Algorithms and Bioinformatics

Part II — Comparative Genomics

Laurent Bulteau

Algorithms and Bioinformatics

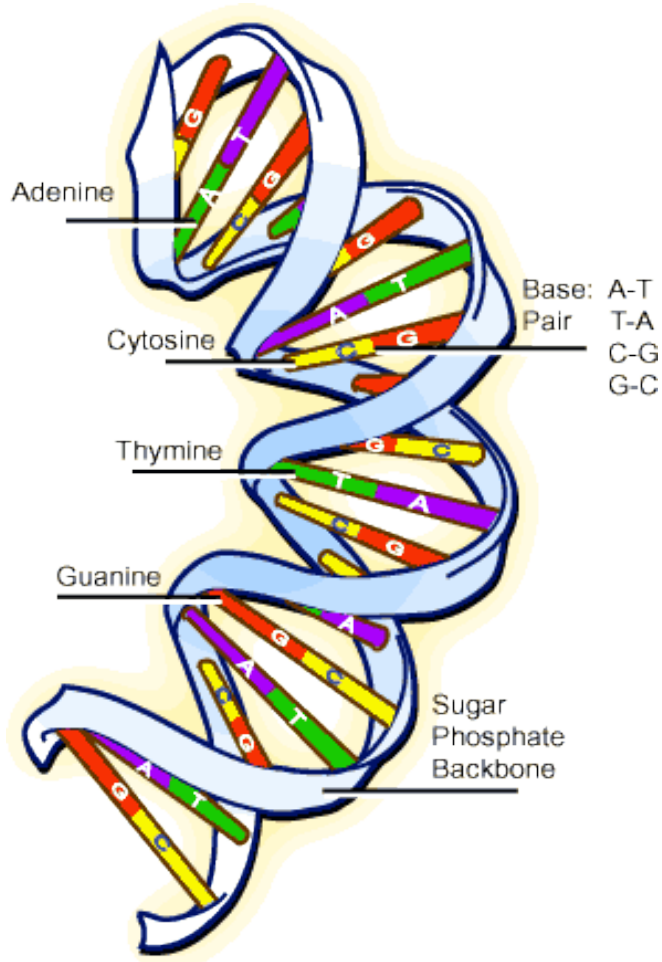
Part II — Comparative Genomics

II.1 — Models and Distances

Laurent Bulteau
Slides from Anthony Labarre

Context and motivations

- **D**eoxyribo**n**ucleic **a**cid: double helix of *nucleotides* (A, C, G, T);
- Complementarity (A-T, C-G): one strip is enough;
- *Gene* = sequence of nucleotides (that codes for a specific protein);
- *Chromosome* = ordered set of genes;
- *Genome* = set of chromosomes;
- **Goal:** compare genomes;



Context and motivations

- Biologists are interested in comparing species, for example:
  - in order to classify them;
  - in order to explain evolution by reconstructing scenarios;
- (Dis)similarity measures are needed;
- Usually based on the sequenced genomes;

At the nucleotide level

- Most comparisons take place at the nucleotide level;

Example (sequence alignment)

$$\begin{array}{lccccccccccccc} S_1: & \cdots & T & C & C & G & C & C & A & - & - & C & T & A & \cdots \\ & & | & | & & | & & | & & & & | & & | & \\ S_2: & \cdots & T & C & G & G & A & C & T & G & G & C & - & A & \cdots \end{array}$$

- Matches, substitutions, insertions and deletions;
- Correspond to mutations;

At the "gene" level

- Some mutations act on *segments* of nucleotides;
- Those large-scale mutations are called **genome rearrangements**;
- Sequence alignment becomes unfit;

At the "gene" level

- Some mutations act on *segments* of nucleotides;
- Those large-scale mutations are called **genome rearrangements**;
- Sequence alignment becomes unfit;

Example (genomes as sequences of segments)

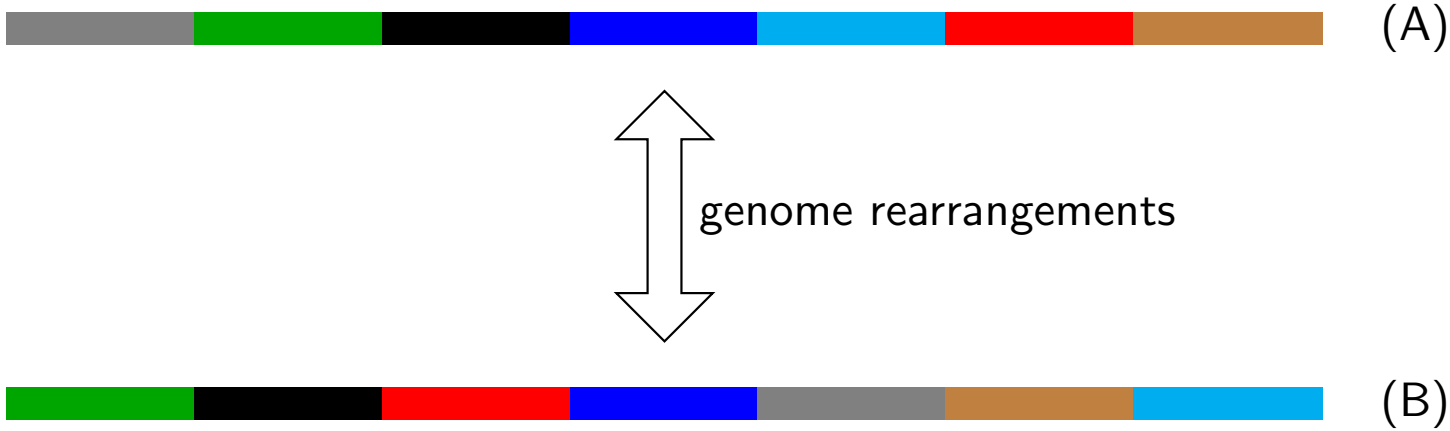
(A)

(B)

At the "gene" level

- Some mutations act on *segments* of nucleotides;
- Those large-scale mutations are called **genome rearrangements**;
- Sequence alignment becomes unfit;

Example (genomes as sequences of segments)



Genome rearrangements

- Our problem:

Problem (PAIRWISE GENOME REARRANGEMENT)

Input: *genomes G_1, G_2, a set S of mutations;*
Goal: *find a shortest sequence of elements of S that transforms G_1 into G_2.*

- Related, simpler problem: compute the **evolutionary distance** $d_S(G_1, G_2)$ (i.e. just the **length** of a shortest sequence);
- Many variants, depending on how genomes are modelled, what (and how) mutations are taken into account, etc.;

Modelling genomes as permutations

- Genomes are seen as permutations if:
  1. they form ordered sequences of genes (or other segments), and
  2. they only differ by order (no duplications or deletions).

Modelling genomes as permutations

- Genomes are seen as permutations if:
  1. they form ordered sequences of genes (or other segments), and
  2. they only differ by order (no duplications or deletions).

Example (genomes $\rightarrow$ permutations)

(A)

(B)

Modelling genomes as permutations

- Genomes are seen as permutations if:
  1. they form ordered sequences of genes (or other segments), and
  2. they only differ by order (no duplications or deletions).

Example (genomes $\rightarrow$ permutations)

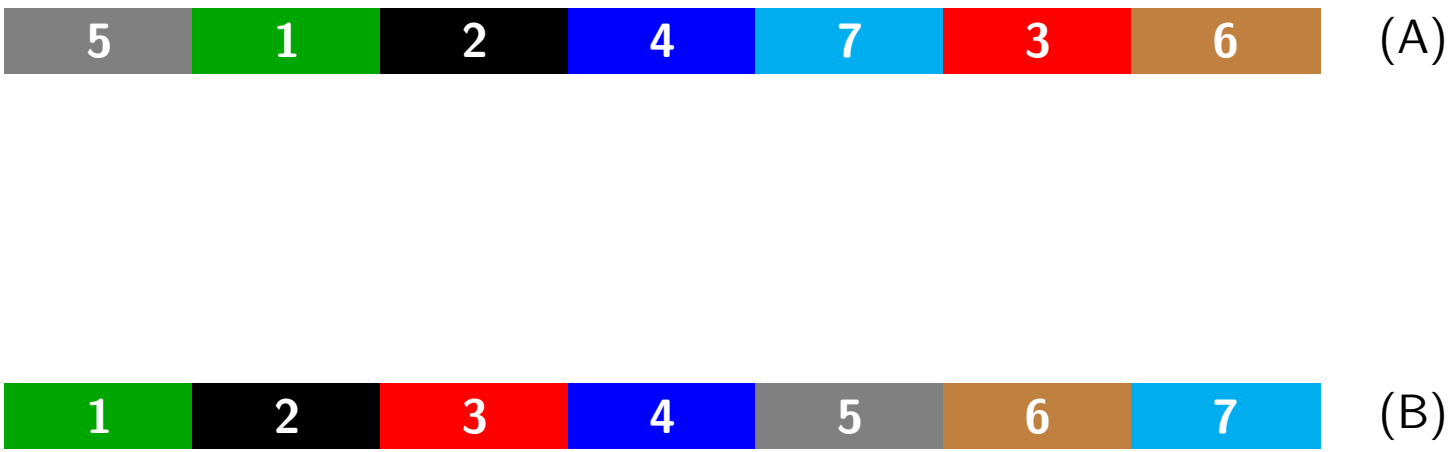
(A)

(B)

Modelling genomes as permutations

- Genomes are seen as permutations if:
  1. they form ordered sequences of genes (or other segments), and
  2. they only differ by order (no duplications or deletions).

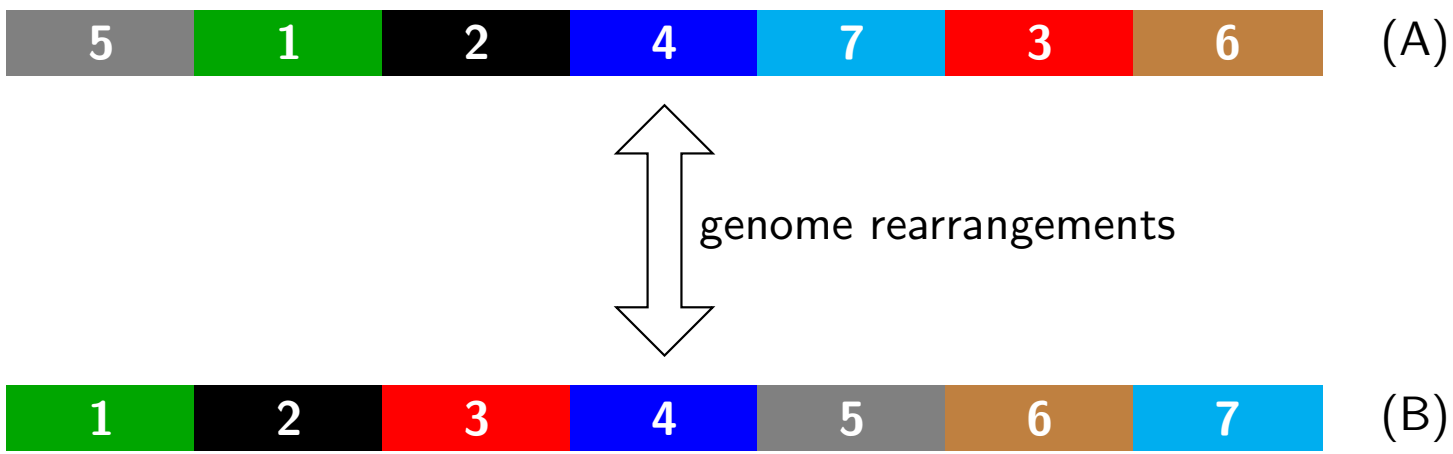
Example (genomes → permutations)



Modelling genomes as permutations

- Genomes are seen as permutations if:
  1. they form ordered sequences of genes (or other segments), and
  2. they only differ by order (no duplications or deletions).

Example (genomes → permutations)



Genome rearrangements for permutations

- Segments can be numbered as we wish, so we assume either genome is the **identity permutation** $\iota = \langle 1\ 2\ \cdots\ n \rangle$ and we wish to **sort** the other genome:

Problem (GENOME REARRANGEMENT (PERMUTATIONS))

Input: *a permutation π in S_n, a set $S \subseteq S_n$ of (per)mutations;*
Goal: *find a shortest sorting sequence of elements of S for π.*

- Again, we can also focus on merely computing $d_S(\pi)$ – the length of an optimal sorting sequence;
- S must generate S_n for any pair of permutations to be a finite distance apart;

Notation and definitions pertaining to permutations

- Permutations can be written in one- or two-row notation:

$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = \langle 4\ 1\ 6\ 2\ 5\ 3 \rangle.$$

Notation and definitions pertaining to permutations

- Permutations can be written in one- or two-row notation:

$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = \langle 4\ 1\ 6\ 2\ 5\ 3 \rangle.$$

- We deal exclusively with $[n] = \{1, 2, \ldots, n\}$;

Notation and definitions pertaining to permutations

- Permutations can be written in one- or two-row notation:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{pmatrix} = \langle 4\ 1\ 6\ 2\ 5\ 3 \rangle.$$

- We deal exclusively with $[n] = \{1, 2, \ldots, n\}$;
- All permutations of $[n]$ with composition form the **symmetric group** S_n;

Notation and definitions pertaining to permutations

- Permutations can be written in one- or two-row notation:

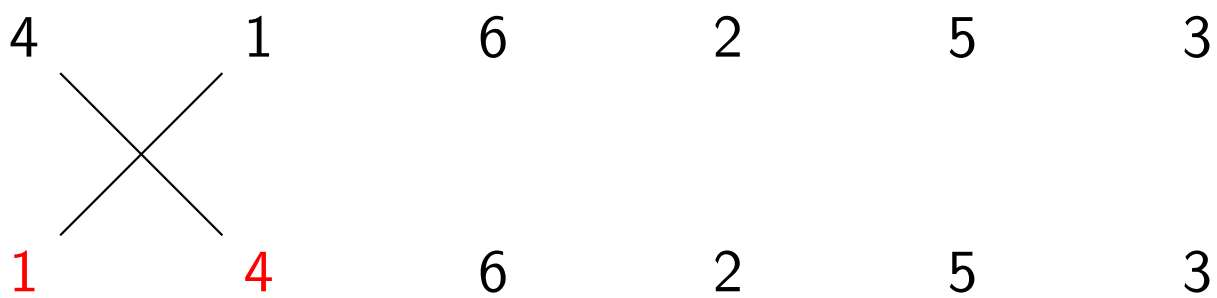
$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{pmatrix} = \langle 4\ 1\ 6\ 2\ 5\ 3 \rangle.$$

- We deal exclusively with $[n] = \{1, 2, \ldots, n\}$;
- All permutations of $[n]$ with composition form the **symmetric group** S_n;
- Composition: the usual $\circ$, which means that in $\pi \circ \sigma$, σ is applied first;

Sorting permutations by adjacent exchanges

- Simple operation : exchange any two adjacent elements:

Example

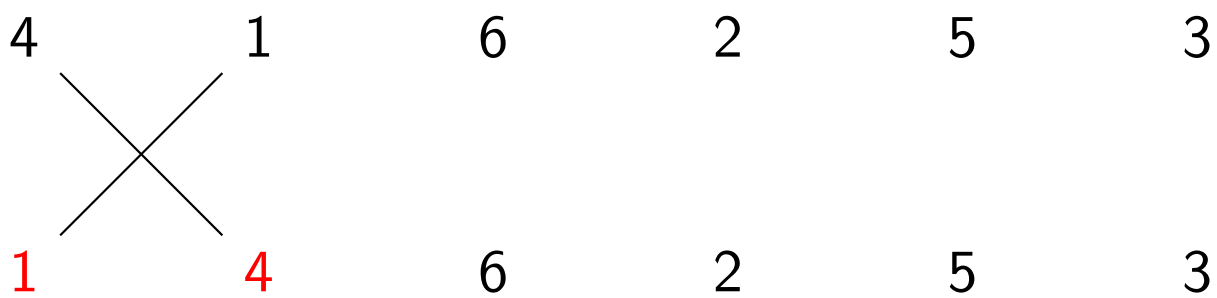


- So we want to sort a permutation by performing as few such exchanges as possible;

Sorting permutations by adjacent exchanges

- Simple operation : exchange any two adjacent elements:

Example

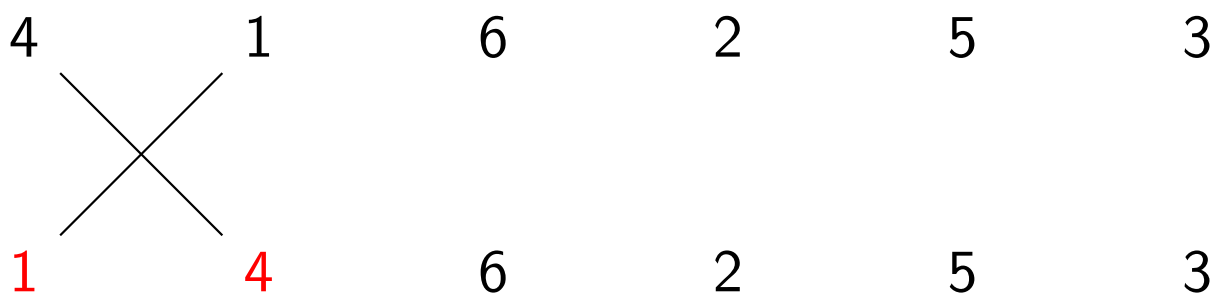


- So we want to sort a permutation by performing as few such exchanges as possible;
- Solution:
  - sorting: use Bubble Sort;

Sorting permutations by adjacent exchanges

- Simple operation : exchange any two adjacent elements:

Example

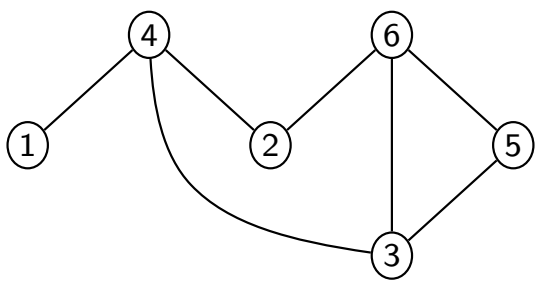


- So we want to sort a permutation by performing as few such exchanges as possible;
- Solution:
  - sorting: use Bubble Sort;
  - distance: the number of swaps used by Bubble Sort;

A closed formula for the adjacent exchange distance

- An **inversion** in a permutation is a pair of misplaced elements: (π_i, π_j) with $i < j$ and $\pi_i > \pi_j$;
- The **permutation graph** of π has $[n]$ as vertex set and the inversions of π as edges;

Example (the permutation graph of $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



A closed formula for the adjacent exchange distance

Theorem

The adjacent exchange distance of π is $|Inv(\pi)| = |E(PG(\pi))|$.

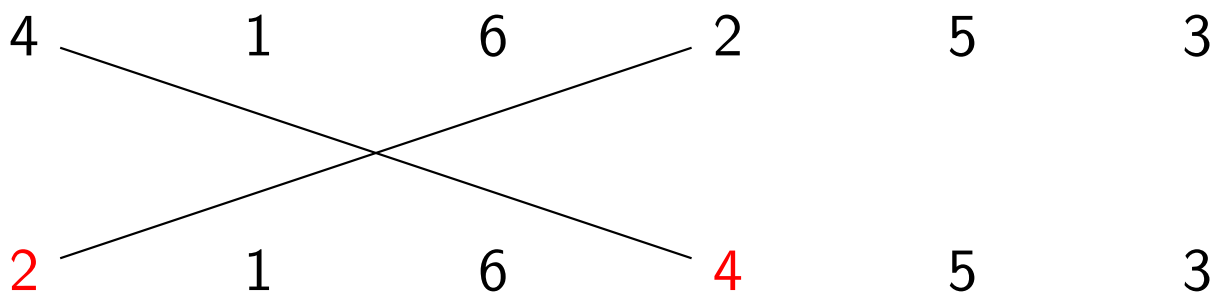
Proof sketch.

Each adjacent swap "fixes" at most one inversion, which is equivalent to removing an edge from PG, and we can always find such a move at every step if $\pi \neq \iota$. □

Sorting permutations by exchanges

- Simple operation : exchange any two elements:

Example



- So we want to sort a permutation by performing as few such exchanges as possible;

Sorting permutations by exchanges

- Here's a more complete example:

Example (a sorting sequence for $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)

$$4 \qquad 1 \qquad 6 \qquad 2 \qquad 5 \qquad 3$$

Sorting permutations by exchanges

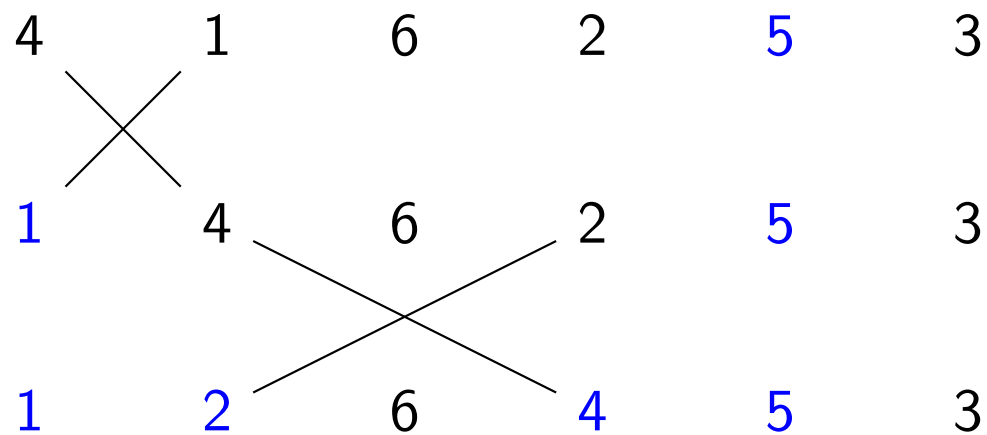
- Here's a more complete example:

Example (a sorting sequence for $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)

Sorting permutations by exchanges

- Here's a more complete example:

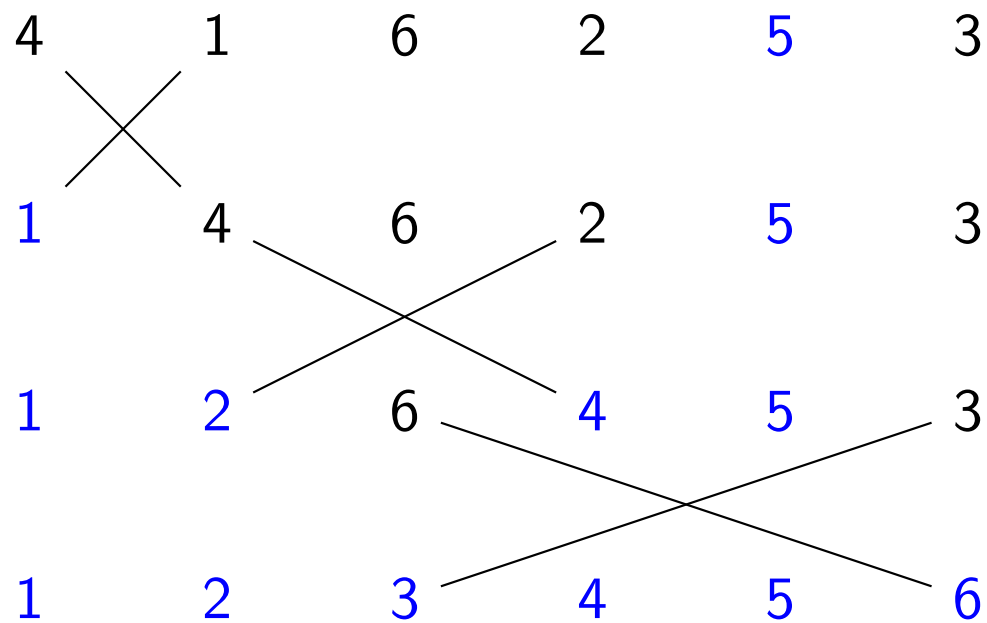
Example (a sorting sequence for $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Sorting permutations by exchanges

- Here's a more complete example:

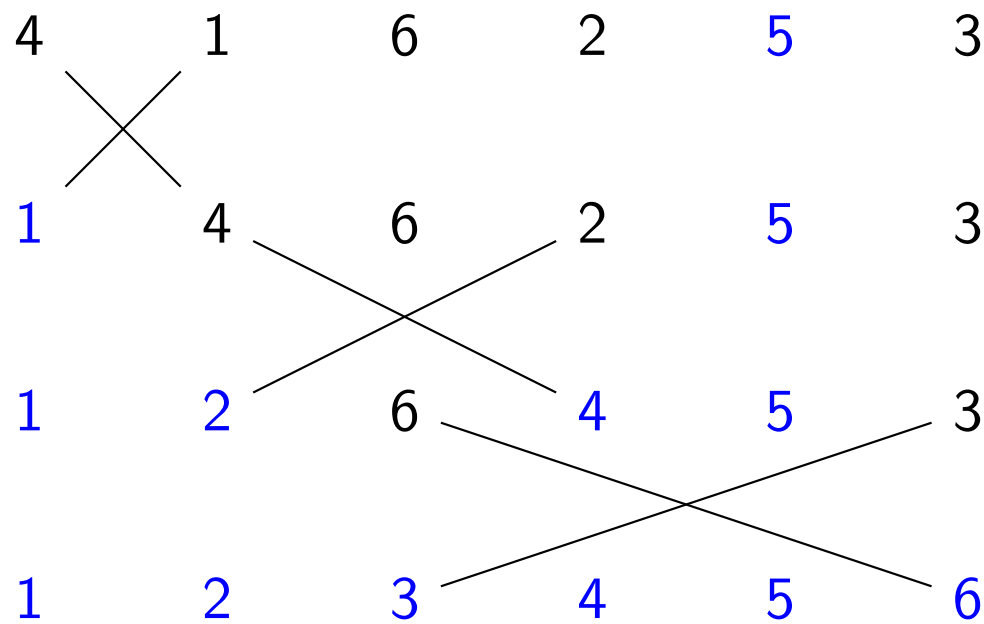
Example (a sorting sequence for $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Sorting permutations by exchanges

- Here's a more complete example:

Example (a sorting sequence for $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



- It works... but can we do better?

Sorting permutations by exchanges

- Our goal: each element should be "at the right place";
- Some elements are already where they should be, so they won't move;
- Strategy: read permutation from left to right, and:
  - if $\pi_i = i$, pass;
  - otherwise, exchange π_i with i;

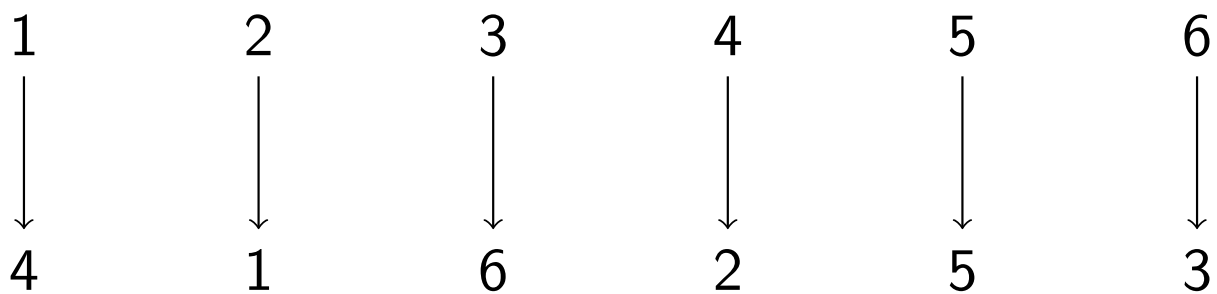
Sorting permutations by exchanges

- The algorithm obviously terminates;
- At every step, we "fix" one or two positions;
- We use the minimum number of exchanges;
- On the other hand, we'd like to be able to compute the distance without sorting;

Cycles

- Computing the distance requires using the **cycles** of the permutation;
- Those cycles are obtained by iterating the permutation's action on $\{1, 2, \ldots, n\}$, stopping when all elements have been visited;

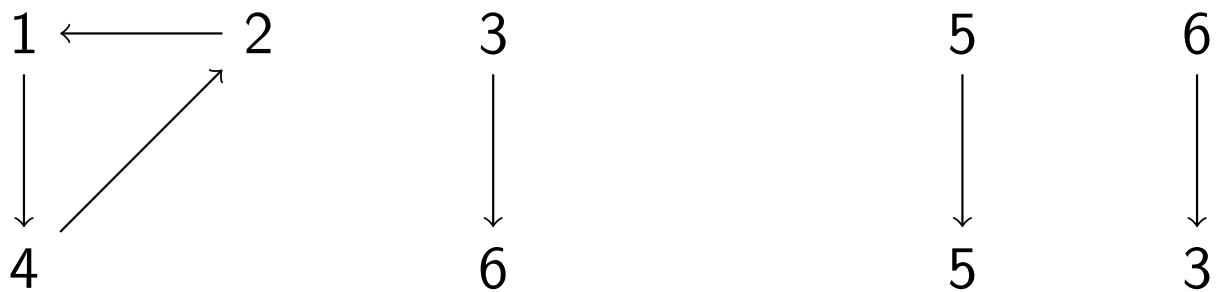
Example (cycles of $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Cycles

- Computing the distance requires using the **cycles** of the permutation;
- Those cycles are obtained by iterating the permutation's action on $\{1, 2, \ldots, n\}$, stopping when all elements have been visited;

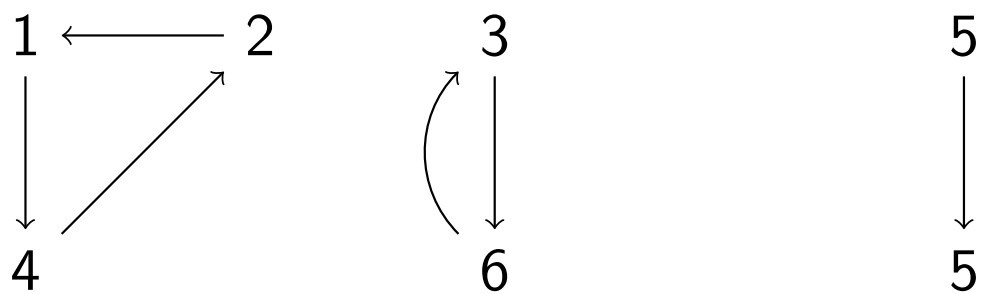
Example (cycles of $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Cycles

- Computing the distance requires using the **cycles** of the permutation;
- Those cycles are obtained by iterating the permutation's action on $\{1, 2, \ldots, n\}$, stopping when all elements have been visited;

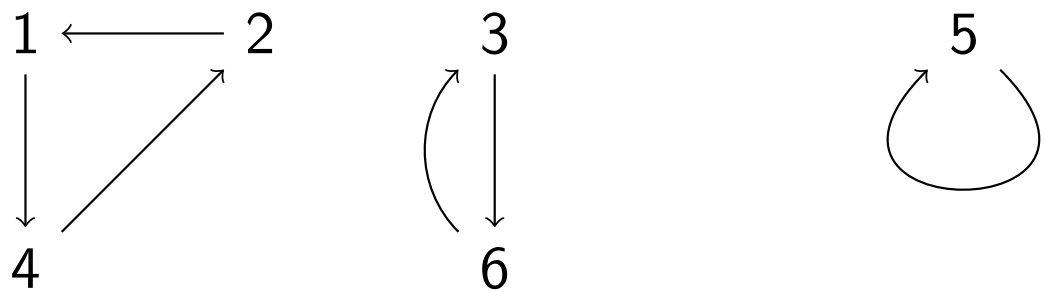
Example (cycles of $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Cycles

- Computing the distance requires using the **cycles** of the permutation;
- Those cycles are obtained by iterating the permutation's action on $\{1, 2, \ldots, n\}$, stopping when all elements have been visited;

Example (cycles of $\langle 4\ 1\ 6\ 2\ 5\ 3 \rangle$)



Disjoint cycle decomposition of permutations

- Each permutation decomposes into disjoint cycles:

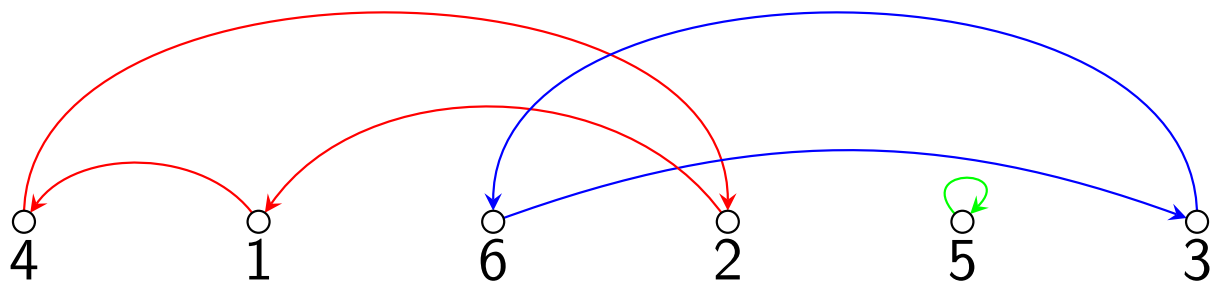
$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = (1,4,2)(3,6)(5).$$

Disjoint cycle decomposition of permutations

- Each permutation decomposes into disjoint cycles:

$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = (1,4,2)(3,6)(5).$$

- The graph of the permutation π, denoted by $\Gamma(\pi)$, pictures this decomposition:

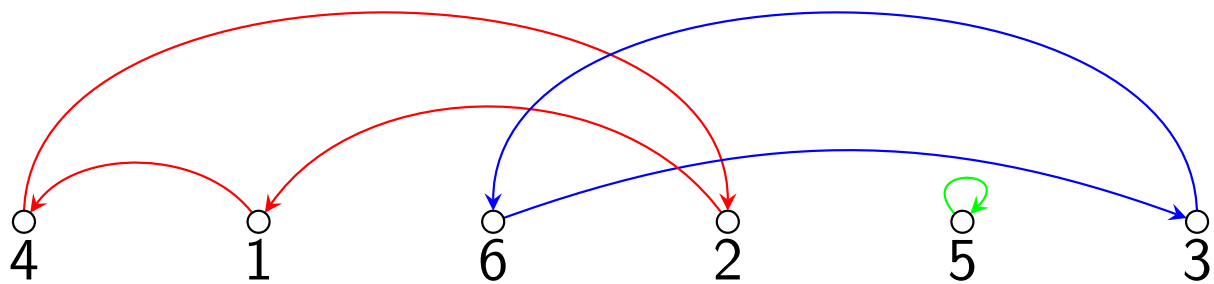


Disjoint cycle decomposition of permutations

- Each permutation decomposes into disjoint cycles:

$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = (1,4,2)(3,6)(5).$$

- The graph of the permutation π, denoted by $\Gamma(\pi)$, pictures this decomposition:



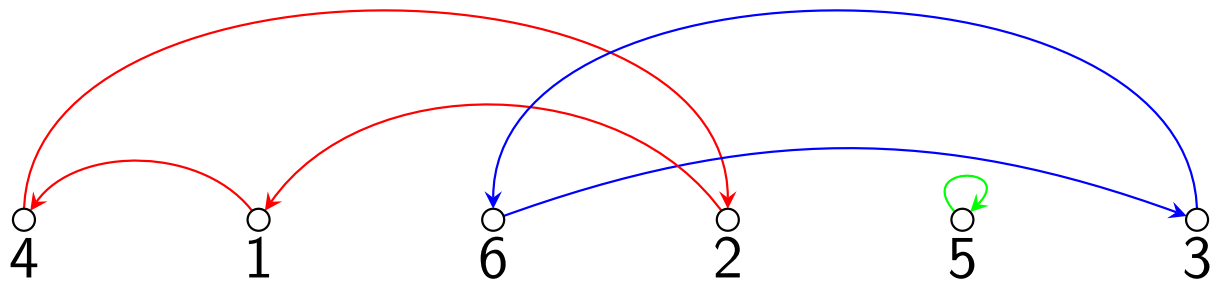
- The number of cycles of π is written $c(\pi)$;

Disjoint cycle decomposition of permutations

- Each permutation decomposes into disjoint cycles:

$$\pi = \left\langle \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 6 & 2 & 5 & 3 \end{array} \right\rangle = (1,4,2)(3,6)(5).$$

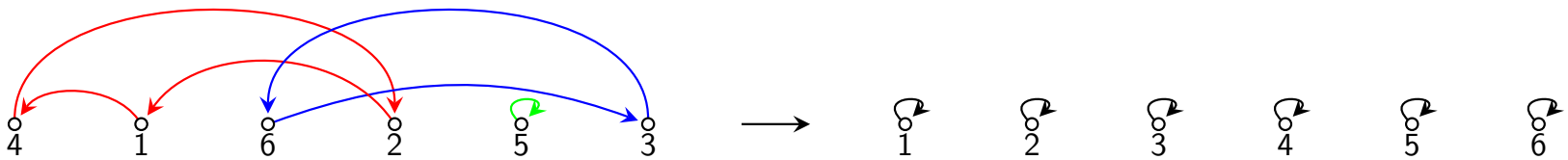
- The graph of the permutation π, denoted by $\Gamma(\pi)$, pictures this decomposition:



- The number of cycles of π is written $c(\pi)$;
- 1-cycles are sometimes omitted;

Cycles and sorting

- Cycles of length 1 correspond to sorted elements; all other cycles consist of elements that are misplaced;



- Sorting comes down to splitting cycles until we only have cycles of length 1;
- Our algorithm repeatedly splits k-cycles into a 1-cycle and a $(k-1)$-cycle;

Computing the "exchange distance" $exc(\cdot)$

- At each step, we can always create a new cycle if $\pi \neq \iota$, so:

$$exc(\pi) \leq n - c(\pi)$$

Computing the "exchange distance" $exc(\cdot)$

- At each step, we can always create a new cycle if $\pi \neq \iota$, so:

$$exc(\pi) \leq n - c(\pi)$$

- And we can't do better, so:

$$exc(\pi) \geq n - c(\pi)$$

Computing the "exchange distance" $exc(\cdot)$

- At each step, we can always create a new cycle if $\pi \neq \iota$, so:

$$exc(\pi) \leq n - c(\pi)$$

- And we can't do better, so:

$$exc(\pi) \geq n - c(\pi)$$

- Therefore:

Computing the "exchange distance" $exc(\cdot)$

- At each step, we can always create a new cycle if $\pi \neq \iota$, so:

$$exc(\pi) \leq n - c(\pi)$$

- And we can't do better, so:

$$exc(\pi) \geq n - c(\pi)$$

- Therefore:

Theorem ([Cayley, 1849])

The exchange distance of π in S_n is $n - c(\pi)$.

Lessons from sorting by exchanges

- Note that ι is the only permutation with n cycles;
- The formula $exc(\pi) = n - c(\pi)$ expresses:
  - the difference between the number of cycles we have and the number of cycles we want;
  - and the fact that at each step, we can obtain exactly one new cycle.
- This point of view will be *crucial* to sorting problems;

Practice

For the algorithms below, a permutation is represented as a size-n array with values and indices ranging from 1 to n.

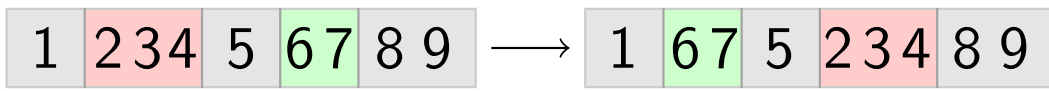
1. Give a linear-time algorithm computing the inverse of a permutation.
2. Give a linear-time algorithm computing an optimal sequence of swaps sorting a permutation.
3. Give a linear-time algorithm computing the decomposition of a permutation into disjoint cycles
4. Let S be a set of permutations defining distance d_S over $\mathcal{S}_n$, such that S is stable by inversion ($\pi \in S \Rightarrow \pi^{-1} \in S$).
   - Prove that $d_S(\pi) = d_S(\pi^{-1})$ for every permutation π.
   - The stability by inversion is a sufficient condition to have the above property, but is it necessary?

Block-interchanges and transpositions

- The mutations we observe in evolution (may) act on intervals;
- We'll now look at two generalisations of exchanges:
  1. block-interchanges;
  2. transpositions;

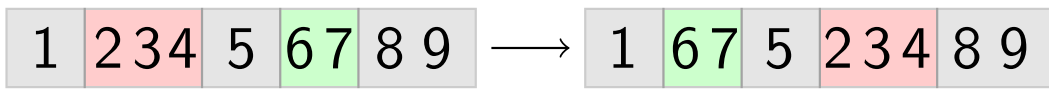
Block-interchanges and transpositions

- The mutations we observe in evolution (may) act on intervals;
- We'll now look at two generalisations of exchanges:
  1. block-interchanges;
  2. transpositions;
- **Block-interchanges** exchange two disjoint intervals in a permutation;

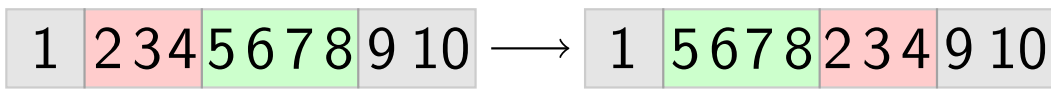


Block-interchanges and transpositions

- The mutations we observe in evolution (may) act on intervals;
- We'll now look at two generalisations of exchanges:
  1. block-interchanges;
  2. transpositions;
- **Block-interchanges** exchange two disjoint intervals in a permutation;



- **Transpositions** displace an interval of the permutation;

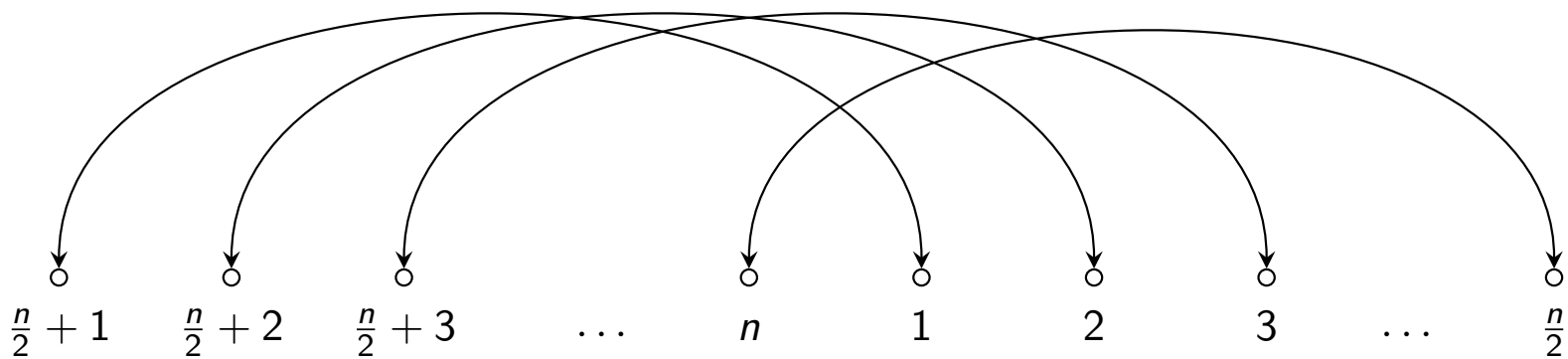


Computing the associated distances

- Does the disjoint cycle technique "work"?

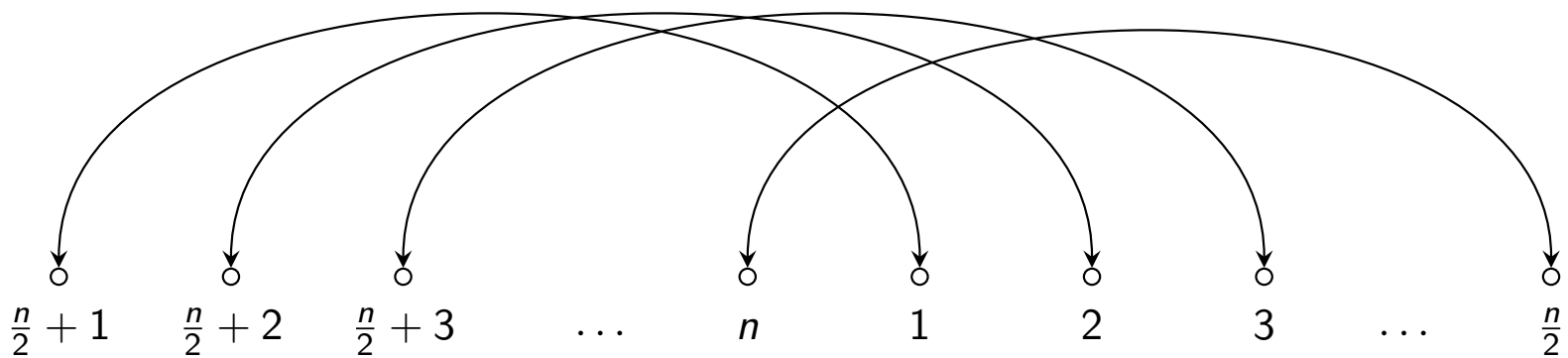
Computing the associated distances

- Does the disjoint cycle technique "work"?
- Unlikely: the following permutation has $n/2$ cycles of length two, but $bid(\pi) = td(\pi) = 1$:



Computing the associated distances

- Does the disjoint cycle technique "work"?
- Unlikely: the following permutation has $n/2$ cycles of length two, but $bid(\pi) = td(\pi) = 1$:



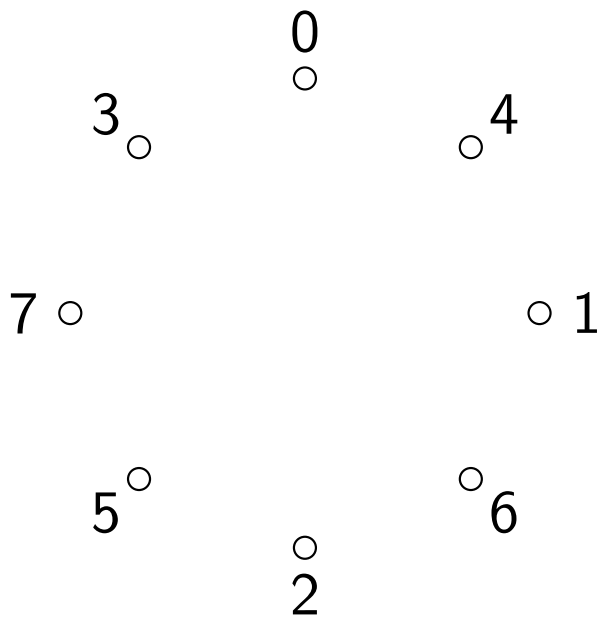
- We'll need something else since we cannot bound the effect of an operation in that setting;

The "directed breakpoint graph"

- (the term "breakpoint" will be explained later);
- Let's build the directed breakpoint graph of $\pi = \langle 4\ 1\ 6\ 2\ 5\ 7\ 3 \rangle$:

The "directed breakpoint graph"

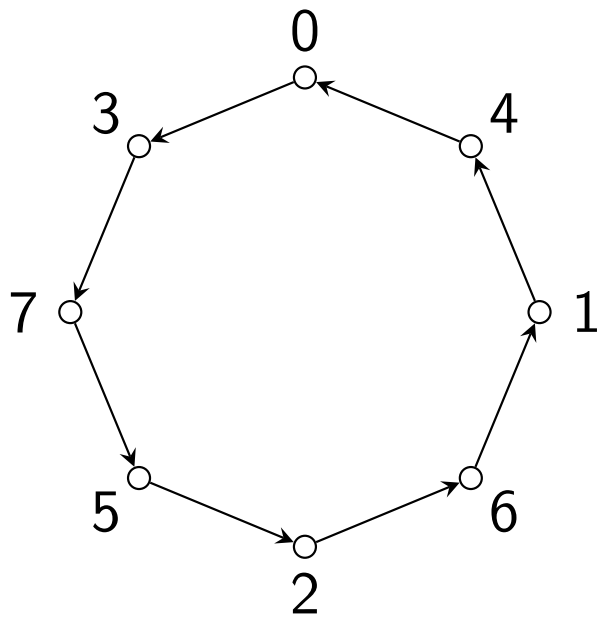
- (the term "breakpoint" will be explained later);
- Let's build the directed breakpoint graph of $\pi = \langle 4\ 1\ 6\ 2\ 5\ 7\ 3 \rangle$:



1. build the ordered vertex set $(\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n)$;

The "directed breakpoint graph"

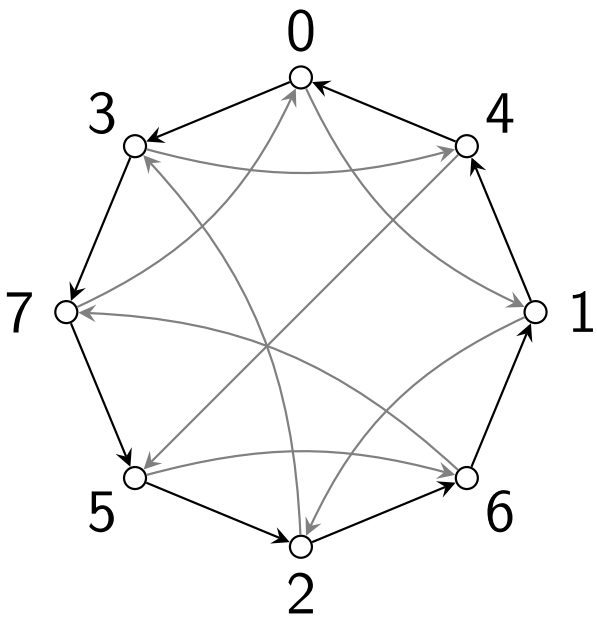
- (the term "breakpoint" will be explained later);
- Let's build the directed breakpoint graph of $\pi = \langle 4\ 1\ 6\ 2\ 5\ 7\ 3 \rangle$:



1. build the ordered vertex set $(\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n)$;
2. add **black arcs** for every ordered pair $(\pi_i, \pi_{i-1 \pmod{n+1}})$;

The "directed breakpoint graph"

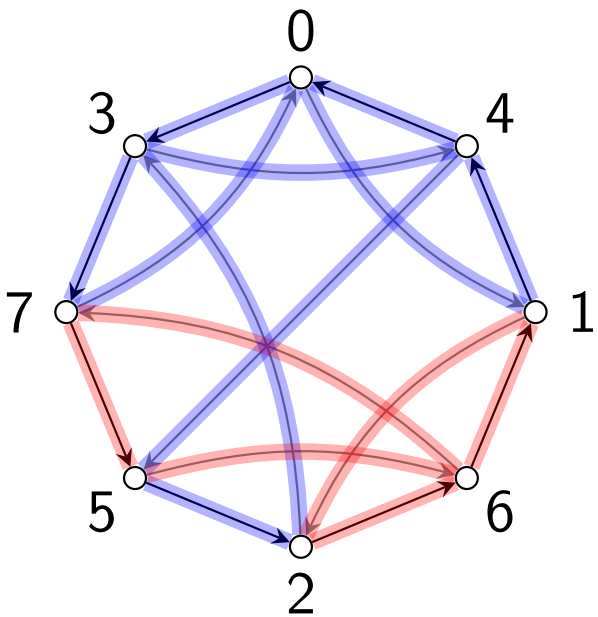
- (the term "breakpoint" will be explained later);
- Let's build the directed breakpoint graph of $\pi = \langle 4\ 1\ 6\ 2\ 5\ 7\ 3 \rangle$:



1. build the ordered vertex set $(\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n)$;
2. add **black arcs** for every ordered pair $(\pi_i, \pi_{i-1 \pmod{n+1}})$;
3. add **grey arcs** for every ordered pair $(i, i+1 \pmod{n+1})$;

The "directed breakpoint graph"

- (the term "breakpoint" will be explained later);
- Let's build the directed breakpoint graph of $\pi = \langle 4\ 1\ 6\ 2\ 5\ 7\ 3 \rangle$:



1. build the ordered vertex set $(\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n)$;
2. add **black arcs** for every ordered pair $(\pi_i, \pi_{i-1 \pmod{n+1}})$;
3. add **grey arcs** for every ordered pair $(i, i+1 \pmod{n+1})$;

$DBG(\pi)$ decomposes in a unique way into **alternating cycles**

Formal definition of the directed breakpoint graph

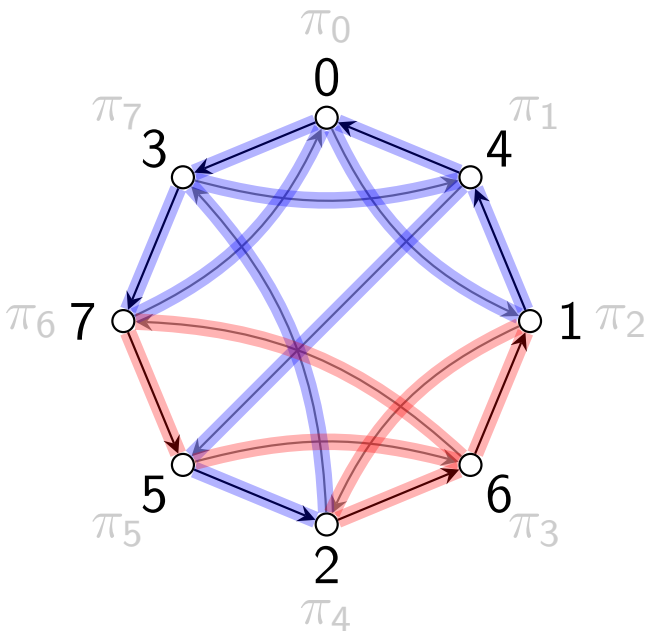
Definition ([Bafna and Pevzner, 1998])

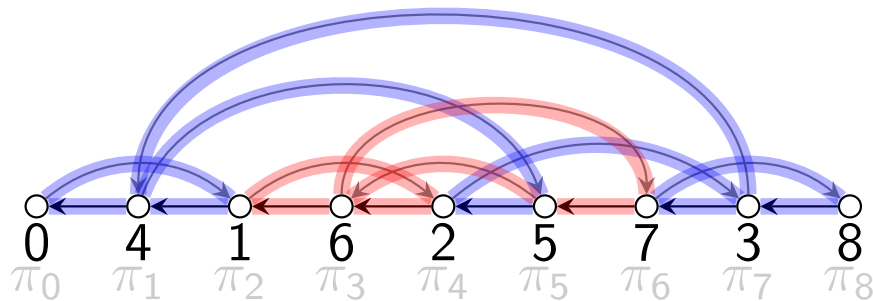
The **directed breakpoint graph** of $\langle \pi_1\ \pi_2\ \cdots\ \pi_n \rangle$ is defined by:

1. an ordered vertex set $V = (\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n)$;
2. a bicoloured arc set $A = A_B \cup A_G$, where:
   - 2.1 $A_B = \{(\pi_i, \pi_{i-1 \pmod{n+1}}) : 0 \leq i \leq n\}$;
   - 2.2 $A_G = \{(i, i+1 \pmod{n+1}) : 0 \leq i \leq n\}$;

Circular vs. linear layout

- One can also represent the directed breakpoint graph using a linear layout without affecting the cycle structure:

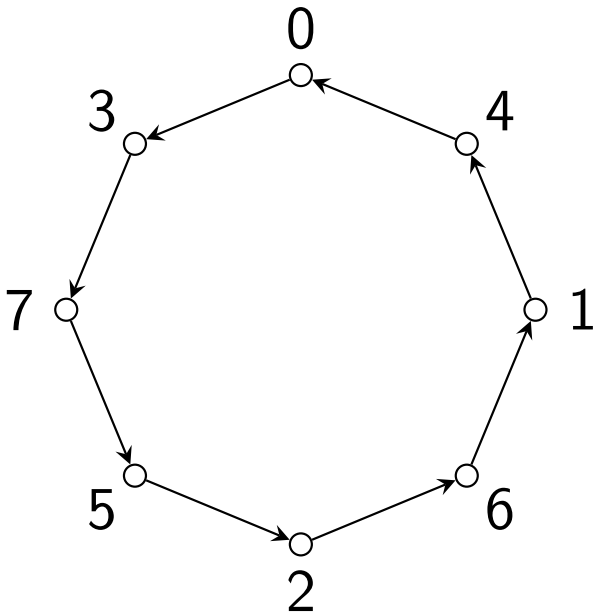


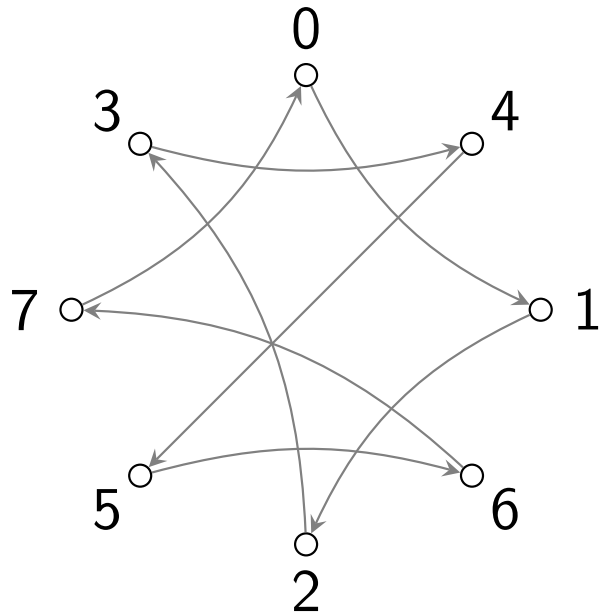


- Circular $\rightarrow$ linear: split 0 into 0 and $n+1$;
- Linear $\rightarrow$ circular: merge 0 and $n+1$ into 0;
- (in both cases: adapt arcs accordingly);

Intuitions behind $DBG(\pi) - 1$

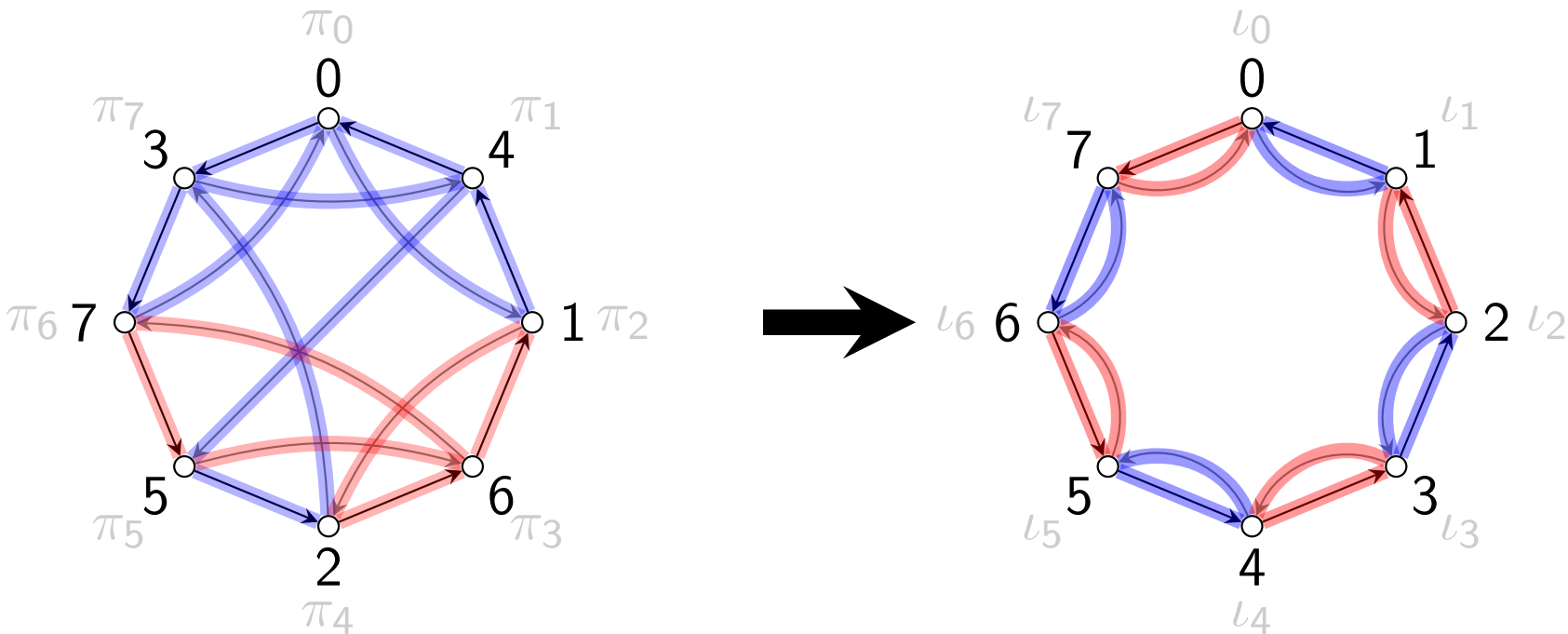
- Both "monochromatic" cycles represent an ordering:
  1. the black one represents the one we have ("reality");
  2. the grey one represents the one we want to obtain ("desire");





Intuitions behind $DBG(\pi)$ – 2

- The alternating cycles are a blend of "reality" and "desire", and we must act on those cycles to turn "reality" into "desire";
- When we are done, we have the largest number of cycles;

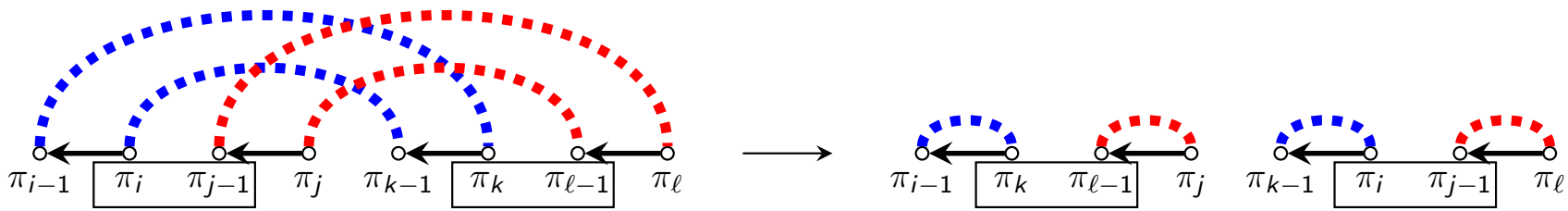


Proving lower bounds

- One way of proving lower bounds on distances: be optimistic:
  1. find out the "best case";
  2. pretend we're always in that case;
- Techniques based on breakpoint graphs follow the same spirit as those based on the disjoint cycle decomposition, but less freedom is allowed (details later);
- Usually: case analysis to determine by how much a parameter of the graph can change with one operation;

Lower bounding the block-interchange distance

A block-interchange $\beta(i, j, k, \ell)$ increases the number of cycles in *DBG* by at most 2:

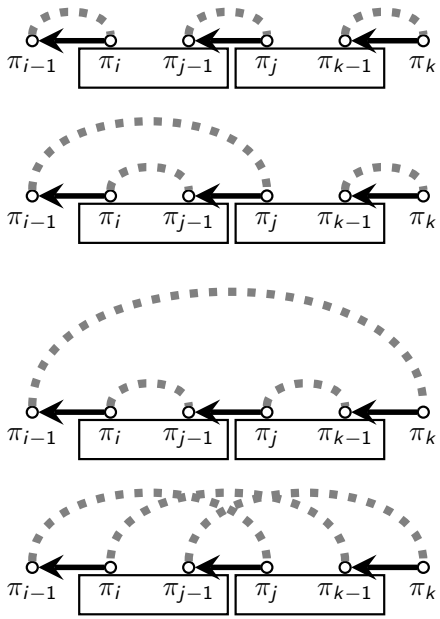


Theorem ([Christie, 1996])

For all π *in* S_n*:* $bid(\pi) \geq \frac{n+1-c(DBG(\pi))}{2}$.

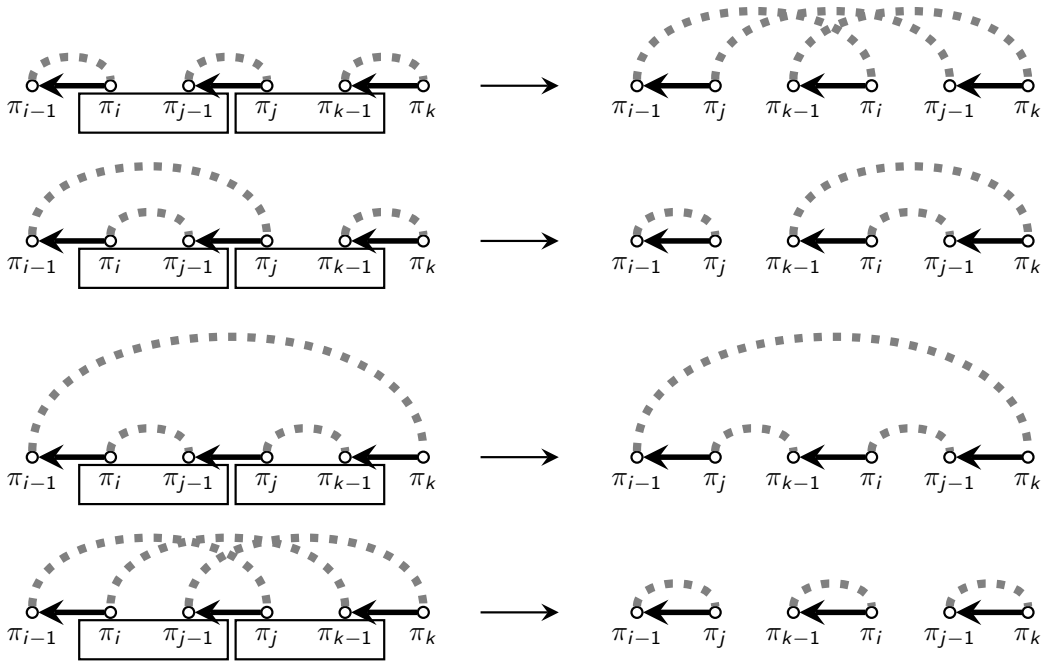
Lower bounding the transposition distance

A transposition $\tau(i, j, k)$ increases the number of **odd** cycles in DBG by at most 2:



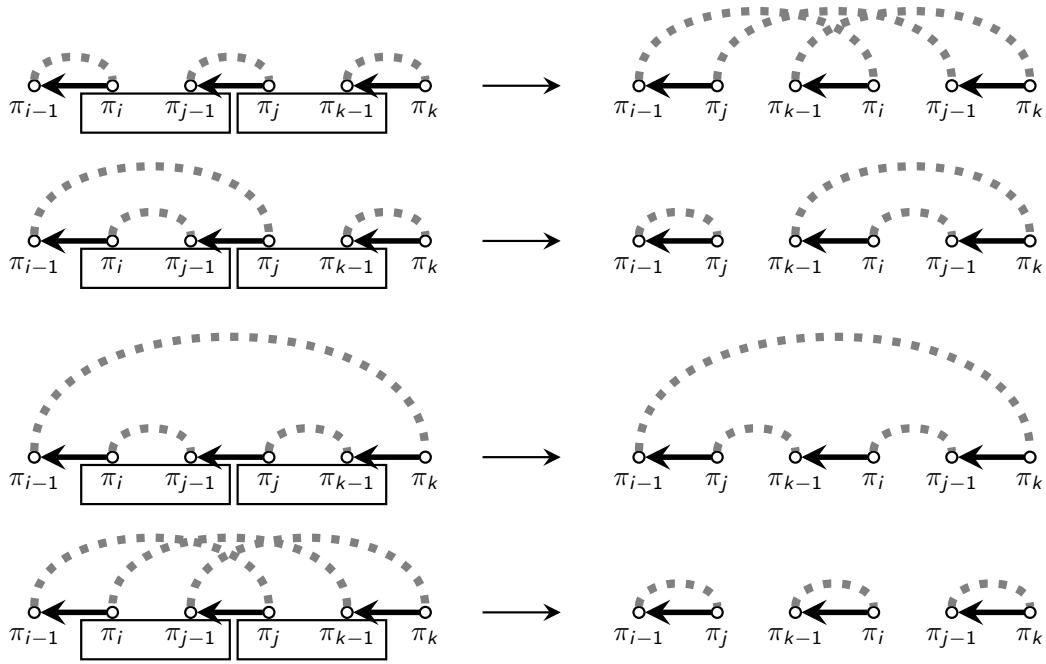
Lower bounding the transposition distance

A transposition $\tau(i, j, k)$ increases the number of **odd** cycles in DBG by at most 2:



Lower bounding the transposition distance

A transposition $\tau(i, j, k)$ increases the number of **odd** cycles in DBG by at most 2:



Theorem ([Bafna and Pevzner, 1998])

For all π in S_n: $td(\pi) \geq \frac{n+1-c_{odd}(DBG(\pi))}{2}$.

Practice

5. Give sorting sequences for the following permutations, and prove they are optimal

   - $\langle 654321 \rangle$, using block-interchanges
   - $\langle 3254761 \rangle$, using transpositions

Proving upper bounds

- One way of proving upper bounds on distances: be pessimistic:
  1. find out the "worst case";
  2. pretend we're always in that case;
- For better upper bounds: be less pessimistic:
  - case analyses of varying difficulty;
  - look at *sequences* of moves;

Upper bounds

- For block-interchanges, we have:

Theorem ([Christie, 1996])

For all π in S_n: $bid(\pi) \leq \frac{n+1-c(DBG(\pi))}{2}$.

- Which equals the lower bound and therefore the exact distance;

Upper bounds

- For transpositions:

  Theorem ([Bafna and Pevzner, 1998])

  For all π in S_n: $td(\pi) \leq \frac{3}{4}(n + 1 - c_{odd}(DBG(\pi))) = \frac{3}{2}OPT$.

- Current best approximation: 11/8 [Elias and Hartman, 2006]

Practice

6. Show that $td(\pi) \leq n - LIS(\pi)$, where LIS denotes the length of the longest increasing subsequence.

7. Give a polynomial-time 2-approximation algorithm for the Transposition Distance problem.

Practice

6. Show that $td(\pi) \leq n - LIS(\pi)$, where LIS denotes the length of the longest increasing subsequence.

7. Give a polynomial-time 2-approximation algorithm for the Transposition Distance problem.

Hint 1: Use the lower bound based on the number of cycles: $td(\pi) \geq \frac{n+1-c(DBG(\pi))}{2}$

Practice

6. Show that $td(\pi) \leq n - LIS(\pi)$, where LIS denotes the length of the longest increasing subsequence.

7. Give a polynomial-time 2-approximation algorithm for the Transposition Distance problem.

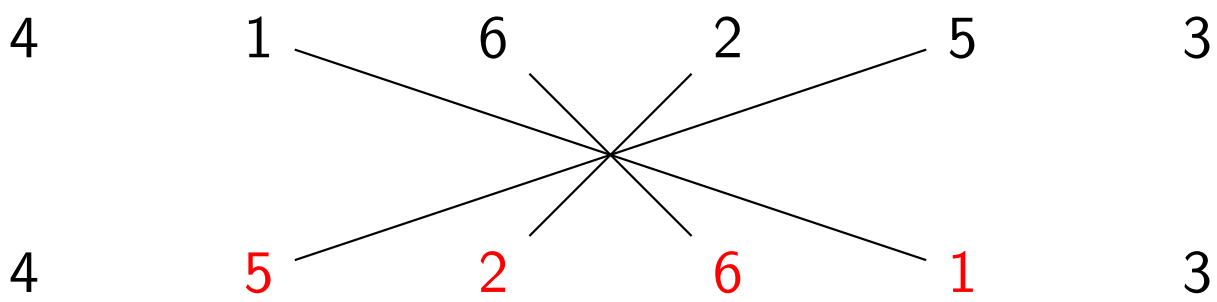
Hint 1: Use the lower bound based on the number of cycles: $td(\pi) \geq \frac{n+1-c(DBG(\pi))}{2}$

Hint 2: If π displays a cycle of the *right* form, give a transposition creating two cycles. Otherwise, turn a cycle into the right form with one transposition.

Reversals

- Another type of mutation occurs frequently: **reversals**, which reverse the order of elements on an interval of the permutation;

Example



Reversals

- Here's an optimal sorting sequence[1]:

Example (optimal sorting sequence of reversals)

6 genes 6 singletons, 0 2-strips Unsigned Reversal Distance: 4

One optimal reversal scenario

Step	Description	
0	(Source)	4 1 6 2 5 3
1	Reversal	4 1 2 6 5 3
2	Reversal	4 5 6 2 1 3
3	Reversal	1 2 6 5 4 3
4	Reversal (Destination)	1 2 3 4 5 6

- Bad news: stuff seen so far doesn't work;

[1] Obtained using GRIMM: `http://grimm.ucsd.edu/cgi-bin/grimm.cgi`

Breakpoints

Definition ([Watterson et al., 1982])

A **breakpoint** in a permutation π is an ordered pair (π_i, π_{i+1}) with $|\pi_{i+1} - \pi_i| \neq 1$ (otherwise it's an **adjacency**).

Breakpoints

Definition ([Watterson et al., 1982])

A **breakpoint** in a permutation π is an ordered pair (π_i, π_{i+1}) with $|\pi_{i+1} - \pi_i| \neq 1$ (otherwise it's an **adjacency**).

Example (breakpoints of $\langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$)

$$3 \bullet 1 \bullet 5 \quad 4 \bullet 2 \bullet 8 \bullet 6 \quad 7$$

Breakpoints

Definition ([Watterson et al., 1982])

A **breakpoint** in a permutation π is an ordered pair (π_i, π_{i+1}) with $|\pi_{i+1} - \pi_i| \neq 1$ (otherwise it's an **adjacency**).

Example (breakpoints of $\langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$)

$$3 \bullet 1 \bullet 5 \quad 4 \bullet 2 \bullet 8 \bullet 6 \quad 7$$

- This notion characterises elements that are "relatively misplaced": they're not consecutive in ι, nor in $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;
- That's also the case of $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$;

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;
- That's also the case of $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$;
- To distinguish them, we **frame** permutations:

$$\langle \pi_1\ \pi_2\ \cdots\ \pi_n \rangle \mapsto \langle \mathbf{0}\ \pi_1\ \pi_2\ \cdots\ \pi_n\ \mathbf{n+1} \rangle$$

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;
- That's also the case of $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$;
- To distinguish them, we **frame** permutations:

$$\langle \pi_1\ \pi_2\ \cdots\ \pi_n \rangle \mapsto \langle \mathbf{0}\ \pi_1\ \pi_2\ \cdots\ \pi_n\ \mathbf{n+1} \rangle$$

- Those artificial elements are denoted by π_0 and π_{n+1};

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;
- That's also the case of $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$;
- To distinguish them, we **frame** permutations:

$$\langle \pi_1\ \pi_2\ \cdots\ \pi_n \rangle \mapsto \langle \mathbf{0}\ \pi_1\ \pi_2\ \cdots\ \pi_n\ \mathbf{n+1} \rangle$$

- Those artificial elements are denoted by π_0 and π_{n+1};
- Which leads to the following definition:

Definition

The **number of breakpoints** of a permutation π in S_n is

$$b(\pi) = |\{(\pi_i, \pi_{i+1}) \mid 0 \leq i \leq n \text{ and } |\pi_{i+1} - \pi_i| \neq 1\}|.$$

Breakpoints

- Note that $\iota = \langle 1\ 2\ \cdots\ n \rangle$ has no breakpoint;
- That's also the case of $\langle n\ n-1\ \cdots\ 2\ 1 \rangle$;
- To distinguish them, we **frame** permutations:

$$\langle \pi_1\ \pi_2\ \cdots\ \pi_n \rangle \mapsto \langle \mathbf{0}\ \pi_1\ \pi_2\ \cdots\ \pi_n\ \mathbf{n+1} \rangle$$

- Those artificial elements are denoted by π_0 and π_{n+1};
- Which leads to the following definition:

Definition

The **number of breakpoints** of a permutation π in S_n is

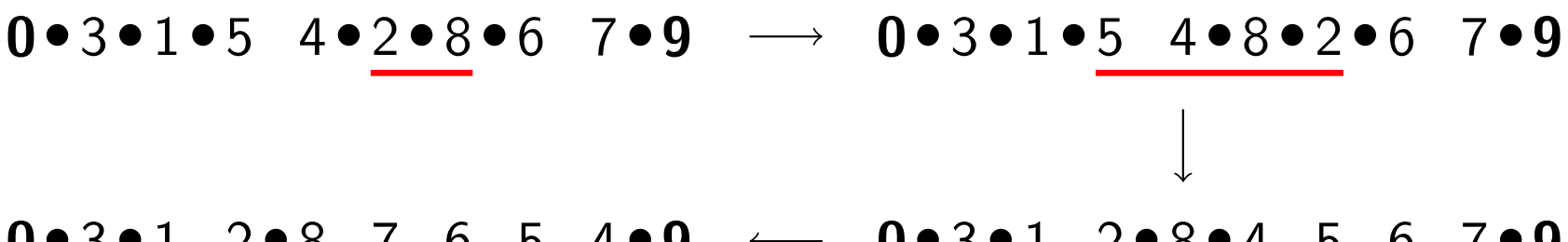
$$b(\pi) = |\{(\pi_i, \pi_{i+1}) \mid 0 \leq i \leq n \text{ and } |\pi_{i+1} - \pi_i| \neq 1\}|.$$

- Example: $\langle \mathbf{0} \bullet 3 \bullet 1 \bullet 5\ 4 \bullet 2 \bullet 8 \bullet 6\ 7 \bullet \mathbf{9} \rangle \Rightarrow b(\pi) = 7$;

Usefulness of breakpoints

- Observation: reversals can "fix" breakpoints:

Example



Lower bound

- A reversal can fix at most two breakpoints;
- If we're lucky, we are always in that case;
- This yields the following lower bound:

Theorem ([Kececioglu and Sankoff, 1995])

For every permutation π:

$$rd(\pi) \geq b(\pi)/2.$$

Upper bound

- On the other hand we can always fix at least one breakpoint (details: [Kececioglu and Sankoff, 1995])
- So the algorithm is a 2-approximation:

$$b(\pi)/2 \leq rd(\pi) \leq b(\pi)$$

- Can we do better?

The *undirected breakpoint graph*

- Yes, but we need a more appropriate structure:

Definition ([Bafna and Pevzner, 1996])

The **undirected breakpoint graph** of the permutation π in S_n, written $UBG(\pi) = (V, E)$, is defined by:

- $V = (\pi_0 = 0, \pi_1, \pi_2, \ldots, \pi_n, \pi_{n+1} = n + 1)$;
- $E = \underbrace{\{\{\pi_i, \pi_{i+1}\} \mid 0 \leq i \leq n\}}_{\text{black edges}} \cup \underbrace{\{\{i, i+1\} \mid 0 \leq i \leq n\}}_{\text{grey edges}}$.

The *undirected breakpoint graph*: example

- Let us build the undirected breakpoint graph $\pi = \langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$:

Example

3 1 5 4 2 8 6 7

The *undirected breakpoint graph*: example

- Let us build the undirected breakpoint graph $\pi = \langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$:

Example

0 3 1 5 4 2 8 6 7 **9**

1. frame the permutation;

The *undirected breakpoint graph*: example

- Let us build the undirected breakpoint graph $\pi = \langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$:

Example

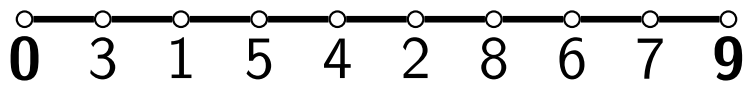
$$\overset{\circ}{\mathbf{0}}\ \overset{\circ}{3}\ \overset{\circ}{1}\ \overset{\circ}{5}\ \overset{\circ}{4}\ \overset{\circ}{2}\ \overset{\circ}{8}\ \overset{\circ}{6}\ \overset{\circ}{7}\ \overset{\circ}{\mathbf{9}}$$

1. frame the permutation;
2. build ordered vertex set ($\pi_0 = 0$, π_1, π_2, ..., $\pi_{n+1} = n + 1$);

The *undirected breakpoint graph*: example

- Let us build the undirected breakpoint graph $\pi = \langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$:

Example

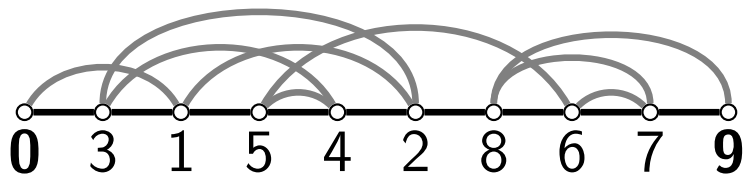


1. frame the permutation;
2. build ordered vertex set ($\pi_0 = 0$, π_1, π_2, ..., $\pi_{n+1} = n + 1$);
3. add **black edges** for every pair $\{\pi_i, \pi_{i+1}\}$;

The *undirected breakpoint graph*: example

- Let us build the undirected breakpoint graph $\pi = \langle 3\ 1\ 5\ 4\ 2\ 8\ 6\ 7 \rangle$:

Example

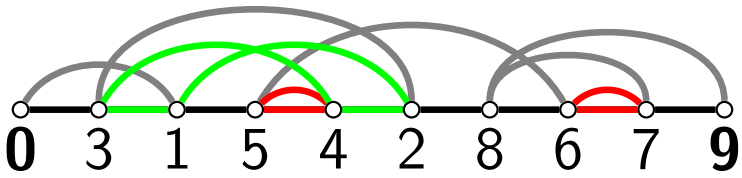


1. frame the permutation;
2. build ordered vertex set ($\pi_0 = 0$, π_1, π_2, ..., $\pi_{n+1} = n + 1$);
3. add **black edges** for every pair $\{\pi_i, \pi_{i+1}\}$;
4. add **grey edges** for every pair $\{i, i + 1\}$;

Decomposition

- That graph decomposes into cycles:

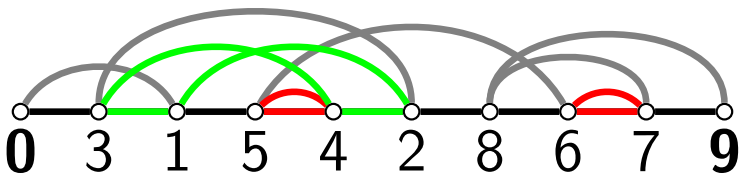
Example



Decomposition

- That graph decomposes into cycles:

Example

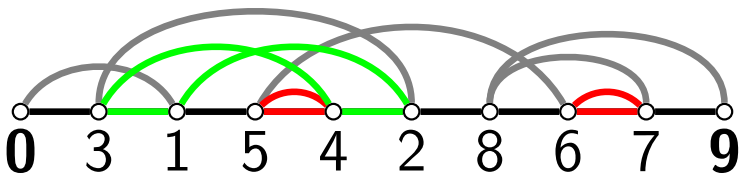


- ... but the decomposition is no longer unique!

Decomposition

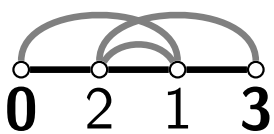
- That graph decomposes into cycles:

Example



- ... but the decomposition is no longer unique!

Example



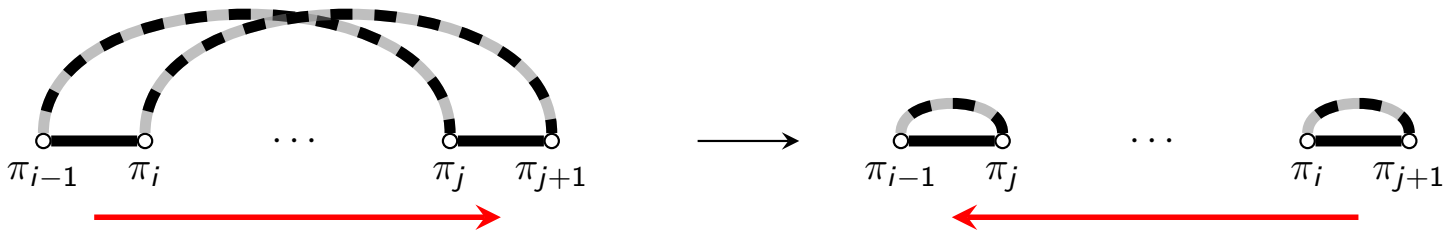
has either one 3-cycle or a 1-cycle and a 2-cycle.

Cycle decompositions

- We use decompositions to derive lower bounds;

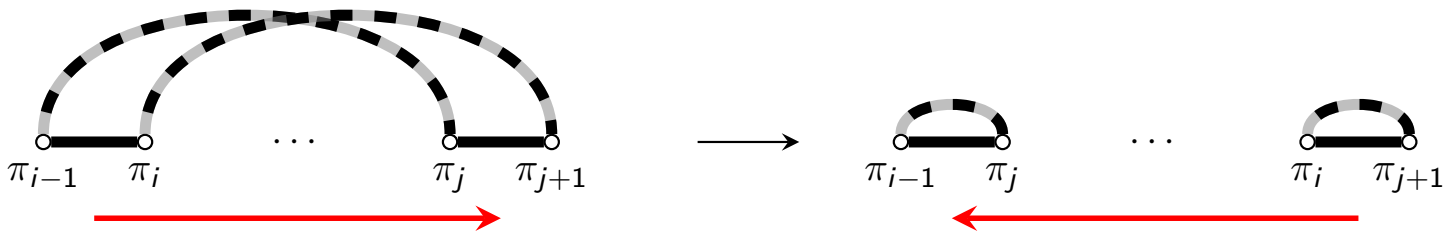
Cycle decompositions

- We use decompositions to derive lower bounds;
- A reversal acts on one or two cycles and can split / merge cycles;



Cycle decompositions

- We use decompositions to derive lower bounds;
- A reversal acts on one or two cycles and can split / merge cycles;



- So we're tempted to say:

$$rd(\pi) \geq n + 1 - c(UBG(\pi))$$

Decomposition

- ... but recall that the decomposition is not unique!!!

Decomposition

- ... but recall that the decomposition is not unique!!!
- The more cycles we have, the closer we are to $UBG(\iota)$;

Decomposition

- ... but recall that the decomposition is not unique!!!
- The more cycles we have, the closer we are to $UBG(\iota)$;
- Therefore, we have in fact:

Decomposition

- ... but recall that the decomposition is not unique!!!
- The more cycles we have, the closer we are to $UBG(\iota)$;
- Therefore, we have in fact:

Theorem ([Bafna and Pevzner, 1996])

For all π in S_n:

$$rd(\pi) \geq n + 1 - c^*(UBG(\pi)),$$

where $c^*(UBG(\pi))$ is the number of cycles in a maximum cardinality decomposition.

Decomposition

- ... but recall that the decomposition is not unique!!!
- The more cycles we have, the closer we are to $UBG(\iota)$;
- Therefore, we have in fact:

Theorem ([Bafna and Pevzner, 1996])

For all π in S_n:

$$rd(\pi) \geq n + 1 - c^*(UBG(\pi)),$$

where $c^*(UBG(\pi))$ is the number of cycles in a maximum cardinality decomposition.

- Unfortunately, finding a maximum cardinality decomposition is NP-hard [Caprara, 1999];

Results on sorting permutations

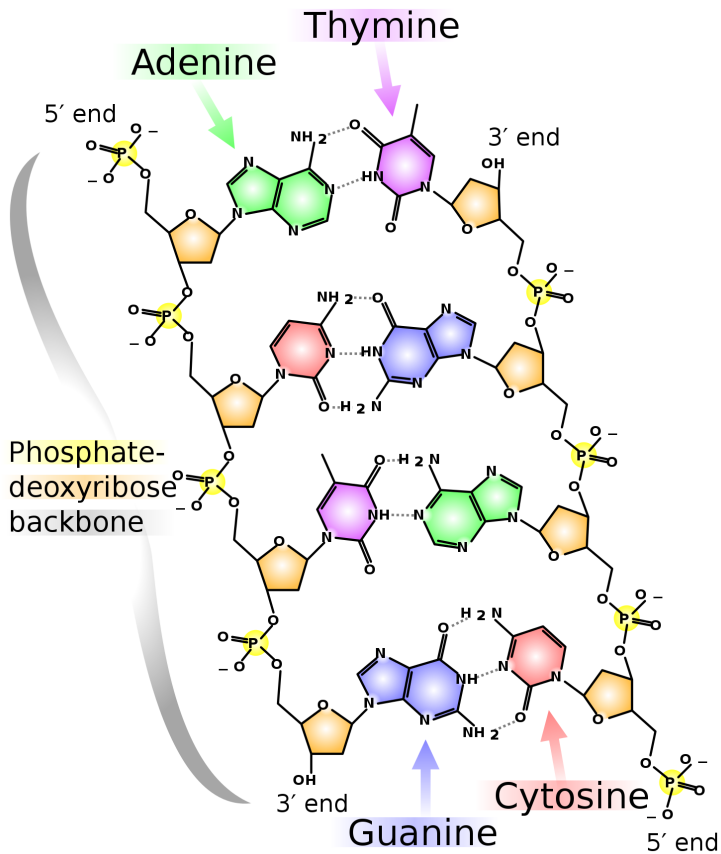
- Here's a nonexhaustive summary on sorting permutations using various operations:

	Operation	Sorting	Distance	Best approximation
	exchange	$O(n)$ [Knuth, 1995]		1
	block-interchange	$O(n)$ [Christie, 1996]		1
	double cut-and-joins	NP-hard [Chen, 2010]		?
	reversal	NP-hard [Caprara, 1999]		11/8 [Berman et al., 2002]
	transposition	NP-hard [Bulteau et al., 2012]		11/8 [Elias and Hartman, 2006]
prefix	exchange	$O(n)$ [Akers et al., 1987]		1
prefix	reversal	NP-hard [Bulteau et al., 2015]		2 [Fischer and Ginzinger, 2005]
prefix	transposition	?	?	2 [Dias and Meidanis, 2002]

- Let us move on to our next model: **signed** permutations;

Motivation

- Permutations lack realism: DNA segments are oriented;
- We need to take orientation into account;
- Therefore, two DNA segments match if:
  - they are the same, or
  - one is the **reverse complement** of the other.

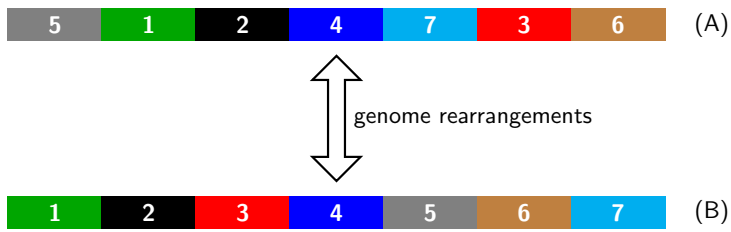


(picture by Madeleine Price Ball, taken from Wikimedia)

Motivation

- So instead of permutations:

Example (genomes → permutations)

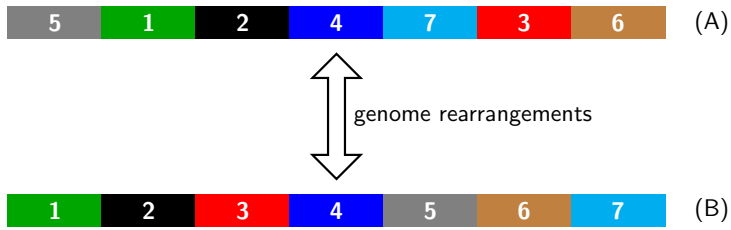


- We now have *signed* permutations:

Motivation

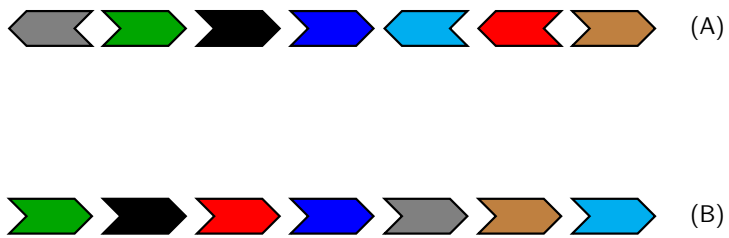
- So instead of permutations:

Example (genomes → permutations)



- We now have *signed* permutations:

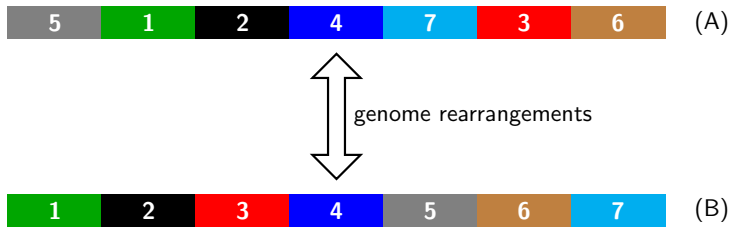
Example (genomes → signed permutations)



Motivation

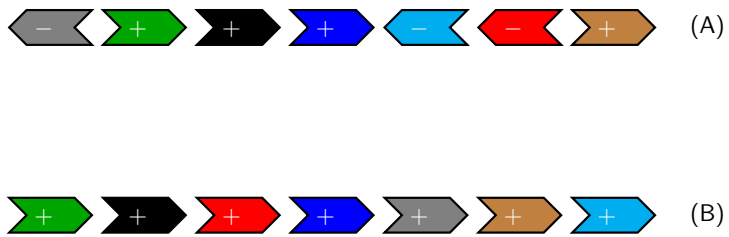
- So instead of permutations:

Example (genomes → permutations)



- We now have *signed* permutations:

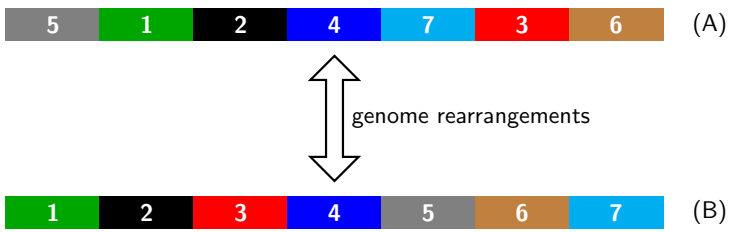
Example (genomes → signed permutations)



Motivation

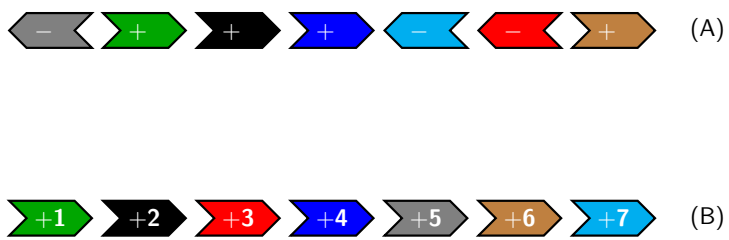
- So instead of permutations:

Example (genomes → permutations)



- We now have *signed* permutations:

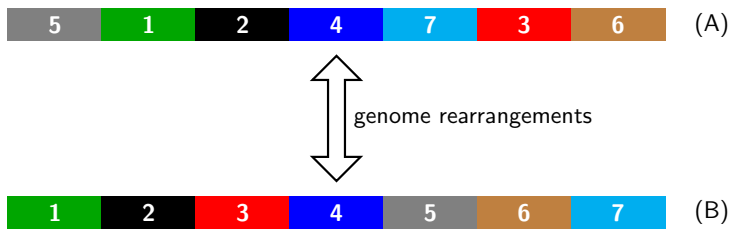
Example (genomes → signed permutations)



Motivation

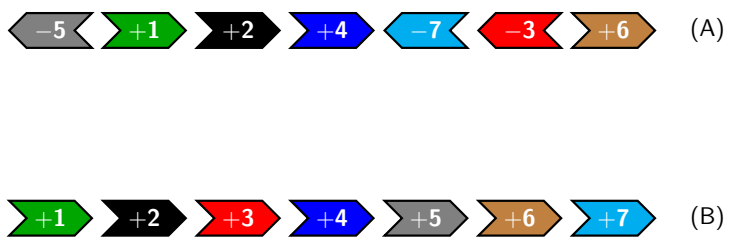
- So instead of permutations:

Example (genomes → permutations)



- We now have *signed* permutations:

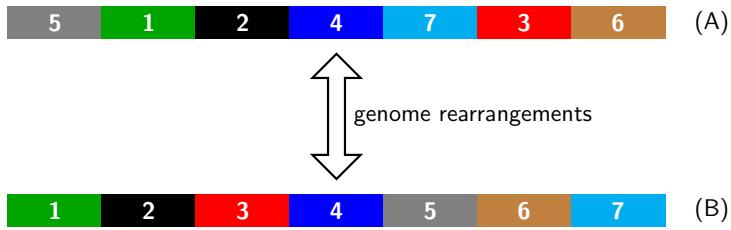
Example (genomes → signed permutations)



Motivation

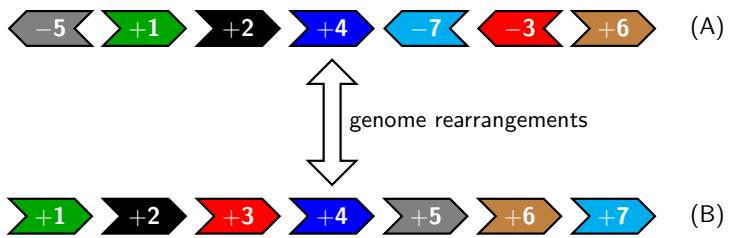
- So instead of permutations:

Example (genomes → permutations)



- We now have *signed* permutations:

Example (genomes → signed permutations)



Signed (per)mutations

- Note: this does not mean that everything you know about unsigned comparisons is useless:
  1. orientation information is not always available;
  2. ideas from unsigned comparisons lead to ideas for signed comparisons;
- Mutations may now act on a segment's place **and** orientation;

Tools

- As expected, tools we've seen previously cannot be used here because they do not take signs into account;
- Then again, some ideas can be adapted;

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;
- Convention: $\pi(-i) = -\pi(i)$ for $1 \leq i \leq n$;

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;
- Convention: $\pi(-i) = -\pi(i)$ for $1 \leq i \leq n$;
- Permutations can be written in one- or two-row notation:

$$\pi = \left\langle \begin{array}{cccccccc} -4 & -3 & -2 & -1 & 1 & 2 & 3 & 4 \\ 2 & 4 & -1 & 3 & -3 & 1 & -4 & -2 \end{array} \right\rangle = \langle -3\ 1\ -4\ -2 \rangle.$$

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;
- Convention: $\pi(-i) = -\pi(i)$ for $1 \leq i \leq n$;
- Permutations can be written in one- or two-row notation:

$$\pi = \left\langle \begin{array}{cccccccc} -4 & -3 & -2 & -1 & 1 & 2 & 3 & 4 \\ 2 & 4 & -1 & 3 & -3 & 1 & -4 & -2 \end{array} \right\rangle = \langle -3\ 1\ -4\ -2 \rangle.$$

- We will restrict ourselves to the mapping of positive elements;

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;
- Convention: $\pi(-i) = -\pi(i)$ for $1 \leq i \leq n$;
- Permutations can be written in one- or two-row notation:

$$\pi = \begin{pmatrix} -4 & -3 & -2 & -1 & 1 & 2 & 3 & 4 \\ 2 & 4 & -1 & 3 & -3 & 1 & -4 & -2 \end{pmatrix} = \langle -3\ 1\ -4\ -2 \rangle.$$

- We will restrict ourselves to the mapping of positive elements;
- Composition works as before;

Notation and definitions pertaining to signed permutations

- We deal exclusively with $\{\pm 1, \pm 2, \ldots, \pm n\}$;
- Convention: $\pi(-i) = -\pi(i)$ for $1 \leq i \leq n$;
- Permutations can be written in one- or two-row notation:

$$\pi = \begin{pmatrix} -4 & -3 & -2 & -1 & 1 & 2 & 3 & 4 \\ 2 & 4 & -1 & 3 & -3 & 1 & -4 & -2 \end{pmatrix} = \langle -3\ 1\ -4\ -2 \rangle.$$

- We will restrict ourselves to the mapping of positive elements;
- Composition works as before;
- The corresponding group is the **hyperoctahedral group** $S_n^{\pm}$;

Signed reversals

- Reversals can be generalised to **signed reversals**, which not only reverse an interval but also flip signs along the interval;

Signed reversals

- Reversals can be generalised to **signed reversals**, which not only reverse an interval but also flip signs along the interval;

Example (signed reversal)

-5	1	2	4	-7	-3	6

-5	1	3	7	-4	-2	6

Signed reversals

- Reversals can be generalised to **signed reversals**, which not only reverse an interval but also flip signs along the interval;

Example (signed reversal)

-5 1 2 4 -7 -3 6

-5 1 3 7 -4 -2 6

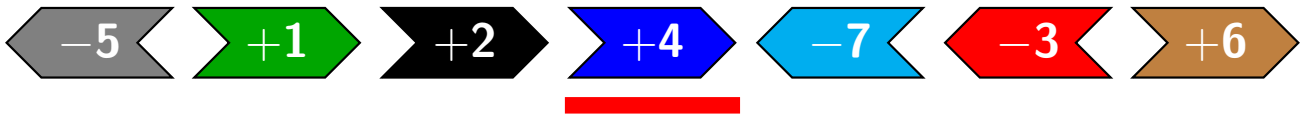
- As before, we're interested in sorting a given signed permutation using as few signed reversals as possible (or merely computing the length of a shortest sequence);

Example: sorting by signed reversals

$\pi =$ 

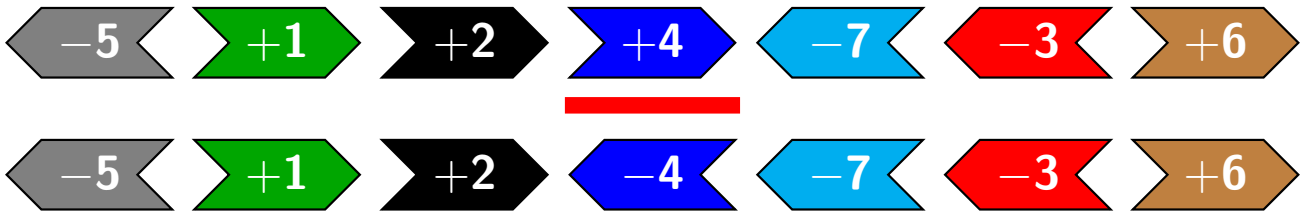
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

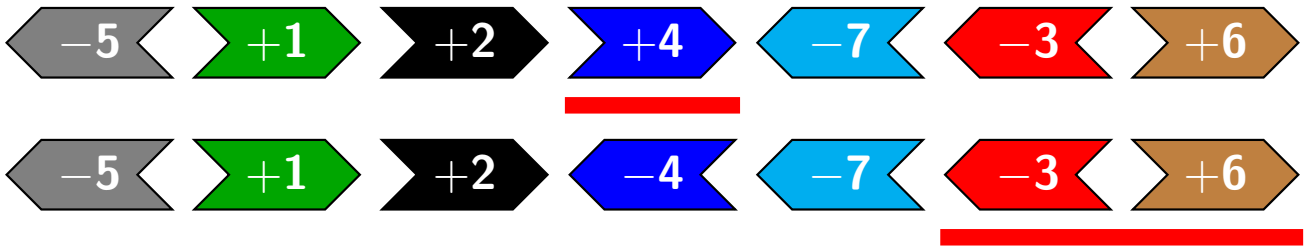
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

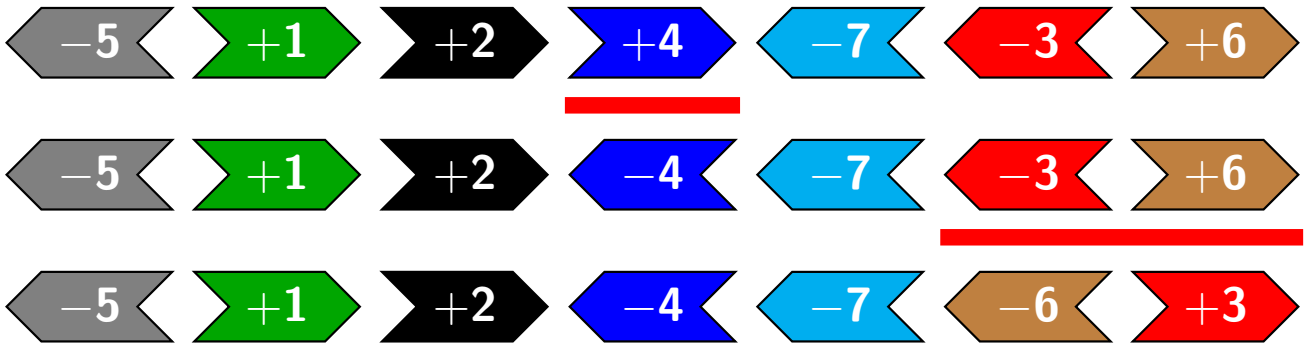
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

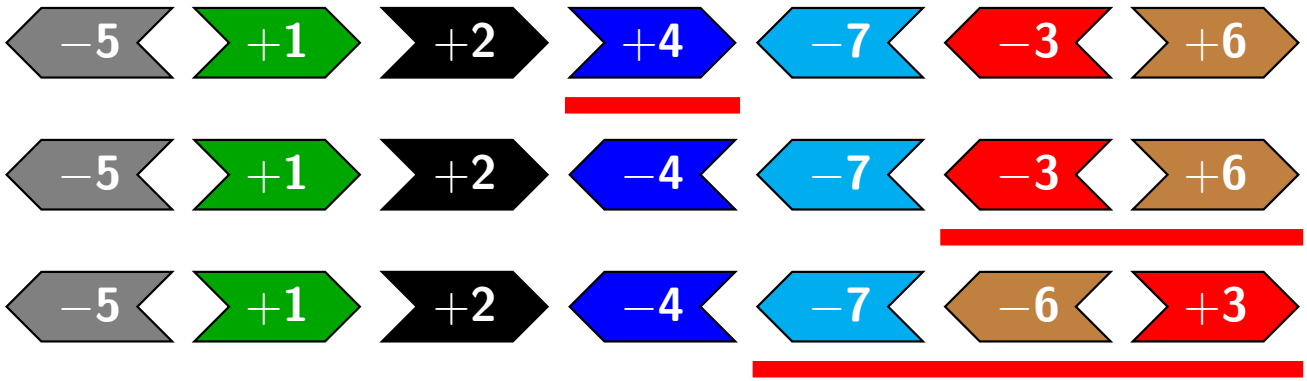
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

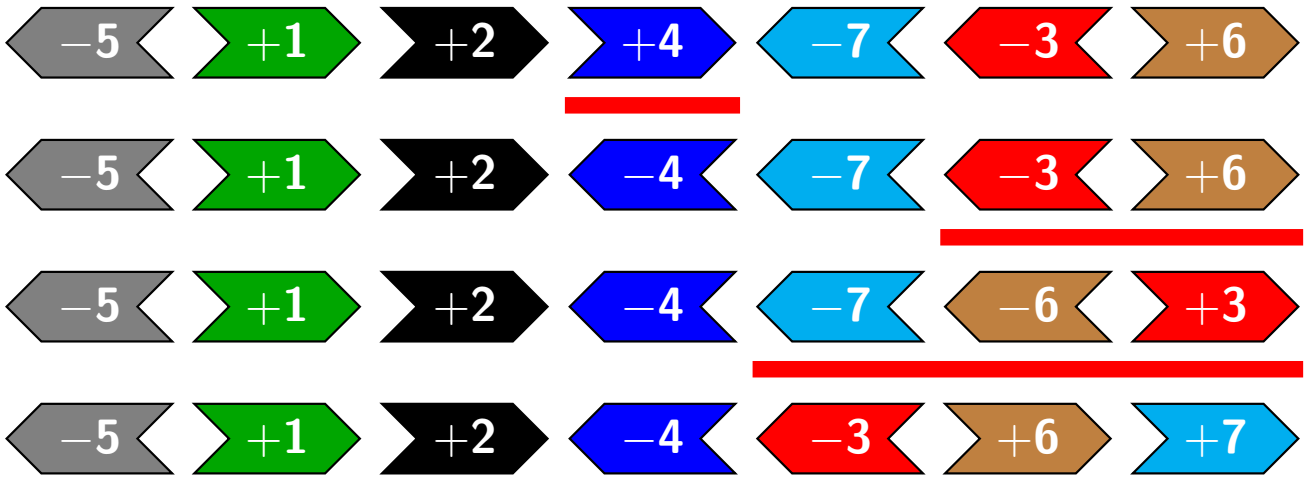
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

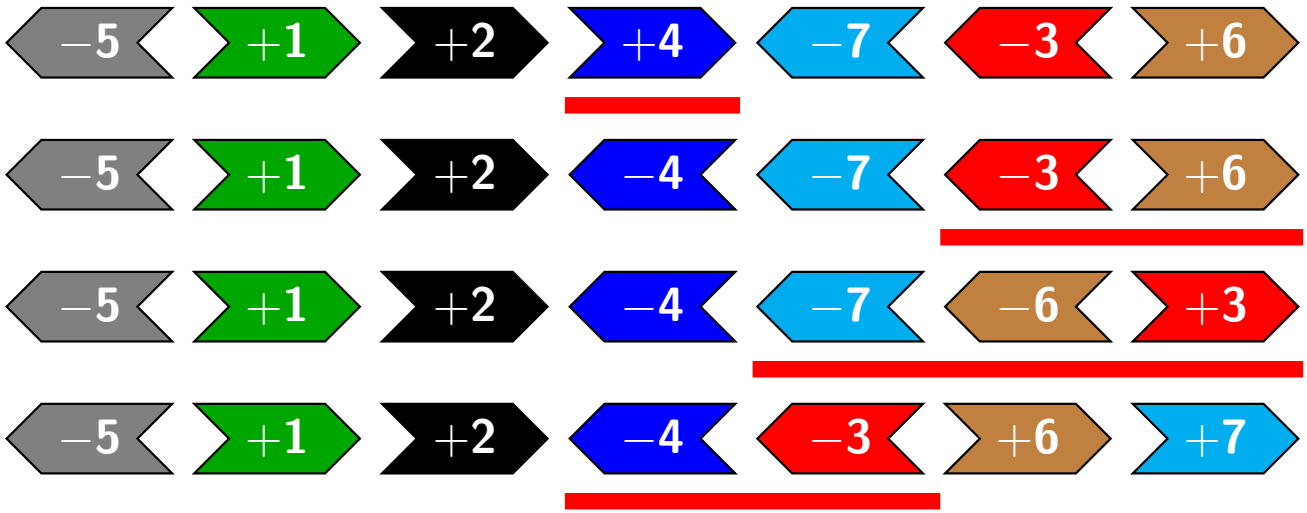
$\sigma =$ 

Example: sorting by signed reversals

$\pi =$ 

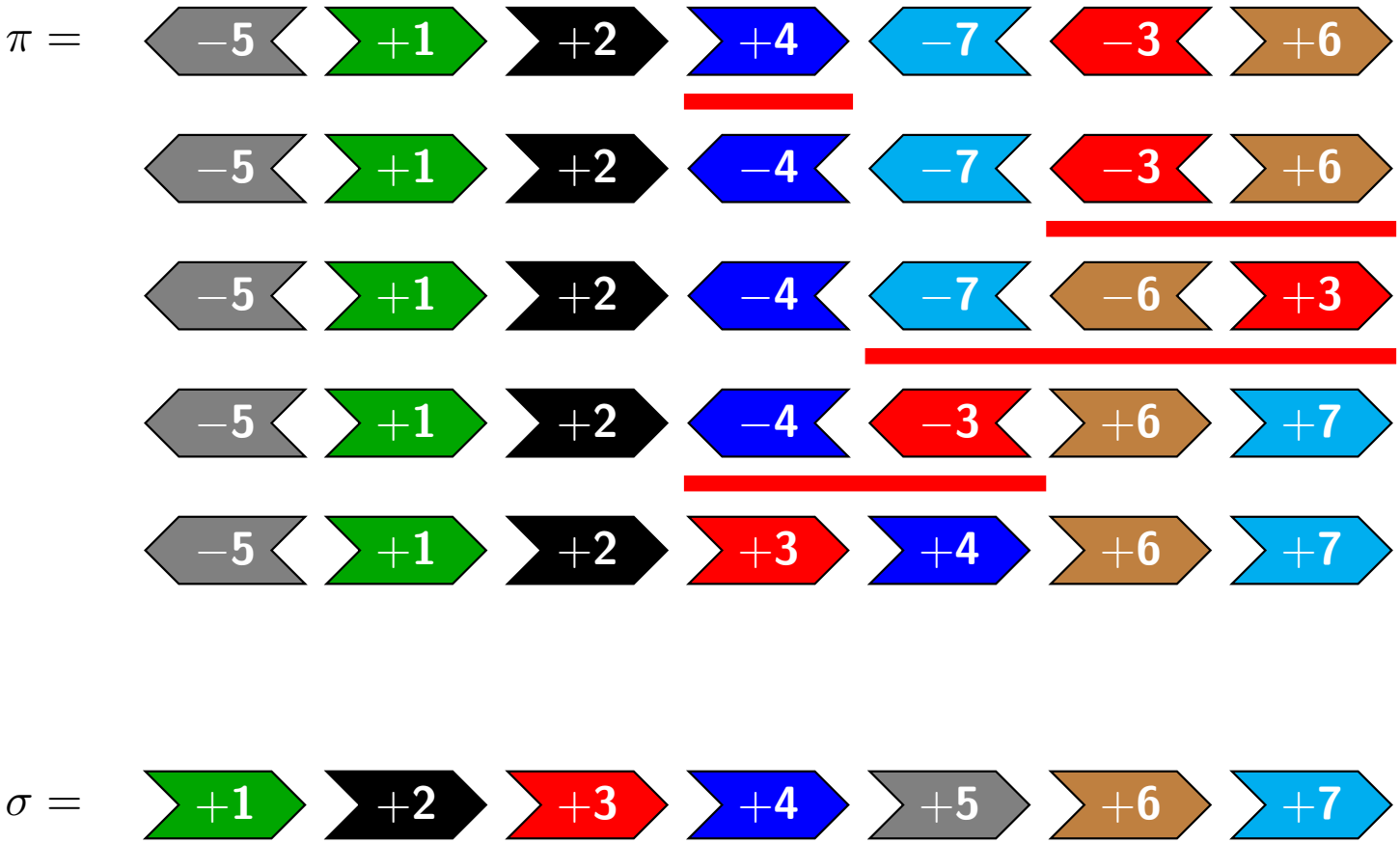
$\sigma =$ 

Example: sorting by signed reversals

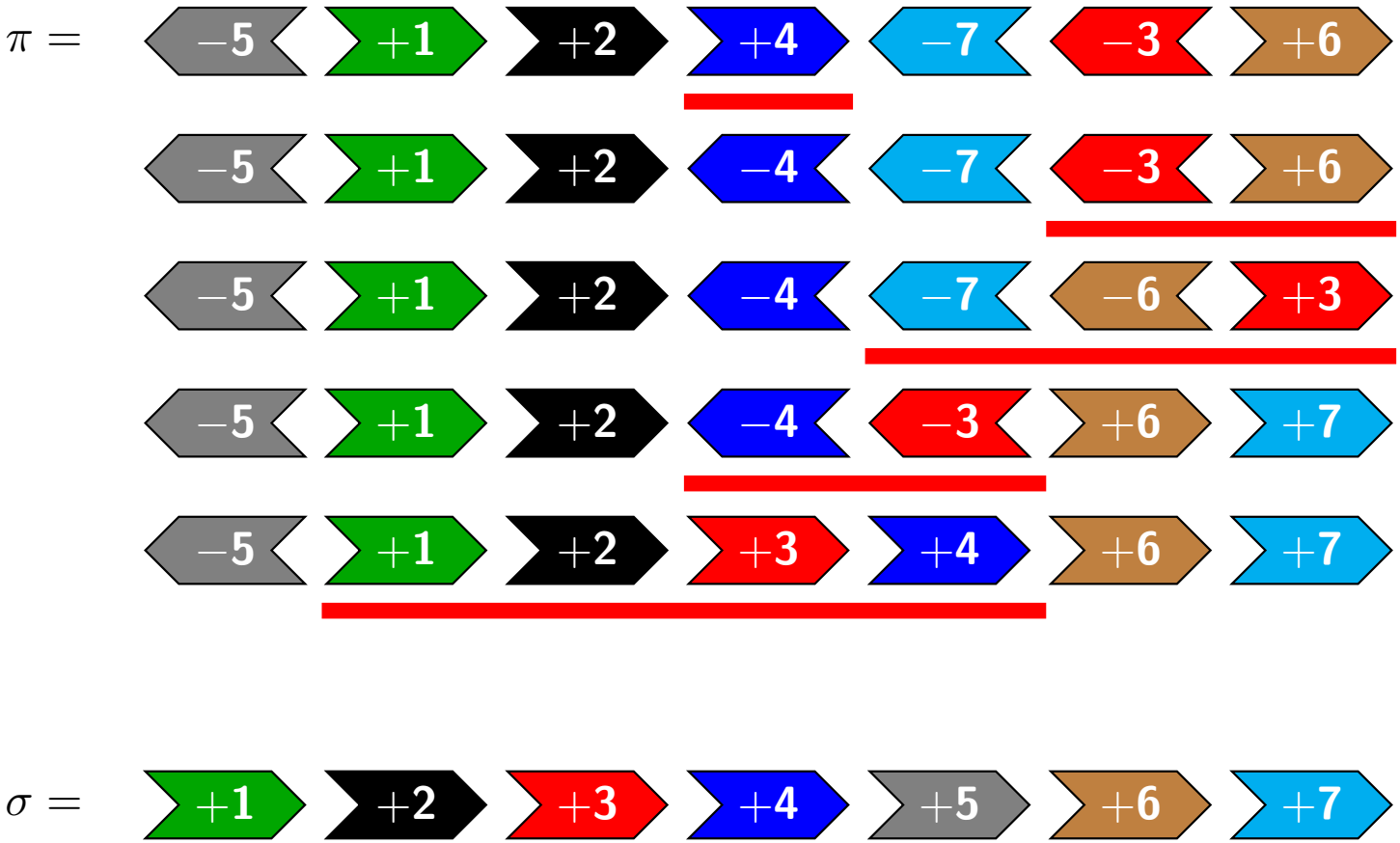
$\pi =$ 

$\sigma =$ 

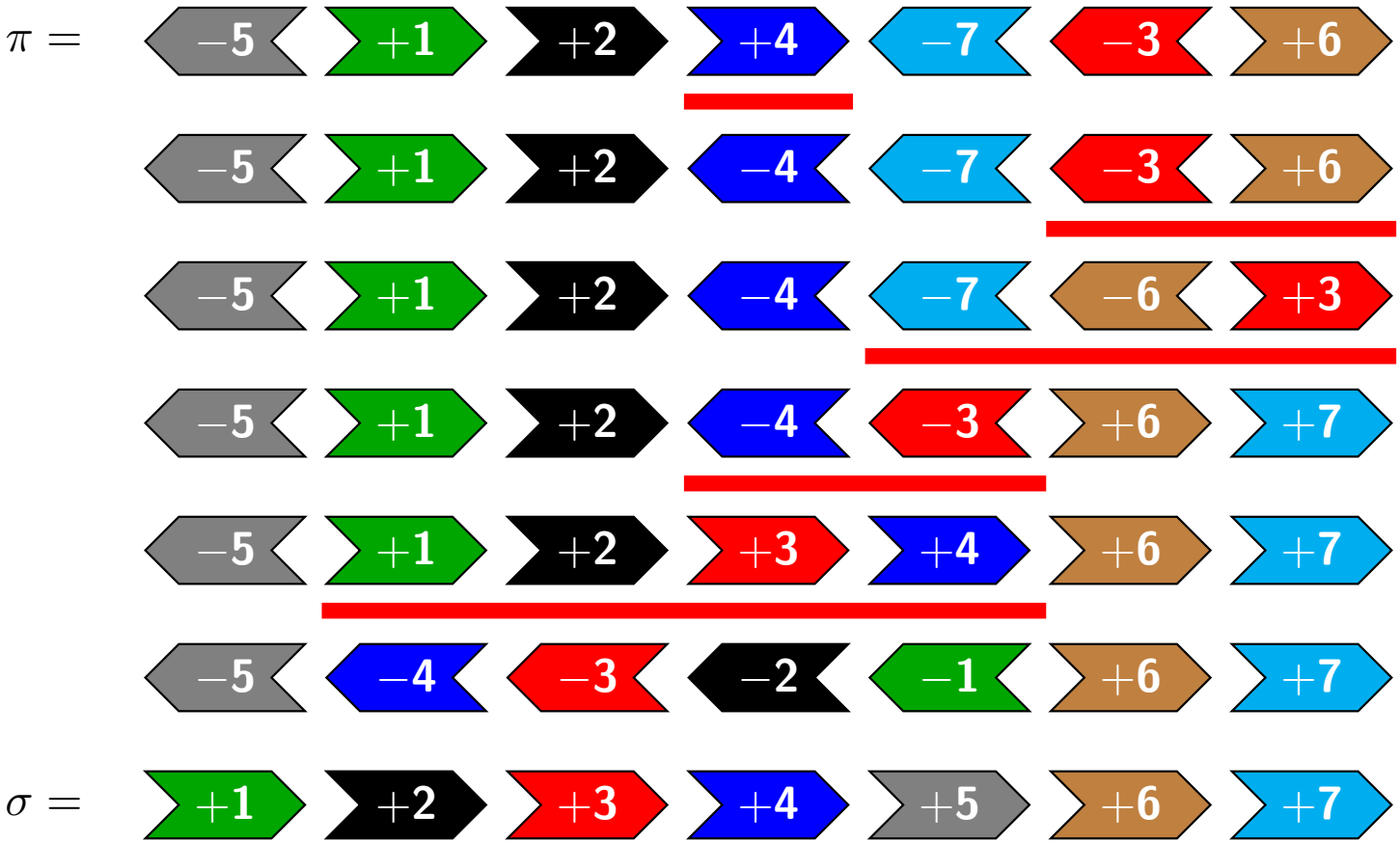
Example: sorting by signed reversals



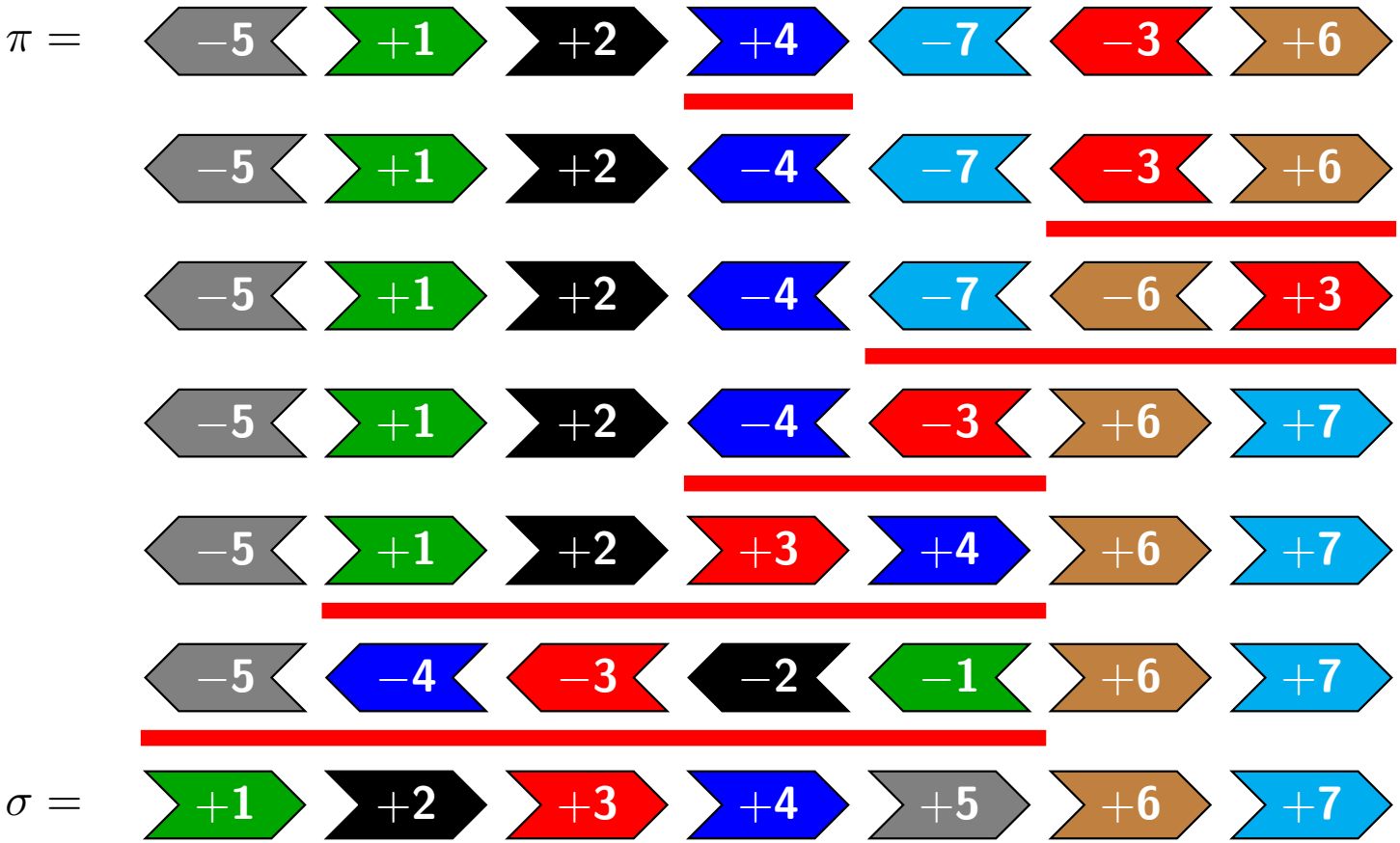
Example: sorting by signed reversals



Example: sorting by signed reversals



Example: sorting by signed reversals



$$srd(\pi, \sigma) \leq 6$$

Solving the problem

- How do we attack this problem?

Solving the problem

- How do we attack this problem?
- Breakpoints can be generalised to the signed setting ...

Solving the problem

- How do we attack this problem?
- Breakpoints can be generalised to the signed setting ...
- ... but you already know / guess that this will at best provide an approximation;

Solving the problem

- How do we attack this problem?
- Breakpoints can be generalised to the signed setting ...
- ... but you already know / guess that this will at best provide an approximation;
- Instead, we're going to adapt the breakpoint graph to the signed setting;

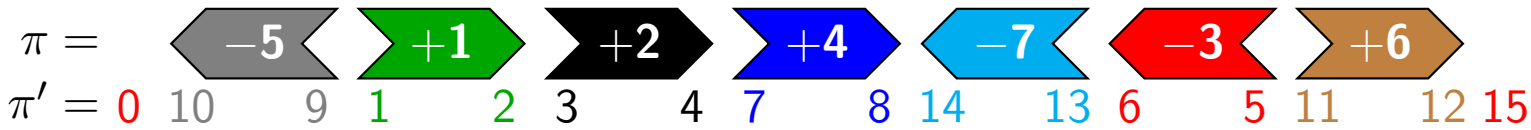
The *breakpoint graph*

- The breakpoint graph in the signed case is slightly different:

$\pi =$ 

The *breakpoint graph*

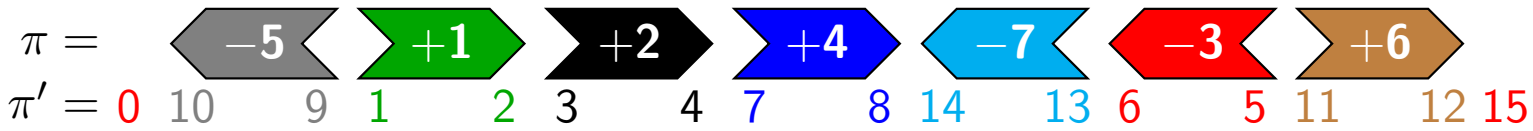
- The breakpoint graph in the signed case is slightly different:



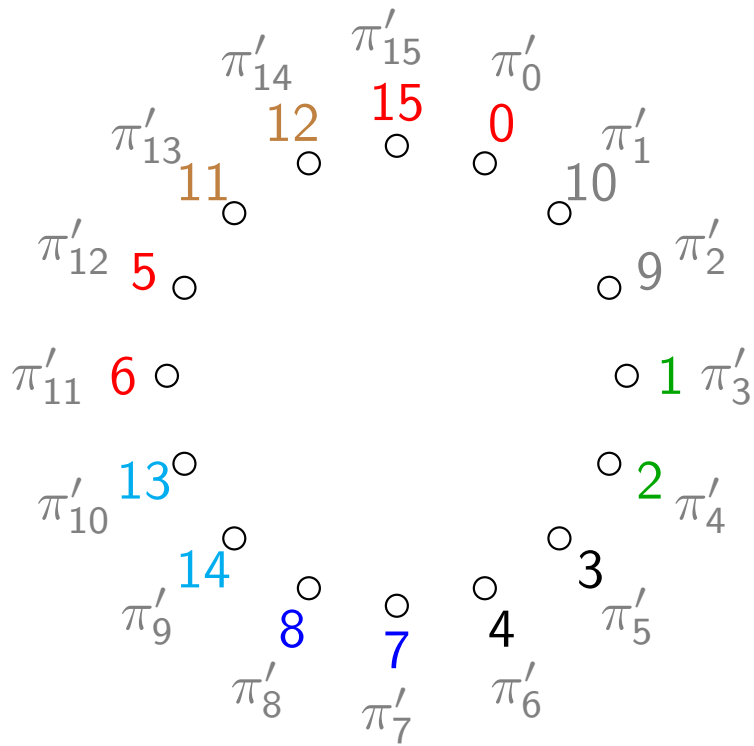
1. double π's elements
   $(i \mapsto \{2|i| - 1, 2|i|\})$ and add **0**
   and **2n + 1**

The *breakpoint graph*

- The breakpoint graph in the signed case is slightly different:

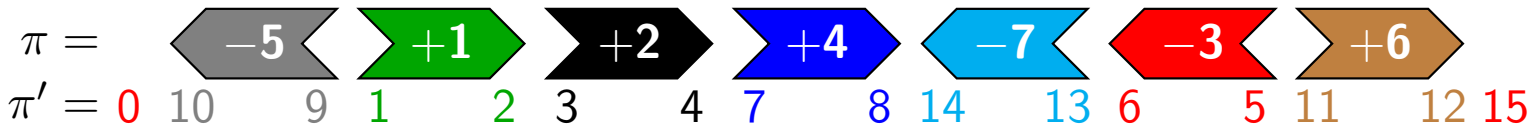


1. double π's elements
   $(i \mapsto \{2|i| - 1, 2|i|\})$ and add **0**
   and $\mathbf{2n + 1}$
2. elements of π' = vertices

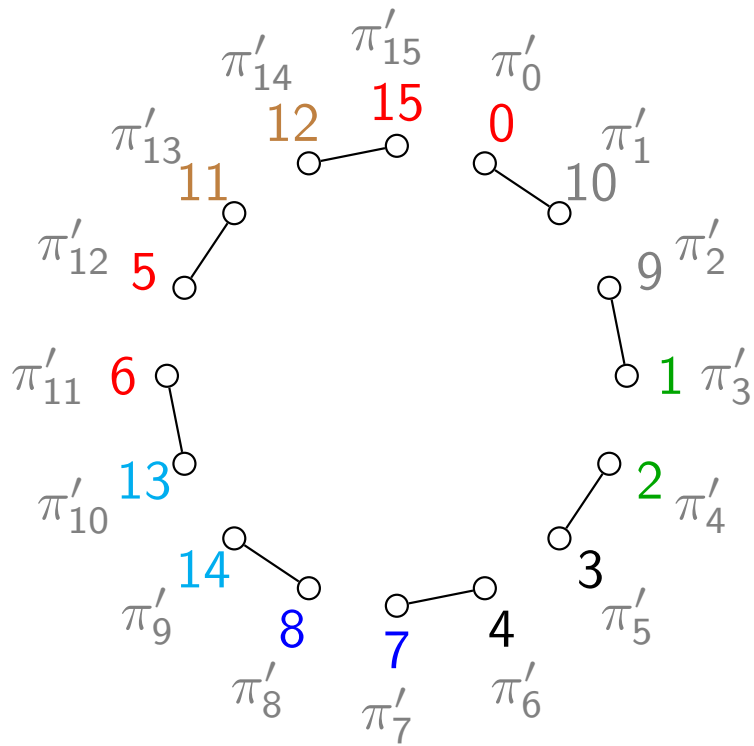


The *breakpoint graph*

- The breakpoint graph in the signed case is slightly different:

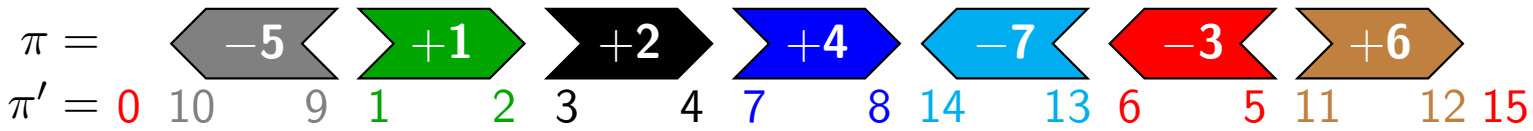


1. double π's elements ($i \mapsto \{2|i| - 1, 2|i|\}$) and add **0** and **2n + 1**
2. elements of π' = vertices
3. **black edges** connect distinct adjacent genes

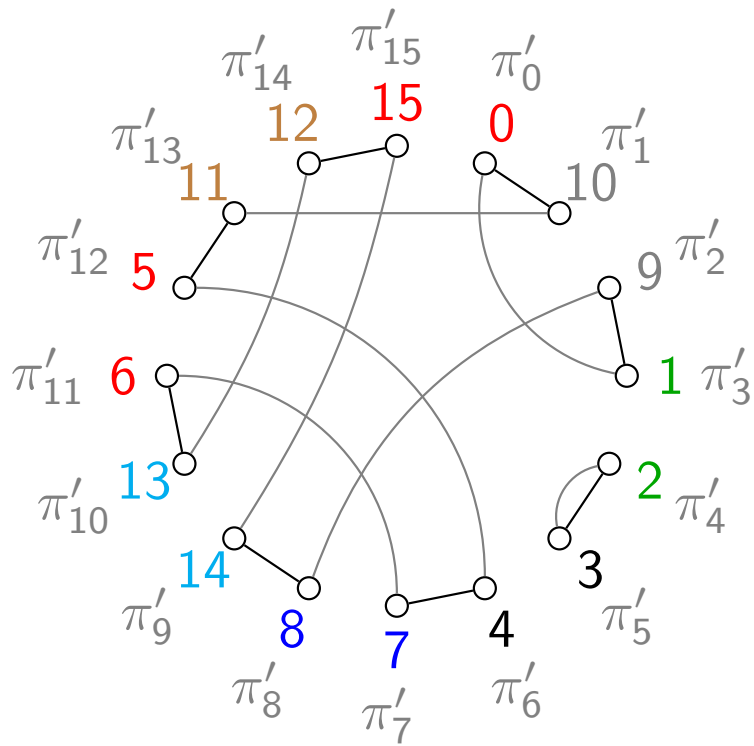


The *breakpoint graph*

- The breakpoint graph in the signed case is slightly different:

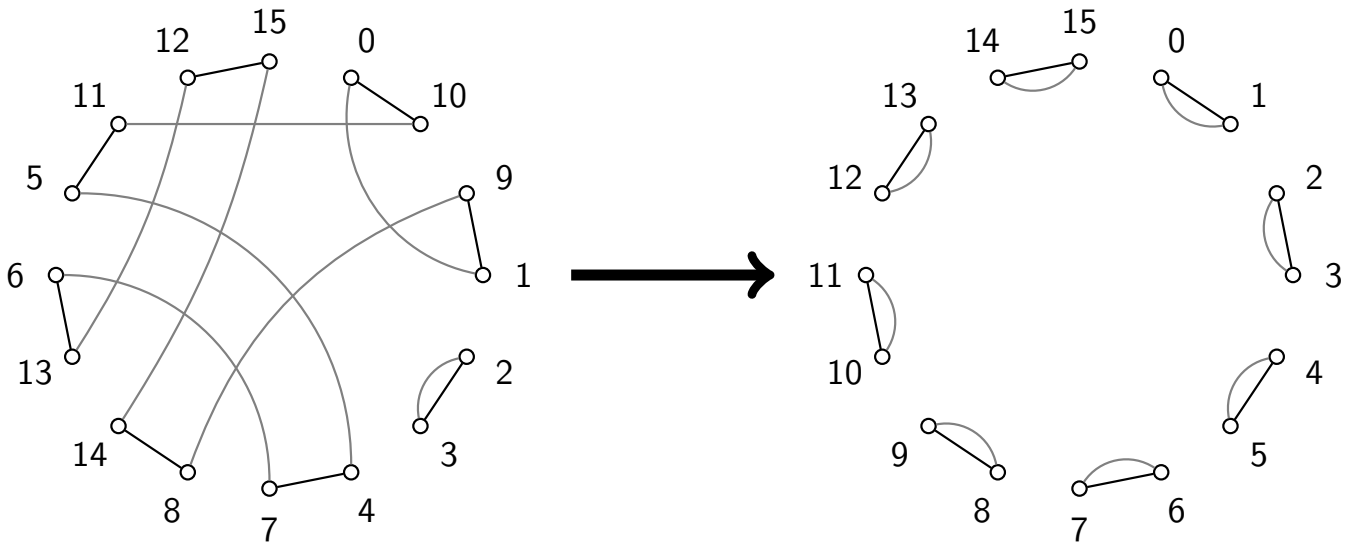


1. double π's elements ($i \mapsto \{2|i| - 1, 2|i|\}$) and add **0** and **2n + 1**
2. elements of π' = vertices
3. **black edges** connect distinct adjacent genes
4. **grey edges** connect distinct consecutive genes



Using the breakpoint graph

- The breakpoint graph is 2-regular and decomposes as such into **alternating cycles** in a unique way;
- The breakpoint graph of $\langle 1\ 2\ \cdots\ n \rangle$ contains the largest number of cycles:



- $\Rightarrow$ goal: create new cycles in as few moves as possible;

Practice

8. Draw the breakpoint graph for the signed permutation

$$\langle -4, 3, -2, -5, 1 \rangle$$

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);
  2. Find an optimal sorting sequence for $\tilde{\pi}$;

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);
  2. Find an optimal sorting sequence for $\tilde{\pi}$;
  3. Convert it back to an optimal sorting sequence for π;

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);
  2. Find an optimal sorting sequence for $\tilde{\pi}$;
     2.1 identify "good" and "bad" cycles in $BG(\tilde{\pi})$;
  3. Convert it back to an optimal sorting sequence for π;

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);
  2. Find an optimal sorting sequence for $\tilde{\pi}$;
     - 2.1 identify "good" and "bad" cycles in $BG(\tilde{\pi})$;
     - 2.2 identify "good" and "bad" components in $BG(\tilde{\pi})$;
  3. Convert it back to an optimal sorting sequence for π;

Overview of Hannenhalli and Pevzner's solution

- [Hannenhalli and Pevzner, 1999] came up with the first polynomial-time algorithm for this problem:
  1. Transform π into $\tilde{\pi}$ (simpler, does not affect distance);
  2. Find an optimal sorting sequence for $\tilde{\pi}$;
     - 2.1 identify "good" and "bad" cycles in $BG(\tilde{\pi})$;
     - 2.2 identify "good" and "bad" components in $BG(\tilde{\pi})$;
     - 2.3 "sort" those components to optimality;
  3. Convert it back to an optimal sorting sequence for π;

Transformation into simple permutations

- A permutation π is **simple** if $BG(\pi)$ contains only cycles of length ≤ 2;

Transformation into simple permutations

- A permutation π is **simple** if $BG(\pi)$ contains only cycles of length ≤ 2;
- The transformation was introduced to simplify analysis and preserves the distance: if $\tilde{\pi}$ is the "simplified" version of π, then $srd(\tilde{\pi}) = srd(\pi)$;

Transformation into simple permutations

- A permutation π is **simple** if $BG(\pi)$ contains only cycles of length ≤ 2;
- The transformation was introduced to simplify analysis and preserves the distance: if $\tilde{\pi}$ is the "simplified" version of π, then $srd(\tilde{\pi}) = srd(\pi)$;
- So we can assume from now on that the permutation to sort is simple;

Transformation into simple permutations

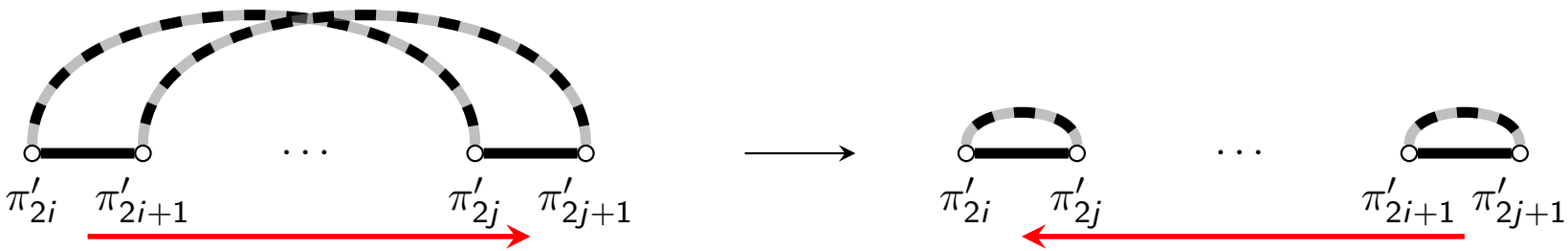
- A permutation π is **simple** if $BG(\pi)$ contains only cycles of length ≤ 2;
- The transformation was introduced to simplify analysis and preserves the distance: if $\tilde{\pi}$ is the "simplified" version of π, then $srd(\tilde{\pi}) = srd(\pi)$;
- So we can assume from now on that the permutation to sort is simple;
- [Gog and Bader, 2008] give fast algorithms to achieve the conversions;

A lower bound on the signed reversal distance

- A signed reversal involves black edges belonging to at most two cycles;

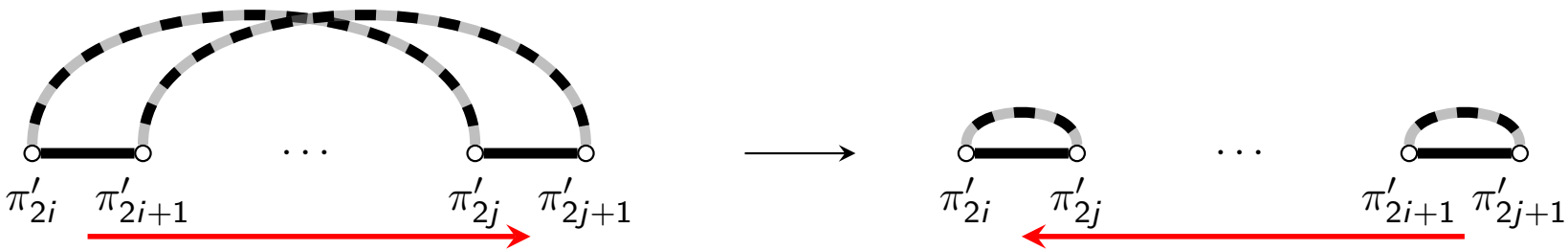
A lower bound on the signed reversal distance

- A signed reversal involves black edges belonging to at most two cycles;
- The only way to increase $c(BG(\pi))$ is to split cycles:



A lower bound on the signed reversal distance

- A signed reversal involves black edges belonging to at most two cycles;
- The only way to increase $c(BG(\pi))$ is to split cycles:



- Therefore, for all π in $S_n^{\pm}$:

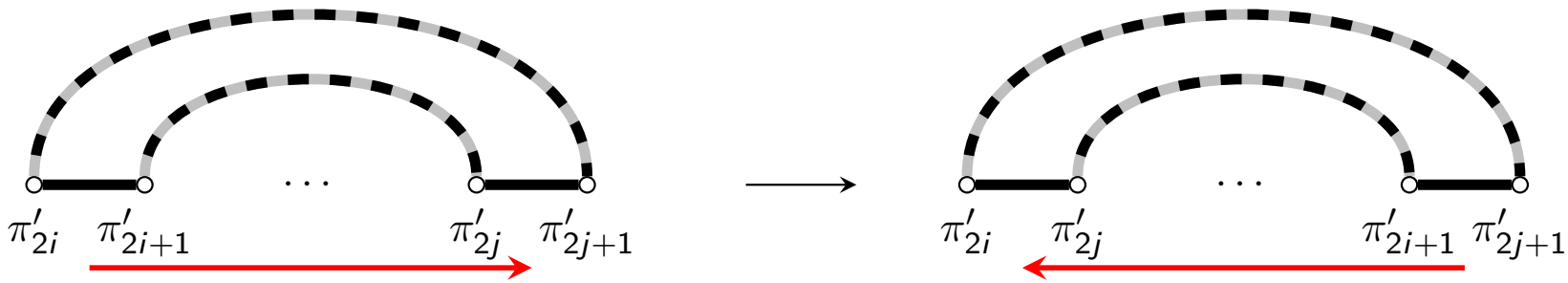
$$srd(\pi) \geq n + 1 - c(BG(\pi)).$$

Good and bad cycles

- However, we cannot always split cycles:

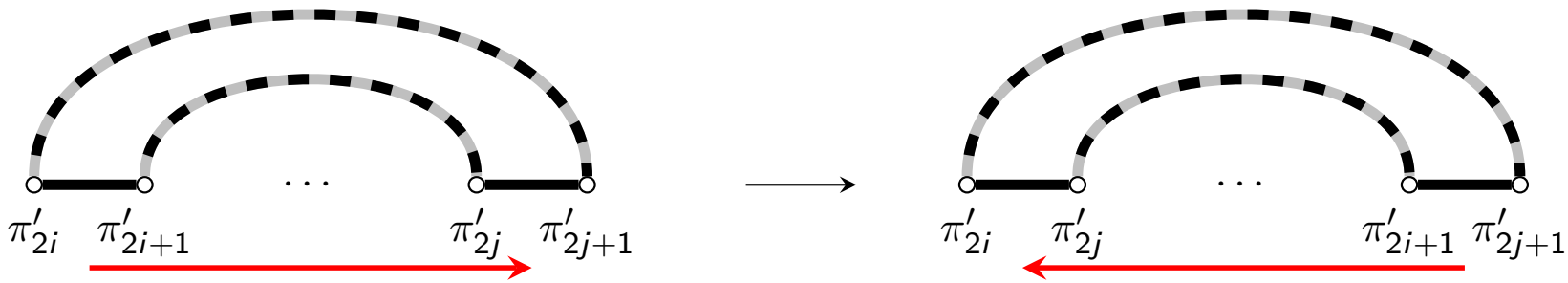
Good and bad cycles

- However, we cannot always split cycles:



Good and bad cycles

- However, we cannot always split cycles:

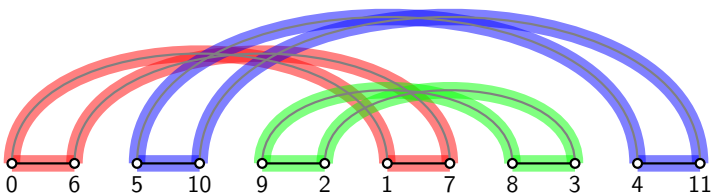


- Hence the inequality: we can split "good" cycles, and we cannot split "bad" cycles;
  - standard terminology: "good" = **oriented**, "bad" = **unoriented**;

Handling good cycles

- Things are actually more complicated than that;
- Even when we don't have bad cycles, the order in which we do things matters!

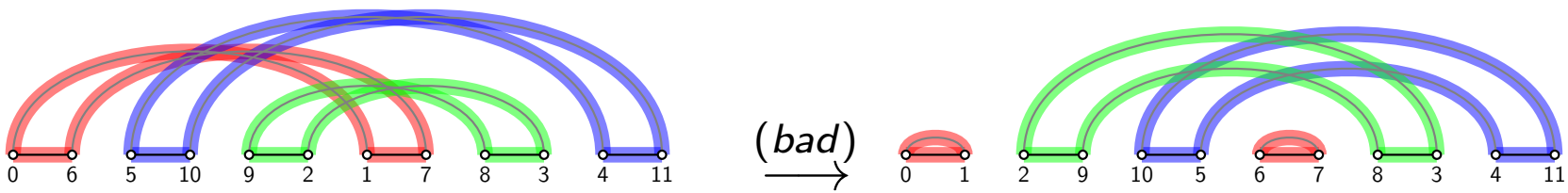
Example (careless and careful cycle splitting)



Handling good cycles

- Things are actually more complicated than that;
- Even when we don't have bad cycles, the order in which we do things matters!

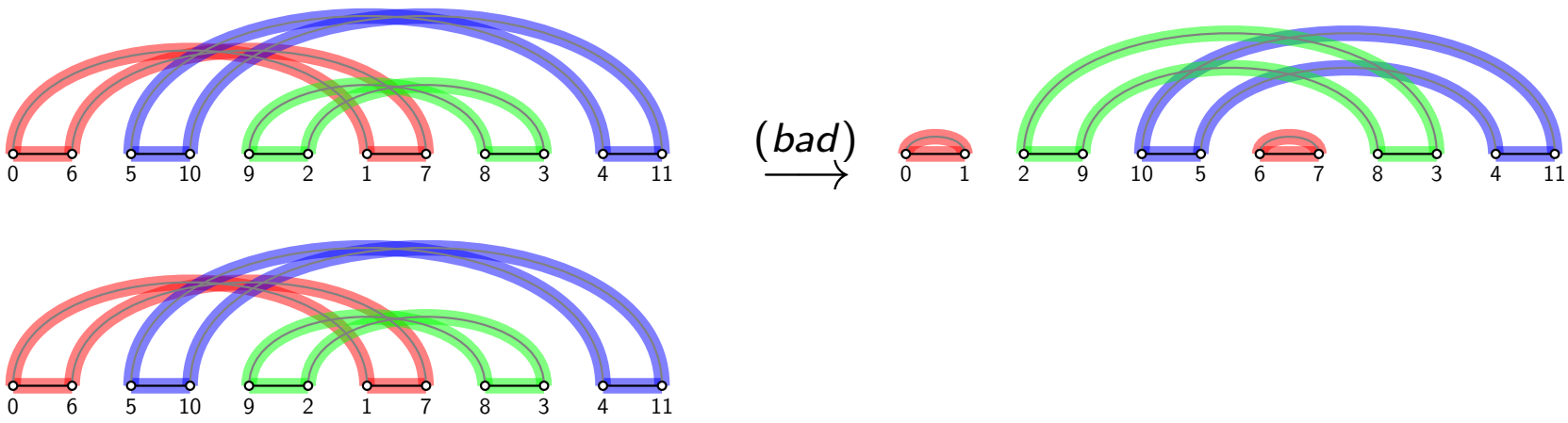
Example (careless and careful cycle splitting)



Handling good cycles

- Things are actually more complicated than that;
- Even when we don't have bad cycles, the order in which we do things matters!

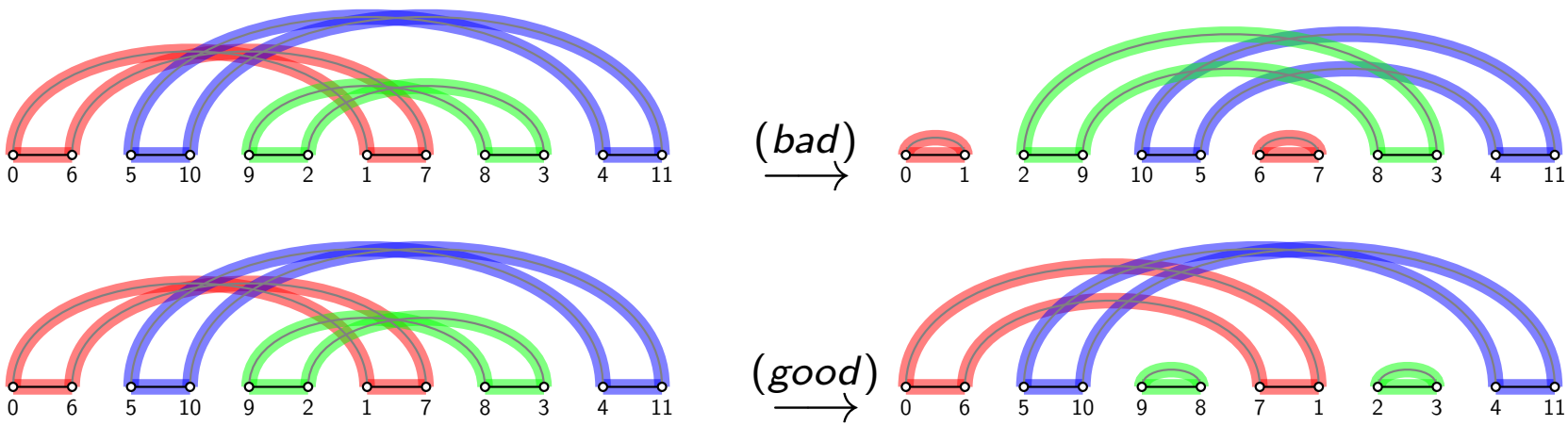
Example (careless and careful cycle splitting)



Handling good cycles

- Things are actually more complicated than that;
- Even when we don't have bad cycles, the order in which we do things matters!

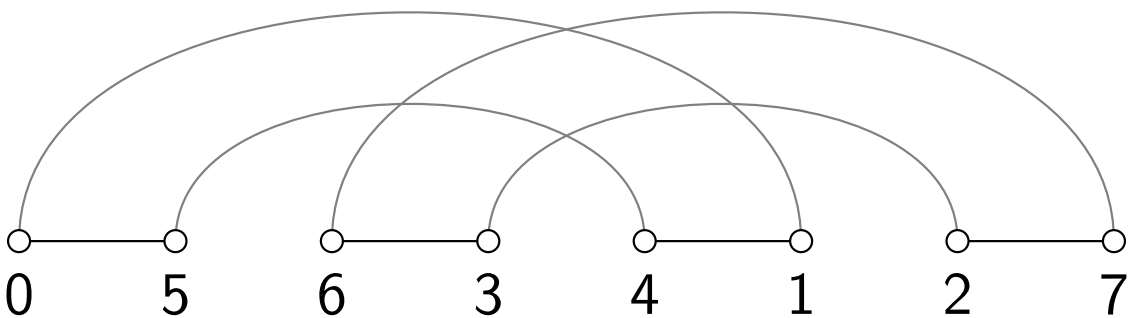
Example (careless and careful cycle splitting)



Handling bad cycles

- "Bad" cycles are not so bad if we can make them "good" (see previous example);
- But sometimes we can't:

Example (a minimal permutation with only bad cycles)



Components

- Although reversals only modify the "contents" of a single cycle, they may also modify the configuration of some other cycles;
- This suggests that cycles are not the right "unit" to deal with;
- We need to consider collections of cycles, or **components** instead;

Interleaving cycles

- Grey edge $\{\pi'_i, \pi'_j\}$ **spans** the interval $[i, j]$ in π';
- Two grey edges **interleave** if their spans intersect properly;
- Two cycles **interleave** if they contain interleaving edges;
- The **interleaving graph** I_π is defined by:
  - $V(I_\pi) =$ cycles of $BG(\pi)$;
  - $E(I_\pi) =$ pairs of interleaving cycles in $BG(\pi)$;
- A **component** of the breakpoint graph is a connected component of the interleaving graph;

The interleaving graph I_π

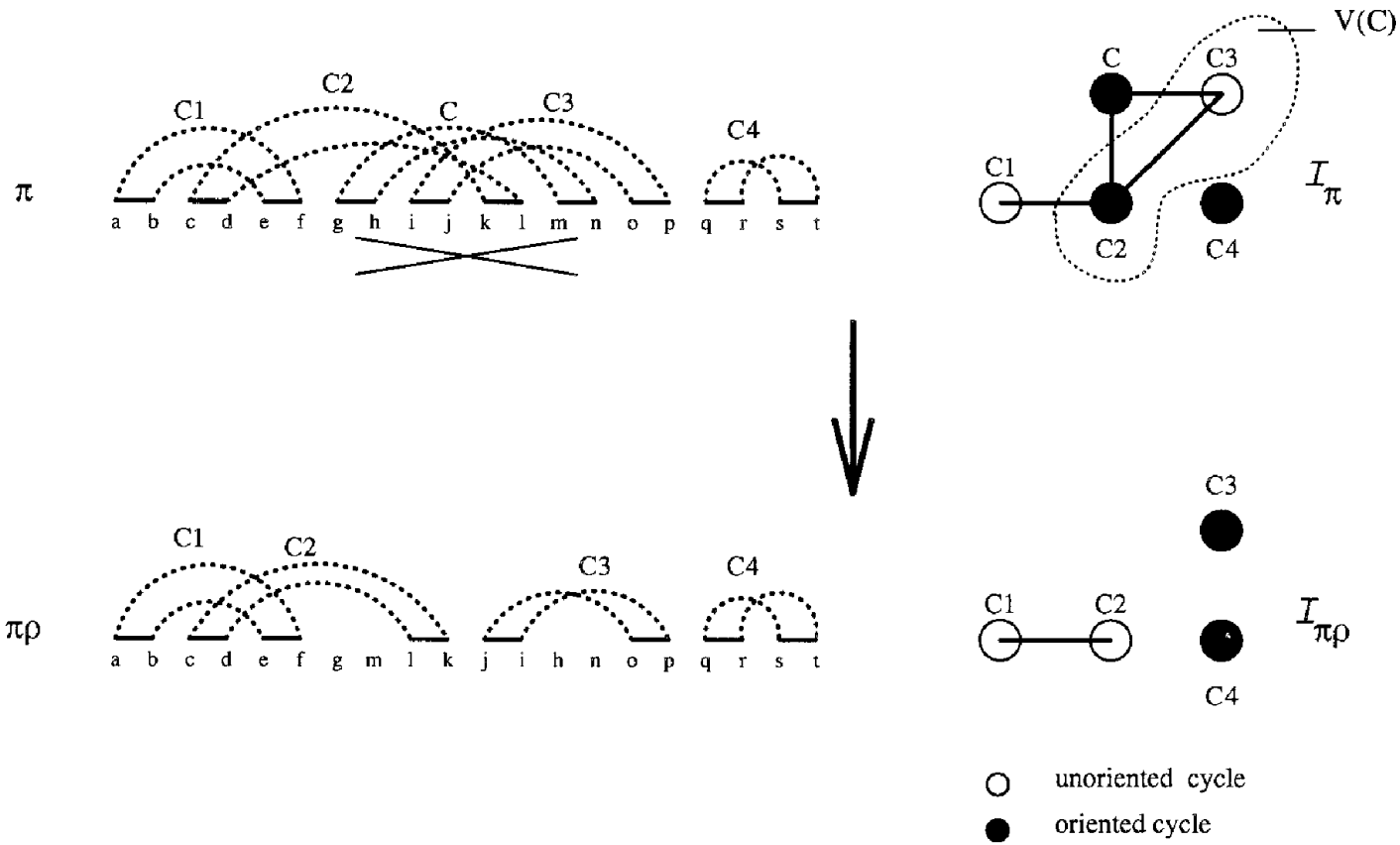


FIG. 8. Reversal on a cycle C (i) deletes vertex C from the interleaving graph; (ii) changes the orientation of vertices in $V(C)$; (iii) complements the subgraph induced by $V(C)$.

(source: [Hannenhalli and Pevzner, 1999])

Good and bad components

- A component is **bad** if it contains only bad cycles, and **good** otherwise;
- Although we must be careful (as seen before), good components are not a problem:
  - they contain good cycles, which can be split;
  - applying a signed reversal on a 2-cycle C reverses the orientation of the cycles interleaving with C (and also changes their interleaving relationships);
  - so we just need to make sure at each step that we can keep splitting cycles afterwards;

Hurdles

- "Bad" (unoriented) components are called **hurdles** and are a problem;
- There are two ways of getting rid of them:
  1. "cutting" them;
  2. "merging" them;
- Either way, one move must be wasted for each hurdle to turn them into "good" components;
- Therefore, for all π in $S_n^{\pm}$:

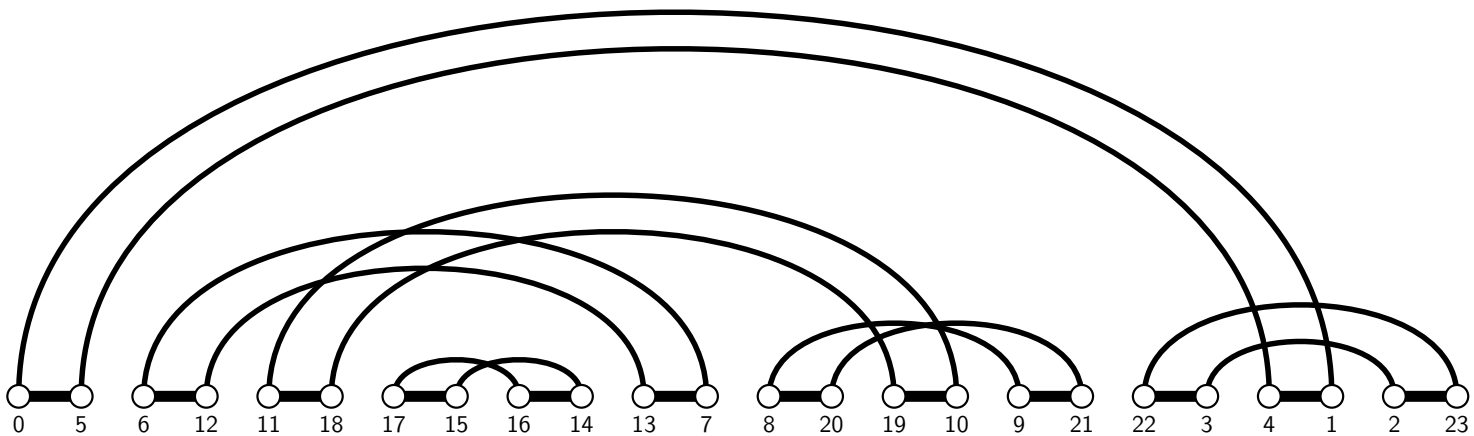
$$srd(\pi) \geq n + 1 - c(BG(\pi)) + h(\pi).$$

Practice

9. In the breakpoint graph of

$$\langle 3, -6, -9, 8, -7, 4, -10, -5, 11, 2, 1 \rangle :$$

- Identify good and bad cycles, components and hurdles
- Give a lower-bound for its signed reversal distance and an optimal sorting scenario



An actual formula

- All this (and many concealed details) leads to a formula for the signed reversal distance:

Theorem ([Hannenhalli and Pevzner, 1999])

For all π in $S_n^{\pm}$:

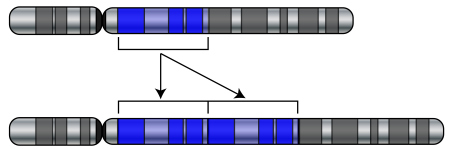
$$srd(\pi) = n + 1 - c(BG(\pi)) + \underbrace{h(\pi)}_{\text{number of hurdles}} + \underbrace{f(\pi)}_{\text{special ``fortress'' case}} .$$

Motivation

- We saw a model for representing genomes without directionality;
- We saw another model for taking directionality into account;
- Both of them lack realism in a crucial way: they don't allow duplications;
- And duplications / insertions / deletions account for a very large part of what happens in evolution [Ohno, 1970];

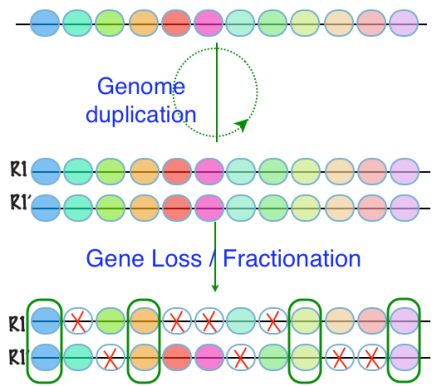
Two examples of duplications

Example (tandem duplications)



(source: K. Aainsqatsi on Wikimedia)

Example (whole genome duplication)



(source: Eric Lyons on CoGePedia)

Strings

- Since duplications pervade genomes, we should take them into account;
- We now see genomes as **strings** on an alphabet Σ;
- Be careful: similar segments have been identified, so $\Sigma = \{\text{segments}\}$ and not $\{A, C, G, T\}$;
- Our goal is still to explain evolution using most parsimonious scenarios made of fixed transformations;

Strings

- Note: the restriction to sorting problems does not work anymore;
  - if you have two A's, which one should be "number one"?
- So we really are interested in transforming one string into another, which is **not** equivalent to sorting another string;
- Sorting problems have been considered in that model, but they're just a special case of a more general problem;

Strings

- We can distinguish between several approaches based on gene contents;
- Either we have exactly the same contents in both genomes (and duplications are of course allowed);
- Or we have duplications but with different amounts of repetitions (e.g. three 1's in genome A but only two in genome B);
- This time the breakpoint graph cannot save us anymore, since we would not know how to connect elements or decompose the graph;

Balanced strings

- The number of occurrences of a character c in a string S is denoted by $occ(c, S)$;

Definition (balanced strings)

Two strings S and T on an alphabet Σ are **balanced** if:

$$\forall\ c \in \Sigma : occ(c, S) = occ(c, T).$$

- Basically, S and T are anagrams;
- Straightforward generalisation of permutations: we have duplications, but we actually still have the same content in both genomes;

Comparing balanced strings

- One way of relating genomes' contents is to identify common segments;
- In other words, we want to partition genomes into the same set of segments;
  - this is how we obtained (signed) permutations;
  - but now we want to partition the resulting sequences;

Generalising breakpoints

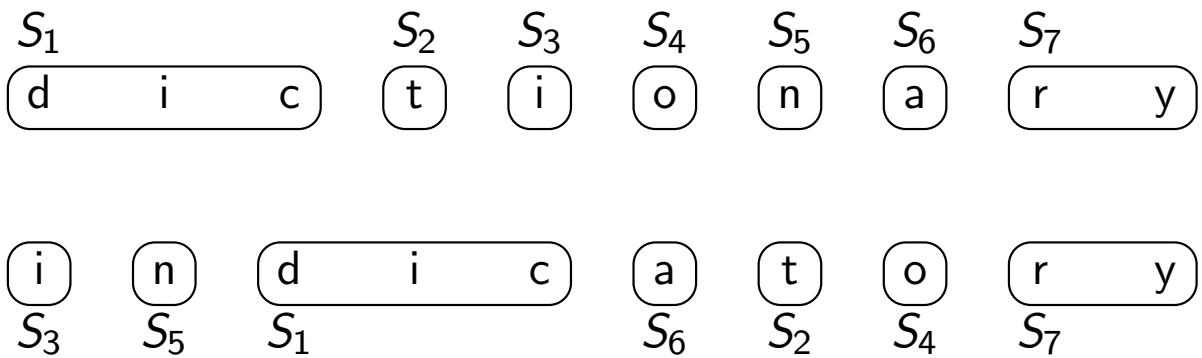
- Recall that, for permutations:
  - **adjacencies** are pairs of adjacent elements in π that are also adjacent in $\iota = \langle 1\ 2\ \cdots\ n \rangle$ (or $\chi = \langle n\ n-1\ \cdots\ 1 \rangle$ for reversals);
  - **breakpoints** are pairs that are not adjacencies;
- Recall that, for signed permutations:
  - **adjacencies** are pairs of adjacent elements in π that are also adjacent in $\iota = \langle 1\ 2\ \cdots\ n \rangle$ (or $\chi = \langle -n\ -(n-1)\ \cdots\ -1 \rangle$ for signed reversals);
  - **breakpoints** are pairs that are not adjacencies;
- Those can be generalised to any pair of permutations;
- And we can do the same thing for strings;

Minimum common string partition

- A **partition** of a string S is a set of strings that can be concatenated to obtain S;
- A **common partition** of two strings S and T is a set of strings that can be concatenated to obtain both S and T;

Example (common string partitions)

Here's a common partition of "dictionary" and "indicatory":



Minimum common string partition

- A common string partition is **minimum** if there is no smaller common string partition for the two strings under consideration;
- This leads to the following decision problem:

Problem (MINIMUM COMMON STRING PARTITION (MCSP))

Instance: *balanced strings S and T, a bound $k \in \mathbb{N}$;*
Question: *is there a common partition of S and T with at most k blocks?*

Relation(s) to rearrangement problems

- Recall that breakpoints were pairs of elements adjacent in one genome but not in the other;
- Common string partitions generalise that point of view to an arbitrary number of elements in each part;
- So if we have a minimum common string partition for S and T, we get the number of breakpoints between strings S and T;

About MCSP

- Bad news about MCSP:
  - NP-hard, even if only one gene family is nontrivial [Blin et al., 2004];
  - APX-hard, even if no character appears more than twice [Goldstein et al., 2005];
- Good news about MCSP:
  - fixed parameter tractable: a solution of size k can be found in time $f(k) \cdot \text{poly}(n)$ ($n = |S| = |T|$) [Bulteau and Komusiewicz, 2014];
- Greedy approach [Goldstein and Lewenstein, 2011]: repeatedly select an LCS without any marked letter;
  - ✓ simple and fast (runs in $O(n)$ time);
  - × approximation ratio between $\Omega(n^{0.43})$ and $O(n^{0.69})$ [Kaplan and Shafrir, 2006];

Practice

10. Give pairwise MCSP distances between the following strings (ignore capitals and whitespaces)
    - `Arrange A String`
    - `A Staring Ranger`
    - `A Garnering Tsar`
11. Find a pair of strings for which Greedy behaves as badly as possible (i.e. maximizing the approximation ratio)

References I

Akers, S. B., Krishnamurthy, B., and Harel, D. (1987).
The star graph: An attractive alternative to the n-cube.
In *ICPP'87*, pages 393–400. Pennsylvania State University Press.

Bafna, V. and Pevzner, P. A. (1996).
Genome rearrangements and sorting by reversals.
SIAM Journal on Computing, 25(2):272–289.

Bafna, V. and Pevzner, P. A. (1998).
Sorting by transpositions.
SIAM Journal on Discrete Mathematics, 11(2):224–240.

Berman, P., Hannenhalli, S., and Karpinski, M. (2002).
1.375-approximation algorithm for sorting by reversals.
In *ESA'02*, volume 2461 of *LNCS*, pages 200–210. Springer-Verlag.

Bulteau, L., Fertin, G., and Rusu, I. (2012).
Sorting by transpositions is difficult.
SIAM Journal on Discrete Mathematics, 26(3):1148–1180.

Bulteau, L., Fertin, G., and Rusu, I. (2015).
Pancake flipping is hard.
Journal of Computer and System Sciences, 81(8):1556–1574.

Caprara, A. (1999).
Sorting permutations by reversals and eulerian cycle decompositions.
SIAM Journal on Discrete Mathematics, 12(1):91–110 (electronic).

References II

Chen, X. (2010).

On sorting permutations by double-cut-and-joins.
In *COCOON'10*, volume 6196 of *LNCS*, pages 439–448. Springer-Verlag.

Christie, D. A. (1996).

Sorting permutations by block-interchanges.
Information Processing Letters, 60(4):165–169.

Dias, Z. and Meidanis, J. (2002).

Sorting by prefix transpositions.
In *SPIRE'02*, volume 2476 of *LNCS*, pages 65–76. Springer-Verlag.

Elias, I. and Hartman, T. (2006).

A 1.375-approximation algorithm for sorting by transpositions.
IEEE/ACM Transactions on Computational Biology and Bioinformatics, 3(4):369–379.

Fischer, J. and Ginzinger, S. W. (2005).

A 2-approximation algorithm for sorting by prefix reversals.
In *ESA'05*, volume 3669 of *LNCS*, pages 415–425. Springer-Verlag.

Gog, S. and Bader, M. (2008).

Fast algorithms for transforming back and forth between a signed permutation and its equivalent simple permutation.
Journal of Computational Biology, 15(8):1–13.

Hannenhalli, S. and Pevzner, P. A. (1999).

Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals.
Journal of the ACM, 46(1):1–27.

References III

Kececioglu, J. and Sankoff, D. (1995).

Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement.
Algorithmica, 13(1-2):180–210.

Knuth, D. E. (1995).

Sorting and Searching, volume 3 of *The art of Computer Programming*.
Addison-Wesley.

Watterson, G. A., Ewens, W. J., Hall, T. E., and Morgan, A. (1982).

The chromosome inversion problem.
Journal of Theoretical Biolooy, 99:1–7.

Algorithms and Bioinformatics

Part II — Comparative Genomics

II.2 — Focus on MCSP and FPT

Laurent Bulteau

Presented at WABI 2013

Workshop on Algorithms in Bioinformatics

Fixed-Parameter Algorithm for Minimum Common String Partition with Few Duplications

Laurent Bulteau, Christian Komusiewicz,
Guillaume Fertin, Irena Rusu

WABI 2013
Full version available at arxiv.org/abs/1307.7842

Outline

1 MCSP Problem

2 $O(d^{2k}n)$ FPT algorithm

3 Implementation

4 Conclusion

MCSP Problem

Minimum Common String Partition

Input: two strings of length n
Output: min. set of blocks partitioning both sequences

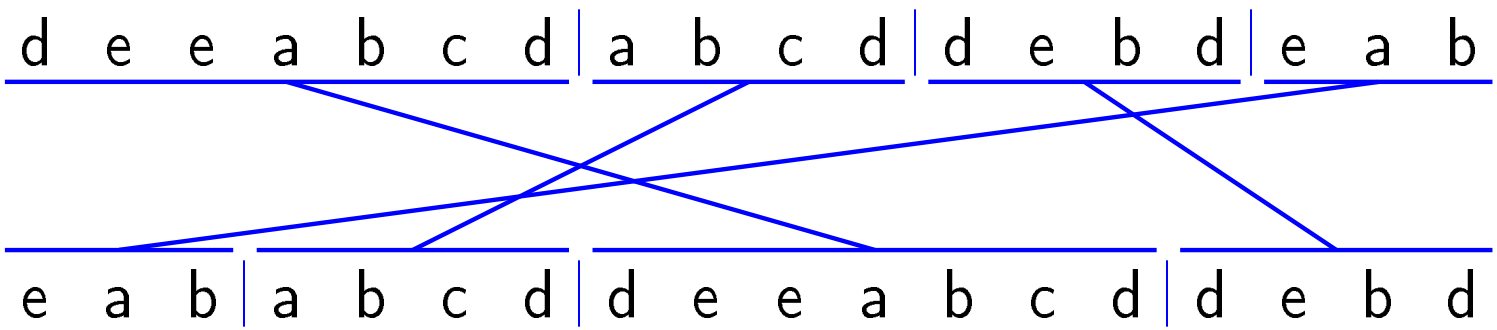
d e e a b c d a b c d d e b d e a b

e a b a b c d d e e a b c d d e b d

Minimum Common String Partition

Input: two strings of length n

Output: min. set of blocks partitioning both sequences



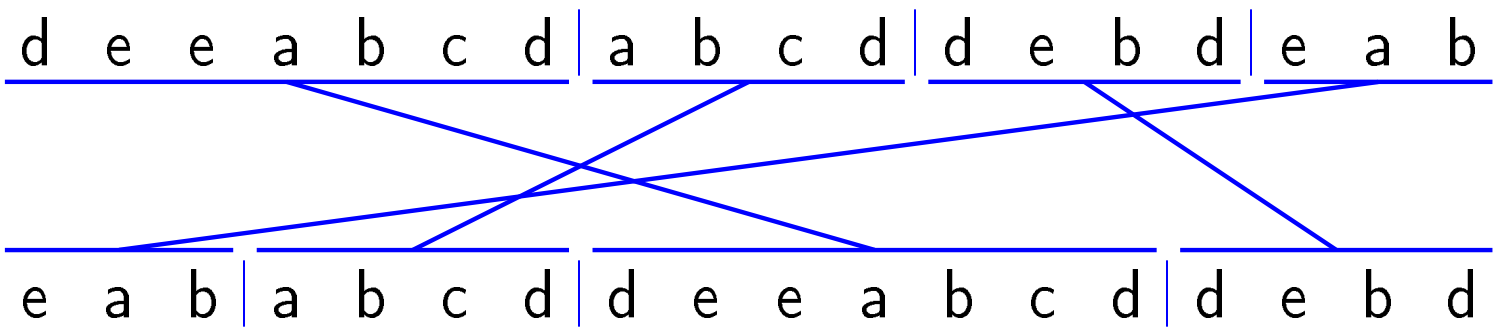
Minimum Common String Partition

Input: two strings of length n

Output: min. set of blocks partitioning both sequences

Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution



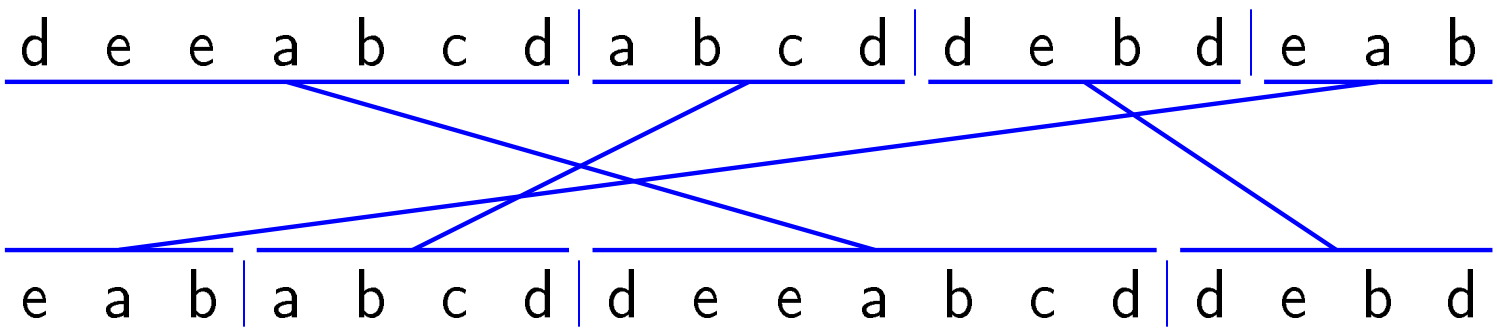
Minimum Common String Partition

Input: two strings of length n

Output: min. set of blocks partitioning both sequences

Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution

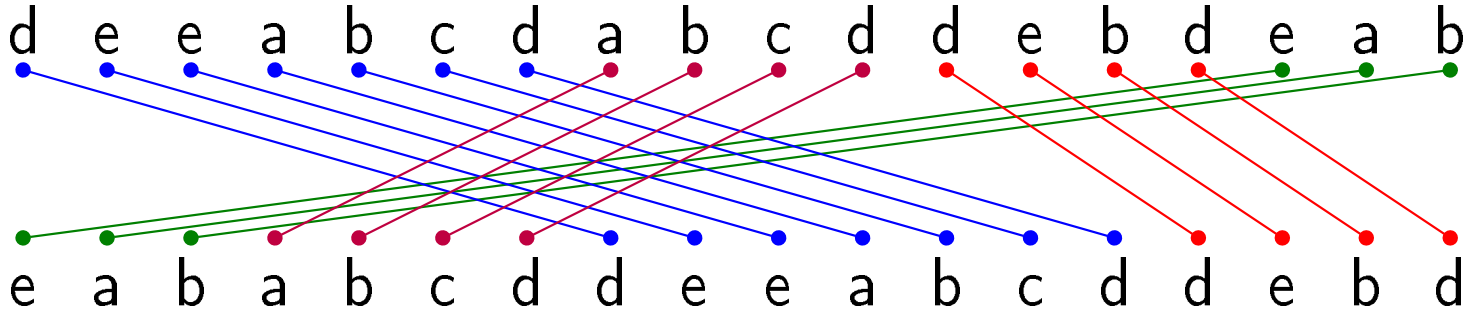


$n = 18$ elements, $d = 5$, $k = 4$

Minimum Common String Partition

Input: two strings of length n
Output: min. set of blocks partitioning both sequences
Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution

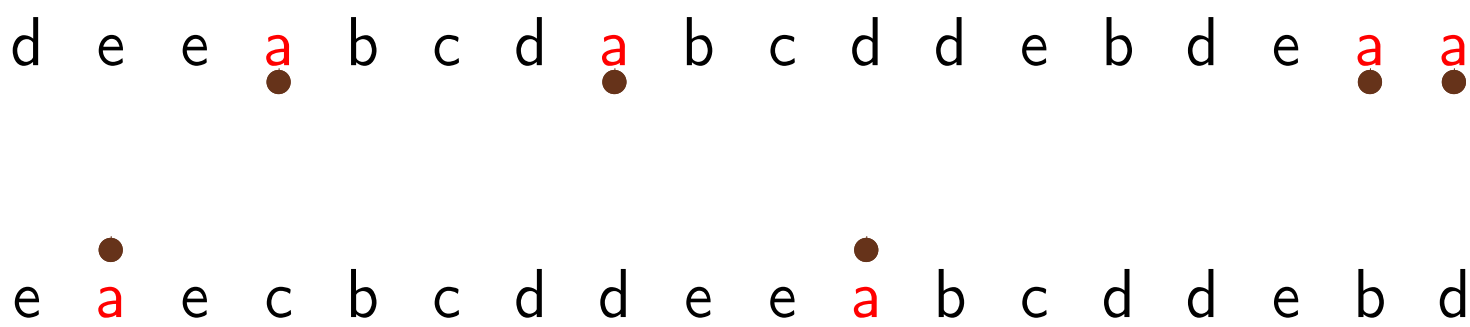


Can be seen as a matching problem

Minimum Common String Partition

Input: two strings of length n
Output: min. set of blocks partitioning both sequences
Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution

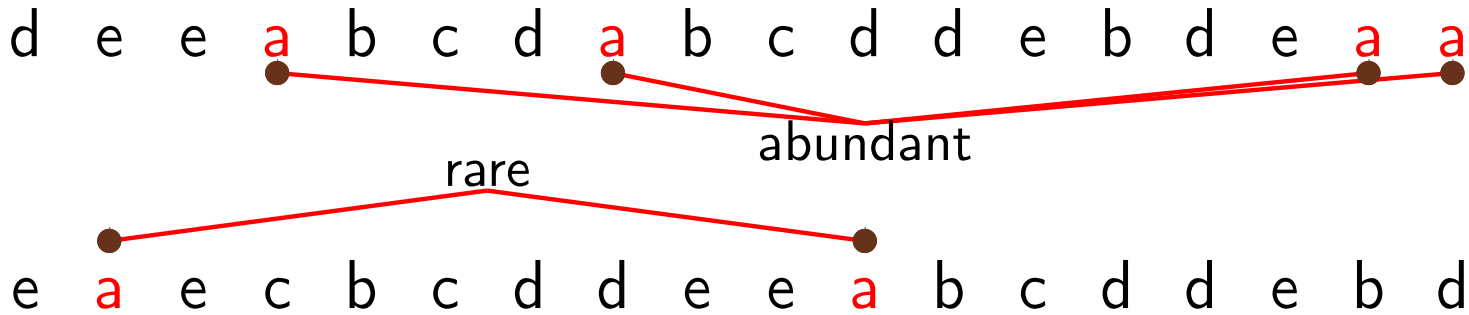


Unbalanced strings: allow deletion of abundant elements between blocks (all rare elements are kept)

Minimum Common String Partition

Input: two strings of length n
Output: min. set of blocks partitioning both sequences
Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution



Unbalanced strings: allow deletion of abundant elements between blocks (all rare elements are kept)

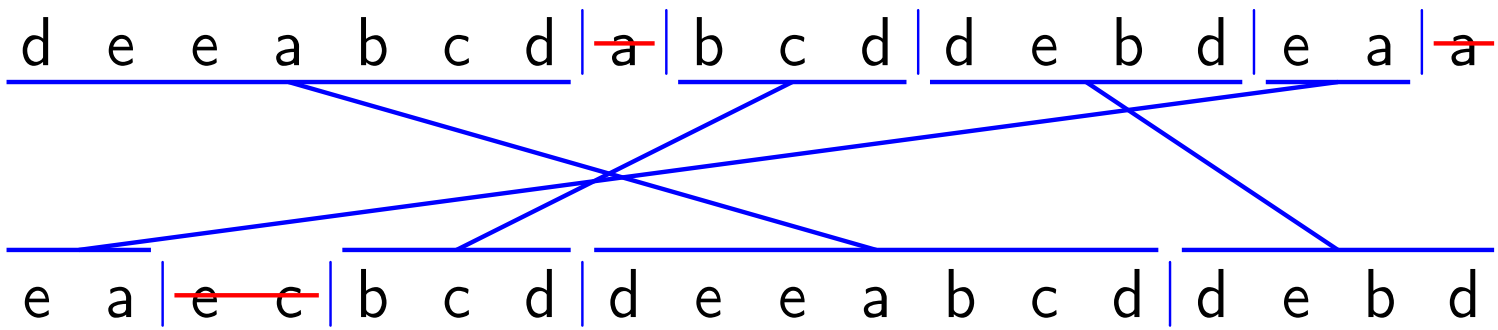
Minimum Common String Partition

Input: two strings of length n

Output: min. set of blocks partitioning both sequences

Parameters:

- d = max. number of occurences of each letter
- k = number of blocks in an optimal solution



Minimum Common String Partition

- NP-hard
- FPT algorithms $d!^k \text{poly}(n)$
  k = number of blocks; d = number of duplications
- FPT algorithm with parameter k only $k^{21k^2}\text{poly}(n)$
- Improved FPT algorithm with k and d $d^{2k}\text{poly}(n)$
  + Allows for unbalanced strings
  + Implemented

Assignment of orthologous genes via genome rearrangement

X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi and T. Jiang 2005

Minimum Common String Partition

- **NP**-hard
- FPT algorithms $d!^k \text{poly}(n)$
  k = number of blocks; d = number of duplications
- FPT algorithm with parameter k only $k^{21k^2}\text{poly}(n)$
- Improved FPT algorithm with k and d $d^{2k}\text{poly}(n)$
  + Allows for unbalanced strings
  + Implemented

Minimum common string partition problem: Hardness and approximations

A. Goldstein, P. Kolman and J. Zheng 2005

Minimum Common String Partition

- **NP**-hard
- FPT algorithms $d!^k \text{poly}(n)$
  k = number of blocks; d = number of duplications
- FPT algorithm with parameter k only $k^{21k^2}\text{poly}(n)$
- Improved FPT algorithm with k and d $d^{2k}\text{poly}(n)$
  + Allows for unbalanced strings
  + Implemented

Minimum common string partition parameterized	
P. Damaschke	2008

Minimum common string partition revisited	
H. Jiang, B. Zhu, D. Zhu and H. Zhu	2012

Minimum Common String Partition

- **NP**-hard
- FPT algorithms $d!^k \text{poly}(n)$
  k = number of blocks; d = number of duplications
- FPT algorithm with parameter k only $k^{21k^2}\text{poly}(n)$
- Improved FPT algorithm with k and d $d^{2k}\text{poly}(n)$
  + Allows for unbalanced strings
  + Implemented

Minimum Common String Partition Parameterized by Partition Size is Fixed-Parameter Tractable

L. Bulteau, C. Komusiewicz — Arxiv 2013

Minimum Common String Partition

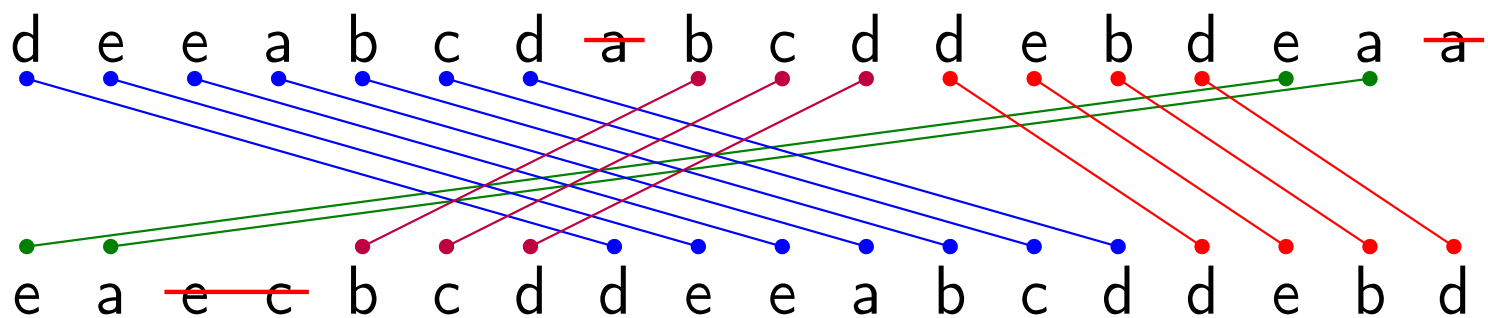
- **NP**-hard
- FPT algorithms $d!^k \text{poly}(n)$
  k = number of blocks; d = number of duplications
- FPT algorithm with parameter k only $k^{21k^2}\text{poly}(n)$
- Improved FPT algorithm with k and d $d^{2k}\text{poly}(n)$
  + Allows for unbalanced strings
  + Implemented

This talk

$O(d^{2k}n)$ FPT algorithm

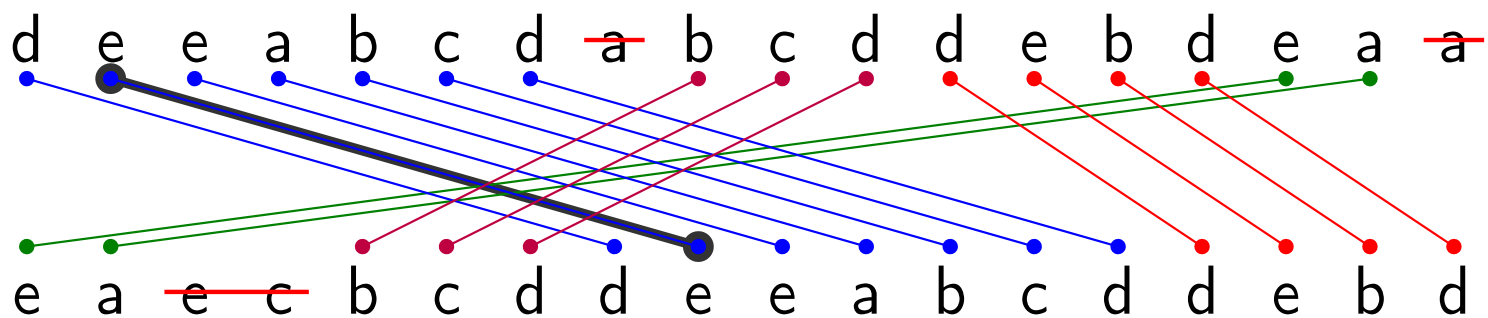
Seeds

- Optimal solution OPT, seen as a matching



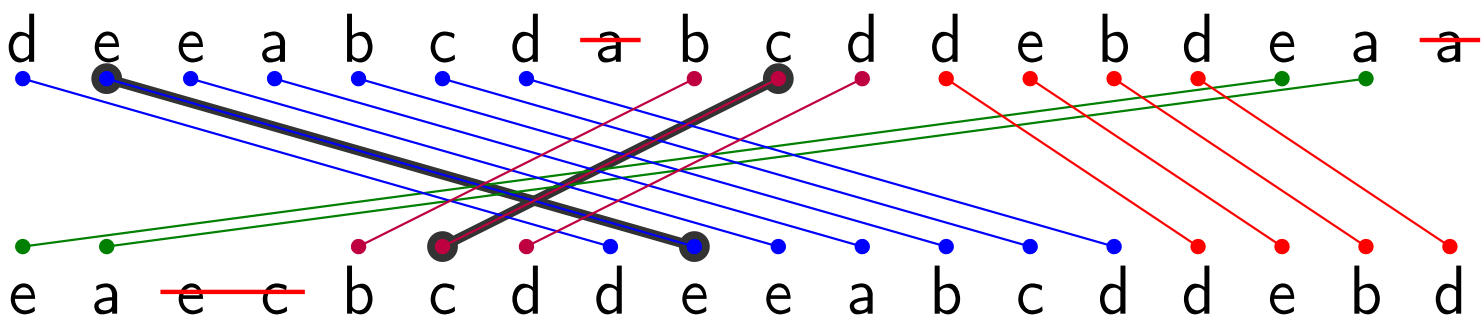
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed



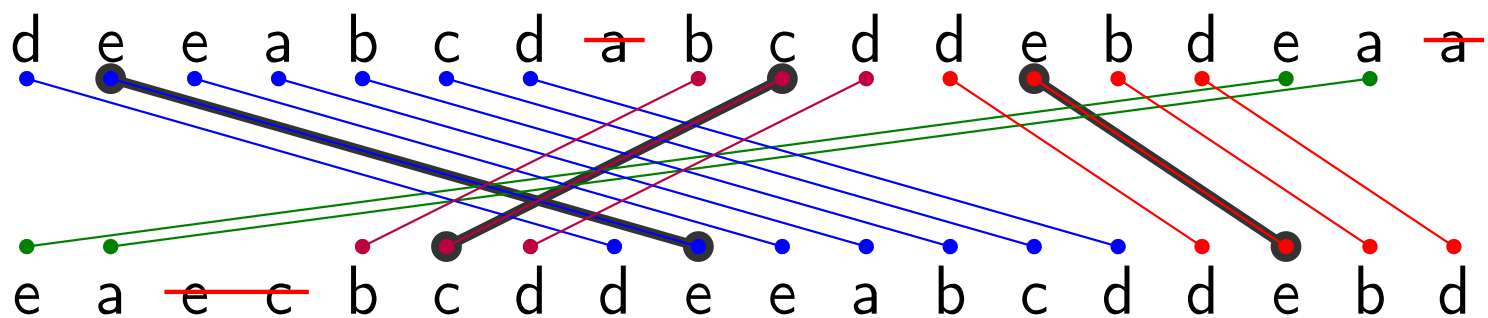
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed



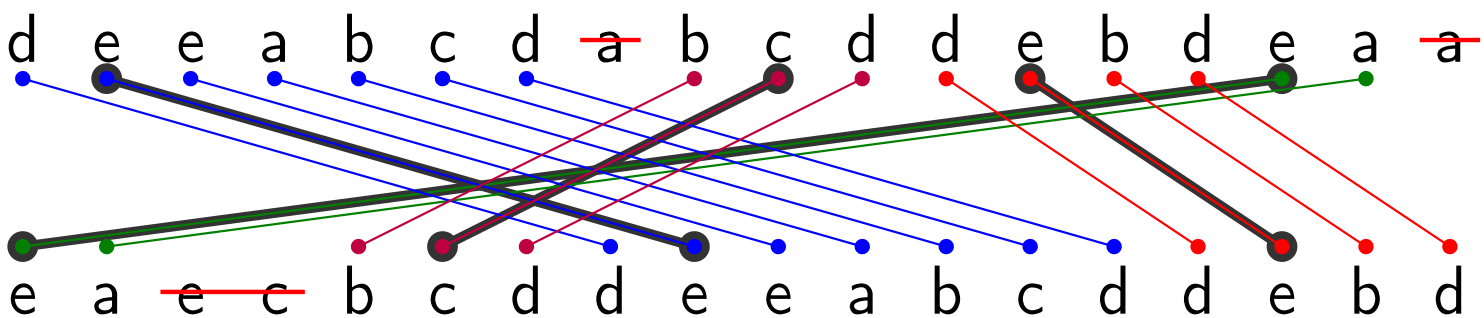
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed



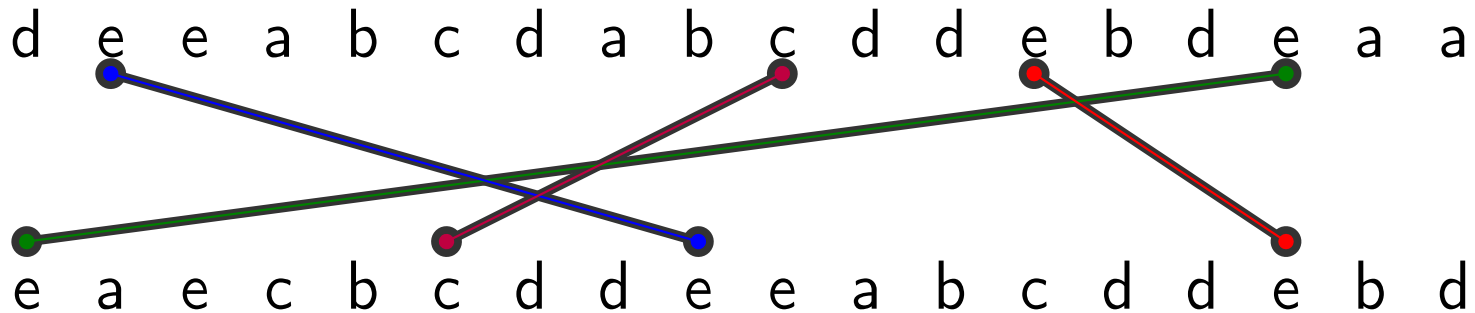
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed



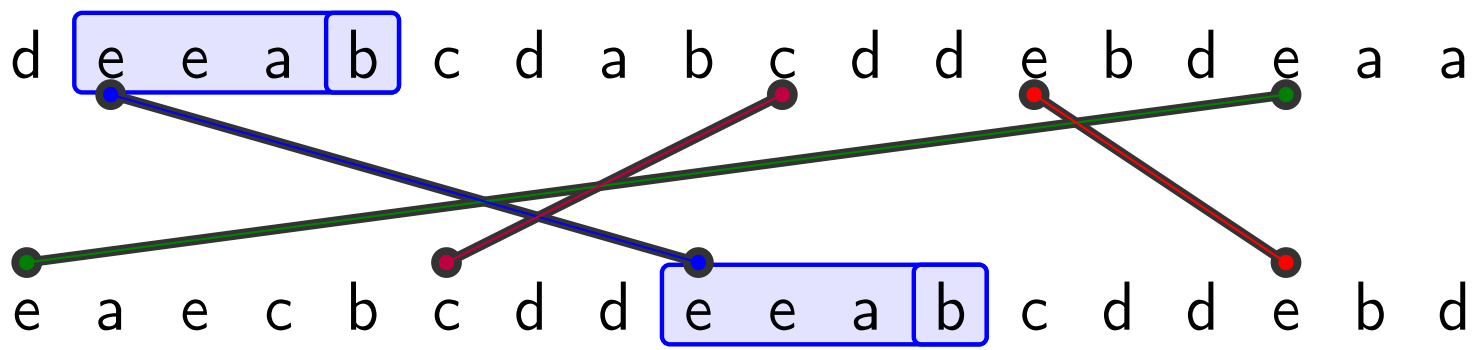
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed
- Goal: find a good set of seeds



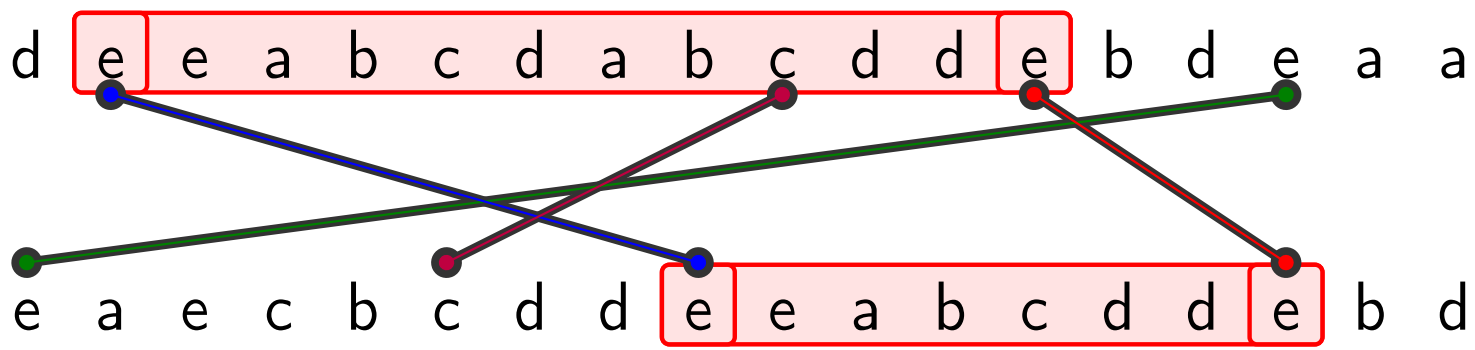
Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed
- Goal: find a good set of seeds
  - Every rare element must be in the same block as a seed: the set is complete



Seeds

- Optimal solution OPT, seen as a matching
- One block $\leftrightarrow$ one seed
- Goal: find a good set of seeds
  - Every rare element must be in the same block as a seed:
    the set is complete
  - Two seeds cannot be in the same block:
    the set is non-redundant



Algorithm Outline

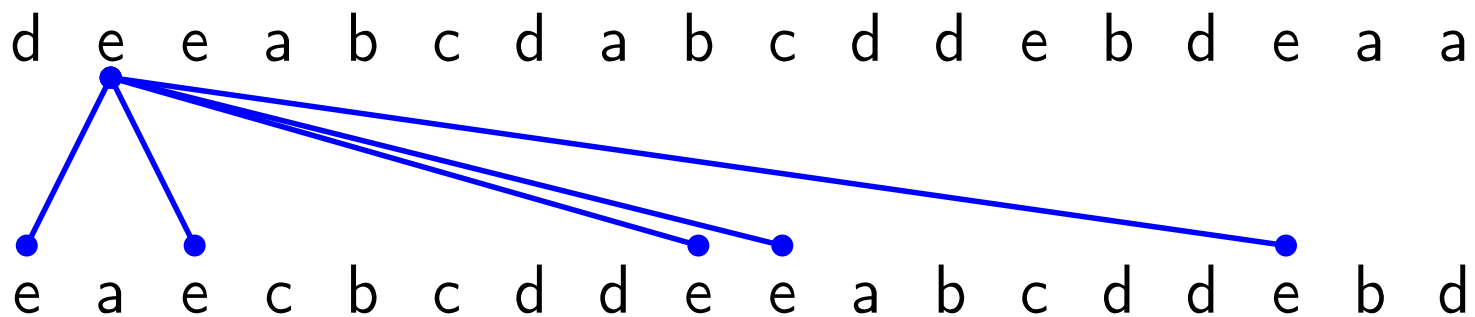
- Start with set of seeds $T = \emptyset$

d e e a b c d a b c d d e b d e a a

e a e c b c d d e e a b c d d e b d

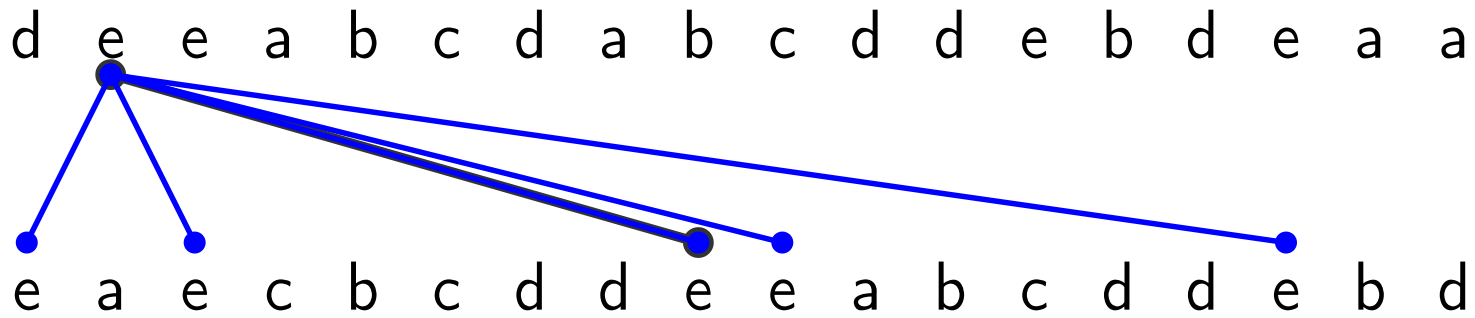
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed



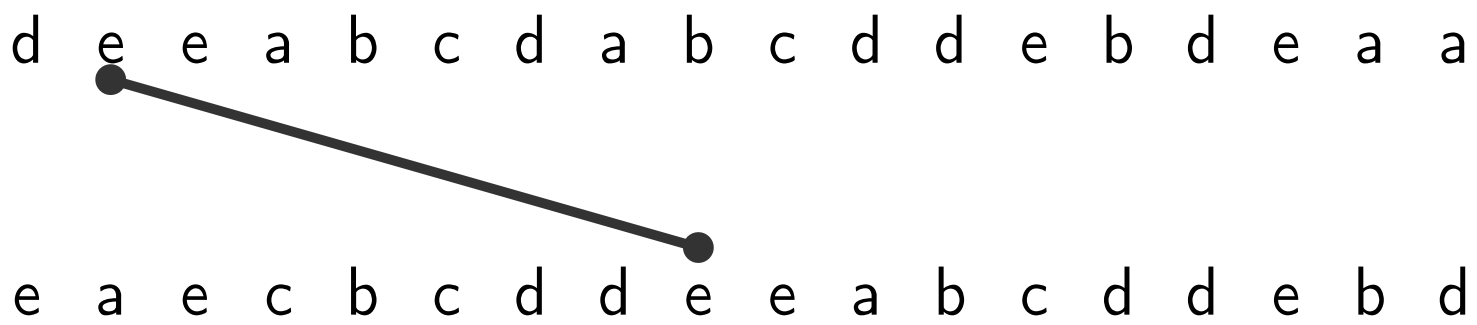
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



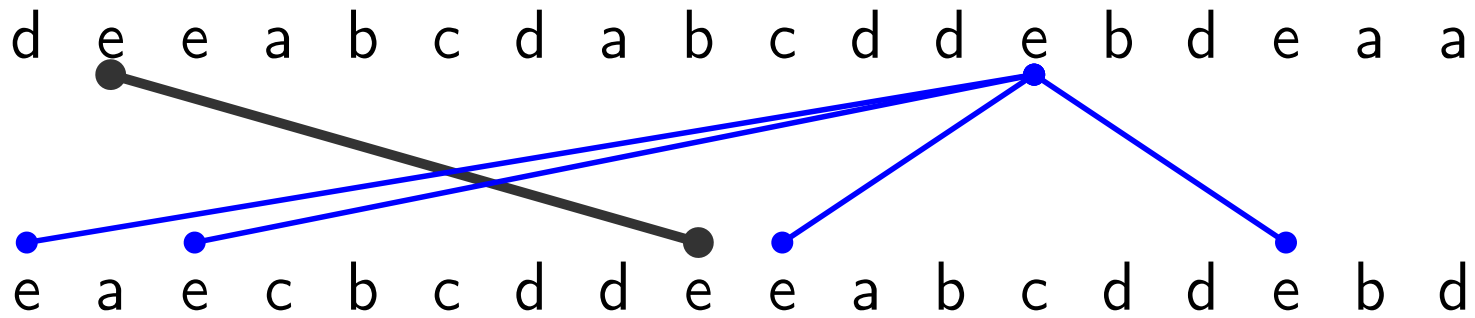
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



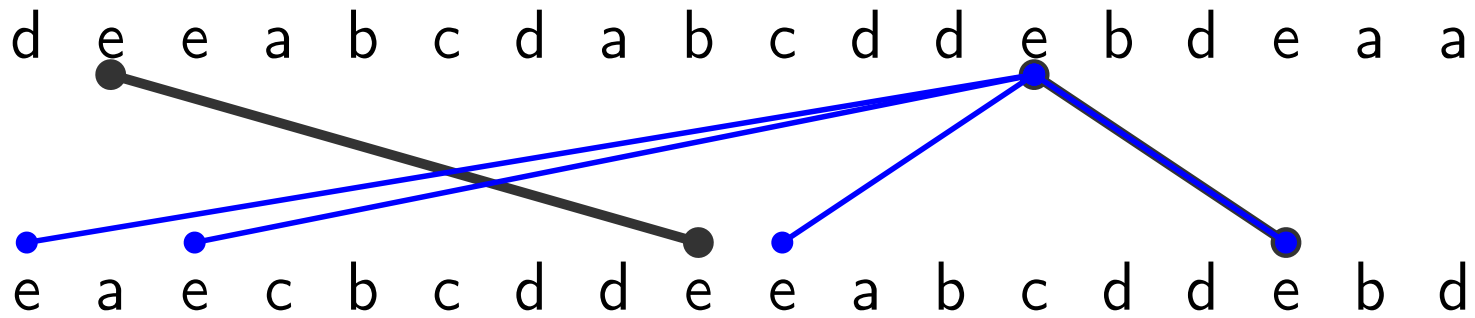
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



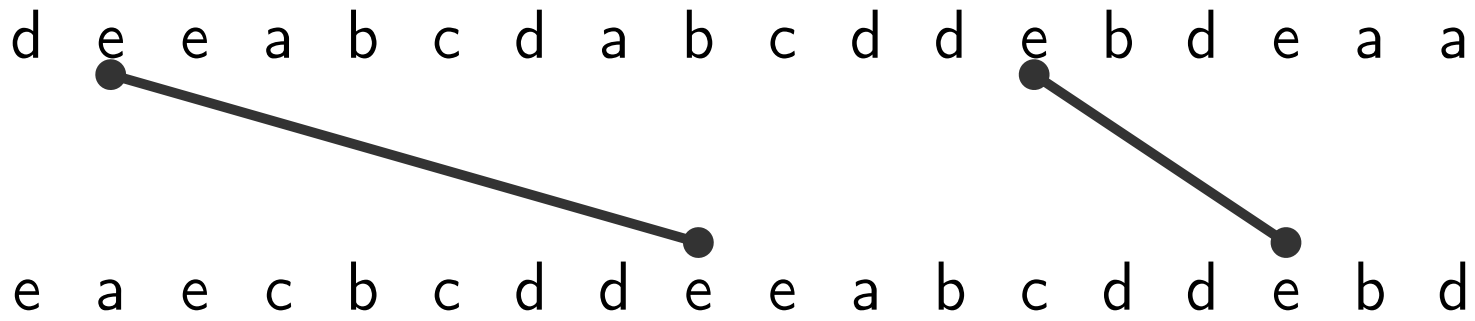
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



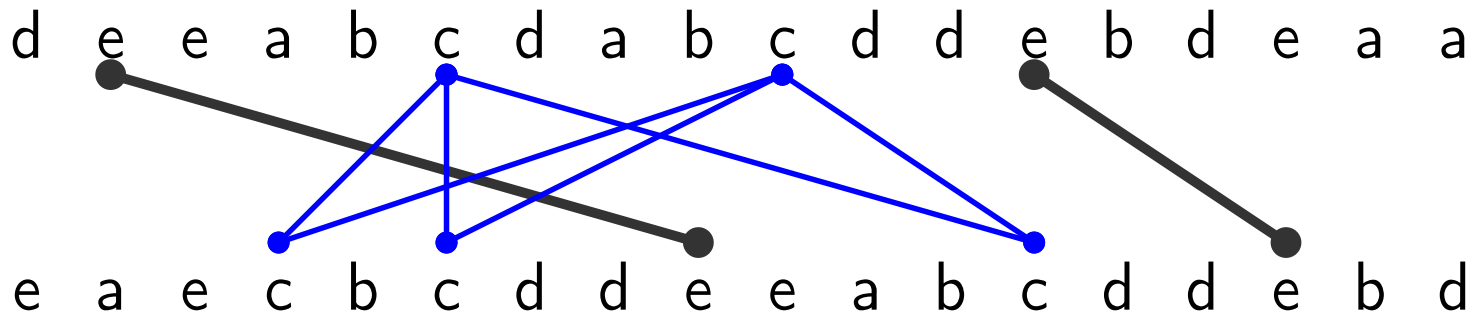
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



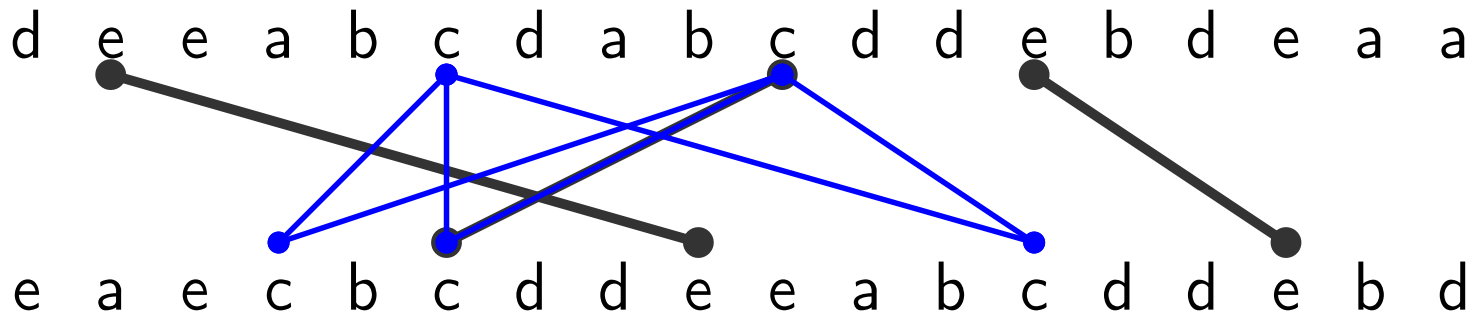
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



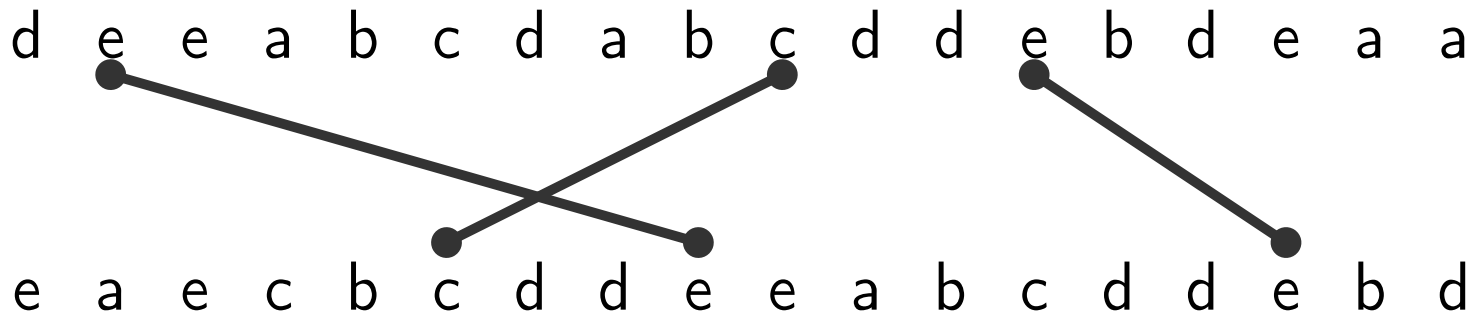
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



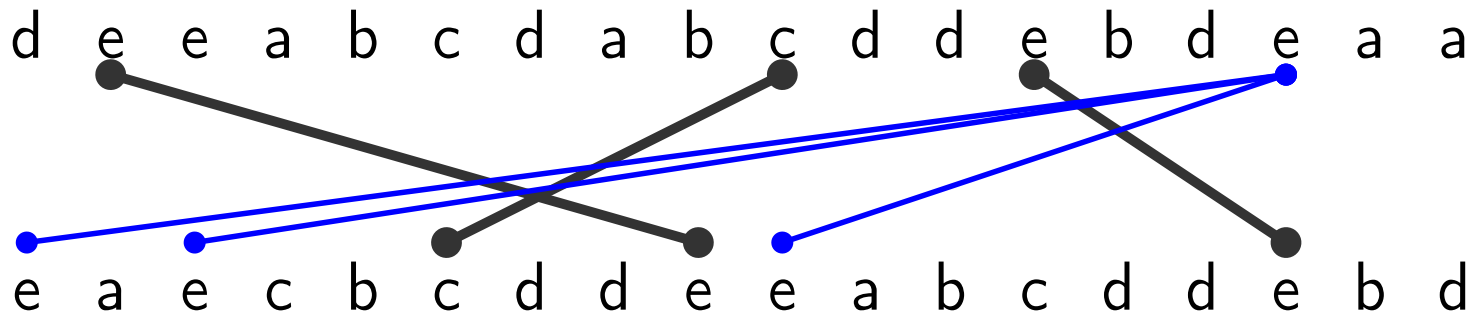
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



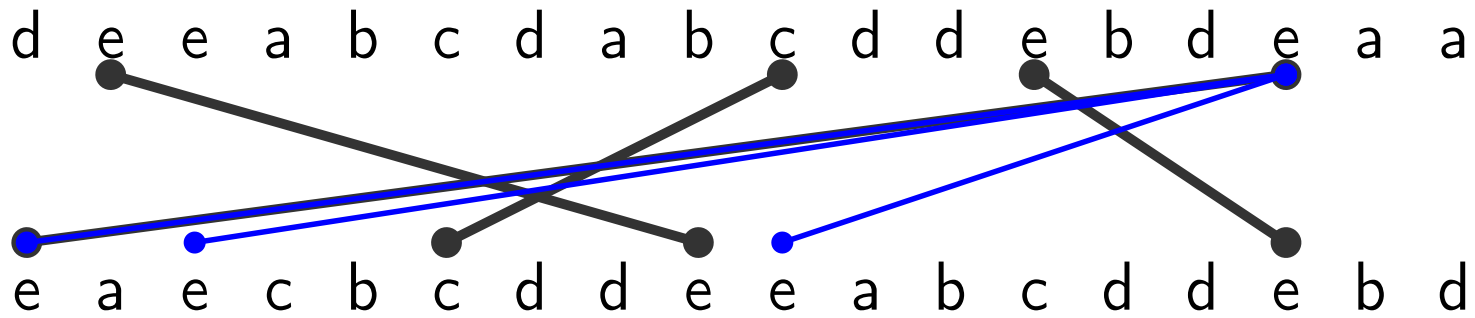
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



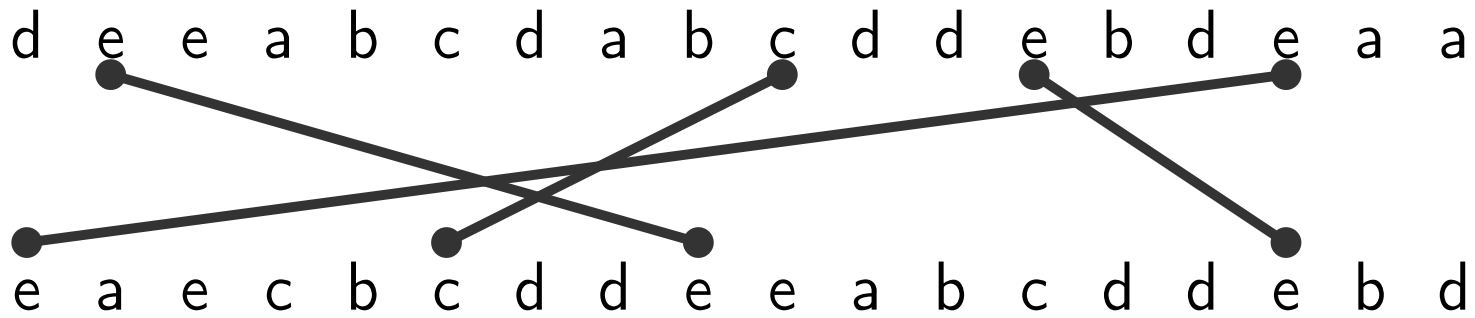
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



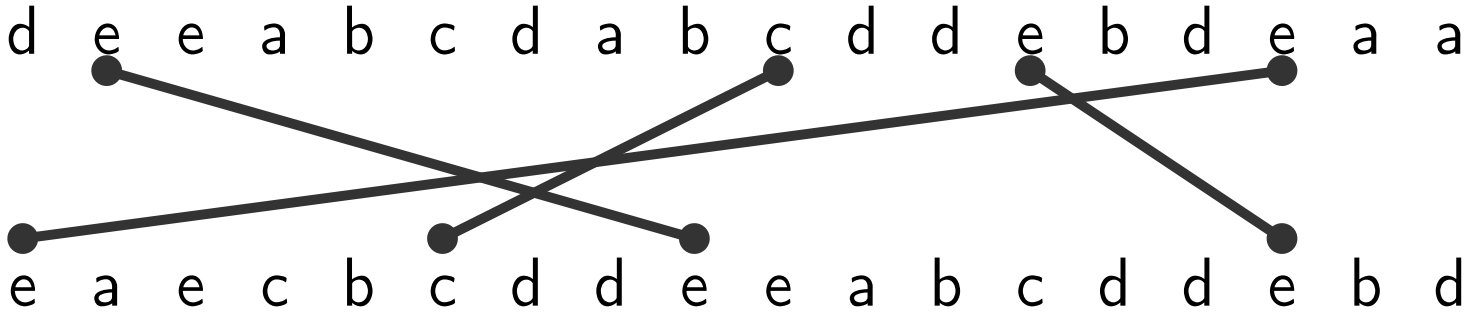
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again



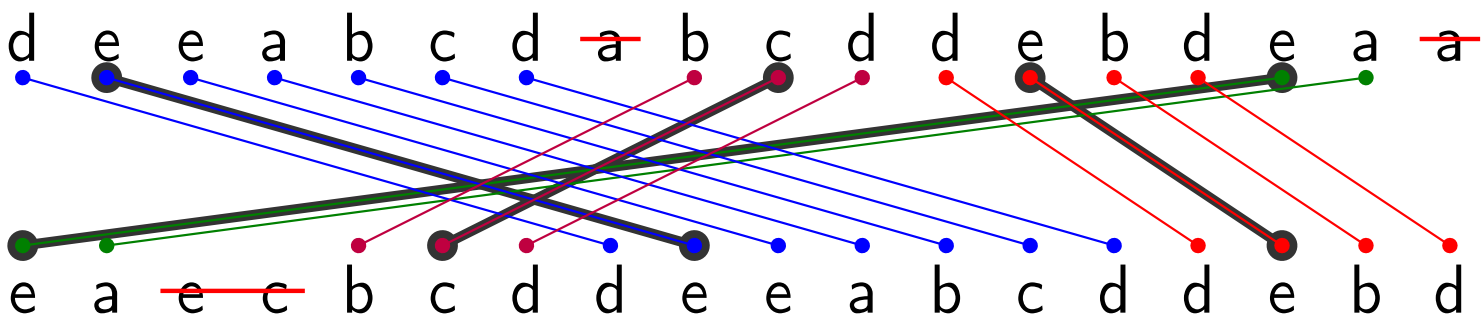
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again
- No more possible set:



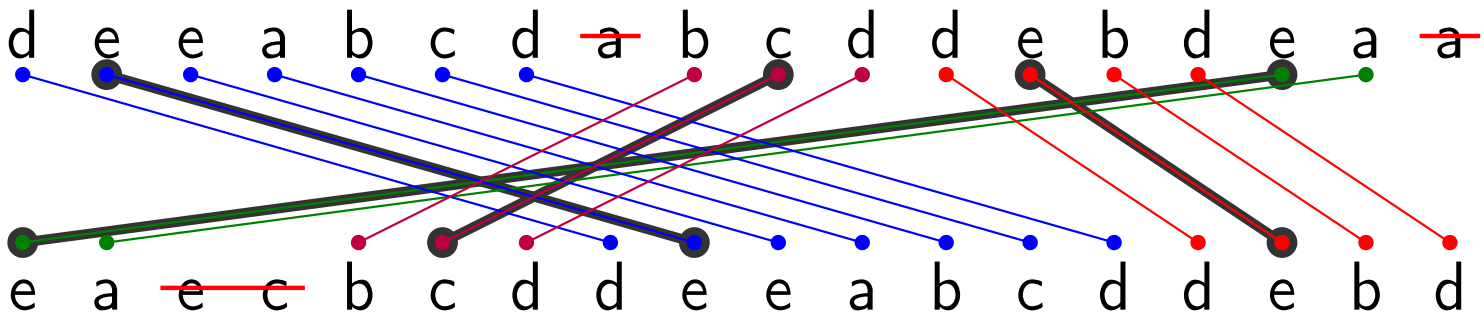
Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again
- No more possible set:
  The set is complete, create a CSP corresponding to the seeds.



Algorithm Outline

- Start with set of seeds $T = \emptyset$
- Identify set of pairs containing 1 correct seed
- Try each candidate in the set as a new seed, start again
- No more possible set:
  The set is complete, create a CSP corresponding to the seeds.

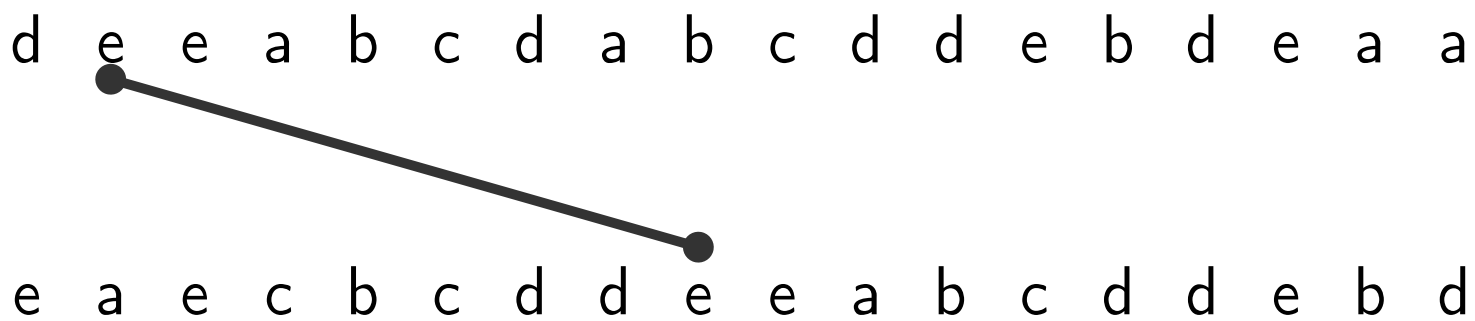


Finding new seeds (1)

Non-redundancy

In OPT, every (x, y) which is not in the same block as an already present seed can be a new seed...

- Every x which cannot be in the same block as any seed can be part of a new seed.

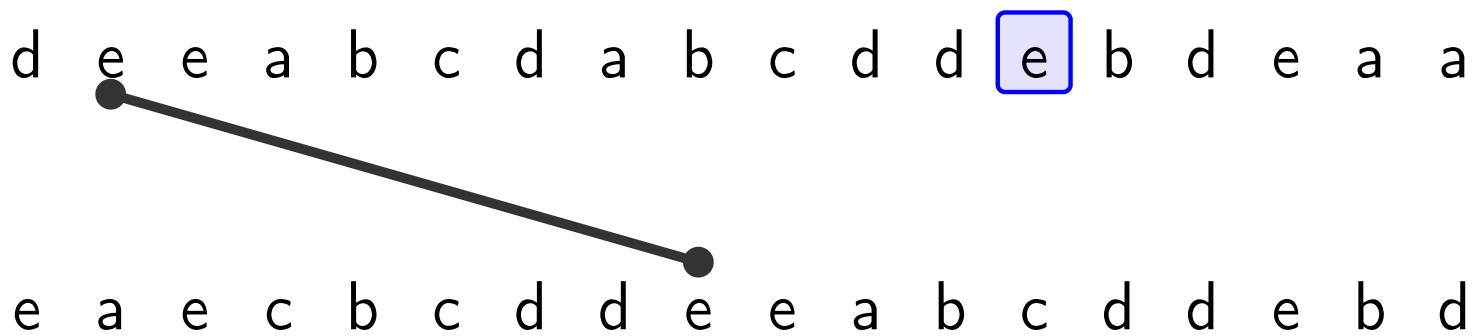


Finding new seeds (1)

Non-redundancy

In OPT, every (x, y) which is not in the same block as an already present seed can be a new seed...

- Every x which cannot be in the same block as any seed can be part of a new seed.

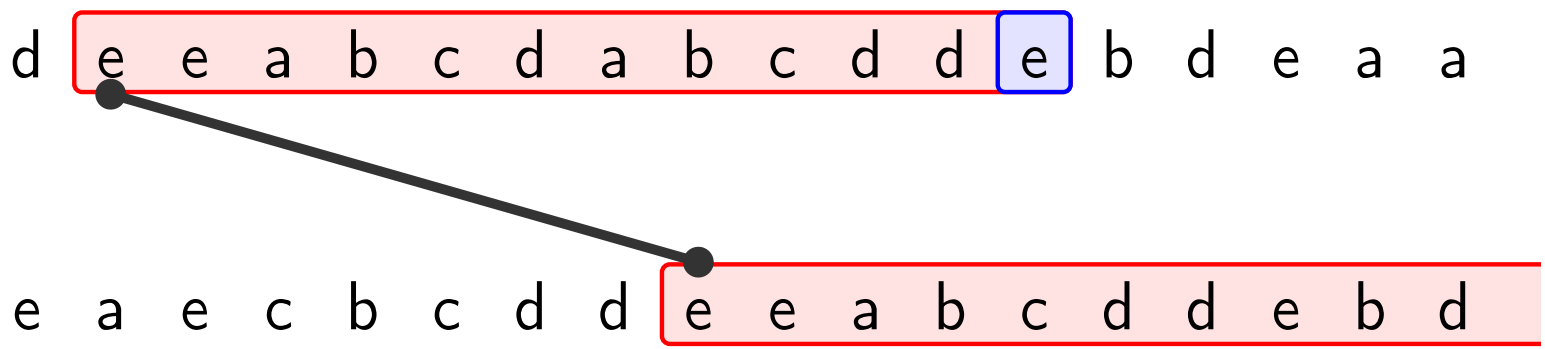


Finding new seeds (1)

Non-redundancy

In OPT, every (x, y) which is not in the same block as an already present seed can be a new seed...

- Every x which cannot be in the same block as any seed can be part of a new seed.

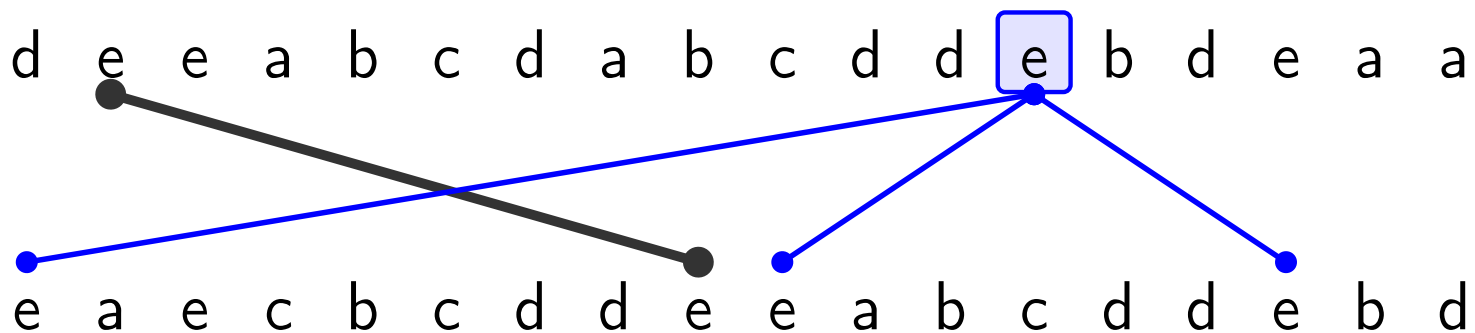


Finding new seeds (1)

Non-redundancy

In OPT, every (x, y) which is not in the same block as an already present seed can be a new seed...

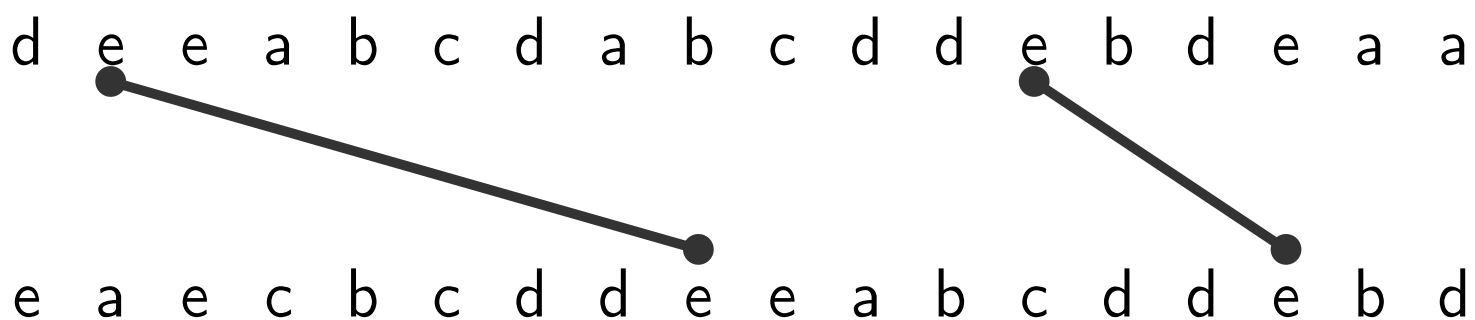
- Every x which cannot be in the same block as any seed can be part of a new seed.
- y is one of the d copies of x in the other sequence: each such pair is a candidate



Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

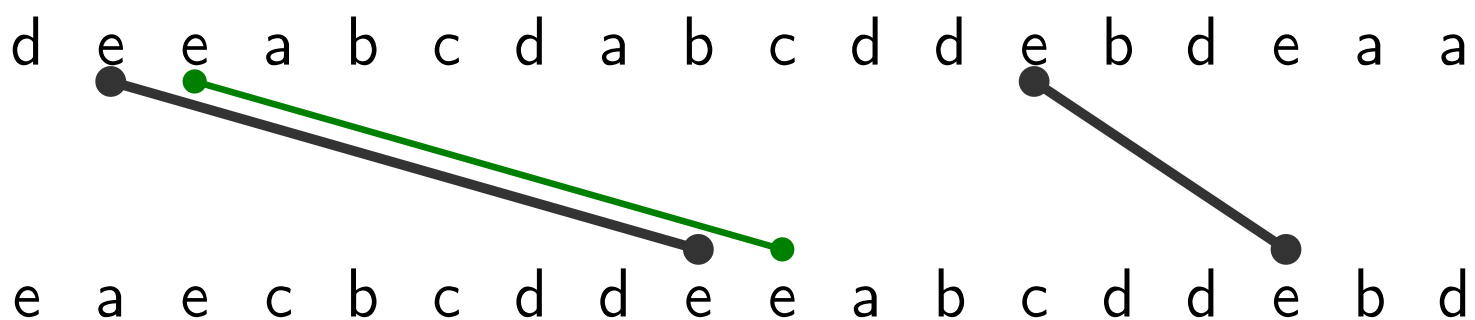


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed

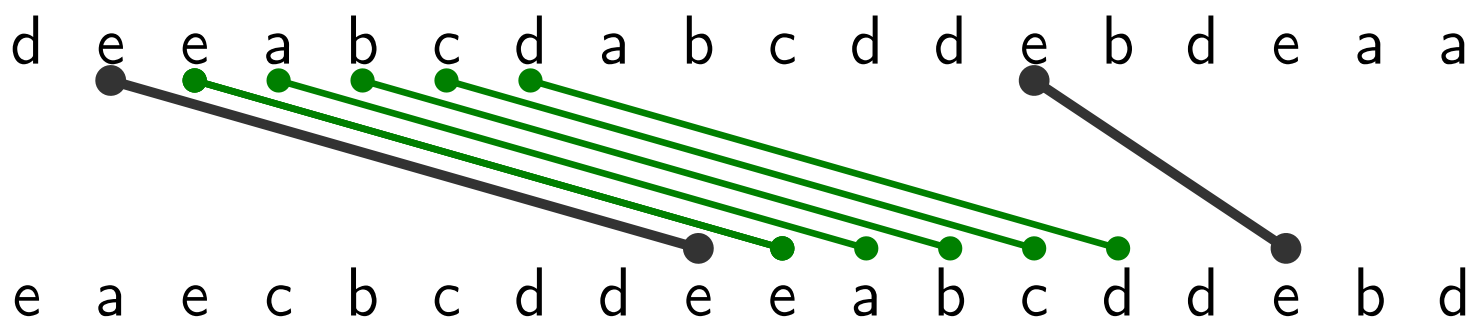


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed

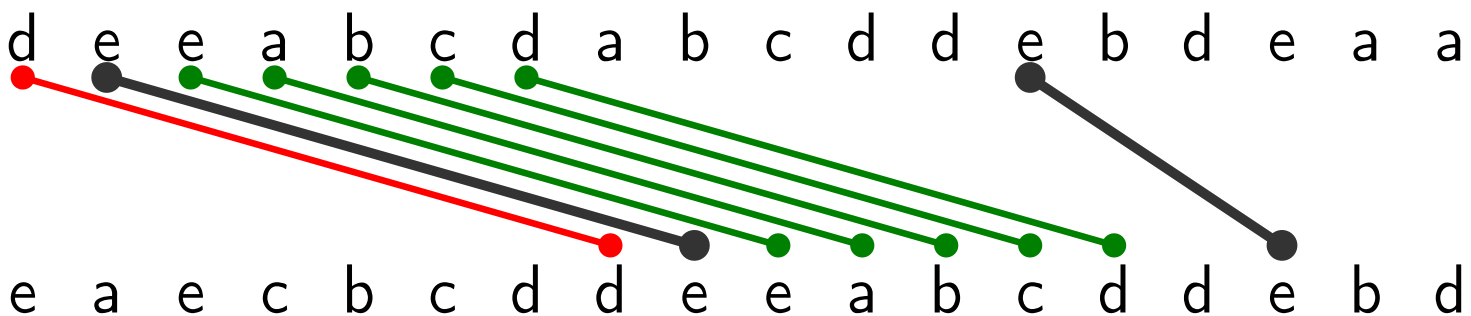


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed
- Red edge: (x, y) can be part of the left-side of a seed

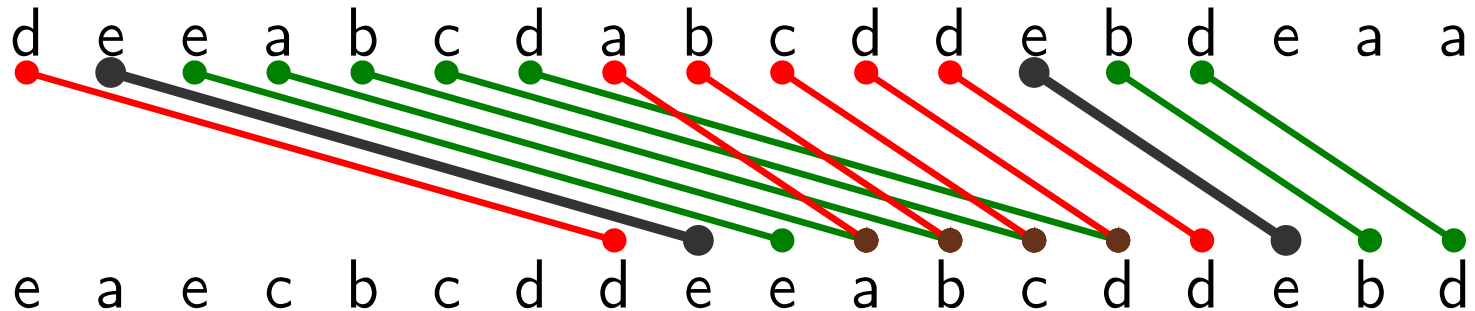


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed
- Red edge: (x, y) can be part of the left-side of a seed

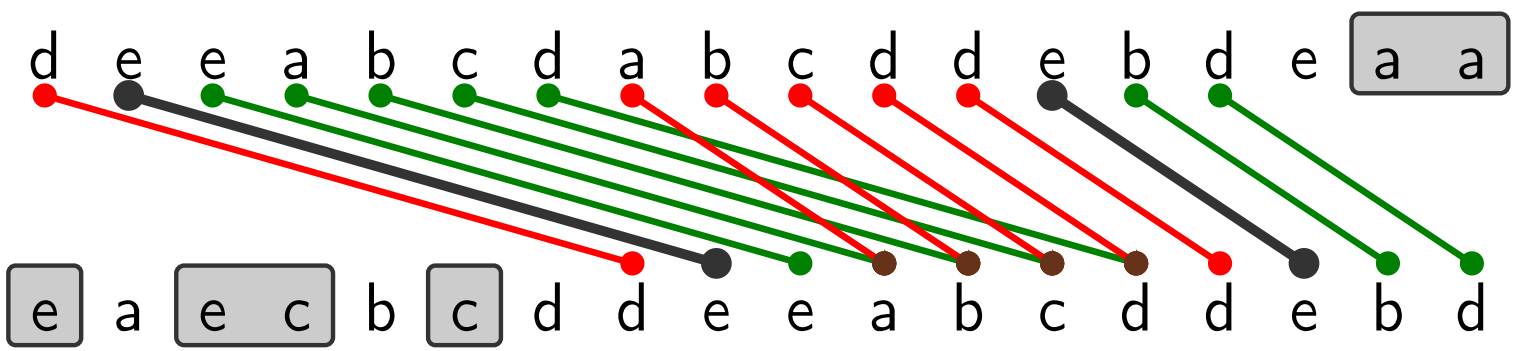


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed
- Red edge: (x, y) can be part of the left-side of a seed
- Abundant vertices can be ignored.

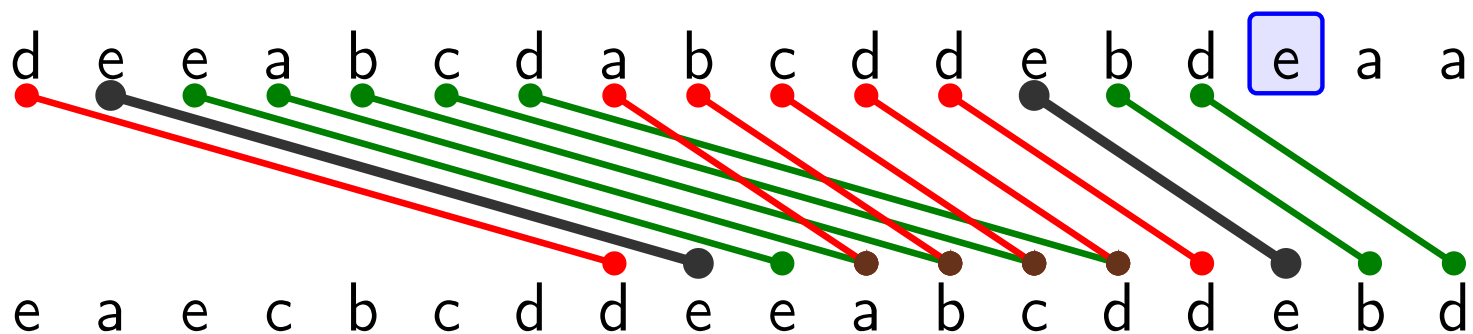


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

- Green edge: (x, y) can be part of the right-side of a seed
- Red edge: (x, y) can be part of the left-side of a seed
- Abundant vertices can be ignored.
- Degree-0 vertices can be part of new seeds (rule 1).

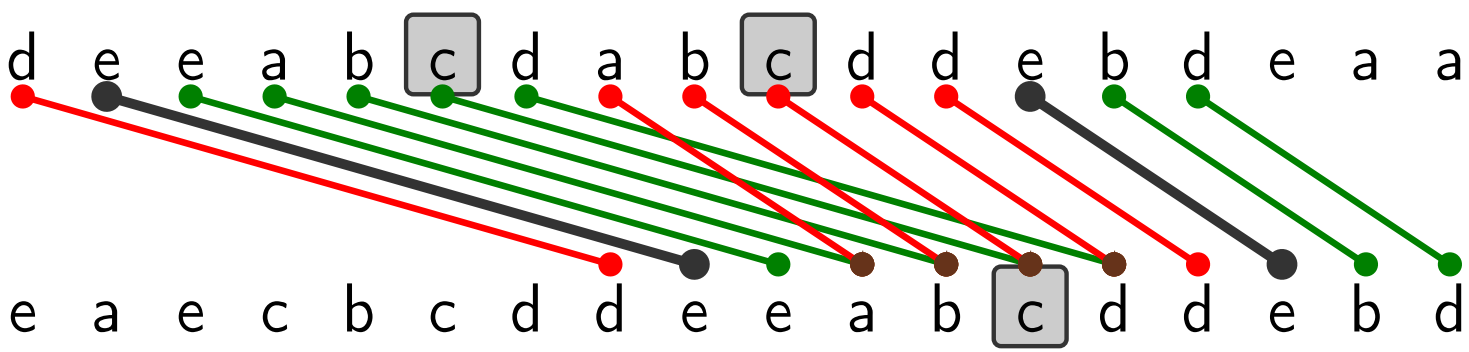


Finding new seeds (2)

Association graph

Link together pairs (x, y) that can be in the same block as a seed

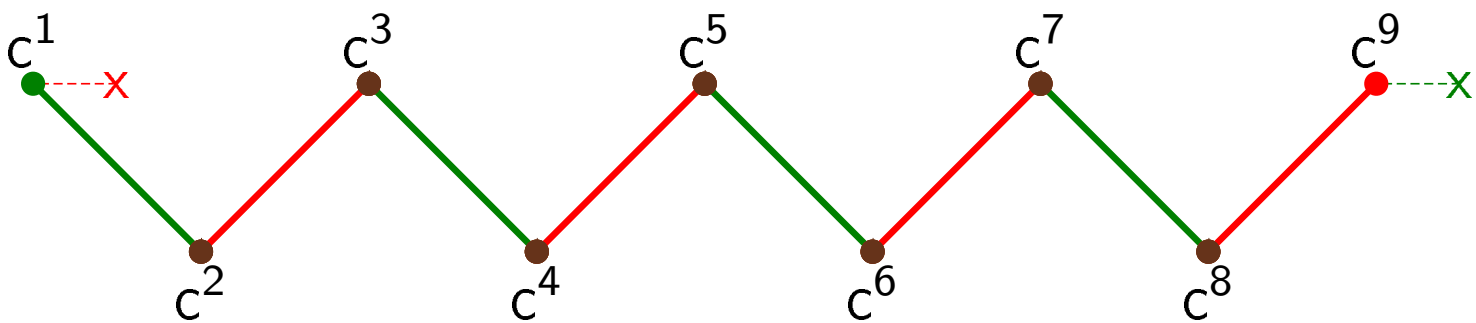
- Green edge: (x, y) can be part of the right-side of a seed
- Red edge: (x, y) can be part of the left-side of a seed
- Abundant vertices can be ignored.
- Degree-0 vertices can be part of new seeds (rule 1).
- What about degree-1 and degree-2 vertices?



Odd Paths

Path of odd length

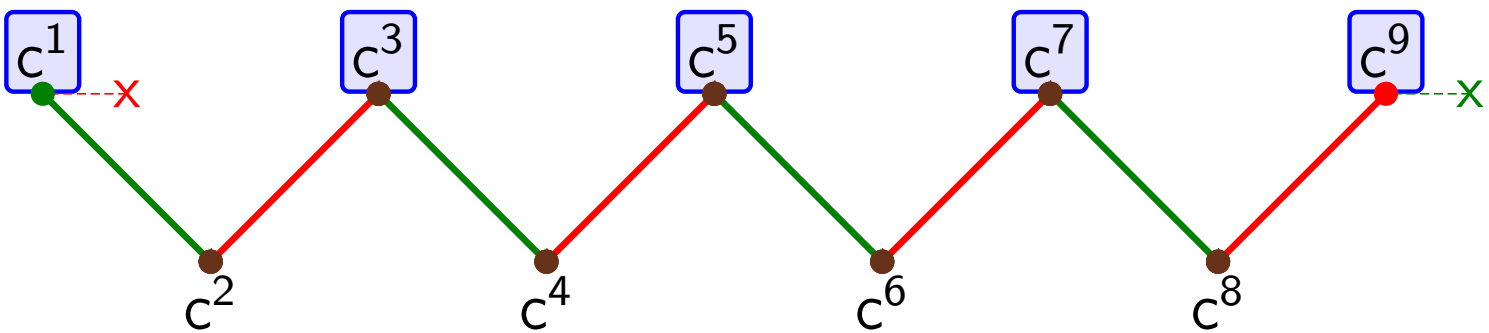
If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare



Odd Paths

Path of odd length

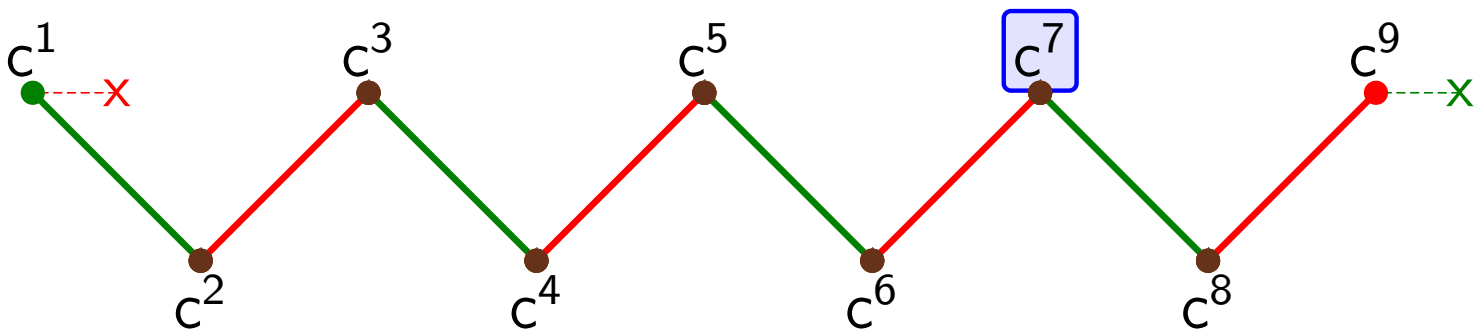
If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare
One of $\{c^{2i+1} \mid i = 1..p\}$ is part of a seed.
NB: $p \leq d$



Odd Paths

Path of odd length

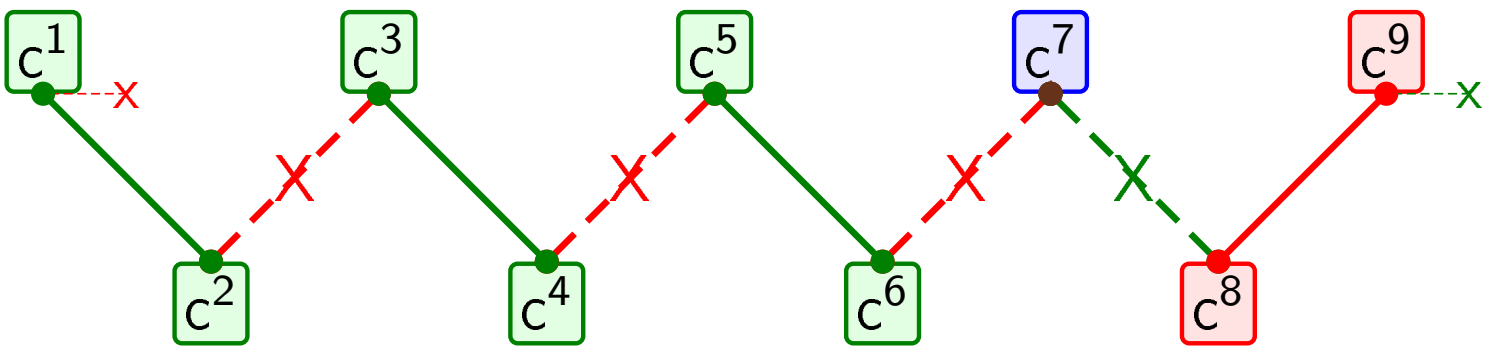
If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare
One of $\{c^{2i+1} \mid i = 1..p\}$ is part of a seed.
NB: $p \leq d$



Odd Paths

Path of odd length

If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare
One of $\{c^{2i+1} \mid i = 1..p\}$ is part of a seed.
NB: $p \leq d$



Odd Paths

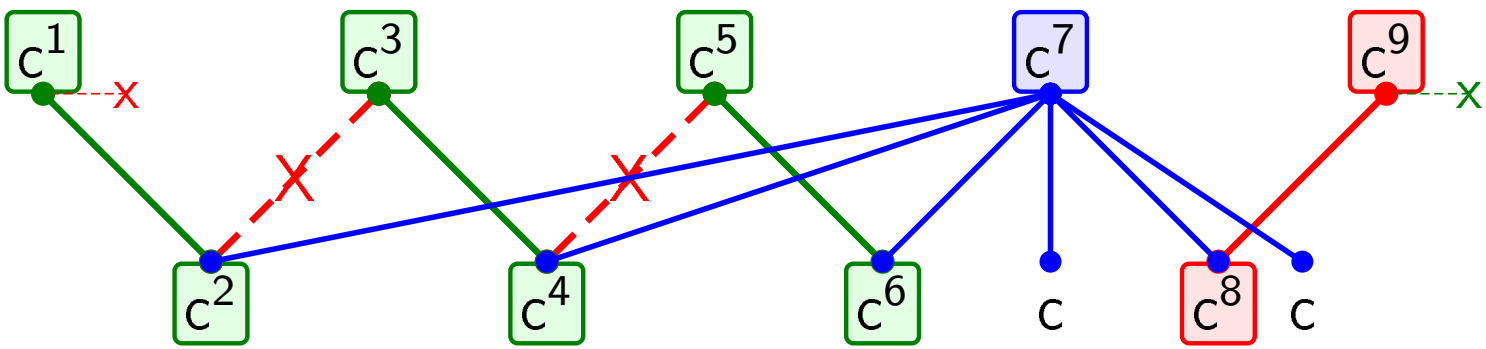
Path of odd length

If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare

One of $\{c^{2i+1} \mid i = 1..p\}$ is part of a seed.

NB: $p \leq d$

Other half of the seed: any letter c in the other sequence.



Odd Paths

Path of odd length

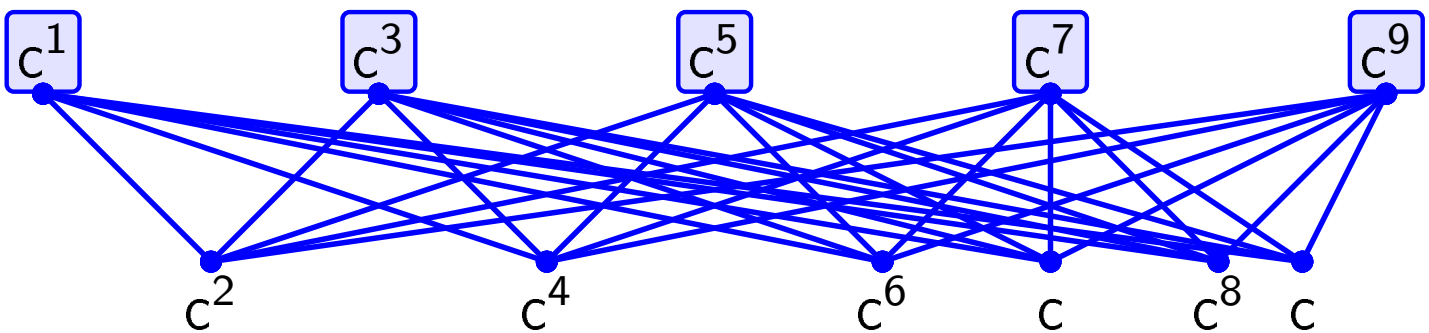
If $(c^1, c^2, c^3, \ldots, c^{2p+1})$ is an odd path with c^1 being rare

One of $\{c^{2i+1} \mid i = 1..p\}$ is part of a seed.

NB: $p \leq d$

Other half of the seed: any letter c in the other sequence.

Total number of seeds to try: at most d^2.



So far...

We know how to deal with single elements and rare odd paths.

What if our association graph has only even paths, cycles, and abundant odd paths?

So far...

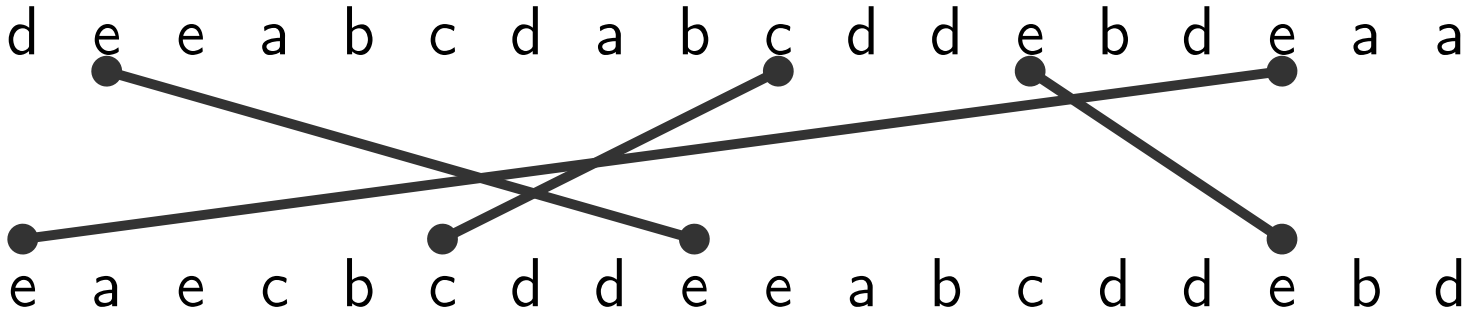
We know how to deal with single elements and rare odd paths.

What if our association graph has only even paths, cycles, and abundant odd paths?

Then we can create a CSP based on our seeds!

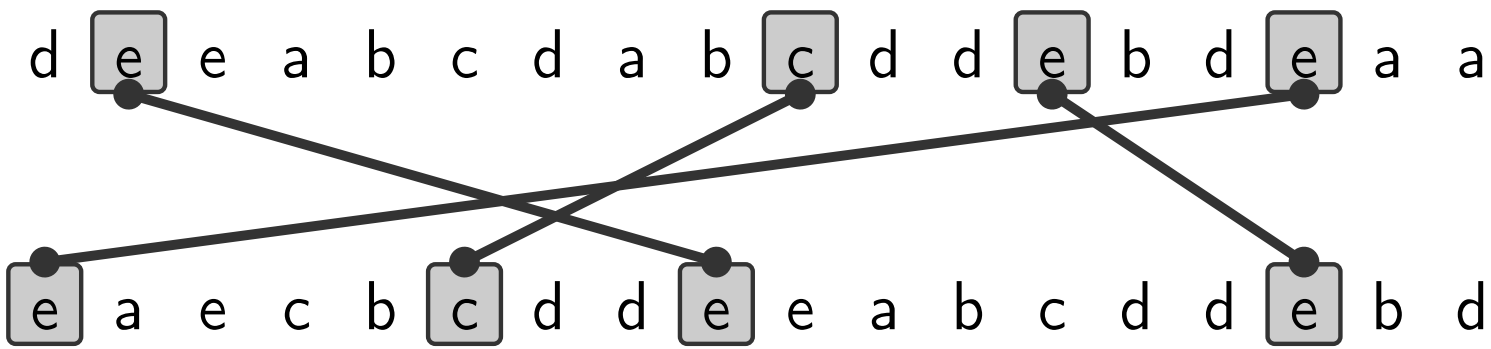
Constructing the solution

- No more seeds can be found...



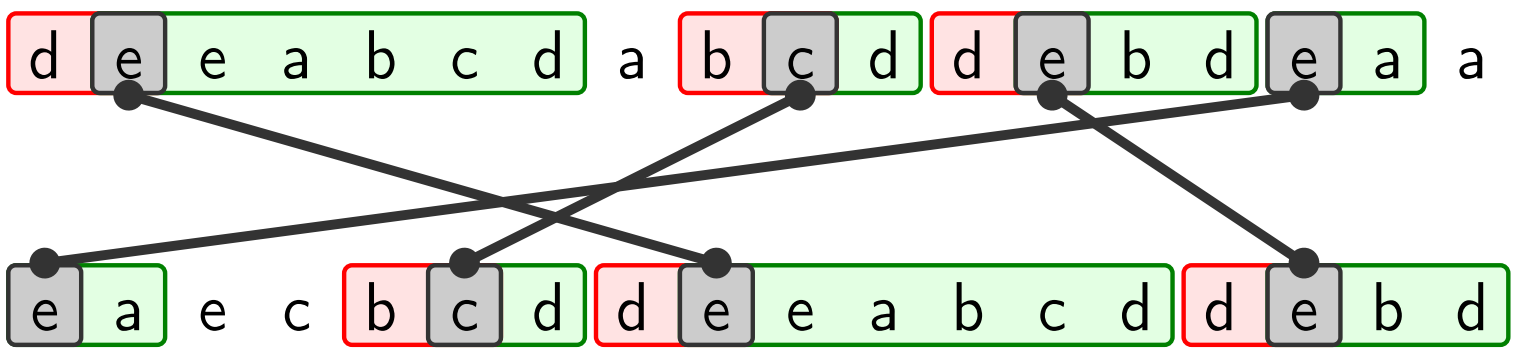
Constructing the solution

- No more seeds can be found...
- Create one block per seed



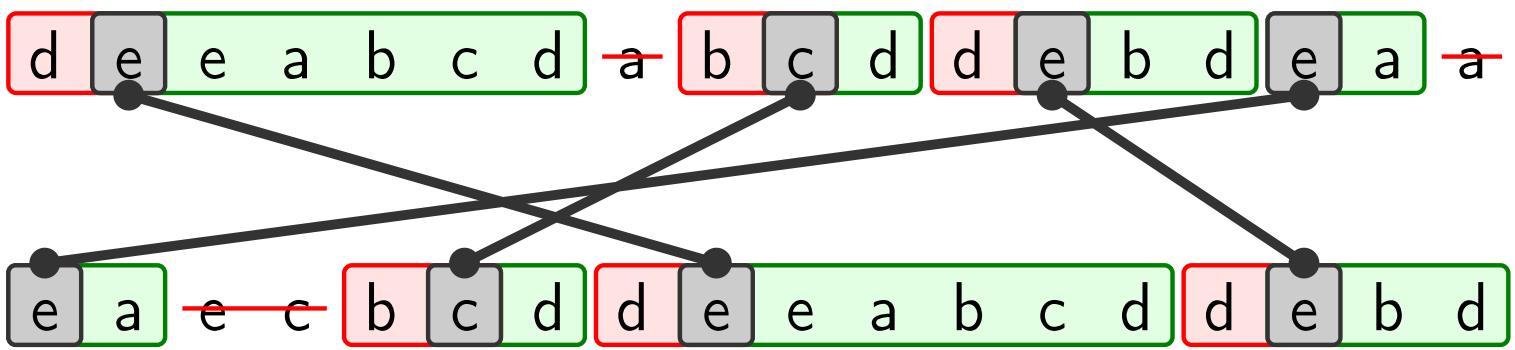
Constructing the solution

- No more seeds can be found...
- Create one block per seed
- Extend the blocks to the left and right (details omitted)



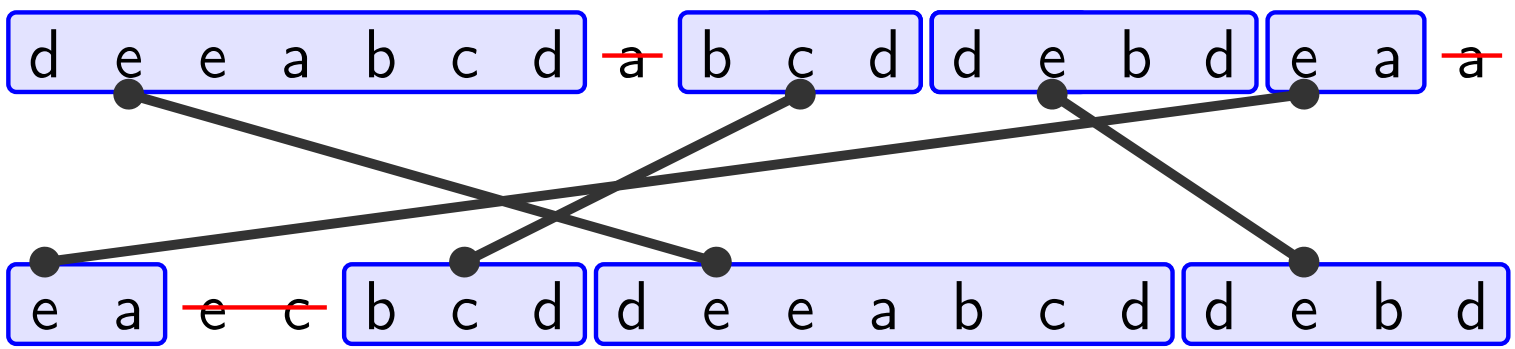
Constructing the solution

- No more seeds can be found...
- Create one block per seed
- Extend the blocks to the left and right (details omitted)
- Delete remaining (abundant) elements



Constructing the solution

- No more seeds can be found...
- Create one block per seed
- Extend the blocks to the left and right (details omitted)
- Delete remaining (abundant) elements
- Output the solution!

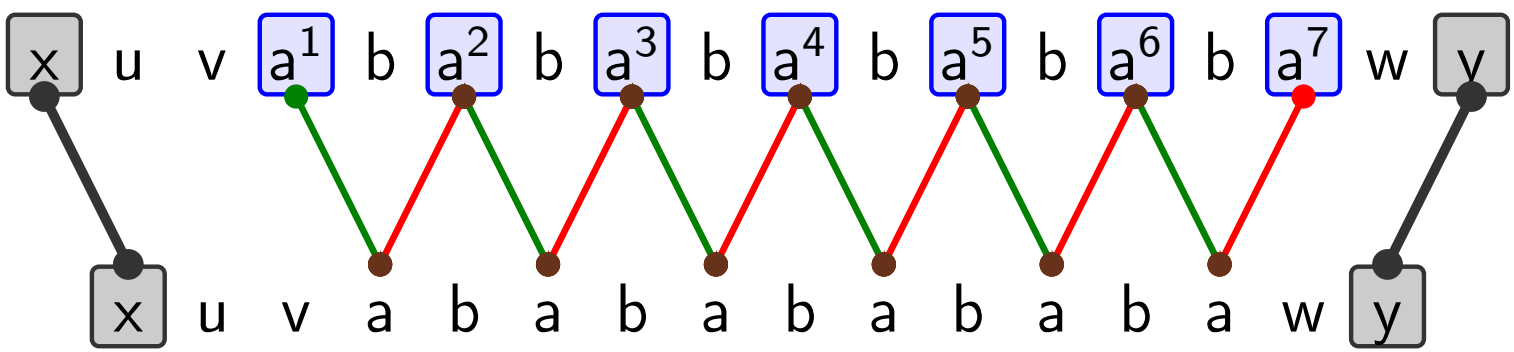


Theoretical Time Complexity

- Time complexity:
  - d or d^2 choices for each of the k seeds: d^{2k-1} branches
  - Computing the graph, looking for paths, etc.: $O(kn)$
  - Overall: $O(d^{2k}n)$
- The algorithm may try to optimize a second objective value.
- Many duplicates, few blocks: from d^2 to $3dk$
  If an odd path visits many times the same intervals, keep only $\leq 3k$ candidate seeds.

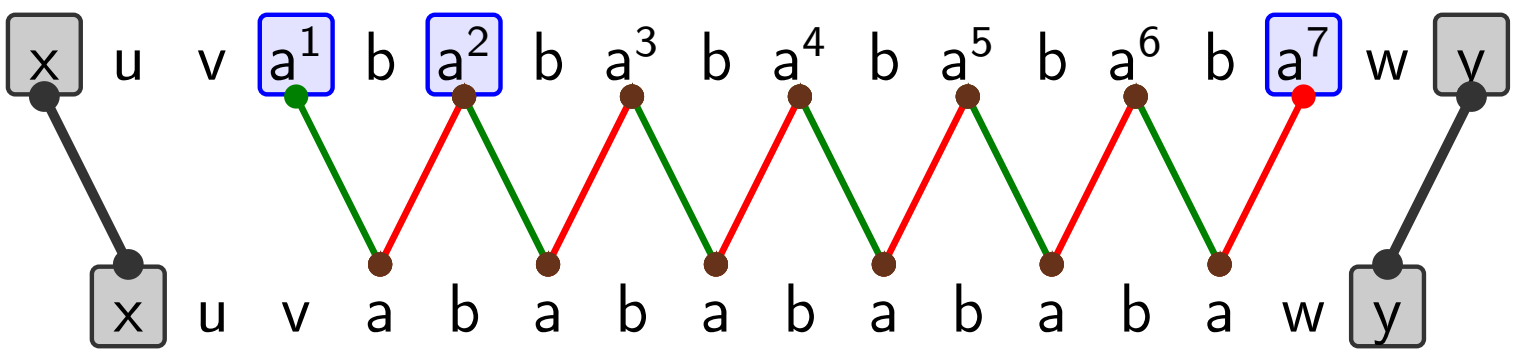
Theoretical Time Complexity

- Time complexity:
  - d or d^2 choices for each of the k seeds: d^{2k-1} branches
  - Computing the graph, looking for paths, etc.: $O(kn)$
  - Overall: $O(d^{2k}n)$
- The algorithm may try to optimize a second objective value.
- Many duplicates, few blocks: from d^2 to $3dk$
  If an odd path visits many times the same intervals, keep only $\leq 3k$ candidate seeds.



Theoretical Time Complexity

- Time complexity:
  - d or d^2 choices for each of the k seeds: d^{2k-1} branches
  - Computing the graph, looking for paths, etc.: $O(kn)$
  - Overall: $O(d^{2k}n)$
- The algorithm may try to optimize a second objective value.
- Many duplicates, few blocks: from d^2 to $3dk$
  If an odd path visits many times the same intervals, keep only $\leq 3k$ candidate seeds.

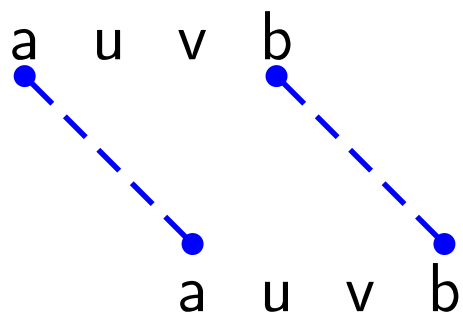


Implementation

Reduction rules

Reduction Rule 1

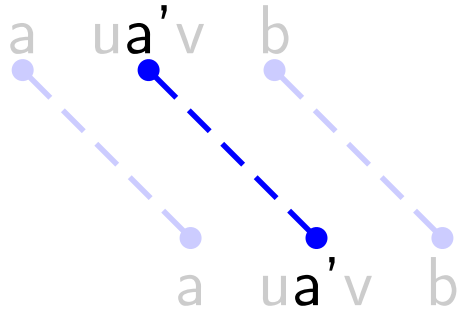
Merge strings with unique letters



Reduction rules

Reduction Rule 1

Merge strings with unique letters



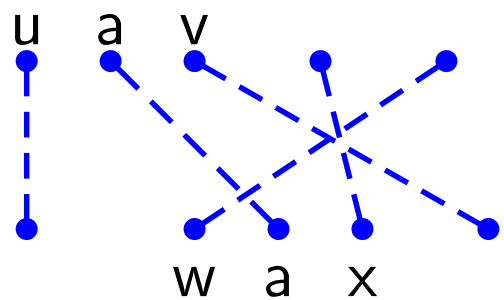
Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks



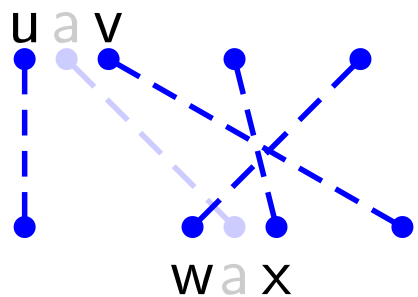
Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks



Decrease k by 1

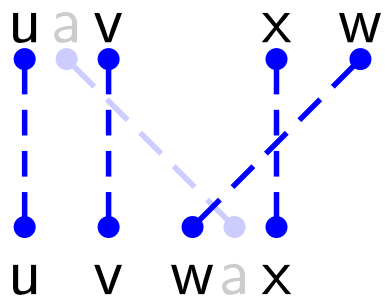
Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks



Decrease k by 2

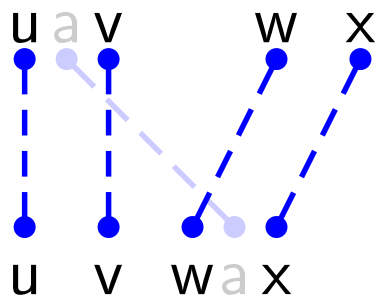
Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks



Decrease k by 3

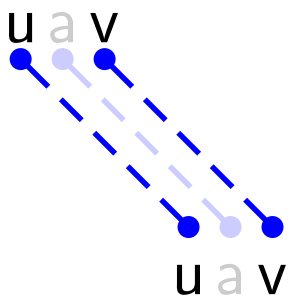
Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks



Leave k unchanged

Reduction rules

Reduction Rule 1

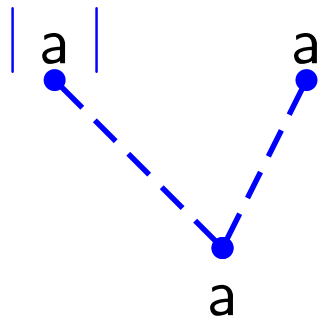
Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks

Reduction Rule 3

Keep best match (one unique letter)



Reduction rules

Reduction Rule 1

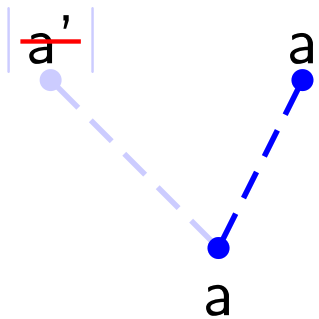
Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks

Reduction Rule 3

Keep best match (one unique letter)



Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

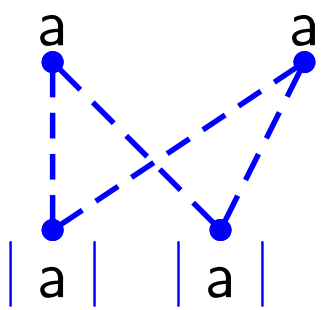
Remove obvious size-1 blocks

Reduction Rule 3

Keep best match (one unique letter)

Reduction Rule 4

Keep best match (≤ 2 occurences)



Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

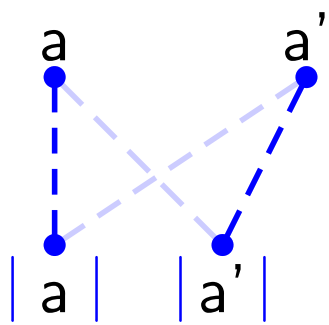
Remove obvious size-1 blocks

Reduction Rule 3

Keep best match (one unique letter)

Reduction Rule 4

Keep best match (≤ 2 occurences)



Reduction rules

Reduction Rule 1

Merge strings with unique letters

Reduction Rule 2

Remove obvious size-1 blocks

Reduction Rule 3

Keep best match (one unique letter)

Reduction Rule 4

Keep best match (≤ 2 occurences)

Reduction of genome size: from 40% to 85%

Running time on biological data

Genomic sequences from: *Borrelia burgdorferi*, *Treponema pallidum*, *Escherichia coli*, *Bacillus subtilis*, and *Bacillus thuringiensis*

Species 1	Species 2	n	k	d	t (s)
B. burg.	*T. pall.*	93	68	3	0.06
B. burg.	*E. coli*	72	59	6	0.22
B. burg.	*B. sub.*	91	63	6	0.15
B. burg.	*B. thur.*	71	51	5	0.09
T. pall.	*E coli*	93	78	5	0.35
T. pall.	*B. sub.*	144	82	7	0.18
T. pall.	*B. thur.*	128	76	6	0.15
E. coli	*B. sub.*	287	234	7	41.06
E. coli	*B. thur.*	282	221	10	18.64
B. sub.	*B. thur.*	693	340	8	249.71

EnsemblBacteria database

Running time on biological data

Genomic sequences from: *Borrelia burgdorferi*, *Treponema pallidum*, *Escherichia coli*, *Bacillus subtilis*, and *Bacillus thuringiensis*

Average letter occurences: from 1.02 to 1.24

Species 1	Species 2	n	k	d	t (s)
B. burg.	*T. pall.*	93	68	3	0.06
B. burg.	*E. coli*	72	59	6	0.22
B. burg.	*B. sub.*	91	63	6	0.15
B. burg.	*B. thur.*	71	51	5	0.09
T. pall.	*E coli*	93	78	5	0.35
T. pall.	*B. sub.*	144	82	7	0.18
T. pall.	*B. thur.*	128	76	6	0.15
E. coli	*B. sub.*	287	234	7	41.06
E. coli	*B. thur.*	282	221	10	18.64
B. sub.	*B. thur.*	693	340	8	249.71

EnsemblBacteria database

Running time on synthetic data

Generated sequences of size $n = 1000$
Average letter occurences: from 2 to 4

k	t (s)	
	$d = 6$	$d = 8$
50	0.06	0.07
60	0.06	0.06
70	0.07	0.08
80	0.09	0.09
90	0.10	0.12
100	0.12	0.16
110	0.13	0.26
120	0.18	1.62
130	0.21	30.42

Conclusion

Conclusion

Practical FPT algorithms for MCSP with association graph:

- $O\left(d^{2k}n\right)$
- $O\left((3dk)^{k}n\right)$

Conclusion

Practical FPT algorithms for MCSP with association graph:

- $O\left(d^{2k}n\right)$
- $O\left((3dk)^{k}n\right)$

Open questions:

- Extend algorithm to signed strings?
- Practical running time with high number of duplications?
- Constant-ratio approximation?