Greedy search

Torgeir R. Hvidsten

This lecture

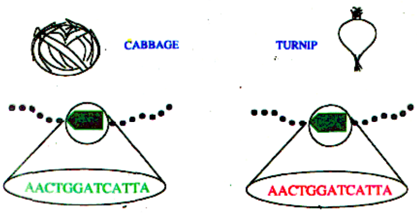
- Genome rearrangements
  - Sorting by reversals
  - Approximation algorithms
  - Breakpoints: a different face of greed
- Finding regulatory motifs in DNA Sequences
- Greedy search methods

Genome rearrangements

Turnip vs cabbage: Look and taste different

Although cabbages and turnips share a recent common ancestor, they look and taste different

Turnip vs cabbage: Comparing gene sequences yields no evolutionary information



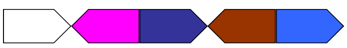
AACTGGATCATTA
AACTGGATCATTA

Turnip vs cabbage: Almost identical mtDNA gene sequences

- In 1980s Jeffrey Palmer studied evolution of plant organelles by comparing mitochondrial genomes of the cabbage and turnip
- 99% similarity between genes
- These surprisingly identical gene sequences differed in gene order
- This study helped pave the way to analyzing genome rearrangements in molecular evolution

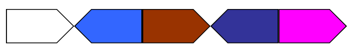
Turnip vs cabbage: Different mtDNA gene order

Gene order comparison:



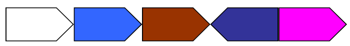
Turnip vs cabbage: Different mtDNA gene order

Gene order comparison:



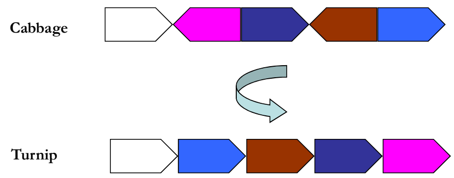
Turnip vs cabbage: Different mtDNA gene order

Gene order comparison:



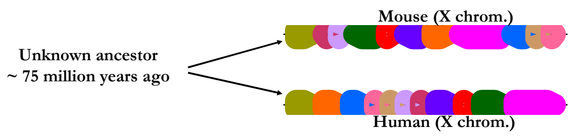
Turnip vs cabbage: Different mtDNA gene order

Gene order comparison:



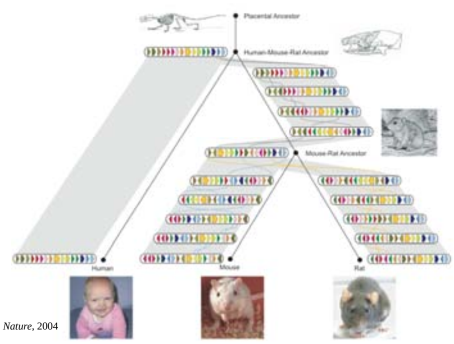
Evolution is manifested as the divergence in gene order

Genome rearrangements



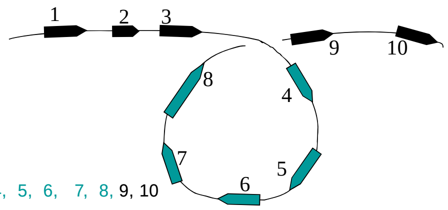
- What are the similarity blocks and how can they be found?
- What is the architecture of the ancestral genome?
- What is the evolutionary scenario for transforming one genome into the other?

History of chromosome X



Rat Consortium, *Nature*, 2004

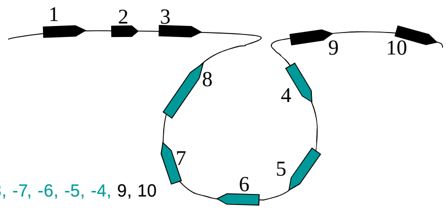
Reversals



1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Blocks represent conserved genes

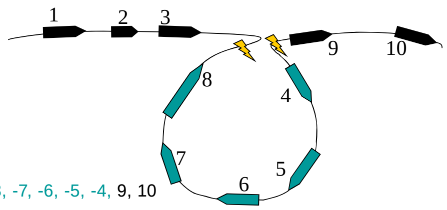
Reversals



1, 2, 3, -8, -7, -6, -5, -4, 9, 10

In the course of evolution, blocks 1,…,10 could be misread as 1, 2, 3, -8, -7, -6, -5, -4, 9, 10

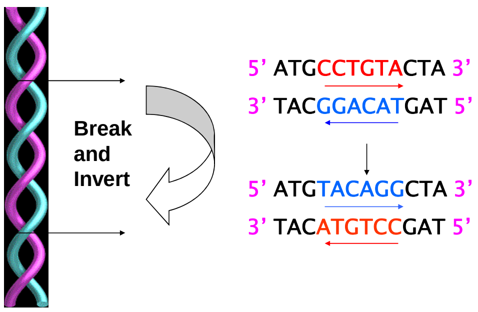
Reversals and breakpoints



1, 2, 3, -8, -7, -6, -5, -4, 9, 10

The reversion introduced two breakpoints  (disruptions in order)

Reversals: Example



Types of rearrangements

Reversal

1 2 3 4 5 6 → 1 2 -5 -4 -3 6

Translocation

1 2 3
4 5 6 → 1 2 6
4 5 3

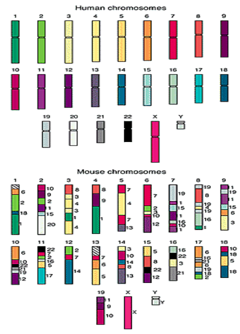
Fusion

1 2 3 4
5 6 ⇄ 1 2 3 4 5 6

Fission

Comparative genomic architectures: Mouse vs human genome

- Humans and mice have similar genomes, but their genes are ordered differently
- ~245 rearrangements
  - Reversals
  - Fusions
  - Fissions
  - Translocation



Waardenburg's syndrome: Mouse provides insight into human genetic disorder

- Waardenburg's syndrome is characterized by pigmentary dysphasia
- Gene implicated in the disease was linked to human chromosome 2 but it was not clear where exactly it is located on chromosome 2

Waardenburg's syndrome and splotch mice

- A breed of mice (with splotch gene) had similar symptoms caused by the same type of gene as in humans
- Scientists succeeded in identifying the location of the gene responsible for disorder in mice
- Finding the gene in mice gives clues to where the same gene is located in humans by analyzing the relative architecture of human and mouse genomes

Reversals: Example

$\pi = 1\ 2\ \underline{3\ 4\ 5}\ 6\ 7\ 8$

$\rho(3,5)$ ↓

$1\ 2\ 5\ 4\ 3\ \underline{6\ 7}\ 8$

$\rho(5,6)$ ↓

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

Reversals and gene orders

- Gene order is represented by a permutation π :

$$\pi = \pi_1 \dots \pi_{i-1}\ \underline{\pi_i \pi_{i+1} \dots \pi_{j-1} \pi_j}\ \pi_{j+1} \dots \pi_n$$

$\rho(i, j)$ ↓

$$\pi \cdot \rho(i, j) = \pi_1 \dots \pi_{i-1}\ \underline{\pi_j \pi_{j-1} \dots \pi_{i+1} \pi_i}\ \pi_{j+1} \dots \pi_n$$

- Reversal $\rho(i, j)$ reverses (flips) the elements from i to j in π

Reversal distance problem

- Goal: Given two permutations, find the shortest series of reversals that transforms one into another
- Input: Permutations π and σ
- Output: A series of reversals $\rho_1, \dots \rho_t$ transforming π into σ, such that t is minimum
- t - reversal distance between π and σ
- $d(\pi, \sigma)$ - smallest possible value of t, given π and σ

Sorting by reversals problem

- Goal: Given a permutation, find a shortest series of reversals that transforms it into the identity permutation ($1\ 2 \dots n$)
- Input: Permutation π
- Output: A series of reversals $\rho_1, \dots \rho_t$ transforming π into the identity permutation such that t is minimum

Sorting by reversals: Example

- $t = d(\pi)$ - reversal distance of π
- Example :

π = 3 4 2 1 5 6 7 10 9 8
4 3 2 1 5 6 7 10 9 8
4 3 2 1 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

So $d(\pi) = 3$

Sorting by reversals: A greedy algorithm

- When sorting permutation π = 1 2 3 6 4 5, the first three elements are already in order so it makes no sense to break them
- The length of the already sorted prefix of π is denoted $prefix(\pi)$
  - $prefix(\pi) = 3$
- This results in an idea for a greedy algorithm: increase $prefix(\pi)$ at every step

Greedy algorithm: An example

- Sorting π :

1 2 3 6 4 5
↓
1 2 3 4 6 5
↓
1 2 3 4 5 6

- Number of steps to sort permutation of length n is at most $(n - 1)$

Greedy Algorithm: Pseudo-code

SimpleReversalSort(π)
for $i \leftarrow 1$ to $n - 1$
  $j \leftarrow$ Index of element i in π (i.e., $\pi_j = i$)
  if $j \neq i$
    $\pi \leftarrow \pi \cdot \rho(i, j)$
    output π
  if π is the identity permutation
    return

Analyzing SimpleReversalSort (I)

- SimpleReversalSort does not guarantee the smallest number of reversals
- It takes five steps on $\pi = 6\ 1\ 2\ 3\ 4\ 5$:
  - Step 1: *1 6 2 3 4 5*
  - Step 2: *1 2 6 3 4 5*
  - Step 3: *1 2 3 6 4 5*
  - Step 4: *1 2 3 4 6 5*
  - Step 5: *1 2 3 4 5 6*

Analyzing SimpleReversalSort (II)

- But it can be sorted in two steps: $\pi = 6\ 1\ 2\ 3\ 4\ 5$
  - Step 1: *5 4 3 2 1 6*
  - Step 2: *1 2 3 4 5 6*
- So, SimpleReversalSort(π) is not optimal
- SimpleReversalSort is not correct (according to the definition in lecture 1)
- Optimal/correct algorithms are unknown for many problems; approximation algorithms are used

Approximation algorithms

- These algorithms find approximate solutions rather than optimal solutions
- The approximation ratio of an algorithm A on input π is:

$$A(\pi) \ / \ OPT(\pi)$$

where

A(π) - solution produced by algorithm A
OPT(π) - optimal solution of the problem

Performance guarantee

- Approximation ratio (performance guarantee) of algorithm A is the maximal approximation ratio of all inputs of size n
- For algorithm A that minimizes the objective function (minimization algorithm):
  - $\max_{|\pi| = n} A(\pi) \ / \ OPT(\pi)$
- For maximization algorithms
  - $\min_{|\pi| = n} A(\pi) \ / \ OPT(\pi)$

Performance guarantee: Example

- SimpleReversalSort is a minimization algorithm
- Performance guarantee:
  - $\max_{|\boldsymbol{\pi}| = n} \mathrm{A}(\boldsymbol{\pi}) \ / \ \mathrm{OPT}(\boldsymbol{\pi})$
  - We have seen that $\mathrm{A}(\boldsymbol{\pi}) = n\text{-}1$ for a problem that could be solved in two steps
  - So, the approximation ratio is at least $(n\text{-}1)/2$

Adjacencies

$\boldsymbol{\pi} = \pi_1 \pi_2 \pi_3 \ldots \pi_{n-1} \pi_n$

- A pair of elements π_i and π_{i+1} are adjacent if

  $\pi_{i+1} = \pi_i \pm 1$
- For example:

  $\boldsymbol{\pi} = 1 \ 9 \ \underline{3 \ 4} \ \underline{7 \ 8} \ 2 \ \underline{6 \ 5}$
- (3, 4), (7, 8) and (6,5) are adjacent pairs

Breakpoints: An example

- There is a breakpoint between any neighboring elements that are non-consecutive (not adjacent):

  $\boldsymbol{\pi} = 1 | 9 | 3 \ 4 | 7 \ 8 | 2 | 6 \ 5$
- Pairs (1,9), (9,3), (4,7), (8,2) and (2,6) form breakpoints of permutation $\boldsymbol{\pi}$
- $b(\boldsymbol{\pi})$ – number of breakpoints in permutation $\boldsymbol{\pi}$

Adjacencies and breakpoints

- An adjacency - a pair of neighboring elements that are consecutive
- A breakpoint - a pair of neighboring elements that are not consecutive

---

$\boldsymbol{\pi} = 5 \ 6 \ 2 \ 1 \ 3 \ 4$ ⟶ Extend $\boldsymbol{\pi}$ with $\pi_0 = 0$ and $\pi_7 = 7$

adjacencies

$0 \ 5 \ 6 \ 2 \ 1 \ 3 \ 4 \ 7$

breakpoints

Reversal distance and breakpoints

Each reversal eliminates at most 2 breakpoints i.e. reversal distance $\geq$ b(π) / 2

π = *2 3 1 4 6 5*

0 | *2 3* | *1* | *4* | *6 5* | *7*	$b(\pi) = 5$
0 1 | *3 2* | *4* | *6 5* | *7*	$b(\pi) = 4$
0 1 2 3 4 | *6 5* | *7*	$b(\pi) = 2$
0 1 2 3 4 5 6 7	$b(\pi) = 0$

Sorting by reversals: A better greedy algorithm

```
BreakPointReversalSort(π)
while b(π) > 0
    Among all possible reversals, choose reversal
    ρ minimizing b(π · ρ)
    π ← π · ρ(i,j)
    output π
return
```

Problem: It is not obvious that this algorithm will terminate!

Strips

Strip: an interval between two consecutive breakpoints in a permutation

- Decreasing strip: strip of elements in decreasing order (e.g. *6 5* and *3 2*).
- Increasing strip: strip of elements in increasing order (e.g. *7 8*)

0 1 9 4 3 7 8 2 5 6 10

- A single-element strip can be declared either increasing or decreasing. We will choose to declare them as decreasing with exception of the strips with *0* and *n+1*

Reducing the Number of Breakpoints

Theorem 1:

If permutation π contains at least one decreasing strip, then there exists a reversal ρ which decreases the number of breakpoints (i.e. $b(\pi \cdot \rho) < b(\pi)$)

"Proof"

For π = *1 4 6 5 7 8 3 2*

0 1|4|6 5|7 8|3 2|9 $b(\pi) = 5$

– Choose the decreasing strip with the smallest element k in π (k = 2 in this case)

– Find $k - 1$ in the permutation

– Reverse the segment between k and k-*1*:

0 1 4 6 5 7 8 3 2 9 $b(\pi) = 5$

↓

0 1 2 3 8 7 5 6 4 9 $b(\pi) = 4$

Reducing the number of breakpoints again

- If there is no decreasing strip, there may be no reversal ρ that reduces the number of breakpoints
- By reversing an increasing strip (the number of breakpoints stay unchanged), we will create a decreasing strip at the next step. Then the number of breakpoints will be reduced in the next step (Theorem 1)

ImprovedBreakpointReversalSort

```
ImprovedBreakpointReversalSort(π)
while b(π) > 0
    if π has a decreasing strip
        Among all possible reversals, choose reversal ρ
        that minimizes b(π·ρ)
    else
        Choose a reversal ρ that flips an increasing strip in π
    π ← π·ρ
    output π
return
```


ImprovedBreakpointReversalSort: Performance guarantee

ImprovedBreakPointReversalSort is an approximation algorithm with a performance guarantee of at most *4*

– It eliminates at least one breakpoint in every two steps; at most $2b(\pi)$ steps

– Approximation ratio: $2b(\pi) \; / \; d(\pi)$

– Optimal algorithm eliminates at most *2* breakpoints in every step: $d(\pi) \geq b(\pi) \; / \; 2$

– Performance guarantee:

$$(2b(\pi) \; / \; d(\pi)) \leq [\, 2b(\pi) \; / \; (b(\pi) \; / \; 2) \,] = 4$$

Finding regulatory motifs in DNA sequences

Implanting motif **AAAAAAGGGGGGG** with four random mutations

```
atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaatacAAtAAAAcGGcGGGa
tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag
gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAtAAAcAAGGaGGGccaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatActAAAAAGGaGcGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa
```

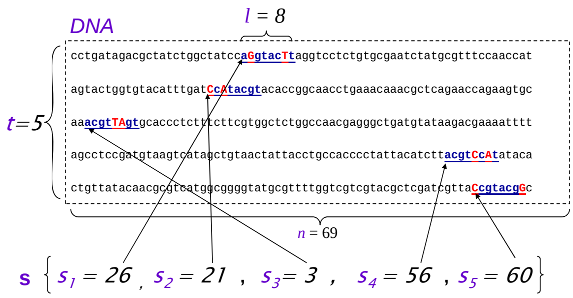

Where is the motif?

```
atgaccgggatactgatagaagaaaggttgggggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaatacaataaaacggcggga
tgagtatccctgggatgacttaaaataatggagtggtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgcaaaaaaagggattgtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatataataaaggaagggcttatag
gtcaatcatgttcttgtgaatggatttaacaataagggctgggaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtataaacaaggagggccaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttaaaaaatagggagccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatactaaaaaggagcggaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttactaaaaaggagcgga
```


Definitions

t	number of sample DNA sequences
n	length of each DNA sequence
DNA	sample of DNA sequences (t x n array)
l	length of the motif (l - mer)
s_i	starting position of an l -mer in sequence i
$\mathbf{s}=(s_1, s_2,\ldots s_t)$	array of motif starting positions

Parameters



Scoring motifs: consensus score

- Given $\boldsymbol{s} = (s_1, \ldots s_t)$ and $\boldsymbol{DNA}$:
- Score($\boldsymbol{s}$,$\boldsymbol{DNA}$) =

$$\sum_{i=1}^{l} \max_{k \in \{A,T,C,G\}} count(k,i)$$

```
a G g t a c T t
C c A t a c g t
a c g t T A g t
a c g t C c A t
C c g t a c g G
```

A	3	0	1	0	3	1	1	0
C	2	4	0	0	1	4	0	0
G	0	1	4	0	0	0	3	1
T	0	0	0	5	1	0	1	4

Consensus a c g t a c g t

Score 3+4+4+5+3+4+3+4=**30**

Greedy motif finding

- Partial score: Score($\boldsymbol{s}$, i, $\boldsymbol{DNA}$)
  - The consensus score for the first i sequences
- Algorithm:
  - Find the optimal motif for the two first sequences
  - Scan the remaining sequences only once, and choose the motif with the best contribution to the partial score

Greedy motif finding

```
GreedyMotifSearch(DNA, t, n, l)
    s ← (1,1, …, 1)
    bestMotif ← s
    for s1 ← 1 to n – l + 1
        for s2 ← 1 to n – l + 1
            if Score(s, 2, DNA) > Score(bestMotif, 2, DNA)
                bestMotif1 ← s1
                bestMotif2 ← s2
    s1 ← bestMotif1
    s2 ← bestMotif2
    for i ← 3 to t
        for si ← 1 to n – l + 1
            if Score(s, i, DNA) > Score(bestMotif, i, DNA)
                bestMotifi ← si
        si ← bestMotifi
    return bestMotif
```

Running time

- Optimal motif for the two first sequences
  - $l(n - l + 1)^2$ operations
- The remaining t-2 sequence
  - $(t - 2)l(n - l + 1)$ operations
- Running time
  - $O(ln^2 + tln)$ or $O(ln^2)$ if $n >> t$
- Vastly better than
  - BruteForceMotifSearch: $(n - l + 1)^t$
  - BruteForceMedianStringSearch: 4^l