Minimum Mosaic Inference of a Set of Recombinants

Guillaume Blin[1] Florian Sikora[1] Romeo Rizzi[2]
Stéphane Vialette[1]

[1]LIGM, Université Paris-Est Marne-la-Vallée - France

[2]DIMI, Università di Udine - Italy

Minimum Mosaic Inference of a Set of Recombinants

Guillaume Blin[1] Florian Sikora[1] Romeo Rizzi[2]
Stéphane Vialette[1]

[1]LIGM, Université Paris-Est Marne-la-Vallée - France

[2]DIMI, Università di Udine - Italy

Minimum Mosaic Inference of a Set of Recombinants

Guillaume Blin[1] Florian Sikora[1] Romeo Rizzi[2]
Stéphane Vialette[1]

[1]LIGM, Université Paris-Est Marne-la-Vallée - France

[2]DIMI, Università di Udine - Italy

Minimum Mosaic Inference of a Set of Recombinants

Guillaume Blin[1] Florian Sikora[1] Romeo Rizzi[2]
Stéphane Vialette[1]

[1]LIGM, Université Paris-Est Marne-la-Vallée - France

[2]DIMI, Università di Udine - Italy

Outline

Introduction

NP-Hardness

Exact Algorithms

Conclusion

Outline

Introduction

NP-Hardness

Exact Algorithms

Conclusion

SNPs

- SNP (Single Nucleotide Polymorphism)
- When a single nucleotide (A,C,G,T) differs in the genome of two members of a specie (or paired chromosome in a individual)

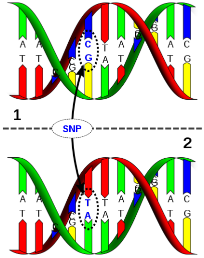


Figure by D. Hall

SNPs

- SNP (Single Nucleotide Polymorphism)
- When a single nucleotide (A,C,G,T) differs in the genome of two members of a specie (or paired chromosome in a individual)
- Represents 90% of the human genetic variation
- Must cheaper to collect than full sequence data

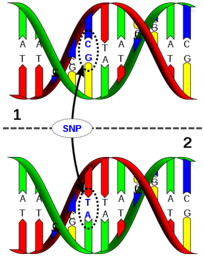


Figure by D. Hall

SNPs

- In most SNPs, **two** (of four) different nucleotides occurs

SNPs

- In most SNPs, **two** (of four) different nucleotides occurs
- Can use **0** and **1** – **binary** data

Recombination

- Principal process inducing these genetic variations

Recombination

- Principal process inducing these genetic variations
- Two equal length sequences...

110001111111001

000110000001111

Recombination

- Principal process inducing these genetic variations
- Two equal length sequences...
- ...generates a third of same length

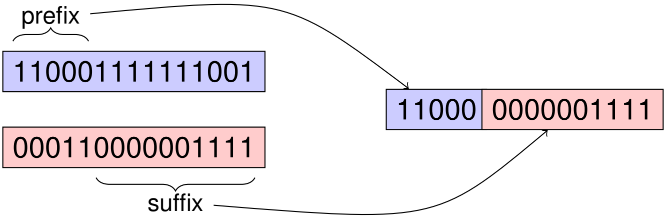
110001111111001

000110000001111

11000 0000001111

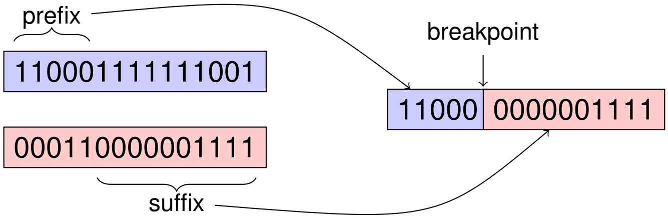
Recombination

- Principal process inducing these genetic variations
- Two equal length sequences...
- ...generates a third of same length
- Concatenation of a prefix in the first one and a suffix in the second one [KOIVISTO ET AL. 04]



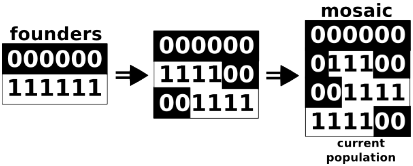
Recombination

- Principal process inducing these genetic variations
- Two equal length sequences...
- ...generates a third of same length
- Concatenation of a prefix in the first one and a suffix in the second one [Koivisto et al. 04]



Founders sequences

- Current sequences are descendant of a small number of **founders sequences**
- A current sequence is composed of **blocks from the founders**, due to recombination



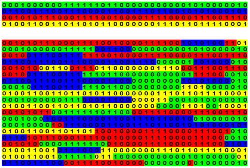
Mosaic

- Really look like a mosaic !

Generated by RecBlock

Mosaic Problem [UKKONEN 02]

- **Input :** A set of m **sequences** (current population) of length n, an integer K
- **Output :** A set of K **founders sequences** that induce a **minimum number of breakpoints**



State of the art

- Polynomial in $\mathcal{O}(mn)$ if $K = 2$ [Ukkonen 02, Wu et al. 07]
- Exact exponentials algorithms [Ukkonen 02, Wu et al. 07]
- Heuristics [Roly & Blum 09]
- Lower bounds on the minimum number of breakpoints needed [Wu 10]

State of the art

- Polynomial in $\mathcal{O}(mn)$ if $K = 2$ [Ukkonen 02, Wu et al. 07]
- Exact exponentials algorithms [Ukkonen 02, Wu et al. 07]
- Heuristics [Roly & Blum 09]
- Lower bounds on the minimum number of breakpoints needed [Wu 10]
- What about the complexity if $K > 2$?

Outline

Introduction

NP-Hardness

Exact Algorithms

Conclusion

Hardness

- A first step in a answer : the problem is NP-Complete if the number of founders is not bounded
- Just some tricks for the proof...

Tool 1: Using arbitrary string

- If the problem on **arbitrary strings** is NP-hard, then so is the problem on binary strings

Tool 1: Using arbitrary string

- If the problem on **arbitrary strings** is NP-hard, then so is the problem on binary strings
  - Suppose an alphabet Σ
- Example
  - $\Sigma = A, B, C$

Tool 1: Using arbitrary string

- If the problem on **arbitrary strings** is NP-hard, then so is the problem on binary strings
  - Suppose an alphabet Σ
  - Take any encoding δ of symbols in Σ by binary strings of length $\lceil \log_2 |\Sigma| \rceil$
- Example
  - $\Sigma = A, B, C$
  - $\delta(A) = 00$
  - $\delta(B) = 01$
  - $\delta(C) = 10$

Tool 1: Using arbitrary string

- ($\Rightarrow$) Any solution with strings over Σ maps into a solution for binary strings without changing the number of breakpoints

- Example
  - $\Sigma = A, B, C$
  - $\delta(A) = 00$
  - $\delta(B) = 01$
  - $\delta(C) = 10$

Tool 1: Using arbitrary string

- ($\Rightarrow$) Any solution with strings over Σ maps into a solution for binary strings without changing the number of breakpoints
- ($\Leftarrow$) If we cannot map the binary founders sequence to symbols of Σ, then we can replace the missing ”word” by its longest suffix in common in Σ without increasing the cost

- Example
  - $\Sigma = A, B, C$
  - $\delta(A) = 00$
  - $\delta(B) = 01$
  - $\delta(C) = 10$

Tool 2: Forcing Founders

- One can force $K' < K$ founders to be part of the solution
- **Add *nm* copies of each forced founders** in the input

Tool 2: Forcing Founders

- One can force $K' < K$ founders to be part of the solution
- **Add *nm* copies of each forced founders** in the input
- If the ”forced founder” is not in the solution founders:

Tool 2: Forcing Founders

- One can force $K' < K$ founders to be part of the solution
- **Add *nm* copies of each forced founders** in the input
- If the ”forced founder” is not in the solution founders:
  - Induce at least 1 breakpoint for one sequence

Tool 2: Forcing Founders

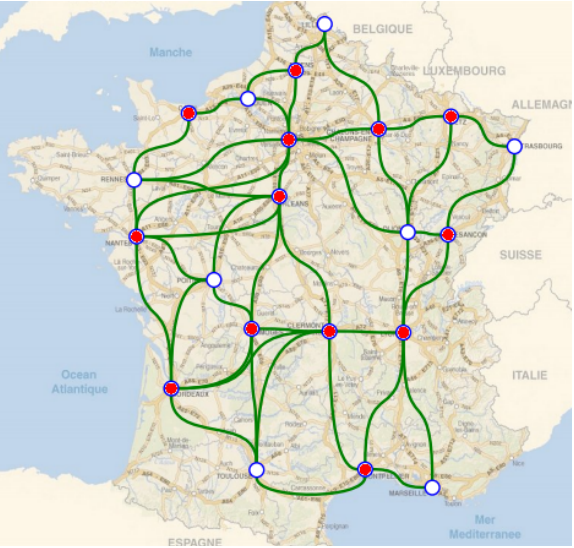
- One can force $K' < K$ founders to be part of the solution
- **Add *nm* copies of each forced founders** in the input
- If the ”forced founder” is not in the solution founders:
  - Induce at least 1 breakpoint for one sequence
  - Therefore induce *nm* breakpoints on the whole...

Proof idea

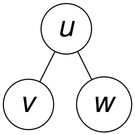
- From the NP-Complete problem VERTEX COVER

Vertex Cover

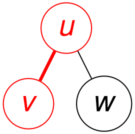
Vertex Cover



Reduction idea



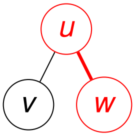
Reduction idea



Input :

$$\underbrace{\begin{matrix} ZZX_uX_uZZZZX_uX_uZZ \\ ZZX_vX_vZZZZX_wX_wZZ \end{matrix}}_{6.|E|}$$

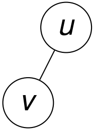
Reduction idea



Input :

$$\underbrace{\begin{array}{l} ZZX_uX_uZZZZX_uX_uZZ \\ ZZX_vX_vZZZZX_wX_wZZ \end{array}}_{6.|E|}$$

Reduction idea



Input :

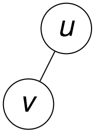
$Z\ Z\ X_u X_u\ Z\ Z$

$Z\ Z\ X_v X_v\ Z\ Z$

$X_u X_u X_u X_u X_u X_u (\times 6.|E| + 1 = 7)$

$X_v X_v X_v X_v X_v X_v \qquad (\times 7)$

Reduction idea



Input :

$Z\ Z\ X_u X_u\ Z\ Z$

$Z\ Z\ X_v X_v\ Z\ Z$

$X_u X_u X_u X_u X_u X_u\ (\times 7)$

$X_v X_v X_v X_v X_v X_v\ (\times 7)$

Forced founders :

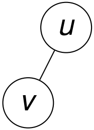
$X_u X_u X_u\ Z\ Z\ Z$

$Z\ Z\ Z\ X_u X_u X_u$

$X_v X_v X_v\ Z\ Z\ Z$

$Z\ Z\ Z\ X_v X_v X_v$

Reduction idea



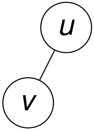
Input :

Z	Z	X_u	X_u	Z	Z	
Z	Z	X_v	X_v	Z	Z	
X_u	X_u	X_u	X_u	X_u	X_u	$(\times 7)$
X_v	X_v	X_v	X_v	X_v	X_v	$(\times 7)$

Forced founders :

X_u	X_u	X_u	Z	Z	Z
Z	Z	Z	X_u	X_u	X_u
X_v	X_v	X_v	Z	Z	Z
Z	Z	Z	X_v	X_v	X_v

Reduction idea



Input :

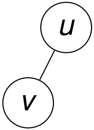
Z	Z	X_u	X_u	Z	Z	
Z	Z	X_v	X_v	Z	Z	
X_u	X_u	X_u	X_u	X_u	X_u	$(\times 7)$
X_v	X_v	X_v	X_v	X_v	X_v	$(\times 7)$

Forced founders :

X_u	X_u	X_u	Z	Z	Z
Z	Z	Z	X_u	X_u	X_u
X_v	X_v	X_v	Z	Z	Z
Z	Z	Z	X_v	X_v	X_v

- Remains $|$Vertex Cover$|$ founders (here 1)

Reduction idea



Input :

Z	Z	X_u	X_u	Z	Z	
Z	Z	X_v	X_v	Z	Z	
X_u	X_u	X_u	X_u	X_u	X_u	$(\times 7)$
X_v	X_v	X_v	X_v	X_v	X_v	$(\times 7)$

Forced founders :

X_u	X_u	X_u	Z	Z	Z
Z	Z	Z	X_u	X_u	X_u
X_v	X_v	X_v	Z	Z	Z
Z	Z	Z	X_v	X_v	X_v

- Remains |Vertex Cover| founders (here 1)
- Will be sequences "$X_i X_i \ldots$" due to $(\times 7)$
- It is a vertex cover otherwise first sequences generate more breakpoints

Outline

Introduction

NP-Hardness

Exact Algorithms

Conclusion

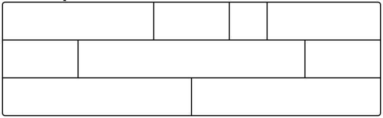
Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**

Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**

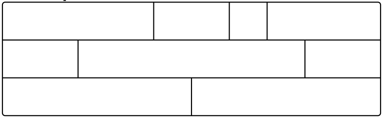
Input :



Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**

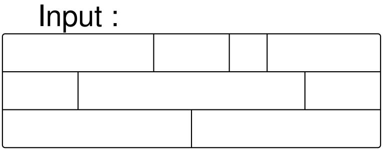
Input :



- Each substring without breakpoints must by definition appears in the solution

Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**

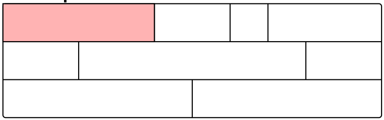


- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

Polynomial-time Algorithm

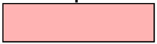
- Suppose one **knows where the breakpoints are**

Input :



- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

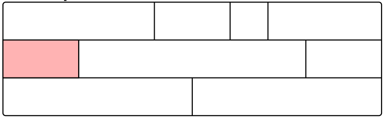
Output :



Polynomial-time Algorithm

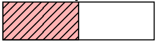
- Suppose one **knows where the breakpoints are**

Input :



- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

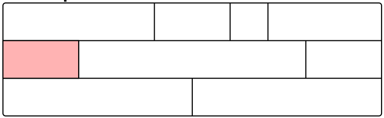
Output :



Polynomial-time Algorithm

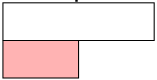
- Suppose one **knows where the breakpoints are**

Input :



- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

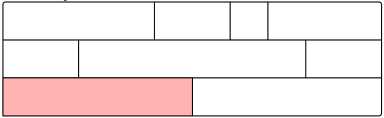
Output :



Polynomial-time Algorithm

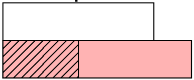
- Suppose one **knows where the breakpoints are**

Input :



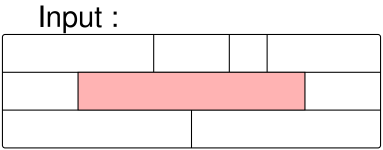
- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

Output :

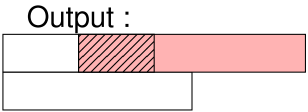


Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**



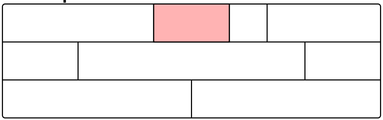
- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output



Polynomial-time Algorithm

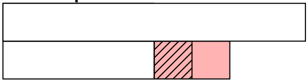
- Suppose one **knows where the breakpoints are**

Input :



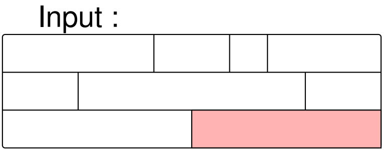
- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

Output :

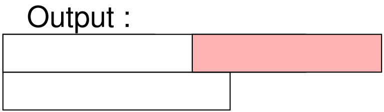


Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**

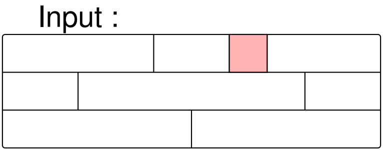


- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

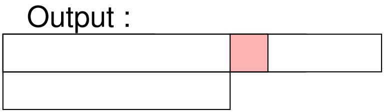


Polynomial-time Algorithm

- Suppose one **knows where the breakpoints are**



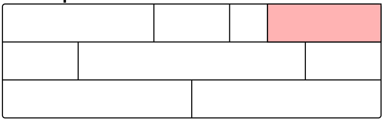
- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output



Polynomial-time Algorithm

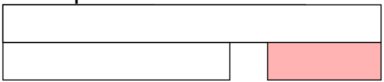
- Suppose one **knows where the breakpoints are**

Input :



- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output

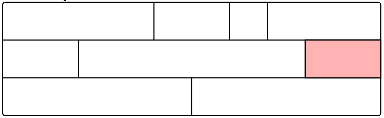
Output :



Polynomial-time Algorithm

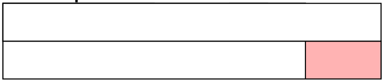
- Suppose one **knows where the breakpoints are**

Input :



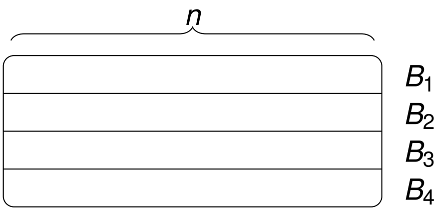
- Each substring without breakpoints must by definition appears in the solution
- Add the substring with the leftmost startpoint in the output
- $\mathcal{O}(|\text{Breakpoints}| \times |\text{Output}| \times |\text{Longest block}|)$

Output :



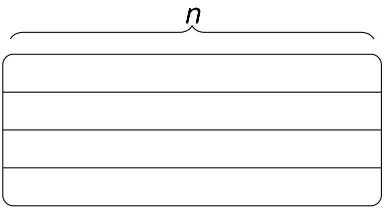
Polynomial-time algorithm

- If one only **knows the number of breakpoints B_i for each input sequence** of size n:



Polynomial-time algorithm

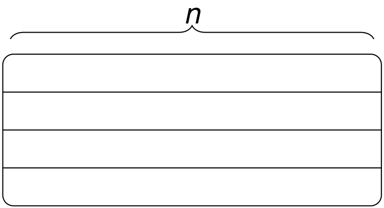
- If one only **knows the number of breakpoints B_i for each input sequence** of size n:
- One can "guess" where all breakpoints can be :



$$B_1 \Rightarrow \binom{n}{B_1} = \mathcal{O}(n^{B_1})$$
$$B_2 \Rightarrow \binom{n}{B_2} = \mathcal{O}(n^{B_2})$$
$$B_3 \Rightarrow \binom{n}{B_3} = \mathcal{O}(n^{B_3})$$
$$B_4 \Rightarrow \binom{n}{B_4} = \mathcal{O}(n^{B_4})$$

Polynomial-time algorithm

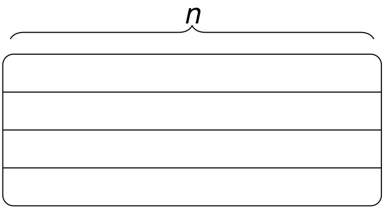
- If one only **knows the number of breakpoints B_i for each input sequence** of size n:
- One can ”guess” where all breakpoints can be :
- And launch the previous algorithm



$$B_1 \Rightarrow \binom{n}{B_1} = \mathcal{O}(n^{B_1})$$
$$B_2 \Rightarrow \binom{n}{B_2} = \mathcal{O}(n^{B_2})$$
$$B_3 \Rightarrow \binom{n}{B_3} = \mathcal{O}(n^{B_3})$$
$$B_4 \Rightarrow \binom{n}{B_4} = \mathcal{O}(n^{B_4})$$

Polynomial-time algorithm

- If one only **knows the number of breakpoints B_i for each input sequence** of size n:
- One can "guess" where all breakpoints can be :
- And launch the previous algorithm
- Overall complexity : $\mathcal{O}(n^{B_1}.n^{B_2}\dots n^{B_m}.BKn) = \mathcal{O}(n^B.BKn)$



$$B_1 \Rightarrow \binom{n}{B_1} = \mathcal{O}(n^{B_1})$$
$$B_2 \Rightarrow \binom{n}{B_2} = \mathcal{O}(n^{B_2})$$
$$B_3 \Rightarrow \binom{n}{B_3} = \mathcal{O}(n^{B_3})$$
$$B_4 \Rightarrow \binom{n}{B_4} = \mathcal{O}(n^{B_4})$$

Polynomial-time algorithm

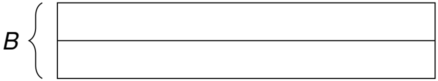
- If one only **knows the number of overall breakpoints** B

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences $= B$



Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences $= B + K$

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences $= B + K$

- Decide which have the breakpoints : $\binom{K+B}{B} = \mathcal{O}((K+B)^B)$

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences $= B + K$

- Decide which have the breakpoints : $\binom{K+B}{B} = \mathcal{O}((K+B)^B)$
- For each, run the $\mathcal{O}(nK^{2m})$ Ukkonen's algorithm

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints B**
- Maximum number of different input sequences $= B + K$

- Decide which have the breakpoints : $\binom{K+B}{B} = \mathcal{O}((K+B)^B)$
- For each, run the $\mathcal{O}(nK^{2m})$ Ukkonen's algorithm
  - Our sequences of interest are $m = B$

Polynomial-time algorithm

- If one only **knows the number of overall breakpoints** B
- Maximum number of different input sequences $= B + K$

- Decide which have the breakpoints : $\binom{K+B}{B} = \mathcal{O}((K+B)^B)$
- For each, run the $\mathcal{O}(nK^{2m})$ Ukkonen's algorithm
  - Our sequences of interest are $m = B$
- Overall complexity : $\mathcal{O}((K+B)^B.nK^{2B})$

Outline

Introduction

NP-Hardness

Exact Algorithms

Conclusion

Conclusion

- If $K = 2$, Mosaic Problem is polynomial time solvable
- If K is not bounded, NP-Complete

Conclusion

- If $K = 2$, Mosaic Problem is polynomial time solvable
- If K is not bounded, NP-Complete
- What about the complexity when K is bounded? FPT?
- What about the existence of a PTAS?

Questions?

Guillaume Blin[1] Florian Sikora[1] Romeo Rizzi[2]
Stéphane Vialette[1]

[1]LIGM, Université Paris-Est Marne-la-Vallée - France

[2]DIMI, Università di Udine - Italy