);
            }
        }
        return G;
    }
```

```
template<typename T>
T get_bipartition_size_difference(const vector<vector
   <T>>& G)
{
    if (G.size() < 2)
    {
        return false;
    }
    vector<bool> blue(G.size(), false);
    vector<bool> red(G.size(), false);

    list<T> curr_round, next_round;
    curr_round.push_back(G.front().front());
    bool is_blue_round = true;
    while (!curr_round.empty())
    {
        for (const T& curr_node : curr_round)
        {
            if (is_blue_round)
            {
                if (red[curr_node])
                {
                    return 0;
                }
                if (!blue[curr_node])
                {
                    blue[curr_node] = true;
                    for (const T& new_node : G[
                       curr_node])
                    {
                        next_round.push_back(new_node
                           );
                    }
                }
            }
            else
            {
                if (blue[curr_node])
                {
                    return 0;
                }
                if (!red[curr_node])
                {
                    red[curr_node] = true;
```

```
                        for (const T& new_node : G[
                           curr_node])
                        {
                            next_round.push_back(new_node
                               );
                        }
                    }
                }
            }
            curr_round.clear();
            swap(curr_round, next_round);
            is_blue_round = !is_blue_round;
        }
        const T count_blue = count(blue.begin(), blue.end
           (), true);
        const T count_red = count(red.begin(), red.end(),
            true);
        if (count_blue > count_red)
        {
            return count_blue - count_red;
        }
        else
        {
            return count_red - count_blue;
        }
    }


    template<typename T>
    T get_count_of_degree(const vector<vector<T>>& G,
       const T degree)
    {
        T count = 0;
        for (const vector<T>& adj_list : G)
        {
            if ((T) adj_list.size() == degree)
            {
                count += 1;
            }
        }
        return count;
    }
```

```
template<typename T>
vector<T> get_vertices_of_degree(const vector<vector<
   T>>& G, const T degree)
{
    vector<T> vertices;
    for (size_t i = 0; i < G.size(); i++)
    {
        if ((T) G[i].size() == degree)
        {
            vertices.push_back((T) i);
        }
    }
    return vertices;
}


template<typename T>
bool has_cycle(const vector<vector<T>>& G, const
   vector<T>& cycle)
{
    for (size_t i = 0; i < cycle.size(); i++)
    {
        const T u = cycle[i];
        const T v = (i == cycle.size() - 1) ? cycle
           [0] : cycle[i + 1];
        const bool edge_exists = find(G[u].begin(), G
           [u].end(), v) != G[u].end();
        if (!edge_exists)
        {
            return false;
        }
    }
    return true;
}


template<typename T>
bool has_path(const vector<vector<T>>& G, const
   vector<T>& path)
{
    for (size_t i = 0; i < path.size() - 1; i++)
    {
        const T u = path[i];
        const T v = path[i + 1];
```

```
            const bool edge_exists = find(G[u].begin(), G
               [u].end(), v) != G[u].end();
            if (!edge_exists)
            {
                return false;
            }
        }
        return true;
    }


    template<typename T>
    bool is_disconnected(const vector<vector<T>>& G)
    {
        vector<bool> visited(G.size(), false);
        list<T> q;
        for (const vector<T>& adj_list : G)
        {
            if (!adj_list.empty())
            {
                const T start = adj_list.front();
                q.push_back(start);
                visited[start] = true;
                break;
            }
        }
        while (!q.empty())
        {
            const T curr_node = q.front();
            q.pop_front();
            for (const T& new_node : G[curr_node])
            {
                if (!visited[new_node])
                {
                    q.push_back(new_node);
                    visited[new_node] = true;
                }
            }
        }
        const T count_visited = count(visited.begin(),
           visited.end(), true);
        return count_visited != (T) G.size();
    }
```

```
template<typename T>
bool is_empty(const vector<vector<T>>& G)
{
    return G.size() == 0;
}


template<typename T>
unordered_map<perm_t, T> perm_to_vertex_mapping(const
    vector<perm_t>& S)
{
    // the elements of S cannot be mapped to type T
       if |S| is larger than the maximum value of T
    assert(S.size() < numeric_limits<T>::max());
    unordered_map<perm_t, T> vertex_map(S.size());
    T curr_vertex = 0;
    for (const perm_t& perm : S)
    {
        vertex_map[perm] = curr_vertex += 1;
    }
    return vertex_map;
}


template<typename T>
bool prune_degree_two_vertices(vector<vector<T>>& G)
{
    bool optimization_occured = false;
    const vector<T> degree_two_vertices =
       get_vertices_of_degree(G, (T) 2);
    for (const T& vertex_1 : degree_two_vertices)
    {
        for (const T& neighbor : G[vertex_1])
        {
            for (const T& vertex_2 : G[neighbor])
            {
                if (vertex_1 != vertex_2 && G[
                   neighbor].size() > 2 && find(
                   degree_two_vertices.begin(),
                   degree_two_vertices.end(),
                   vertex_2) != degree_two_vertices.
                   end())
                {
```

```
                        for (const T& neighbor_to : G[
                           neighbor])
                        {
                            if (neighbor_to != vertex_1
                               && neighbor_to != vertex_2
                               )
                            {
                                const typename vector<T
                                   >::iterator it = find(
                                   G[neighbor_to].begin()
                                   , G[neighbor_to].end()
                                   , neighbor);
                                if (it != G[neighbor_to].
                                   end())
                                {
                                    G[neighbor_to].erase(
                                       it);
                                }
                            }
                        }
                        G[neighbor] = {vertex_1, vertex_2
                           };
                        optimization_occured = true;
                    }
                }
            }
        }
        return optimization_occured;
    }


    template<typename T>
    bool read_sol_file(const string in_file, vector<T>&
       solution)
    {
        ifstream file(in_file);
        if (!file.good())
        {
            return false;
        }
        size_t num_vertices;
        file >> num_vertices;
        solution.resize(num_vertices);
        for (size_t i = 0; i < num_vertices; i++)
```

```
    {
        file >> solution[i];
    }
    return true;
}


template<typename T>
bool write_tsp_file(const vector<vector<T>>& G, const
    string pattern, const int n, const bool
   path_problem, const string out_file)
{
    ofstream file(out_file);
    if (!file.good())
    {
        return false;
    }
    file << "NAME : G_" << n << "(" << pattern << ")"
        << endl;
    file << "TYPE : TSP" << endl;
    if (path_problem)
    {
        file << "DIMENSION : " << G.size() + 1 <<
           endl;
        file << "COMMENT : Hamiltonian Path Problem"
           << endl;
    }
    else
    {
        file << "DIMENSION : " << G.size() << endl;
        file << "COMMENT : Hamiltonian Cycle Problem"
            << endl;
    }
    file << "EDGE_WEIGHT_TYPE: EXPLICIT" << endl;
    file << "EDGE_WEIGHT_FORMAT: UPPER_ROW" << endl;
    file << "EDGE_WEIGHT_SECTION" << endl;
    // set edge weight to zero if the edge exists in
       the graph, otherwise
    // set edge weight to one.
    // if hamiltonian path we add an auxiliary vertex
        that is connected to
    // all other vertices with weight zero
    for (size_t i = 0; i < G.size(); i++)
    {
```

```
            vector<bool> adj_vec(G.size(), false);
            for (const T& vertex : G[i])
            {
                adj_vec[vertex] = true;
            }
            for (size_t j = 0; j < G.size(); j++)
            {
                if (i < j)
                {
                    if (adj_vec[j])
                    {
                        file << "␣0";
                    }
                    else
                    {
                        file << "␣1";
                    }
                }
                else
                {
                    file << "␣␣";
                }
            }
            if (path_problem)
            {
                file << "␣0";
            }
            file << endl;
        }
        file << "EOF" << endl;
        return true;
    }


    template vector<vector<int64_t>>
       compute_adjacency_list(const vector<perm_t>& S,
       const int n);

    template int64_t get_bipartition_size_difference(
       const vector<vector<int64_t>>& G);

    template int64_t get_count_of_degree(const vector<
       vector<int64_t>>& G, const int64_t degree);
```

```
template vector<int64_t> get_vertices_of_degree(const
    vector<vector<int64_t>>& G, const int64_t degree)
   ;

template bool has_cycle(const vector<vector<int64_t
   >>& G, const vector<int64_t>& cycle);

template bool has_path(const vector<vector<int64_t>>&
    G, const vector<int64_t>& path);

template bool is_disconnected(const vector<vector<
   int64_t>>& G);

template bool is_empty(const vector<vector<int64_t>>&
    G);

template unordered_map<perm_t, int64_t>
   perm_to_vertex_mapping(const vector<perm_t>& S);

template bool prune_degree_two_vertices(vector<vector
   <int64_t>>& G);

template bool read_sol_file(const string in_file,
   vector<int64_t>& solution);

template bool write_tsp_file(const vector<vector<
   int64_t>>& G, const string pattern, const int n,
   const bool path_problem, const string out_file);

}
```

Listing A.3: pattern_avoidance.h

```
#ifndef _PATTERN_AVOIDANCE_H
#define _PATTERN_AVOIDANCE_H

#include <cstdint>
#include <ostream>
#include <string>
#include <vector>
#include "perm.h"
using namespace std;  // perm.h already enforces
   namespace std...
```

```
namespace pattern_avoidance {

/** Inserts a new digit into a permutation at a
   specific position.
*/
inline perm_t add_digit(const perm_t perm, const int
   pos, const uint64_t new_digit)
{
    return setdigit(addpos(perm, pos), pos, new_digit
       );
}


/** Appends all non-duplicate permutations of length
   n to each string of a
    vector.

    For example calling append_all_permutations
       ({"123, 321"}, 3) returns
    {"123", "123 132", "123 213", "123 231", "123
       312", "123 321",
     "321", "321 123", "321 132", "321 213", "321
        231", "321 312"}.

    Asserts n <= 20 because n! cannot be stored in a
       64-bit integer for larger
    values.
*/
vector<string> append_all_permutations(const vector<
   string>& patterns, const int n);


/** Returns true if and only if the permutation lhs
   is lexographically smaller
    than permutations rhs. Both permutations must be
       of length n.

    TODO: Do we even need this? The < comparison
       should work if all unused data
    in the permutation are zero bits.
*/
bool compare_perm(const perm_t lhs, const perm_t rhs,
    const int n);
```

```
/** Returns all permutations on the set {1, ..., n}.

    Asserts n <= 20 because n! cannot be stored in a
       64-bit integer for larger
    values.
*/
vector<perm_t> compute_all_permutations(const int n);


/** Returns all permutations on the set {1, ..., n}
   as simple strings, i.e.
    there is no separator between digits.

    Asserts n < 10 because permutations of length n
       >= 10 have ambiguous
    representation as simple strings.
*/
vector<string> compute_all_permutations_simple_string
   (const int n);


/** Returns all ascents in a permutation of length n.
    An ascent is the position of a digit that is
       followed by a larger digit.
    Position index starts at zero.

    For example the ascents of 1324 are 0 and 2.
*/
vector<int> get_ascents(const perm_t perm, const int
   n);


/** Returns the number of ascents in a permutation of
    length n.
*/
int get_num_ascents(const perm_t perm, const int n);


/** Returns the position of a digit in a permutation.
    Position index starts at
    zero. Returns -1 if the permutation does not
       contain the digit.
*/
```

```
int get_digit_position(const perm_t perm, const
   uint64_t digit);


/** Returns the inversion number of a permutation of
   length n.
*/
int inversion_number(const perm_t perm, const int n);


/** Returns the identity permutation (increasing
   sequence) of length n.
*/
perm_t identity_permutation(const int n);


/** Returns true if and only if a permutation has an
   even inversion number.
*/
inline bool is_even(const perm_t perm, const int n)
{
    return inversion_number(perm, n) % 2 == 0;
}


/** Returns true if and only if a permutation is tame
   , i.e. the largest digit
    is not at the left or right-most position.
*/
inline bool is_tame(const perm_t perm, const int n)
{
    const uint64_t max_digit = n - 1;
    return getdigit(perm, 0) != max_digit && getdigit
       (perm, n - 1) != max_digit;
}


/** Returns all permutations up to length n that
   avoid the pattern.

    The return value is a two-dimensional vector
       where vec[i] contains all patterns
    of length i in no particular order.
*/
```

```
vector<vector<perm_t>> pattern_avoiding_permutations(
    const string pattern, const int n);


/** Returns all permutations up to length n that
    avoid the pattern.

     The return value is a two-dimensional vector
        where vec[i] contains all patterns
     of length i in lexicographical order.
*/
vector<vector<perm_t>>
    pattern_avoiding_permutations_sorted(const string
    pattern, const int n);


/** Returns a string representation of a permutation
    of length n. Each digit is
     separated by a whitespace.

     For example 1324 is represented as "1 3 2 4".
*/
string perm_to_str(const perm_t perm, const int n);


/** Returns a string representation of a permutation
    of length n. There is no
     separator between digits. Therefore the
        representation is ambiguous for
     permutations of length 10 and larger.
*/
string perm_to_str_simple(const perm_t perm, const
    int n);


/** Returns the permutation corresponding to a simple
     string.
*/
perm_t string_to_permutation(const string permutation
    );

} // namespace pattern_avoidance

#endif
```

Listing A.4: pattern_avoidance.cpp

```
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <iterator>
#include <list>
#include <sstream>
#include <string>
#include <utility>
#include <vector>
#include "countavoiders.h"
#include "perm.h"
#include "pattern_avoidance.h"
using namespace std;


namespace pattern_avoidance {

int64_t factorial(const int n)
{
    assert(n <= 20);
    int64_t result = 1;
    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }
    return result;
}


vector<string> append_all_permutations(const vector<
   string>& patterns, const int n)
{
    vector<string> new_patterns;
    new_patterns.reserve(patterns.size() * (size_t)(
       factorial(n)));
    const vector<string> all_permutations =
       compute_all_permutations_simple_string(n);
    for (const string& pattern : patterns)
    {
        new_patterns.push_back(pattern);
        for (const string& permutation :
           all_permutations)
```

```
        {
            if (pattern != permutation)
            {
                new_patterns.push_back(pattern + "␣"
                   + permutation);
            }
        }
    }
    assert(new_patterns.capacity() == new_patterns.
       size());
    return new_patterns;
}


bool compare_perm(const perm_t lhs, const perm_t rhs,
    const int n)
{
    for (int i = 0; i < n; i++)
    {
        const uint64_t lhs_digit = getdigit(lhs, i);
        const uint64_t rhs_digit = getdigit(rhs, i);
        if (lhs_digit < rhs_digit)
        {
            return true;
        }
        else if (lhs_digit > rhs_digit)
        {
            return false;
        }
    }
    return false;
}


vector<perm_t> compute_all_permutations(const int n)
{
    assert(n <= 20);
    vector<uint64_t> digits(n);
    for (int i = 0; i < n; i++)
    {
        digits[i] = i;
    }
    vector<perm_t> all_permutations;
    all_permutations.reserve(factorial(n));
```

```
    do {
        perm_t perm = 0;
        for (int i = 0; i < n; i++)
        {
            perm = setdigit(perm, i, digits[i]);
        }
        all_permutations.push_back(perm);
    }
    while (next_permutation(digits.begin(), digits.
       end()));
    return all_permutations;
}


vector<string> compute_all_permutations_simple_string
   (const int n)
{
    assert(n < 10);
    string permutation = perm_to_str_simple(
       identity_permutation(n), n);
    vector<string> all_permutations;
    all_permutations.reserve(factorial(n));
    do {
        all_permutations.push_back(permutation);
    }
    while (next_permutation(permutation.begin(),
       permutation.end()));
    return all_permutations;
}


vector<int> get_ascents(const perm_t perm, const int
   n)
{
    vector<int> ascents;
    uint64_t curr_digit = getdigit(perm, 0);
    for (int i = 0; i < n - 1; i++)
    {
        uint64_t next_digit = getdigit(perm, i + 1);
        if (curr_digit < next_digit)
        {
            ascents.push_back(i);
        }
        curr_digit = next_digit;
```

```
    }
    ascents.shrink_to_fit();
    assert(ascents.capacity() == ascents.size());
    return ascents;
}


int get_digit_position(const perm_t perm, const
   uint64_t digit)
{
    for (int i = 0; i < MAXPERMSIZE; i++)
    {
        if (getdigit(perm, i) == digit)
        {
            return i;
        }
    }
    return -1;
}


int get_num_ascents(const perm_t perm, const int n)
{
    int ascents = 0;
    uint64_t prev_digit = getdigit(perm, 0);
    for (int i = 1; i < n; i++)
    {
        uint64_t curr_digit = getdigit(perm, i);
        if (prev_digit < curr_digit)
        {
            ascents += 1;
        }
        prev_digit = curr_digit;
    }
    return ascents;
}


int inversion_number(const perm_t perm, const int n)
{
    int result = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
```

```
            {
                if (getdigit(perm, i) > getdigit(perm, j)
                   )
                {
                    result += 1;
                }
            }
        }
        return result;
    }


    perm_t identity_permutation(const int n)
    {
        perm_t perm = 0;
        for (int i = 0; i < n; i++)
        {
            perm = setdigit(perm, i, i);
        }
        return perm;
    }


    vector<vector<perm_t>> pattern_avoiding_permutations(
       const string pattern, const int n)
    {
        vector<list<perm_t>> avoiders_list;
        buildavoidersfrompatternlist(pattern, n,
           avoiders_list);
        vector<vector<perm_t>> avoiders(n + 1);
        for (int i = 1; i <= n; i++)
        {
            avoiders[i].reserve(avoiders_list[i].size());
            copy(begin(avoiders_list[i]), end(
               avoiders_list[i]), back_inserter(avoiders[
               i]));
        }
        return avoiders;
    }


    vector<vector<perm_t>>
       pattern_avoiding_permutations_sorted(const string
       pattern, const int n)
```

```
{
    vector<vector<perm_t>> avoiders =
       pattern_avoiding_permutations(pattern, n);
    for (int i = 1; i <= n; i++)
    {
        sort(avoiders[i].begin(), avoiders[i].end(),
           [i](perm_t lhs, perm_t rhs) { return
           compare_perm(lhs, rhs, i); });
    }
    return avoiders;
}

string perm_to_str(const perm_t perm, const int n)
{
    ostringstream buffer;
    for (int i = 0; i < n - 1; i++)
    {
        buffer << getdigit(perm, i) + 1 << "␣";
    }
    buffer << getdigit(perm, n - 1) + 1;
    return buffer.str();
}


string perm_to_str_simple(const perm_t perm, const
   int n)
{
    ostringstream buffer;
    for (int i = 0; i < n - 1; i++)
    {
        buffer << getdigit(perm, i) + 1;
    }
    buffer << getdigit(perm, n - 1) + 1;
    return buffer.str();
}


perm_t string_to_permutation(const string permutation
   )
{
    perm_t perm = 0;
    for (size_t i = 0; i < permutation.size(); i++)
    {
        // Convert ASCII character to number and
```

```
            subtract 1 because digits in perm_t are
               zero-based
            const char digit = permutation[i] - 49;
            perm = setdigit(perm, i, digit);
        }
        return perm;
    }

    } // namespace pattern_avoidance
```

Listing A.5: write_algorithm_j.cpp

```
    #include <algorithm>
    #include <iostream>
    #include <string>
    #include <unordered_map>
    #include <unordered_set>
    #include <vector>
    #include "pattern_avoidance.h"
    #include "minimal_jump_graph.h"
    using namespace std;
    using namespace minimal_jump;
    using namespace pattern_avoidance;


    vector<perm_t> algorithm_j(const vector<perm_t>& S,
       const int n, const perm_t start)
    {
        vector<perm_t> sequence;
        sequence.reserve(S.size());
        const unordered_set<perm_t> S_hashtable(S.begin()
           , S.end());
        unordered_map<perm_t, bool> visited(S.size());
        visited[start] = true;
        sequence.push_back(start);
        perm_t most_recent = start;
        bool terminate = false;
        // bool pick_right = true;
        while (!terminate)
        {
            vector<min_jump_t> min_jumps =
               compute_min_jumps_verbose(most_recent, n,
               S_hashtable);
            sort(min_jumps.begin(), min_jumps.end(), [](
               min_jump_t lhs, min_jump_t rhs) { return
```

```
   lhs.digit > rhs.digit; });
terminate = true;
for (size_t i = 0; i < min_jumps.size(); i++)
{
    if (!visited[min_jumps[i].to])
    {
        if (i + 1 < min_jumps.size() &&
           min_jumps[i].digit == min_jumps[i
           + 1].digit && !visited[min_jumps[i
            + 1].to])
        {
            // jump direction is ambiguous
            // cout << "terminating at " <<
               perm_to_str(min_jumps[i].to, n
               ) << " because direction is
               ambiguous" << endl;
            break;
            // resolve by picking alternating
                right or left jump
            // if ((pick_right && min_jumps[i
               ].right) || (!pick_right && !
               min_jumps[i].right))
            // {
            //     most_recent = min_jumps[i
               ].to;
            // }
            // else
            // {
            //     most_recent = min_jumps[i
               ].to;
            // }
            // pick_right = !pick_right;
            // resolve by picking both and
               violate minimal jump property
            // sequence.push_back(min_jumps[i
               ].to);
            // visited[min_jumps[i].to] =
               true;
            // most_recent = min_jumps[i +
               1].to;
        }
        // cout << "selecting " <<
           perm_to_str(min_jumps[i].to, n) <<
            endl;
```

```
                most_recent = min_jumps[i].to;
                sequence.push_back(most_recent);
                visited[most_recent] = true;
                terminate = false;
                break;
            }
        }
    }
    return sequence;
}


vector<perm_t> algorithm_j(const vector<perm_t>& S,
   const int n)
{
    if (S.empty())
    {
        return vector<perm_t>();
    }
    return algorithm_j(S, n, S.front());
}


int main()
{
    vector<string> patternlist = {"1234",
                                  "1324",
                                  "2134",
                                  "2314",
                                  "3124",
                                  "3214",
                                  "4123",
                                  "4132",
                                  "4213",
                                  "4231",
                                  "4312",
                                  "4321"};
    patternlist = append_all_permutations(patternlist
       , 4);
    constexpr int n_min = 4;
    constexpr int n_max = 8;
    for (const string& pattern : patternlist)
    {
        const vector<vector<perm_t>> avoiders =
```

```
            pattern_avoiding_permutations(pattern,
               n_max);
        bool success = true;
        for (int i = n_min; i <= n_max; i++)
        {
            for (const perm_t& perm : avoiders[i])
            {
                const vector<perm_t> sequence =
                   algorithm_j(avoiders[i], i, perm);
                if (sequence.size() != avoiders[i].
                   size())
                {
                    // cout << "Algorithm J failed
                       for S_" << i << "(" << pattern
                        << "): Expected " << avoiders
                       [i].size() << " but got " <<
                       sequence.size() << endl;
                    success = false;
                }
                else
                {
                    cout << "Algorithm␣J␣succeeded␣
                       for␣S_" << i << "(" << pattern
                        << ")␣with␣" << perm_to_str(
                       perm, i) << endl;
                }
            }
        }
        if (success)
        {
            cout << "Algorithm␣J␣succeeded␣for␣up␣to␣
               S_" << n_max << "(" << pattern << ")"
               << endl;
        }
    }
    return 0;
}
```

Listing A.6: write_avoiders.cpp

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "pattern_avoidance.h"
```

```
using namespace std;
using namespace pattern_avoidance;


void write_avoiders(const vector<string>& patternlist
   , const int n_min, const int n_max, const string
   path)
{
    for (const string& pattern : patternlist)
    {
        const vector<vector<perm_t>> avoiders =
           pattern_avoiding_permutations_sorted(
           pattern, n_max);
        for (int i = n_min; i <= n_max; i++)
        {
            const string out_file = path + "S_" +
               to_string(i) + "(" + pattern + ").txt"
               ;
            ofstream file(out_file);
            for (const perm_t& perm : avoiders[i])
            {
                file << perm_to_str(perm, i) << endl;
            }
        }
    }
}


int main()
{
    const vector<string> patterns =
       append_all_permutations({"1234", "1243"}, 4);
    const string path = "/home/manuel/Documents/
       Semester␣8/Bachelor␣Thesis/code/avoiding␣
       patterns/";
    write_avoiders(patterns, 4, 10, path);
}
```

Listing A.7: write_degree_one_vertices.cpp

```
#include <algorithm>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <string>
```

```
#include <vector>
#include "minimal_jump_graph.h"
#include "pattern_avoidance.h"
using namespace std;
using namespace minimal_jump;
using namespace pattern_avoidance;


void write_degree_one_vertices(const string pattern,
   const int n_min, const int n_max, ostream& out)
{
    const vector<vector<perm_t>> avoiders =
       pattern_avoiding_permutations(pattern, n_max);
    for (int i = n_min; i <= n_max; i++)
    {
        const vector<vector<int64_t>> G =
           compute_adjacency_list<int64_t>(avoiders[i
           ], i);
        const vector<int64_t> degree_one_vertices =
           get_vertices_of_degree(G, (int64_t) 1);
        if (degree_one_vertices.size() > 0)
        {
            vector<perm_t> degree_one_permutations;
            degree_one_permutations.reserve(
               degree_one_vertices.size());
            for (const int64_t& vertex :
               degree_one_vertices)
            {
                degree_one_permutations.push_back(
                   avoiders[i][vertex]);
            }
            sort(degree_one_permutations.begin(),
               degree_one_permutations.end(), [i](
               perm_t lhs, perm_t rhs) { return
               compare_perm(lhs, rhs, i); });
            out << pattern << "␣␣n=" << i << endl <<
               endl;
            for (const perm_t& perm :
               degree_one_permutations)
            {
                out << perm_to_str(perm, i) << endl;
            }
            if (i < n_max)
            {
```

```
                out << endl << endl;
            }
        }
    }
}


void write_degree_one_vertices_count(const vector<
   string>& patternlist, const int n_min, const int
   n_max, ostream& out)
{
    vector<vector<int64_t>> degree_one_vertices_count
       (patternlist.size(), vector<int64_t>(n_max +
       1));
    int64_t max_count = 0;
    for (size_t i = 0; i < patternlist.size(); i++)
    {
        const vector<vector<perm_t>> avoiders =
           pattern_avoiding_permutations(patternlist[
           i], n_max);
        for (int j = n_min; j <= n_max; j++)
        {
            const vector<vector<int64_t>> G =
               compute_adjacency_list<int64_t>(
               avoiders[j], j);
            const int64_t count = get_count_of_degree
               (G, (int64_t) 1);
            degree_one_vertices_count[i][j] = count;
            if (count > max_count)
            {
                max_count = count;
            }
        }
    }
    const int padding_formula = max_element(
       patternlist.begin(), patternlist.end(), [](
       string lhs, string rhs) { return lhs.size() <
       rhs.size(); })->size();
    const int padding_degree = to_string(max_count).
       size();
    out << left << setw(padding_formula + 1) << "F";
    for (int i = n_min; i <= n_max; i++)
    {
        out << setw(padding_degree + 1) << i;
```

```
    }
    out << endl;
    for (size_t i = 0; i < patternlist.size(); i++)
    {
        out << setw(padding_formula + 1) <<
           patternlist[i];
        for (int j = n_min; j <= n_max; j++)
        {
            out << setw(padding_degree + 1) <<
               degree_one_vertices_count[i][j];
        }
        out << endl;
    }
    out << endl;
}


int main()
{
    const vector<string> patterns =
       append_all_permutations({"123", "213"}, 3);
    // const vector<string> patterns =
       append_all_permutations({"1234", "1243",
       "2134", "2314", "3124", "3214"}, 4);
    for (const string& pattern : patterns)
    {
        ofstream file_vertices(pattern + ".txt");
        write_degree_one_vertices(pattern, 4, 10,
           file_vertices);
    }
    return 0;
}
```

Listing A.8: write_even_odd_count.cpp

```
#include <algorithm>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;


typedef struct parity_count_t {
```

```
    int64_t even;
    int64_t odd;
    parity_count_t(int64_t even, int64_t odd)
        : even(even)
        , odd(odd)
    { }
} parity_count_t;


/** Returns true if and only if a permutation of
   digits avoids the increasing
    pattern of length k.
*/
bool avoids_increasing_pattern(const vector<int>&
   digits, const int k)
{
    // contains the largest increasing subsequence up
        to this digit
    vector<int> largest_subsequence(digits.size(), 1)
       ;
    for (size_t i = 0; i < digits.size(); i++)
    {
        for (size_t j = i + 1; j < digits.size(); j
           ++)
        {
            if (digits[i] < digits[j])
            {
                if (largest_subsequence[j] <
                   largest_subsequence[i] + 1)
                {
                    largest_subsequence[j] =
                       largest_subsequence[i] + 1;
                    if (largest_subsequence[j] >= k)
                    {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

```
/** Returns the inversion number of a permutation of
   digits.
*/
int64_t inversion_number(const vector<int>& digits)
{
    int64_t result = 0;
    for (size_t i = 0; i < digits.size(); i++)
    {
        for (size_t j = i + 1; j < digits.size(); j
           ++)
        {
            if (digits[i] > digits[j])
            {
                result++;
            }
        }
    }
    return result;
}


/** Returns the number of even and odd permutations
   of length n that avoid the
    increasing pattern of length k.
*/
parity_count_t count_parity(const int n, const int k)
{
    vector<int> digits(n);
    for (int i = 0; i < n; i++)
    {
        digits[i] = i;
    }
    int64_t even_count = 0;
    int64_t odd_count = 0;
    do {
        if (avoids_increasing_pattern(digits, k))
        {
            if (inversion_number(digits) % 2 == 0)
            {
                even_count += 1;
            }
            else
            {
                odd_count += 1;
```

```
                }
            }
        }
        while (next_permutation(digits.begin(), digits.
            end()));
        return parity_count_t(even_count, odd_count);
    }


    void write_even_odd_difference(const int n_max, const
        int k, ostream& out)
    {
        vector<int64_t> even(n_max + 1), odd(n_max + 1),
            diff(n_max + 1);
        for (int i = k; i <= n_max; i++)
        {
            const parity_count_t parity_count =
                count_parity(i, k);
            even[i] = parity_count.even;
            odd[i] = parity_count.odd;
            diff[i] = abs(parity_count.even -
                parity_count.odd);
        }
        const string title_n = "n";
        const string title_even = "even";
        const string title_odd = "odd";
        const string title_diff = "difference";
        const string title_larger = "larger";
        const int max_n_len = to_string(n_max).length();
        const int max_even_len = to_string(*max_element(
            even.begin(), even.end())).length();
        const int max_odd_len = to_string(*max_element(
            odd.begin(), odd.end())).length();
        const int max_diff_len = to_string(*max_element(
            diff.begin(), diff.end())).length();
        const int padding_n = max(max_n_len, (int)
            title_n.length());
        const int padding_even = max(max_even_len, (int)
            title_even.length());
        const int padding_odd = max(max_odd_len, (int)
            title_odd.length());
        const int padding_diff = max(max_diff_len, (int)
            title_diff.length());
        const int padding_larger = title_larger.length();
```

```
    const int column_dist = 2;
    for (int i = 1; i <= k; i++)
    {
        out << i;
    }
    out << endl;
    out << left;
    out << setw(padding_n + column_dist) << title_n;
    out << setw(padding_even + column_dist) <<
       title_even;
    out << setw(padding_odd + column_dist) <<
       title_odd;
    out << setw(padding_diff + column_dist) <<
       title_diff;
    out << setw(padding_larger + column_dist) <<
       title_larger;
    out << endl;
    for (int i = k; i <= n_max; i++)
    {
        out << setw(padding_n + column_dist) << i;
        out << setw(padding_even + column_dist) <<
           even[i];
        out << setw(padding_odd + column_dist) << odd
           [i];
        out << setw(padding_diff + column_dist) <<
           diff[i];
        if (even[i] - odd[i] > 0)
        {
            out << setw(padding_larger + column_dist)
                << "even";
        }
        else if (even[i] - odd[i] < 0)
        {
            out << setw(padding_larger + column_dist)
                << "odd";
        }
        else
        {
            out << setw(padding_larger + column_dist)
                << "-";
        }
        out << endl;
    }
}
```

```
int main()
{
    // this will run for quite some time
    constexpr int max_permutation_length = 14;
    constexpr int min_pattern_length = 3;
    constexpr int max_pattern_length = 6;
    for (int i = min_pattern_length; i <=
       max_pattern_length; i++)
    {
        write_even_odd_difference(
           max_permutation_length, i, cout);
        cout << endl;
    }
    return 0;
}
```

Listing A.9: write_min_jumps.cpp

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <unordered_set>
#include <vector>
#include "pattern_avoidance.h"
#include "minimal_jump_graph.h"
using namespace std;
using namespace minimal_jump;
using namespace pattern_avoidance;


void write_min_jumps(const string pattern, const int
   n_min, const int n_max, const string path)
{
    const vector<vector<perm_t>> avoiders =
       pattern_avoiding_permutations_sorted(pattern,
       n_max);
    for (int i = n_min; i <= n_max; i++)
    {
        const unordered_set<perm_t>
           avoiders_hashtable(avoiders[i].begin(),
           avoiders[i].end());
```

```
        const string out_file = path + "G_" +
           to_string(i) + "(" + pattern + ").txt";
        ofstream file(out_file);
        for (const perm_t& perm : avoiders[i])
        {
            vector<perm_t> min_jumps =
               compute_min_jumps(perm, i,
               avoiders_hashtable);
            sort(min_jumps.begin(), min_jumps.end(),
               [i](perm_t lhs, perm_t rhs) { return
               compare_perm(lhs, rhs, i); });
            file << perm_to_str(perm, i) << endl <<
               endl;
            for (const perm_t& min_jump : min_jumps)
            {
                file << "\t" << perm_to_str(min_jump,
                    i) << endl;
            }
            file << endl << endl;
        }
    }
}


int main()
{
    const vector<string> patterns =
       append_all_permutations({"2134", "2314", "3124
       ", "3214"}, 4);
    const string path = "/home/manuel/Documents/
       Semester␣8/Bachelor␣Thesis/code/minimal␣jump␣
       graphs/";
    for (const string& pattern : patterns)
    {
        write_min_jumps(pattern, 4, 8, path);
    }
}
```

Listing A.10: write_results.cpp

```
#include <algorithm>
#include <cstdio>     /* fgets, popen, pclose */
#include <cstdlib>    /* mkdtemp */
#include <cstdint>
#include <fstream>
```

```
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
#include "minimal_jump_graph.h"
#include "pattern_avoidance.h"
using namespace std;
using namespace minimal_jump;
using namespace pattern_avoidance;


constexpr char CONCORDE[] = "/etc/concorde/TSP/
   concorde";
constexpr int CONCORDE_MAX_INPUT_SIZE = 7000;


bool run_process(const char* cmd)
{
    FILE *process = popen(cmd, "r");
    char buff[128];
    while (fgets(buff, 128, process))
    {
        cout << buff;
    }
    if (!process || pclose(process) < 0)
    {
        return false;
    }
    return true;
}


/** Returns
     1 if the graph has a Hamiltonian cycle
     0 if the graph has no Hamiltonian cycle
    -1 if temporary directory for concorde files
       could not be created
    -2 if .tsp file for concorde could not be created
    -3 if running concorde fails
    -4 if .sol file from concorde could not be read
*/
template<typename T>
int get_hamiltonian_cycle(const vector<vector<T>>& G,
    const string pattern, const int n, vector<T>&
```

```
   solution)
{
    char temp_dir_template[] = "/tmp/XXXXXX";
    if (!mkdtemp(temp_dir_template))
    {
        return -1;
    }
    const string temp_dir = temp_dir_template;
    const string tsp_file = temp_dir + "/a.tsp";
    const string sol_file = temp_dir + "/a.sol";
    const string concorde(CONCORDE);
    const string cmd = "cd␣" + temp_dir + "␣&&␣" +
       concorde + "␣a.tsp";
    if (!write_tsp_file(G, pattern, n, false,
       tsp_file))
    {
        return -2;
    }
    if (!run_process(cmd.c_str()))
    {
        return -3;
    }
    if (!read_sol_file(sol_file, solution))
    {
        return -4;
    }
    return has_cycle(G, solution);
}


/** Returns
     1 if the graph has a Hamiltonian path
     0 if the graph has no Hamiltonian path
    -1 if temporary directory for concorde files
       could not be created
    -2 if .tsp file for concorde could not be created
    -3 if running concorde fails
    -4 if .sol file from concorde could not be read
*/
template<typename T>
int get_hamiltonian_path(const vector<vector<T>>& G,
   const string pattern, const int n, vector<T>&
   solution)
{
```

```
    char temp_dir_template[] = "/tmp/XXXXXX";
    if (!mkdtemp(temp_dir_template))
    {
        return -1;
    }
    const string temp_dir = temp_dir_template;
    const string tsp_file = temp_dir + "/a.tsp";
    const string sol_file = temp_dir + "/a.sol";
    const string concorde(CONCORDE);
    const string cmd = "cd␣" + temp_dir + "␣&&␣" +
       concorde + "␣a.tsp";
    if (!write_tsp_file(G, pattern, n, true, tsp_file
       ))
    {
        return -2;
    }
    if (!run_process(cmd.c_str()))
    {
        return -3;
    }
    if (!read_sol_file(sol_file, solution))
    {
        return -4;
    }
    const typename vector<T>::iterator aux = find(
       solution.begin(), solution.end(), solution.
       size() - 1);
    reverse(solution.begin(), aux);
    reverse(next(aux), solution.end());
    solution.erase(aux);
    return has_path(G, solution);
}


/** Returns
    1 if the graph has a Hamiltonian cycle
    0 if the graph has no Hamiltonian cycle
    -1 if programm was compiled without concorde
    -2 if temporary directory for concorde files
       could not be created
    -3 if .tsp file for concorde could not be created
    -4 if running concorde fails
    -5 if .sol file from concorde could not be read
*/
```

```
template<typename T>
int check_Hamiltonian_cycle(const vector<vector<T>>&
   G, const string pattern, const int n)
{
    vector<T> solution;
    return get_hamiltonian_cycle(G, pattern, n,
       solution);
}


/** Returns
     1 if the graph has a Hamiltonian path
     0 if the graph has no Hamiltonian path
    -1 if programm was compiled without concorde
    -2 if temporary directory for concorde files
       could not be created
    -3 if .tsp file for concorde could not be created
    -4 if running concorde fails
    -5 if .sol file from concorde could not be read
*/
template<typename T>
int check_Hamiltonian_path(const vector<vector<T>>& G
   , const string pattern, const int n)
{
    vector<T> solution;
    return get_hamiltonian_path(G, pattern, n,
       solution);
}


void write_results(const vector<string>& patternlist,
    const int n_min, const int n_max, ostream& out)
{
    const int pattern_padding = max_element(
       patternlist.begin(), patternlist.end(), [](
       string lhs, string rhs) { return lhs.size() <
       rhs.size(); })->size() + 1;
    out << left << setw(pattern_padding) << "F";
    for (int i = n_min; i <= n_max; i++)
    {
        out << setw(4) << i;
    }
    out << endl;
```

```
for (const string& pattern : patternlist)
{
    out << setw(pattern_padding) << pattern;
    vector<vector<perm_t>> avoiders =
       pattern_avoiding_permutations(pattern,
       n_max);
    for (int i = n_min; i <= n_max; i++)
    {
        cout << "building␣graph␣of␣" << avoiders[
           i].size() << "␣nodes␣for␣" << pattern
           << "␣with␣n=" << i << endl;
        const vector<vector<int64_t>> G =
           compute_adjacency_list<int64_t>(
           avoiders[i], i);
        const bool empty = minimal_jump::is_empty
           (G);
        const bool disconnected = is_disconnected
           (G);
        const int64_t degree_one_vertices_count =
            get_count_of_degree(G, (int64_t) 1);
        // bipartition is undefined for
           disconnected graphs
        const int64_t bipartition_size_difference
            = (disconnected) ? 0 :
           get_bipartition_size_difference(G);
        if (bipartition_size_difference > 1 &&
           degree_one_vertices_count > 2)
        {
            out << setw(4) << "Xbd";
        }
        else if (disconnected &&
           degree_one_vertices_count > 2)
        {
            out << setw(4) << "Xcd";
        }
        else if (bipartition_size_difference > 1)
        {
            out << setw(4) << "Xb";
        }
        else if (disconnected)
        {
            out << setw(4) << "Xc";
        }
        else if (degree_one_vertices_count > 2)
```

```
{
    out << setw(4) << "Xd";
}
else if (empty)
{
    out << setw(4) << "Xe";
}
else
{
    if (G.size() >
       CONCORDE_MAX_INPUT_SIZE)
    {
        out << setw(4) << "?";
    }
    else
    {
        const int cycle =
           check_Hamiltonian_cycle(G,
           pattern, i);
        const int path = (cycle == 1) ? 1
            : check_Hamiltonian_path(G,
           pattern, i);
        if (cycle < 0 || path < 0)
        {
            cerr << "cycle␣return␣code:␣"
                << cycle << ",␣path␣
               return␣code:␣" << path <<
               endl;
            out << setw(4) << "FAIL";
        }
        else if (cycle == 1)
        {
            out << setw(4) << "C";
        }
        else if (path == 1)
        {
            if (degree_one_vertices_count
                > 0 &&
               bipartition_size_difference
                > 0)
            {
                out << setw(4) << "Pbd";
            }
            else if (
```

```
   bipartition_size_difference
    > 0)
{
    out << setw(4) << "Pb";
}
else if (
   degree_one_vertices_count
   > 0)
{
    out << setw(4) << "Pd";
}
else
{
    vector<vector<int64_t>>
       G_optimize = G;
    while (
       prune_degree_two_vertices
       (G_optimize));
    const bool
       pruned_disconnected =
       is_disconnected(
       G_optimize);
    const int64_t
       pruned_degree_one_vertices_count
        = get_count_of_degree
       (G_optimize, (int64_t)
        1);
    // bipartition is
       undefined for
       disconnected graphs
    const int64_t
       pruned_bipartition_size_difference
        = (disconnected) ? 0
       :
       get_bipartition_size_difference
       (G_optimize);
    if (
       pruned_bipartition_size_difference
        > 0 &&
       pruned_degree_one_vertices_count
        > 0)
    {
        out << setw(4) << "
           P2bd";
```

```
                        }
                        else if (
                           pruned_disconnected &&

                           pruned_degree_one_vertices_count
                            > 0)
                        {
                            out << setw(4) << "
                               P2cd";
                        }
                        else if (
                           pruned_bipartition_size_difference
                            > 0)
                        {
                            out << setw(4) << "
                               P2b";
                        }
                        else if (
                           pruned_disconnected)
                        {
                            out << setw(4) << "
                               P2c";
                        }
                        else if (
                           pruned_degree_one_vertices_count
                            > 0)
                        {
                            out << setw(4) << "
                               P2d";
                        }
                        else
                        {
                            out << setw(4) << "P?
                               ";
                        }
                    }
                }
                else
                {
                    out << setw(4) << "X?";
                }
            }
        }
    }
```

```
            out << endl;
        }
    }


    int main()
    {
        const vector<string> patterns =
           append_all_permutations({"1234", "1243", "2134
           ", "2314", "3124", "3214"}, 4);
        ofstream out_file("hamiltonicity.txt");
        write_results(patterns, 4, 12, out_file);
        return 0;
    }
```

Bibliography

[1] David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver. http://www.math.uwaterloo.ca/tsp/concorde.html, 2003.

[2] Samuel Connolly, Zachary Gabor, and Anant Godbole. The location of the first ascent in a 123-avoiding permutation. *Integers*, 15:Paper No. A13, 8, 2015.

[3] P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Compositio Math.*, 2:463–470, 1935.

[4] Elizabeth Hartung, Hung Phuc Hoang, Torsten Mütze, and Aaron Williams. Combinatorial generation via permutation languages. i. fundamentals, 2019.

[5] OEIS Foundation Inc. The on-line encyclopedia of integer sequences. https://oeis.org/A246138, 2020.

[6] William Kuszmaul. Fast algorithms for finding pattern avoiders and counting pattern occurrences in permutations. *Math. Comp.*, 87(310):987–1011, 2018.

[7] Julian West. *Permutations with forbidden subsequences and stack-sortable permutations*. ProQuest LLC, Ann Arbor, MI, 1990. Thesis (Ph.D.)–Massachusetts Institute of Technology.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Hamiltonicity of minimal jump graphs

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Nowack	Manuel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 10. July 2020	M. Nowack

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.