

Solving hard problems effectively on permutations of small grid-width

Michal Opler
Joint work with Vít Jelínek

Computer Science Institute of Charles University in Prague

Permutation Patterns
June 2019

Permutation-related computational problems

Permutation-related computational problems

PERMUTATION PATTERN MATCHING (PPM)

Input: Permutations π of size k and τ of size n .

Question: Is π contained in τ ?

Permutation-related computational problems

PERMUTATION PATTERN MATCHING (PPM)

Input: Permutations π of size k and τ of size n .

Question: Is π contained in τ ?

LONGEST COMMON PATTERN (LCP)

Input: Permutations π of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Permutation-related computational problems

PERMUTATION PATTERN MATCHING (PPM)

Input: Permutations π of size k and τ of size n .

Question: Is π contained in τ ?

LONGEST COMMON PATTERN (LCP)

Input: Permutations π of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Observation

PPM can easily be reduced to LCP.

Permutation-related computational problems

PERMUTATION PATTERN MATCHING (PPM)

Input: Permutations π of size k and τ of size n .

Question: Is π contained in τ ?

LONGEST COMMON PATTERN (LCP)

Input: Permutations π of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Observation

PPM can easily be reduced to LCP.

Theorem

Both PPM and LCP are NP-complete.

Permutation-related computational problems

Question

What happens if π comes from some restricted permutation class $\mathcal{C}$? For which classes $\mathcal{C}$ can we solve PPM and LCP efficiently?

Permutation-related computational problems

Question

What happens if π comes from some restricted permutation class $\mathcal{C}$? For which classes $\mathcal{C}$ can we solve PPM and LCP efficiently?

$\mathcal{C}$ -PATTERN PERMUTATION PATTERN MATCHING ($\mathcal{C}$ -PPM)

Input: Permutations $\pi \in \mathcal{C}$ of size k and τ of size n .

Question: Is π contained in τ ?

$\mathcal{C}$ -PATTERN LONGEST COMMON PATTERN ($\mathcal{C}$ -LCP)

Input: Permutations $\pi \in \mathcal{C}$ of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Permutation-related computational problems

Question

What happens if π comes from some restricted permutation class $\mathcal{C}$? For which classes $\mathcal{C}$ can we solve PPM and LCP efficiently?

$\mathcal{C}$ -PATTERN PERMUTATION PATTERN MATCHING ($\mathcal{C}$ -PPM)

Input: Permutations $\pi \in \mathcal{C}$ of size k and τ of size n .

Question: Is π contained in τ ?

$\mathcal{C}$ -PATTERN LONGEST COMMON PATTERN ($\mathcal{C}$ -LCP)

Input: Permutations $\pi \in \mathcal{C}$ of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Example

Av(21)-LCP can be decided in $O(n \log n)$ time.

Permutation-related computational problems

Question

What happens if π comes from some restricted permutation class $\mathcal{C}$? For which classes $\mathcal{C}$ can we solve PPM and LCP efficiently?

$\mathcal{C}$ -PATTERN PERMUTATION PATTERN MATCHING ($\mathcal{C}$ -PPM)

Input: Permutations $\pi \in \mathcal{C}$ of size k and τ of size n .

Question: Is π contained in τ ?

$\mathcal{C}$ -PATTERN LONGEST COMMON PATTERN ($\mathcal{C}$ -LCP)

Input: Permutations $\pi \in \mathcal{C}$ of size k , τ of size n and integer t .

Question: Is there a pattern σ of size t contained in both π and τ ?

Example

Av(21)-LCP can be decided in $O(n \log n)$ time.

Theorem (Jelínek, Kynčl¹)

Av(321)-PPM is NP-complete.

¹V. Jelínek and J. Kynčl. Hardness of permutation pattern matching. SODA '17.

Restricted classes - outline

1. Separable permutations²
2. Finitely many simple permutations²
3. Bounded grid-width

²D. Rossin, M. Bouvel. The Longest Common Pattern Problem for two Permutations. Pure Mathematics and Applications, 2006.

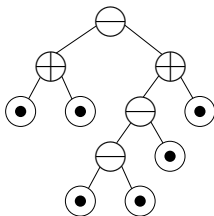
Separable permutations

Definition

A permutation π is *separable* if it can be obtained by direct and skew sums of singleton permutations.

Definition

A *binary separating tree* is a binary ordered tree such that each internal vertex is labeled by $\oplus$ or $\ominus$. Each binary separating tree is naturally associated with a unique separable permutation.



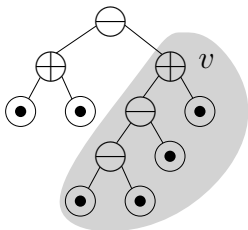
Separable permutations

Definition

A permutation π is *separable* if it can be obtained by direct and skew sums of singleton permutations.

Definition

A *binary separating tree* is a binary ordered tree such that each internal vertex is labeled by $\oplus$ or $\ominus$. Each binary separating tree is naturally associated with a unique separable permutation.

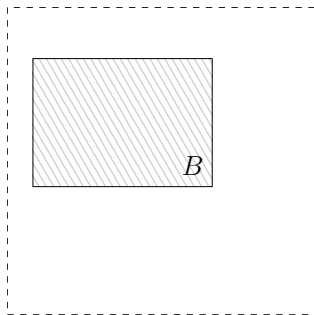


For a vertex v of the tree, we use π_v to denote the permutation associated to the subtree rooted in v .

Separable permutations

Definition

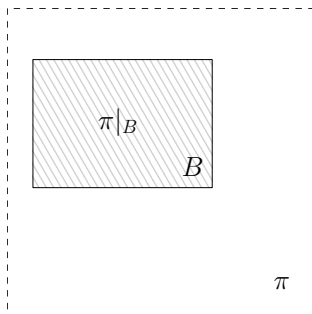
A *box* is the Cartesian product of two integer intervals.



Separable permutations

Definition

A *box* is the Cartesian product of two integer intervals.



For a box B , we use $\pi|_B$ to denote the subpermutation of π corresponding to the points of permutation diagram that lie inside B .

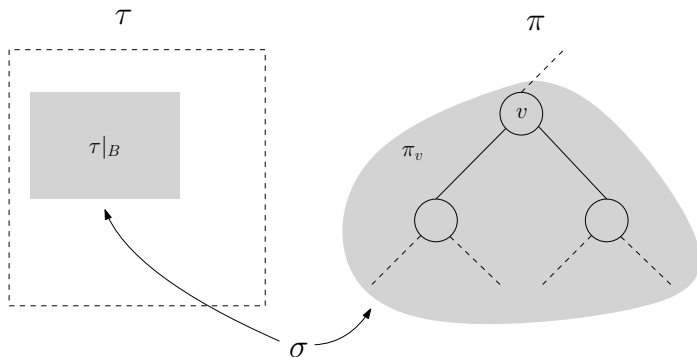
Algorithm 1

Dynamic programming with the following procedure:

LARGESTSUBPATTERN(v , B)

Input: Vertex v of the separating tree of π , and a box B .

Output: Size of the largest σ contained in both π_v and $\tau|_B$.



Algorithm 1

Case 1: v is a leaf

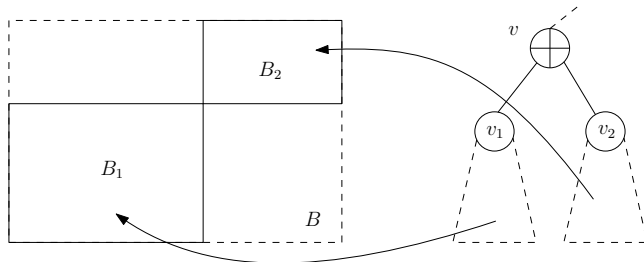
$$\text{LARGESTSUBPATTERN}(\bullet, B) = \begin{cases} 1 & \text{if } \tau|_B \text{ is non-empty} \\ 0 & \text{otherwise.} \end{cases}$$

Algorithm 1

Case 1: v is a leaf

$$\text{LARGESTSUBPATTERN}(\bullet, B) = \begin{cases} 1 & \text{if } \tau|_B \text{ is non-empty} \\ 0 & \text{otherwise.} \end{cases}$$

Case 2: v is a $\oplus$ -node with children v_1 and v_2



Try all such possible splittings of B into B_1 and B_2 and return

$$\max_{B_1, B_2} (\text{LARGESTSUBPATTERN}(v_1, B_1) + \text{LARGESTSUBPATTERN}(v_2, B_2))$$

Algorithm 1

Time complexity

We see that

- ▶ there is $O(k)$ vertices in the separating tree of π ,
- ▶ $O(n^4)$ possible boxes B , and
- ▶ we spend $O(n^2)$ time inside each call of LARGESTSUBPATTERN trying possible splittings of B into B_1 and B_2 .

Algorithm 1

Time complexity

We see that

- ▶ there is $O(k)$ vertices in the separating tree of π ,
- ▶ $O(n^4)$ possible boxes B , and
- ▶ we spend $O(n^2)$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of B into B_1 and B_2 .

Theorem (Bouvel, Rossin)

We can decide Sep-LCP in time $O(kn^6)$.

Finitely many simple permutations

Definition

A *decomposition tree* is an ordered tree such that each internal vertex

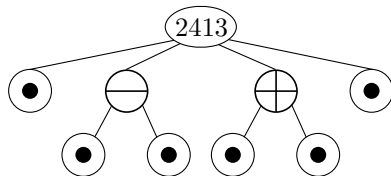
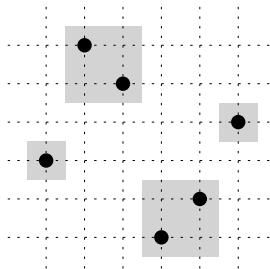
- ▶ is labeled by $\oplus$ or $\ominus$ and has 2 children or
- ▶ is labeled by a simple permutation of size d and has d children.

Finitely many simple permutations

Definition

A *decomposition tree* is an ordered tree such that each internal vertex

- ▶ is labeled by $\oplus$ or $\ominus$ and has 2 children or
- ▶ is labeled by a simple permutation of size d and has d children.



Algorithm 2

LARGESTSUBPATTERN(v, B)

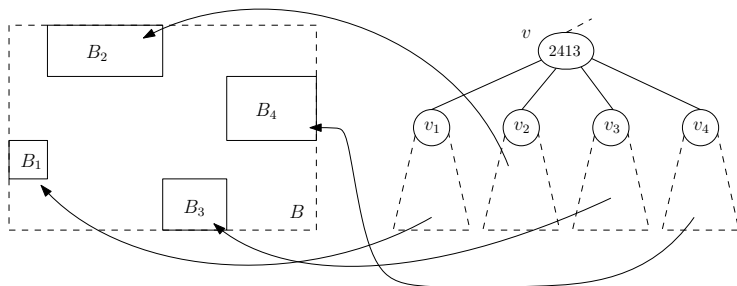
$\vdots$

Algorithm 2

LARGESTSUBPATTERN(v , B)

$\vdots$

Case 3: v is labeled by a simple permutation with children $v_1, \dots, v_d$



Try all such possible splittings of B into $B_1, \dots, B_d$ and return

$$\max_{B_1, \dots, B_d} \left(\sum_{i=1}^d \text{LARGESTSUBPATTERN}(v_i, B_i) \right)$$

Algorithm 2

Time complexity

For π whose decomposition tree has maximum arity d ,

- ▶ there is $O(k)$ vertices in the decomposition tree of π ,
- ▶ $O(n^4)$ possible boxes B , and

Algorithm 2

Time complexity

For π whose decomposition tree has maximum arity d ,

- ▶ there is $O(k)$ vertices in the decomposition tree of π ,
- ▶ $O(n^4)$ possible boxes B , and
- ▶ we spend $O(n^{2d-2})$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of B .

Algorithm 2

Time complexity

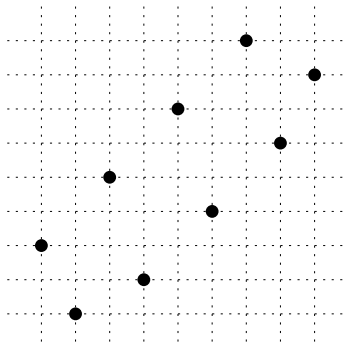
For π whose decomposition tree has maximum arity d ,

- ▶ there is $O(k)$ vertices in the decomposition tree of π ,
- ▶ $O(n^4)$ possible boxes B , and
- ▶ we spend $O(n^{2d-2})$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of B .

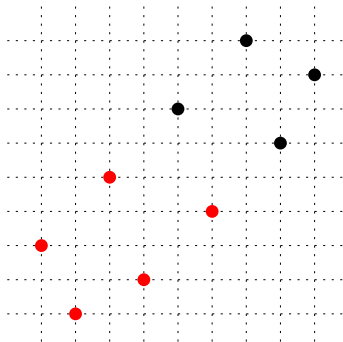
Theorem (Bouvel, Rossin)

Let $\mathcal{C}$ be a permutation class with finitely many simples. We can decide $\mathcal{C}$ -LCP in time $O(kn^{2d+2})$ where d is the size of the largest simple in $\mathcal{C}$.

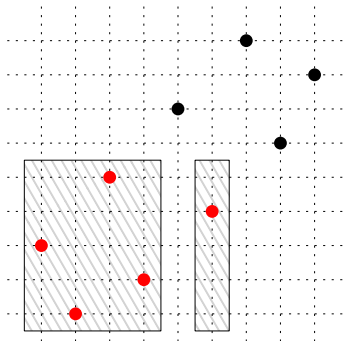
Almost intervals



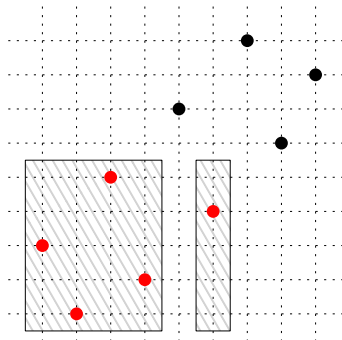
Almost intervals



Almost intervals



Almost intervals



Definition

- ▶ The *intervalicity* of a set $A \subseteq [n]$ is the size of the smallest integer interval family whose union is equal to A .
- ▶ The *grid-width* of a subpattern of π is the maximum of the intervalicities of its index set and its value set.

Grid trees

Definition

- ▶ The *grid tree* of a permutation π is a rooted ordered binary tree T with leaves corresponding to the points of the permutation diagram.
- ▶ The *grid-width* of T is the maximum grid-width of π_v over all vertices v .
- ▶ The *grid-width* of π , denoted by $\text{gw}(\pi)$, is the minimum grid-width of T over all possible grid trees T of π .

Grid tree - example

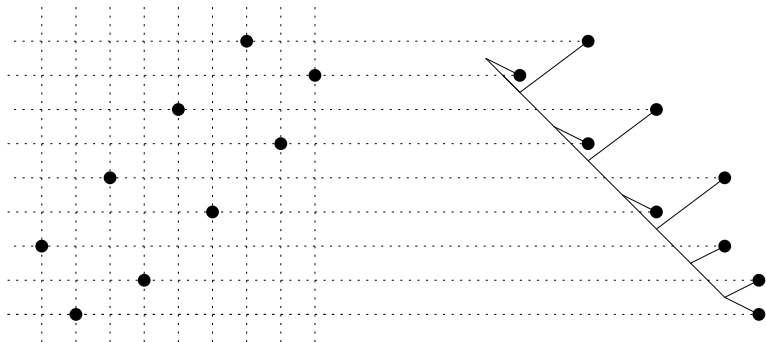
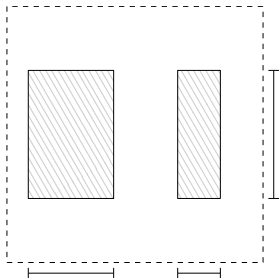


Figure: Increasing oscillation of size 9 and its grid tree of grid-width 2.

Subgrids

Definition

The (k, l) -subgrid is a Cartesian product of $A, B \subseteq [n]$ such that the intervalicity of A is k and the intervalicity of B is l .



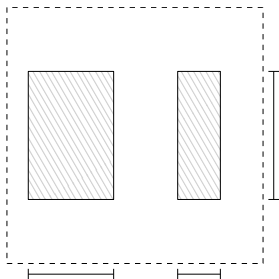
Observation

The $(1, 1)$ -subgrids are precisely the boxes.

Subgrids

Definition

The (k, l) -subgrid is a Cartesian product of $A, B \subseteq [n]$ such that the intervalicity of A is k and the intervalicity of B is l .



Observation

The $(1, 1)$ -subgrids are precisely the boxes.

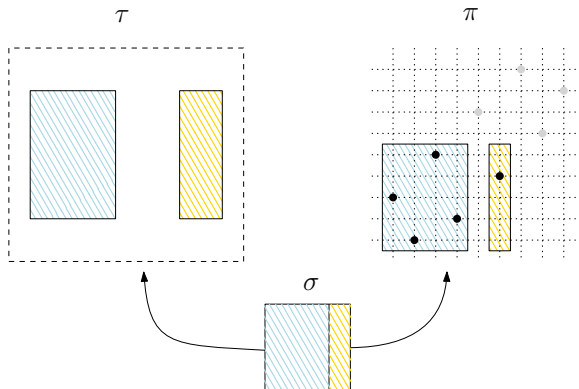
For a subgrid G , we use $\pi|_G$ to denote the **gridded** subpermutation of π corresponding to the points of permutation diagram that lie inside G .

Algorithm 3

LARGESTSUBPATTERN(v , G)

Input: Vertex v of the grid tree of π and a subgrid G of appropriate dimensions.

Output: Size of the largest gridded σ contained in both π_v and $\tau|_G$.



Algorithm 3

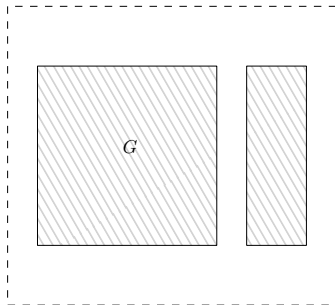
Case 1: v is a leaf ...

Algorithm 3

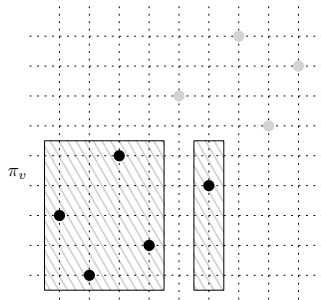
Case 1: v is a leaf ...

Case 2:

$\mathcal{T}$



π

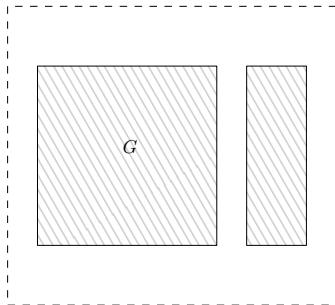


Algorithm 3

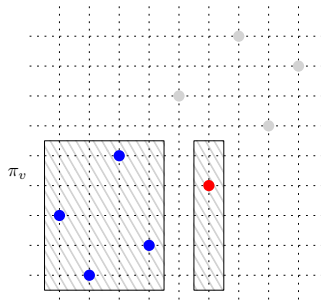
Case 1: v is a leaf ...

Case 2:

$\mathcal{T}$



π

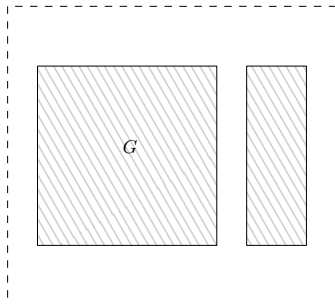


Algorithm 3

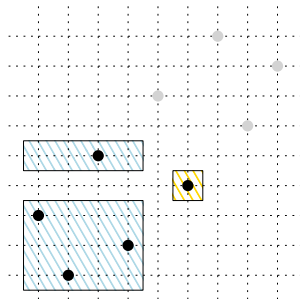
Case 1: v is a leaf ...

Case 2:

$\mathcal{T}$



π

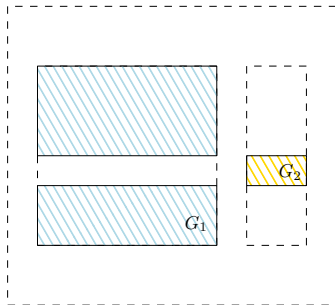


Algorithm 3

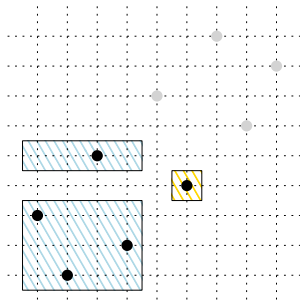
Case 1: v is a leaf ...

Case 2:

τ



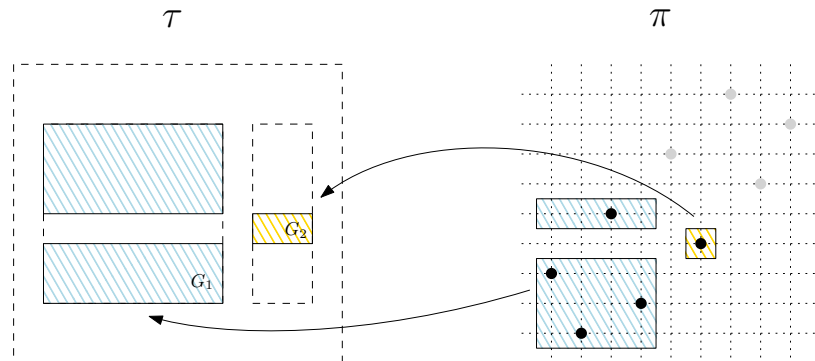
π



Algorithm 3

Case 1: v is a leaf ...

Case 2:



Try all such possible splittings of G into G_1 , G_2 and return

$$\max_{G_1, G_2} (\text{LARGESTSUBPATTERN}(v_1, G_1) + \text{LARGESTSUBPATTERN}(v_2, G_2))$$

Algorithm 3

Time complexity

For π with grid-width g

- ▶ there is $O(k)$ vertices in the grid tree of π ,
- ▶ $O(n^{4g})$ possible subgrids G , and

Algorithm 3

Time complexity

For π with grid-width g

- ▶ there is $O(k)$ vertices in the grid tree of π ,
- ▶ $O(n^{4g})$ possible subgrids G , and
- ▶ we spend $O(n^{4g-2})$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of G .

Algorithm 3

Time complexity

For π with grid-width g

- ▶ there is $O(k)$ vertices in the grid tree of π ,
- ▶ $O(n^{4g})$ possible subgrids G , and
- ▶ we spend $O(n^{4g-2})$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of G .

Theorem

We can decide $\mathcal{C}$ -LCP in time $O(kn^{O(g)})$ for a permutation class $\mathcal{C}$ with grid-width bounded by g .

Algorithm 3

Time complexity

For π with grid-width g

- ▶ there is $O(k)$ vertices in the grid tree of π ,
- ▶ $O(n^{4g})$ possible subgrids G , and
- ▶ we spend $O(n^{4g-2})$ time inside each call of `LARGESTSUBPATTERN` trying possible splittings of G .

Theorem

We can decide $\mathcal{C}$ -LCP in time $O(kn^{O(g)})$ for a permutation class $\mathcal{C}$ with grid-width bounded by g .

Theorem

PPM is $W[1]$ -hard when parametrized by grid-width of π , i.e. under standard parametrized complexity hardness assumptions there is no algorithm for solving PPM in time $O(f(\text{gw}(\pi))n^{O(1)})$ for any computable function f .

Grid-width

- ▶ A permutation π has grid-width equal to 1 if and only if it is separable.

Grid-width

- ▶ A permutation π has grid-width equal to 1 if and only if it is separable.
- ▶ Grid-width measures how far is permutation from being a separable.

Grid-width

- ▶ A permutation π has grid-width equal to 1 if and only if it is separable.
- ▶ Grid-width measures how far is permutation from being a separable.
- ▶ Grid-width of π is equal to the maximum grid-width over all simple permutations contained in π .

Grid-width

- ▶ A permutation π has grid-width equal to 1 if and only if it is separable.
- ▶ Grid-width measures how far is permutation from being a separable.
- ▶ Grid-width of π is equal to the maximum grid-width over all simple permutations contained in π .
- ▶ Up to a constant multiplicative factor equivalent to tree-complexity introduced by Ahal and Rabinovich³.

³S. Ahal and Y. Rabinovich. On complexity of the subpattern problem. SIAM J. Discrete Math., 2008.

Grid-width

- ▶ A permutation π has grid-width equal to 1 if and only if it is separable.
- ▶ Grid-width measures how far is permutation from being a separable.
- ▶ Grid-width of π is equal to the maximum grid-width over all simple permutations contained in π .
- ▶ Up to a constant multiplicative factor equivalent to tree-complexity introduced by Ahal and Rabinovich³.

Theorem (Ahal, Rabinovich)

$\mathcal{C}$ -PPM can be solved in polynomial time for any permutation class $\mathcal{C}$ with bounded grid-width.

³S. Ahal and Y. Rabinovich. On complexity of the subpattern problem. SIAM J. Discrete Math., 2008.

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

We don't know.

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

We don't know. In general we can get an approximation.

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

We don't know. In general we can get an approximation.

Definition

For a permutation $\pi \in \mathcal{S}_n$, its *adjacency graph* G_π is defined as follows. The set of vertices of G_π is the set $\{(i, \pi_i); i \in [n]\}$. Two vertices $(x_1, y_1), (x_2, y_2)$ are adjacent if $|x_1 - x_2| = 1$ or $|y_1 - y_2| = 1$.

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

We don't know. In general we can get an approximation.

Definition

For a permutation $\pi \in \mathcal{S}_n$, its *adjacency graph* G_π is defined as follows. The set of vertices of G_π is the set $\{(i, \pi_i); i \in [n]\}$. Two vertices $(x_1, y_1), (x_2, y_2)$ are adjacent if $|x_1 - x_2| = 1$ or $|y_1 - y_2| = 1$.

Proposition

*For any permutation $\pi \in \mathcal{S}_n$, $\frac{1}{8}\text{tw}(G_\pi) \leq \text{gw}(\pi) \leq \text{tw}(G_\pi) + 2$.
Moreover, given a tree decomposition of width k for G_π , we can compute a grid tree of π of grid-width at most $k + 2$ in linear time.*

Computing grid-width

Can we obtain an optimal grid-tree in polynomial time?

We don't know. In general we can get an approximation.

Definition

For a permutation $\pi \in \mathcal{S}_n$, its *adjacency graph* G_π is defined as follows. The set of vertices of G_π is the set $\{(i, \pi_i); i \in [n]\}$. Two vertices $(x_1, y_1), (x_2, y_2)$ are adjacent if $|x_1 - x_2| = 1$ or $|y_1 - y_2| = 1$.

Proposition

For any permutation $\pi \in \mathcal{S}_n$, $\frac{1}{8}\text{tw}(G_\pi) \leq \text{gw}(\pi) \leq \text{tw}(G_\pi) + 2$. Moreover, given a tree decomposition of width k for G_π , we can compute a grid tree of π of grid-width at most $k + 2$ in linear time.

Corollary

We can compute a grid tree of permutation π with grid-width at most $5\text{gw}(\pi) + O(1)$ in time $2^{O(\text{gw}(\pi))}n$.

Moreover, in many cases the structure of the class provides us optimal grid trees for free.

Grid-width of various classes

Permutation class	Grid-width
-------------------	------------

Grid-width of various classes

Permutation class	Grid-width
separable	1

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2
Widdershin permutations	2

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2
Widdershin permutations	2
acyclic $k \times k$ monotone grid class	$\leq k$

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2
Widdershin permutations	2
acyclic $k \times k$ monotone grid class	$\leq k$
finitely many simples	$\leq$ size of the largest simple

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2
Widdershin permutations	2
acyclic $k \times k$ monotone grid class	$\leq k$
finitely many simples	$\leq$ size of the largest simple
$Av(123)$	∞
skew-merged	∞

Grid-width of various classes

Permutation class	Grid-width
separable	1
$Av(132)$	1
infinite oscillations	2
Widdershin permutations	2
acyclic $k \times k$ monotone grid class	$\leq k$
finitely many simples	$\leq$ size of the largest simple
$Av(123)$	∞
skew-merged	∞
cyclic monotone grid classes	$\infty(?)$

Open questions

- ▶ Can we compute the grid-width of a given permutation exactly in polynomial time or is it NP-hard?
- ▶ What other computational problems can be solved efficiently for classes with bounded grid-width?
- ▶ Is $\mathcal{C}$ -PPM NP-hard for every $\mathcal{C}$ with unbounded grid-width?
- ▶ Does the property of having bounded grid-width translate into interesting algebraic or enumeration properties of the class?

Thank you!