# Review

CS 4364/5364
Spring 2021

# Exact String Matching

- Given string *P*, called the pattern, and a longer string *T*, called the text, the **exact matching** problem is to find all occurrences, if any, of *P* in *T*.

- Example:
  - *P* = "`aba`", *T* = "`bbabaxababay`"
  - *P* occurs in *T* at positions: 3, 7, & 9
  - Note, that 2 occurrences overlap

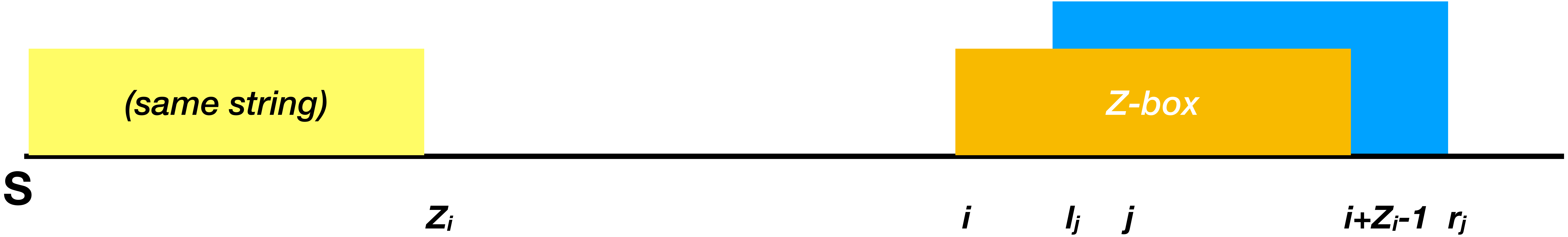
# Exact String Matching

Naïve algorithm

- linearly compare the pattern to each starting position in the text *O(nm)*

*Z-box* preprocessing

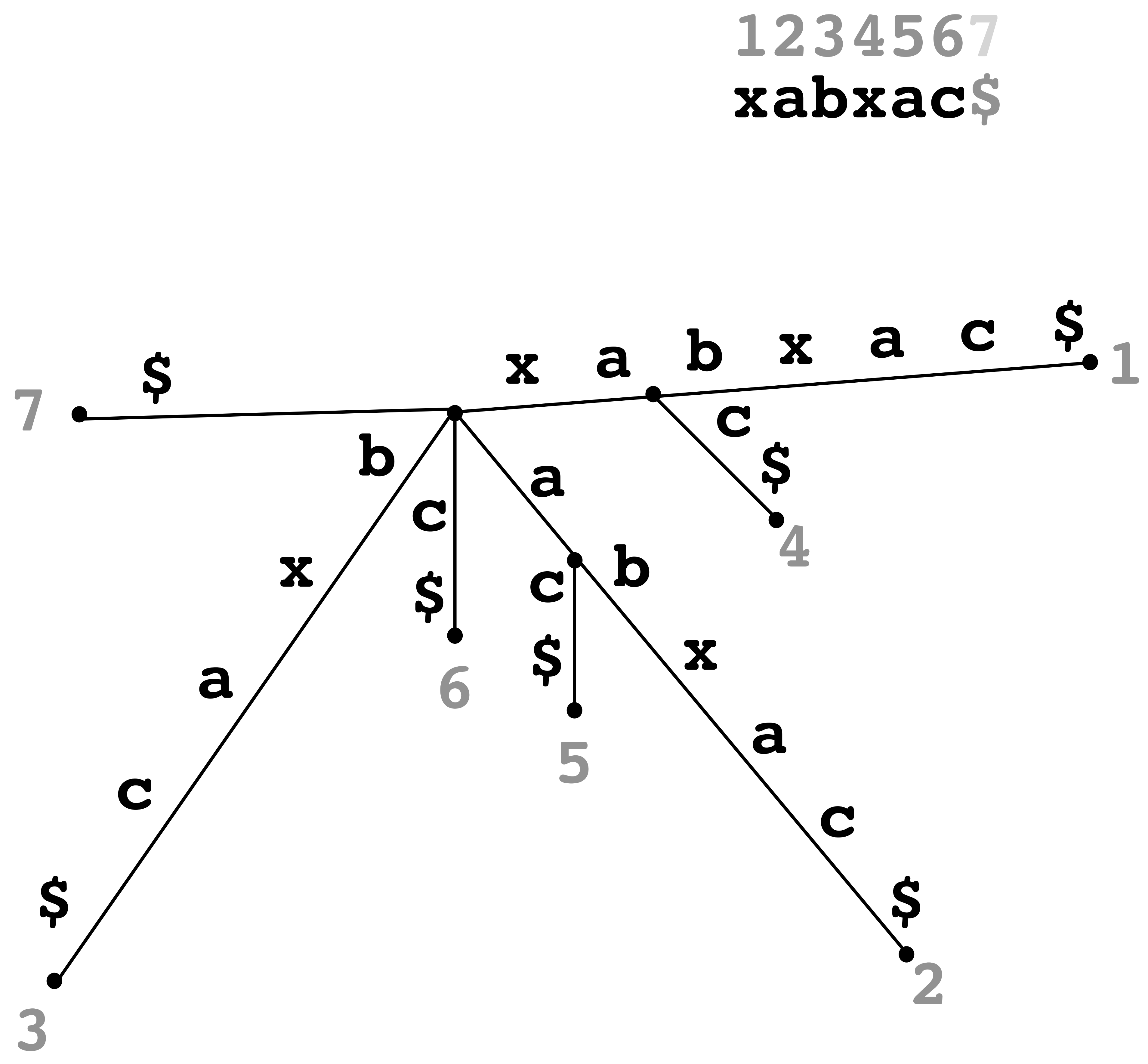
- in linear time identifies the longest string at each position that matches a prefix of that string



Boyer-Moore

- Match from right to left in the pattern, and move by more than one character

# Suffix Trees

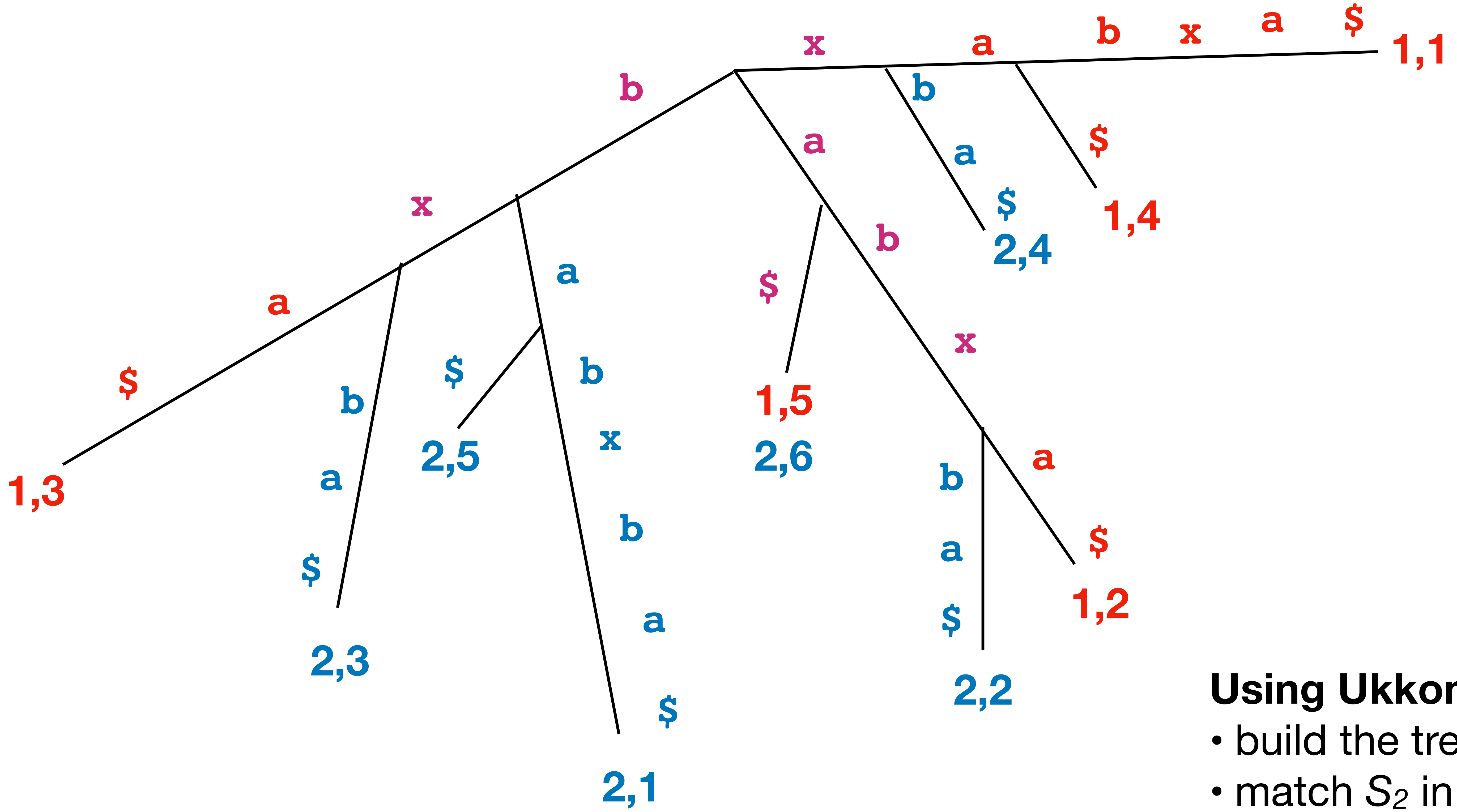


Ukkonen's algorithm builds a suffix tree in *O(m)*-time using 3 rules:

- **Rule 1** In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.
- **Rule 2** *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.
- **Rule 3** Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

# Generalized Suffix Trees

```
     123456
S1 = xabxa
S1 = babxba
```



**Using Ukkonen's Algorithm**

- build the tree for $S_1$
- match $S_2$ in the tree until a mismatch is found at $S_2[j]$
- restart the Ukkonen algorithm from $j$ (all suffixes of $S[1...j-1]$ are already in the tree)
- repeat for $S_3$, $S_4$, ... , $S_k$

# Suffix Arrays

A suffix array contains the starting position of the suffixes of a string when listed in lexicographic order.

One more concept:

> ***lcp(i,j)*** for positions *i* and *j* is the length of the longest common prefix of the suffixes at position *i* and *j* in the suffix array

`s = mississippi`

| | | |
|---|---|---|
| 11: | `i` | **1** |
| 8: | `ippi` | **1** |
| 5: | `issippi` | **4** |
| 2: | `ississippi` | **0** |
| 1: | `mississippi` | **0** |
| 10: | `pi` | **1** |
| 9: | `ppi` | **0** |
| 7: | `sippi` | **2** |
| 4: | `sissippi` | **1** |
| 6: | `ssippi` | **3** |
| 3: | `ssissippi` | **-** |

# Global Alignment Problem

- An **alignment** of two sequences is formed by inserting gap characters,'`-`', in arbitrary locations along the sequences so that they end up wit the same length and there are no two spaces at the same position of the two augmented strings.

```
baseball---
----ballcap
```

# Global Alignment Problem

- An **alignment** of two sequences is formed by inserting gap characters,'`-`', in arbitrary locations along the sequences so that they end up wit the same length and there are no two spaces at the same position of the two augmented strings.

```
baseball---      baseball
----ballcap      ballca-p
```

# Global Alignment Problem

- An **alignment** of two sequences is formed by inserting gap characters,'`-`', in arbitrary locations along the sequences so that they end up wit the same length and there are no two spaces at the same position of the two augmented strings.

```
baseball      baseball---      baseball
-ballcap      ----ballcap      ballca-p
```

# Global Alignment Problem

- An **alignment** of two sequences is formed by inserting gap characters,'`-`', in arbitrary locations along the sequences so that they end up wit the same length and there are no two spaces at the same position of the two augmented strings.

```
baseball         baseball---       baseball
-ballcap         ----ballcap       ballca-p
```

How do we know which one of these is best?

# Needleman-Wunsch

- Define an *n*x*m* array *V*, the cell *V(i,j)* will hold the score of the best sub alignments of *S[1...i]* and *T[1...j]*

# Needleman-Wunsch

- Define an *nxm* array *V*, the cell *V(i,j)* will hold the score of the best sub alignments of *S[1...i]* and *T[1...j]*

- The recurrence relation (the base of any DP)

$$V(i,j) = \max \begin{cases} V(i-1, j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ V(i-1, j) + \delta(S[i], -\ ) & \text{delete} \\ V(i, j-1) + \delta(\ -\ , T[j]) & \text{insert} \end{cases}$$

# Needleman-Wunsch

- Define an *nxm* array *V*, the cell *V(i,j)* will hold the score of the best sub alignments of *S[1...i]* and *T[1...j]*
- The recurrence relation (the base of any DP)

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$

- The initialization is:
  $V(0,0) = 0$
  $V(0,j) = V(0,j-1) + \delta(-,T[j])$
  $V(i,0) = V(i-1,0) + \delta(S[i],-)$

# Needleman-Wunsch

- Define an *nxm* array *V*, the cell *V(i,j)* will hold the score of the best sub alignments of *S[1...i]* and *T[1...j]*

- The recurrence relation (the base of any DP)

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i],-) & \text{delete} \\ V(i,j-1) + \delta(-,T[j]) & \text{insert} \end{cases}$$

- The initialization is:
  $V(0,0) = 0$
  $V(0,j) = V(0,j-1) + \delta(-,T[j])$
  $V(i,0) = V(i-1,0) + \delta(S[i],-)$

Optimal alignment score is in *V(n,m)*

# *Local* Alignment

- Given two strings *S* and *T*, find the two substrings, *A* of *S* and *B* of *T,* with the highest alignment score.
- Brute-force: Align all substrings of *S* with all substrings of *T.* There are $\binom{n}{2}$ substrings of *S*, and $\binom{m}{2}$ substrings of *T*. The total running time would be $O(n^3m^3)$*!*
- Smith and Waterman [1981] developed an algorithm, similar to Needleman-Wunch, that is able to find the optimal local alignment in *O(mn)*-time.

# Smith-Waterman

- The recurrence relation

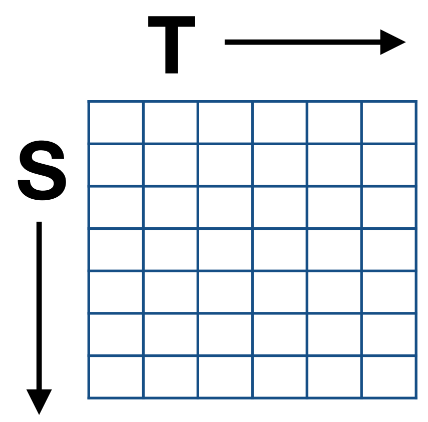
$$V(i,j) = \max \begin{cases} 0 & \text{align empty strings} \\ V(i-1,j-1) + \delta(S[i], T[i]) & \text{match/mismatch} \\ V(i-1,j) + \delta(S[i], -) & \text{delete} \\ V(i,j-1) + \delta(-, T[j]) & \text{insert} \end{cases}$$

- The initialization is:

  $V(0,j) = V(i,0) = 0$

# Semi-global Alignment

| Ignored spaces | Modification |
|---|---|
| The beginning of S | Initialize column 0 to 0s |
| The end of S | Search for the maximum value in the last column |
| The beginning of T | Initialize row 0 to 0s |
| The end of T | Search for the maximum value in the last row |



# Affine Gap Costs

- The one everyone uses!
- Attributed to Gotoh [1982]
- Define the function $f_{a,b}(k) =: a + b * k$ where $a$ and $b$ are tunable parameters (if $a=0$, this is the same as before)
- Can still be solved in $O(mn)$-time and $O(mn)$-space, but we need a bit more sophistication

# Affine Gap Costs

$$f_{\alpha,\beta,\gamma,\delta}(\mathbb{A}) = \alpha \cdot \mathbf{mt}_{\mathbb{A}} - \beta \cdot \mathbf{ms}_{\mathbb{A}} - \gamma \cdot \mathbf{id}_{\mathbb{A}} - \delta \cdot \mathbf{gp}_{\mathbb{A}}$$

- $\mathrm{mt}_{\mathbb{A}}$ -- number of columns where both characters match
- $\mathrm{ms}_{\mathbb{A}}$ -- number of columns where there characters are different (mismatches)
- $\mathrm{id}_{\mathbb{A}}$ -- number of gap characters (indels)
- $\mathrm{gp}_{\mathbb{A}}$ -- number of gaps

# Gotoh's Algorithm

## Recursion

$$F(i,j) = \max \begin{cases} F(i-1,j) - \gamma \\ G(i-1,j) - \gamma - \delta \end{cases}$$

$$E(i,j) = \max \begin{cases} E(i,j-1) - \gamma \\ G(i,j-1) - \gamma - \delta \end{cases}$$

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \alpha & \textbf{if } S[i] = T[i] \\ G(i-1,j-1) - \beta & \textbf{if } S[i] = T[i] \\ E(i,j) \\ F(i,j) \end{cases}$$

## Initialization

$$G(0,j) = E(0,j) = -1 * \left(\gamma + \delta j\right)$$

$$G(i,0) = F(i,0) = -1 * \left(\gamma + \delta j\right)$$

$$E(i,0) = -\infty$$

$$F(0,j) = -\infty$$

# An example

**$s_1$** = AACCCG

**$s_1$** = AAGGCC

$\mathbb{A}_1$

```
AA--CCCG
AAGGCC--
```

$\mathbb{A}_2$

```
AA-CCCG
AAGGCC-
```

$\mathbb{A}_3$

```
AACCCG
AAGGCC
```

$\mathbb{A}_4$

```
AAC-CCG
AAGGCC-
```

| | $\mathbb{A}_1$ | $\mathbb{A}_2$ | $\mathbb{A}_3$ | $\mathbb{A}_4$ |
|---|---|---|---|---|
| **mt** | 4 | 4 | 3 | 4 |
| **ms** | 0 | 1 | 3 | 1 |
| **id** | 4 | 2 | 0 | 2 |
| **gp** | 2 | 2 | 0 | 2 |

Question: what values of α,β,ɣ, and δ should we choose to get the “best” alignment?

# An example

**$s_1$** = AACCCG

**$s_1$** = AAGGCC

$\mathbb{A}_1$
```
AA--CCCG
AAGGCC--
```

$\mathbb{A}_2$
```
AA-CCCG
AAGGCC-
```

$\mathbb{A}_3$
```
AACCCG
AAGGCC
```

$\mathbb{A}_4$
```
AAC-CCG
AAGGCC-
```

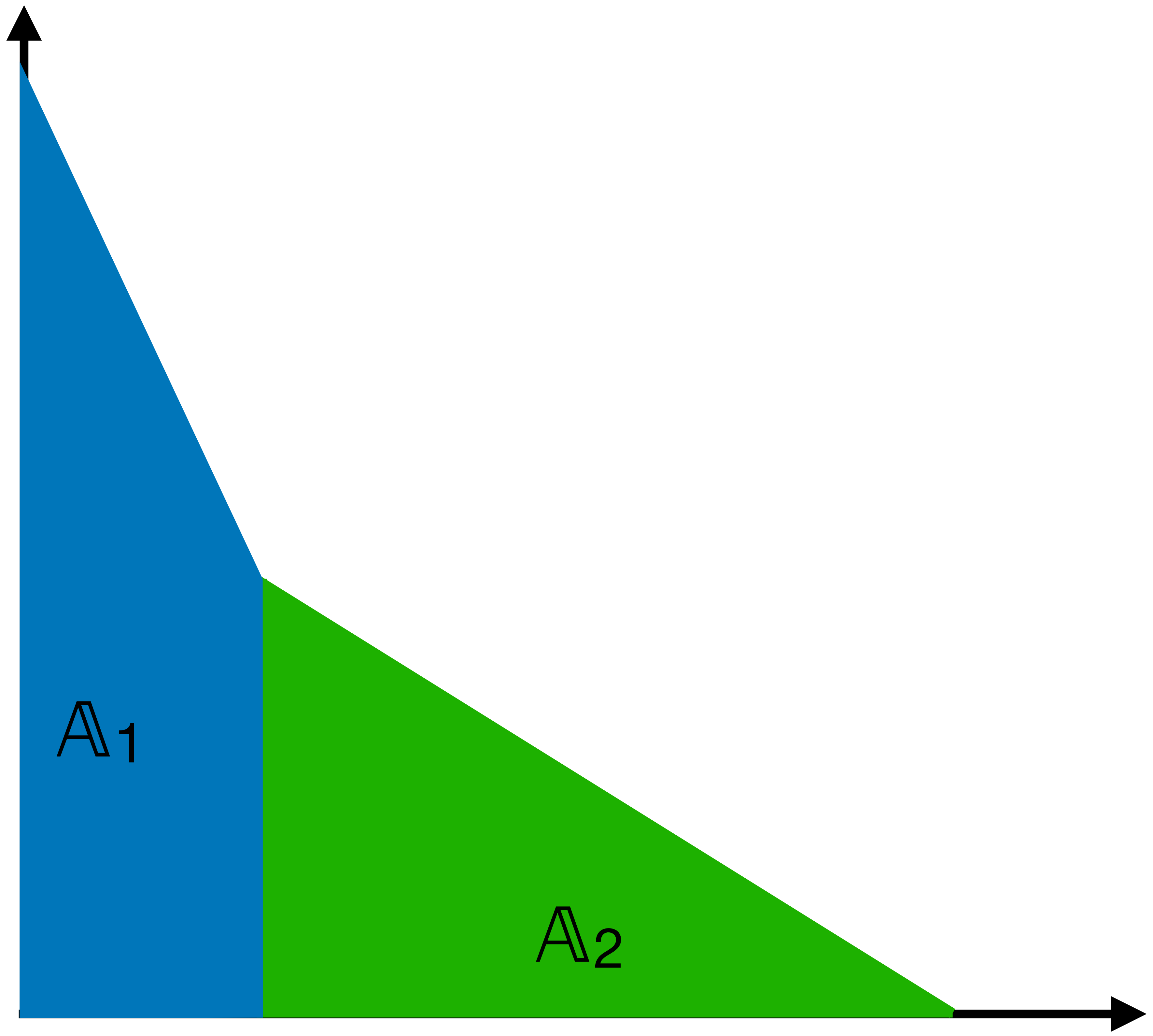
| | $\mathbb{A}_1$ | $\mathbb{A}_2$ | $\mathbb{A}_3$ | $\mathbb{A}_4$ |
|---|---|---|---|---|
| **mt** | 4 | 4 | 3 | 4 |
| **ms** | 0 | 1 | 3 | 1 |
| **id** | 4 | 2 | 0 | 2 |
| **gp** | 2 | 2 | 0 | 2 |

Question: what values of α,β,ɣ, and δ should we choose to get the “best” alignment?

**What do we even mean by "best"?**

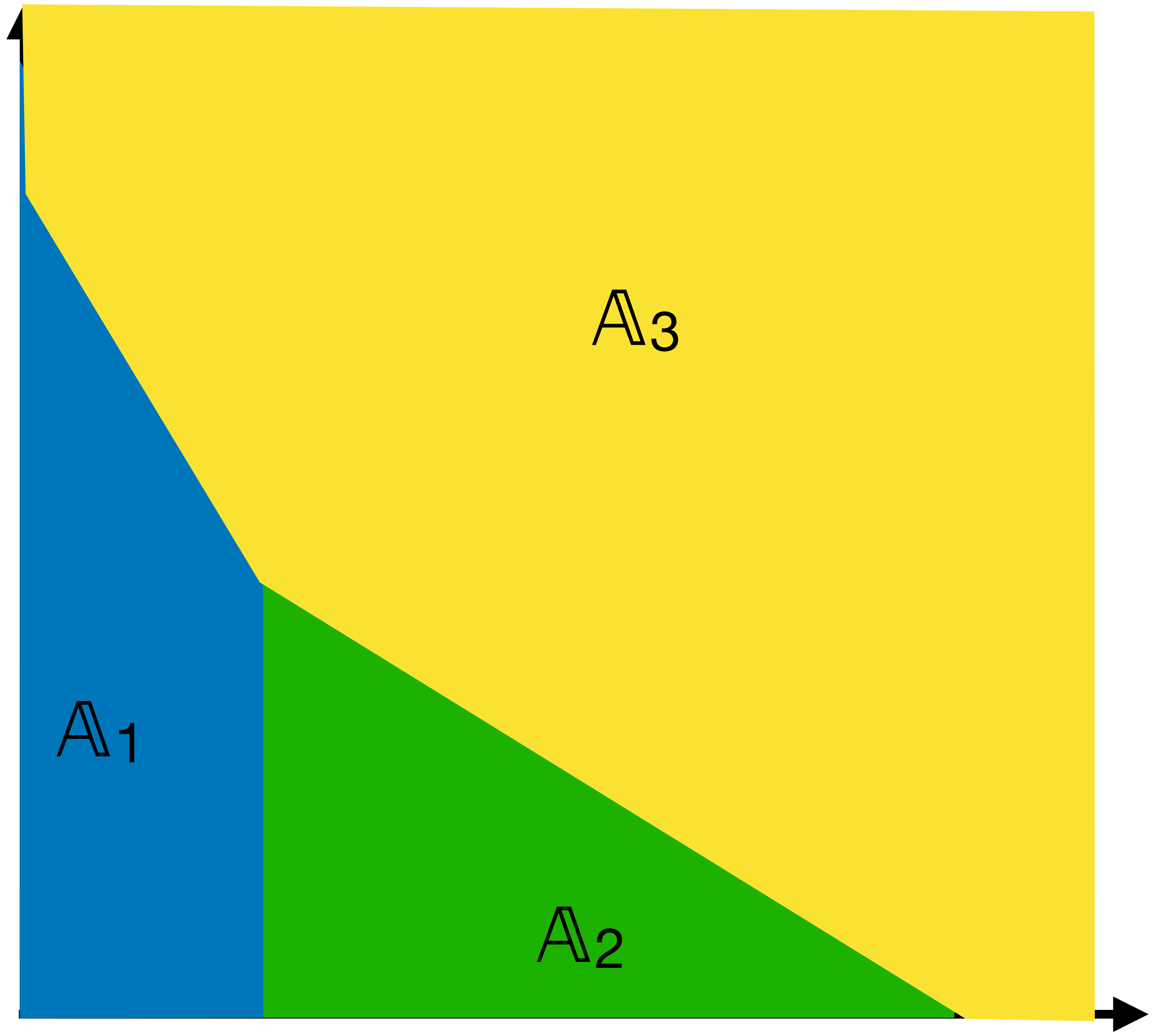
# Parametric Alignment

- when two parameters are free, there are only $O(n^2)$ different regions
- the boundaries are always lines
- the boundaries can be found in $O(n^4)$-time



# Parametric Alignment

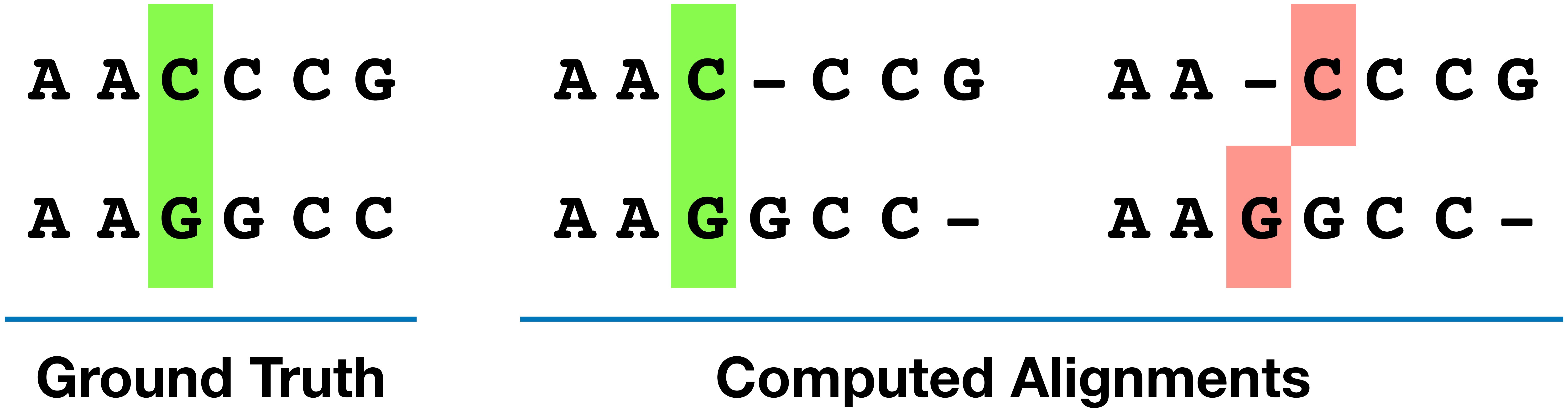
- when two parameters are free, there are only $O(n^2)$ different regions
- the boundaries are always lines
- the boundaries can be found in $O(n^4)$-time



# A Digression on Accuracy

How would we know how accurate an alignment was if we knew the right answer?

The **sum-of-pairs** accuracy measures the fraction of substitutions from the ground truth alignment that are recovered in a computed alignment



```
A A C C C G      A A C - C C G      A A - C C C G
A A G G C C      A A G G C C -      A A G G C C -
```

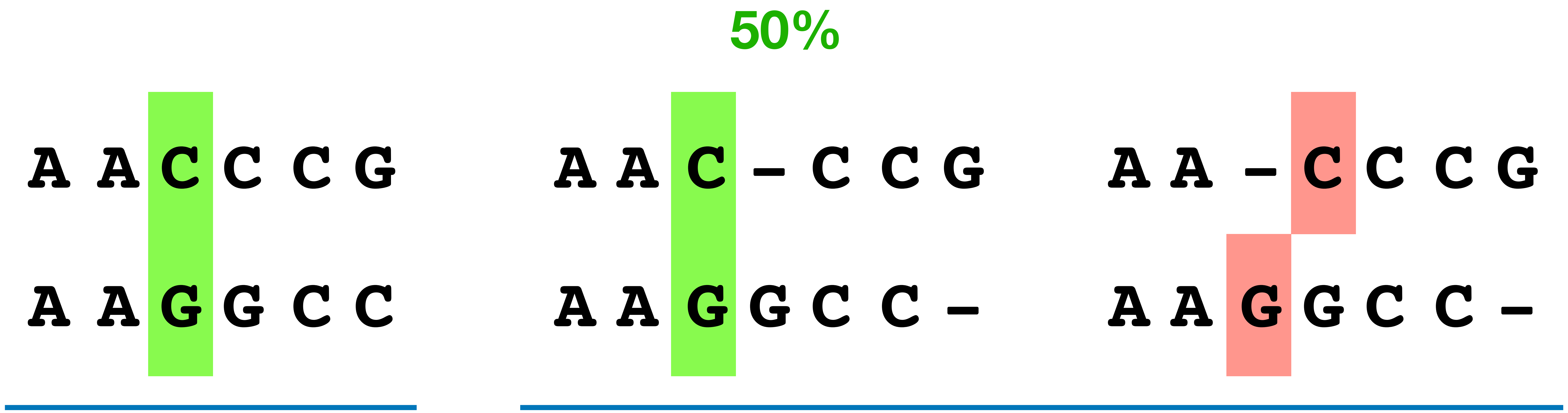
**Ground Truth**

**Computed Alignments**

# A Digression on Accuracy

How would we know how accurate an alignment was if we knew the right answer?

The **sum-of-pairs** accuracy measures the fraction of substitutions from the ground truth alignment that are recovered in a computed alignment



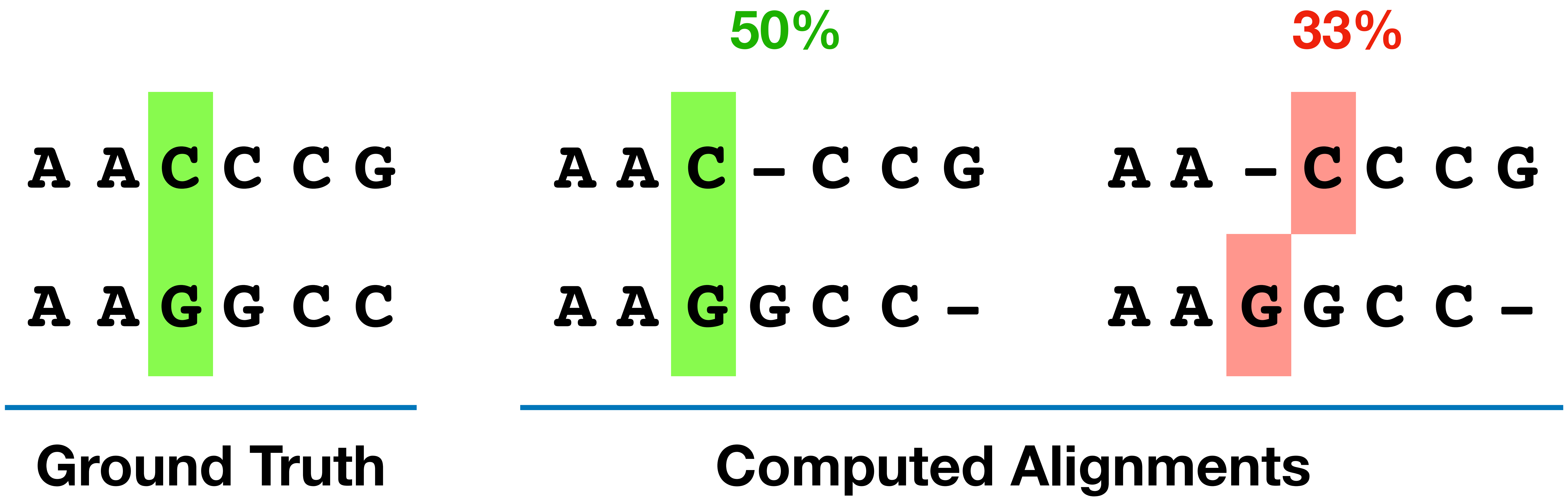
**Ground Truth**

**Computed Alignments**

# A Digression on Accuracy

How would we know how accurate an alignment was if we knew the right answer?

The **sum-of-pairs** accuracy measures the fraction of substitutions from the ground truth alignment that are recovered in a computed alignment



50%

33%

A A C C C G
A A G G C C

**Ground Truth**

A A C – C C G
A A G G C C –

A A – C C C G
A A G G C C –

**Computed Alignments**

# The (Sequence) Database Search Problem

Given a database *D* of sequences (DNA, Protein, Books, Web Pages) and a query string *Q* find the sting(s) *S* in *D* which is/are closest matches to *Q* under a defined scoring function.

# The (Sequence) Database Search Problem

Given a database *D* of sequences (DNA, Protein, Books, Web Pages) and a query string *Q* find the sting(s) *S* in *D* which is/are closest matches to *Q* under a defined scoring function.

Scoring functions are typically either

- **Semi-global alignment** -- The best possible alignment score between a substring *A* of *S* and *Q*, or
- **Local alignment** -- The vest possible alignment score between a substring *A* of *S* and a substring *B* of *Q*.

# FastA/FastP

**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

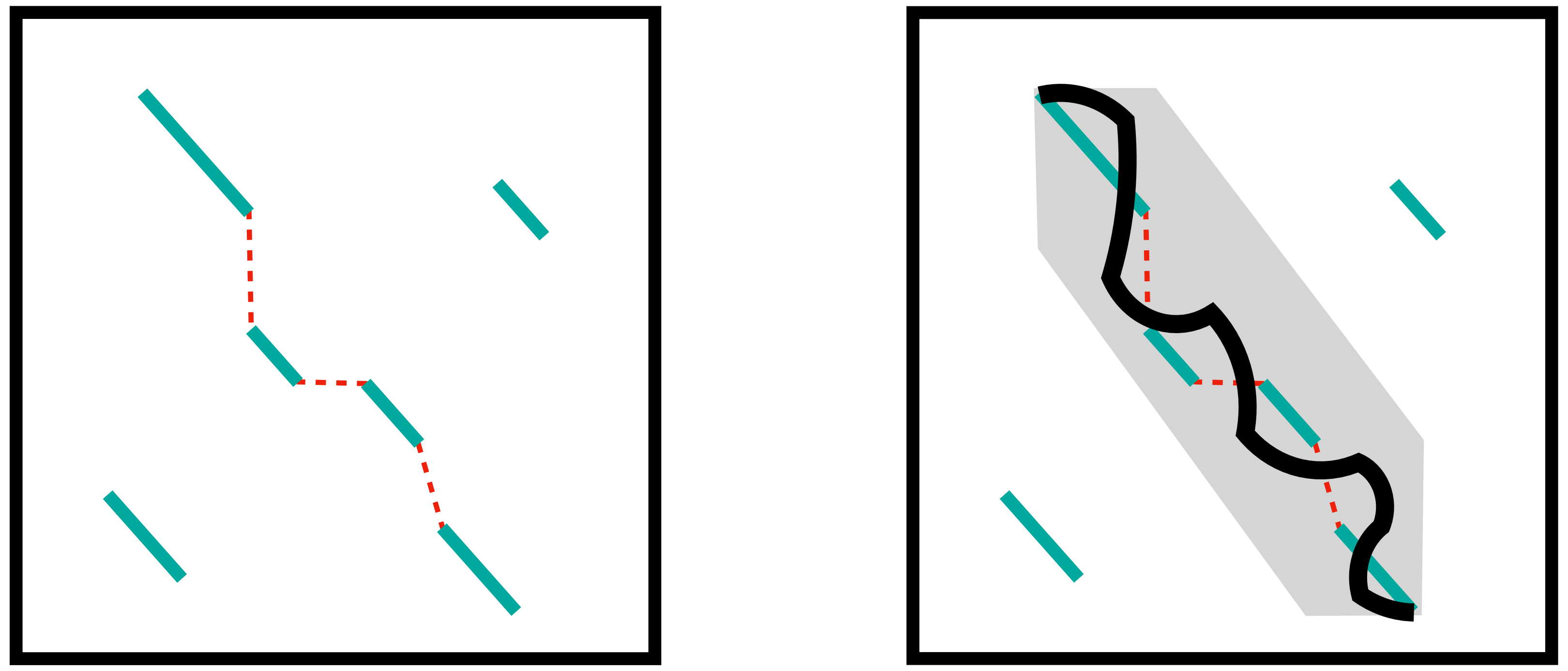
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

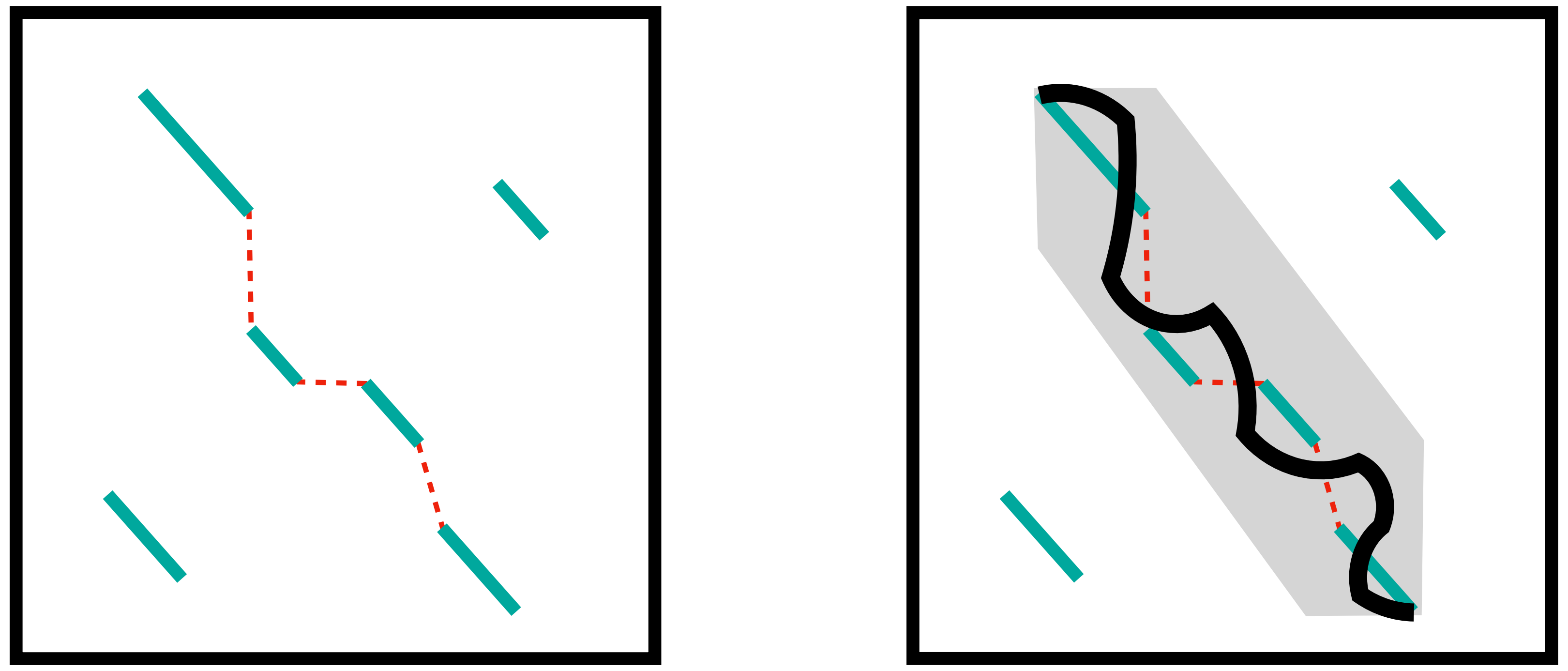
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

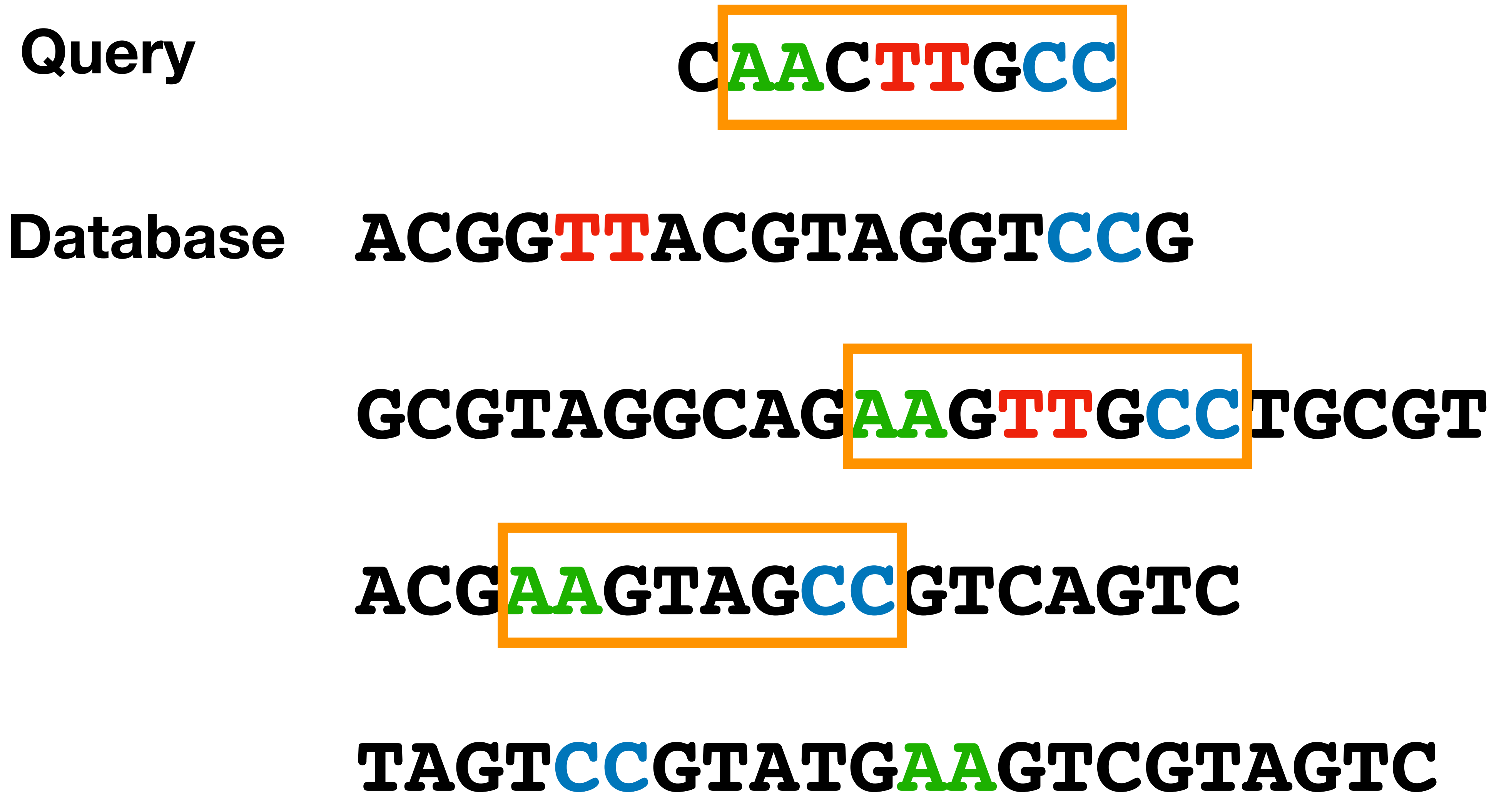
**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is *4*$^k$ for DNA and RNA, *20*$^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost



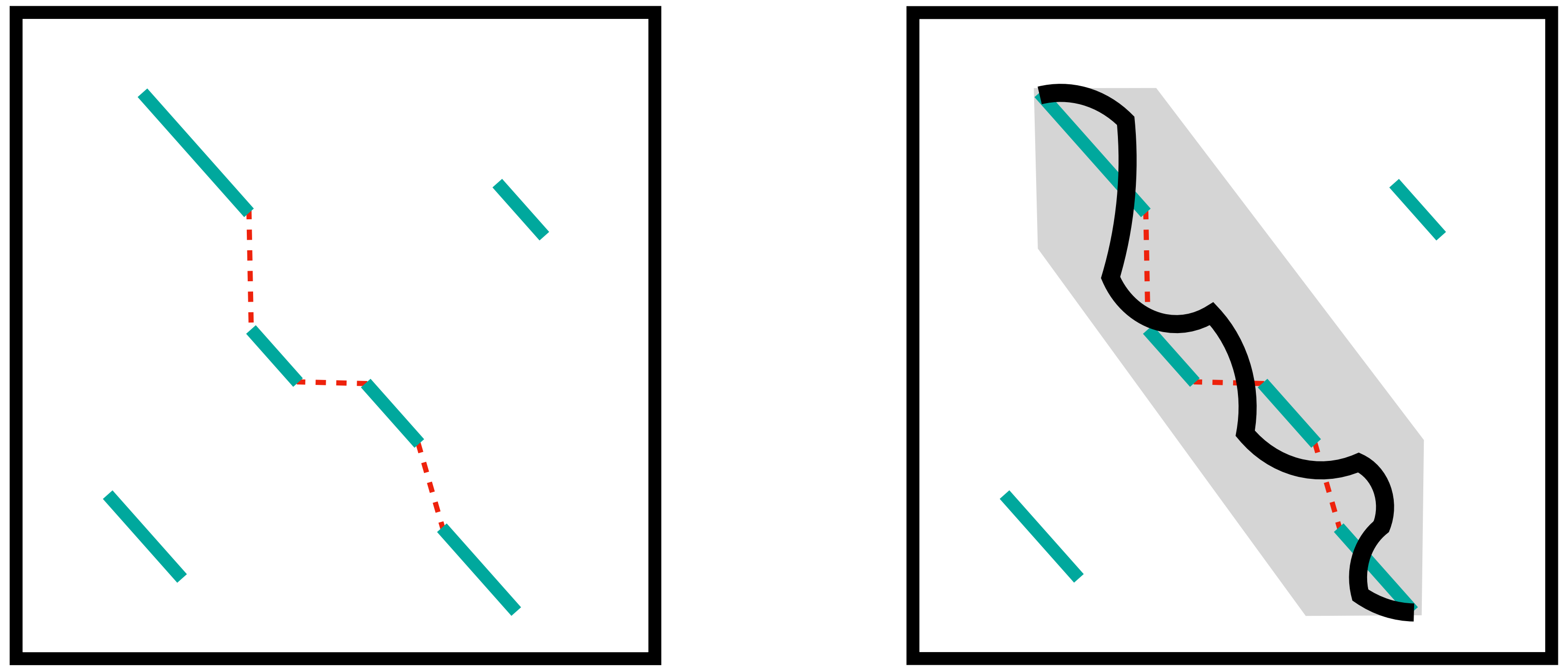
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

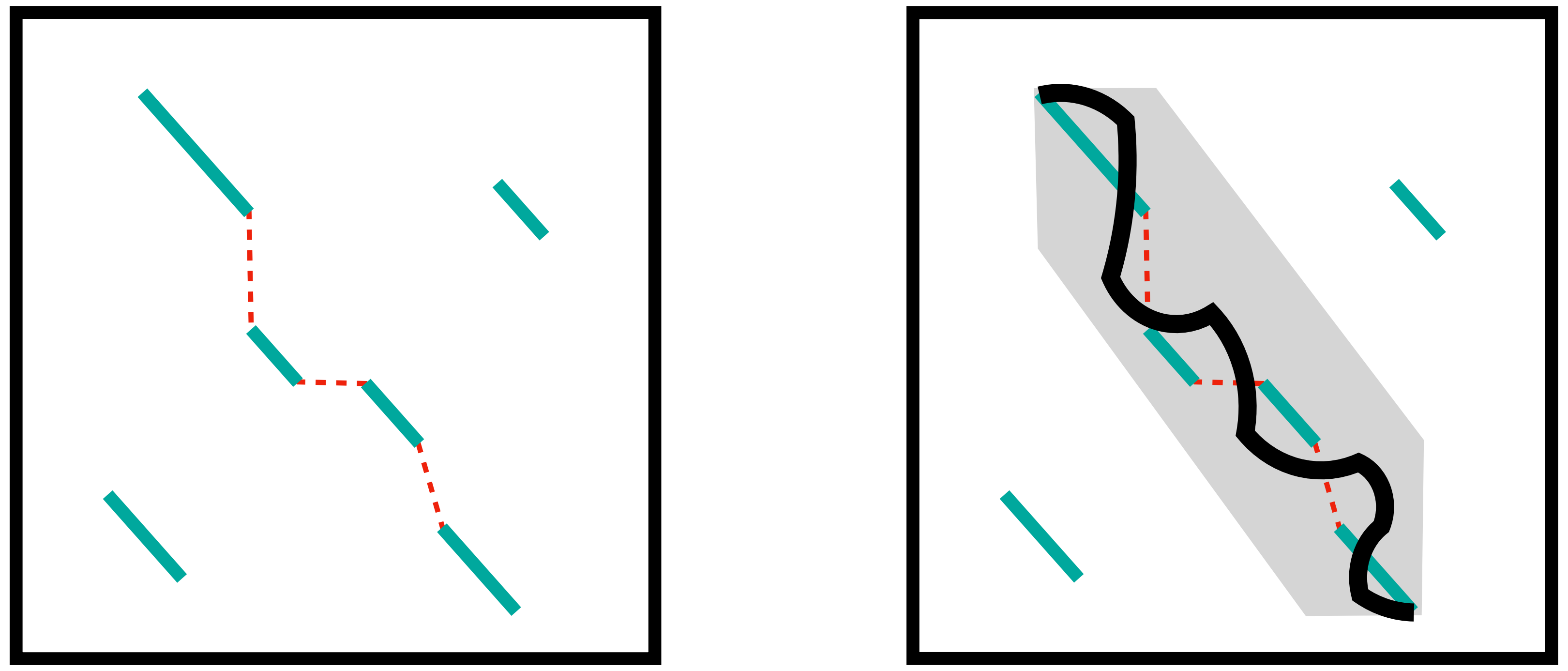
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

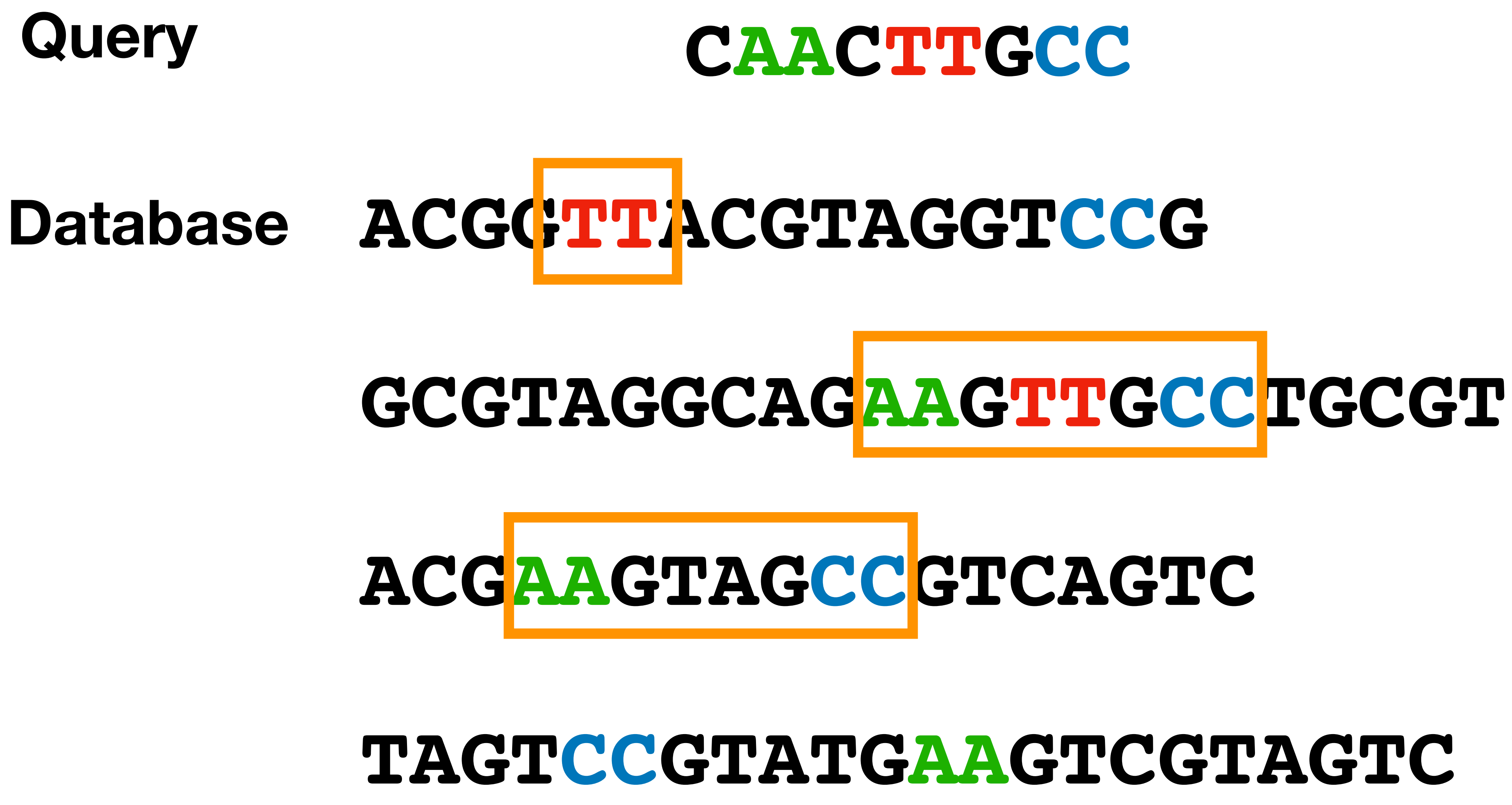
**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost



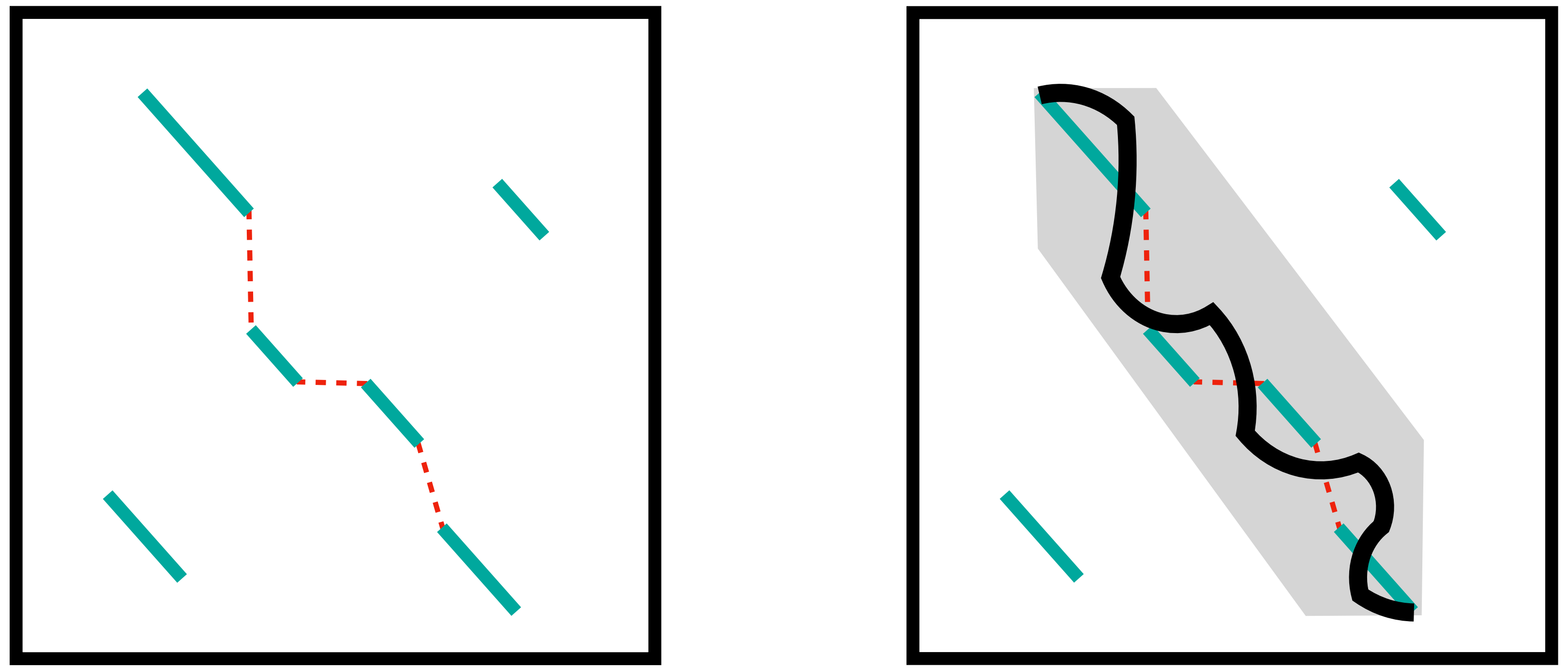
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

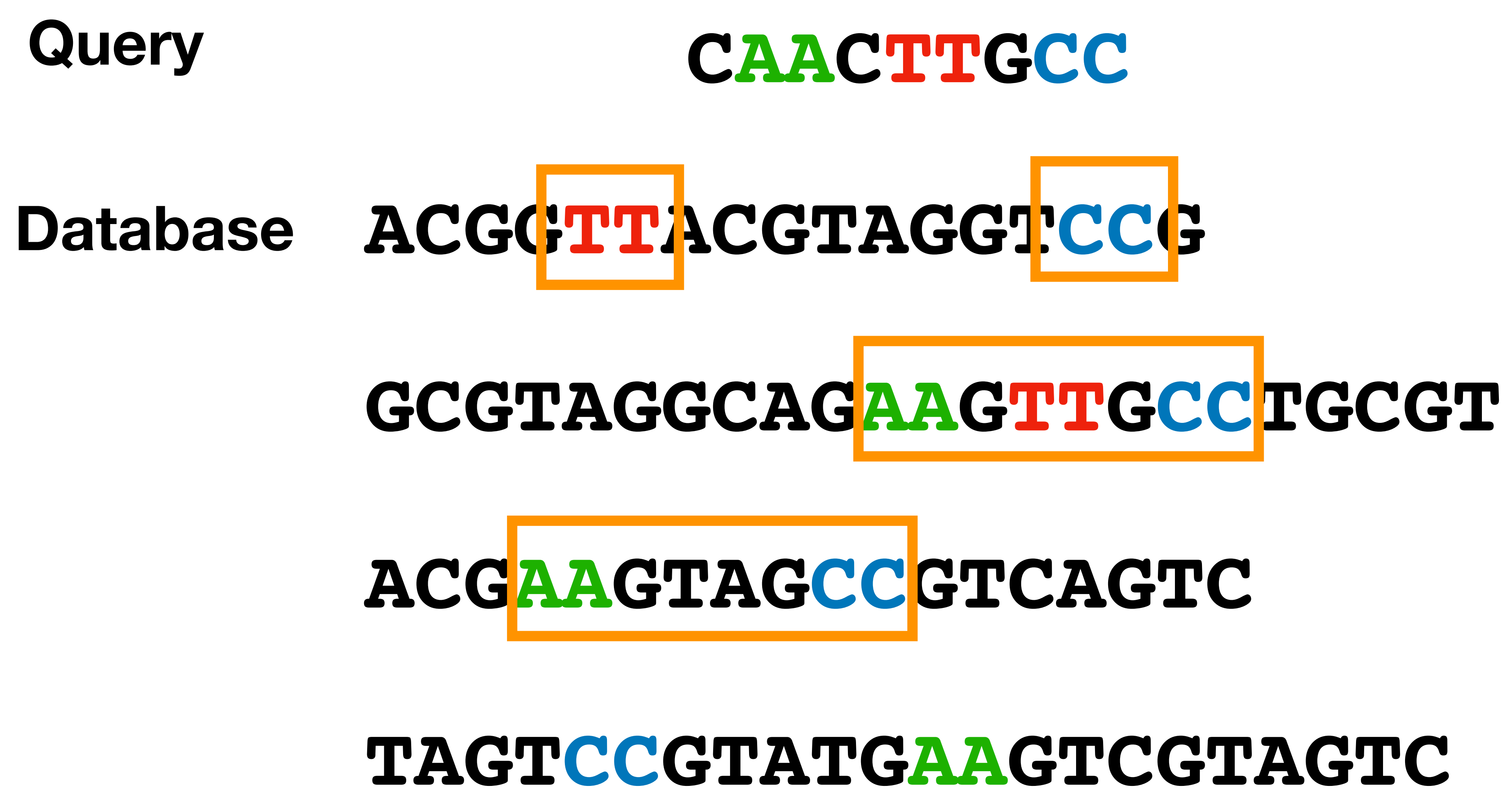
**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

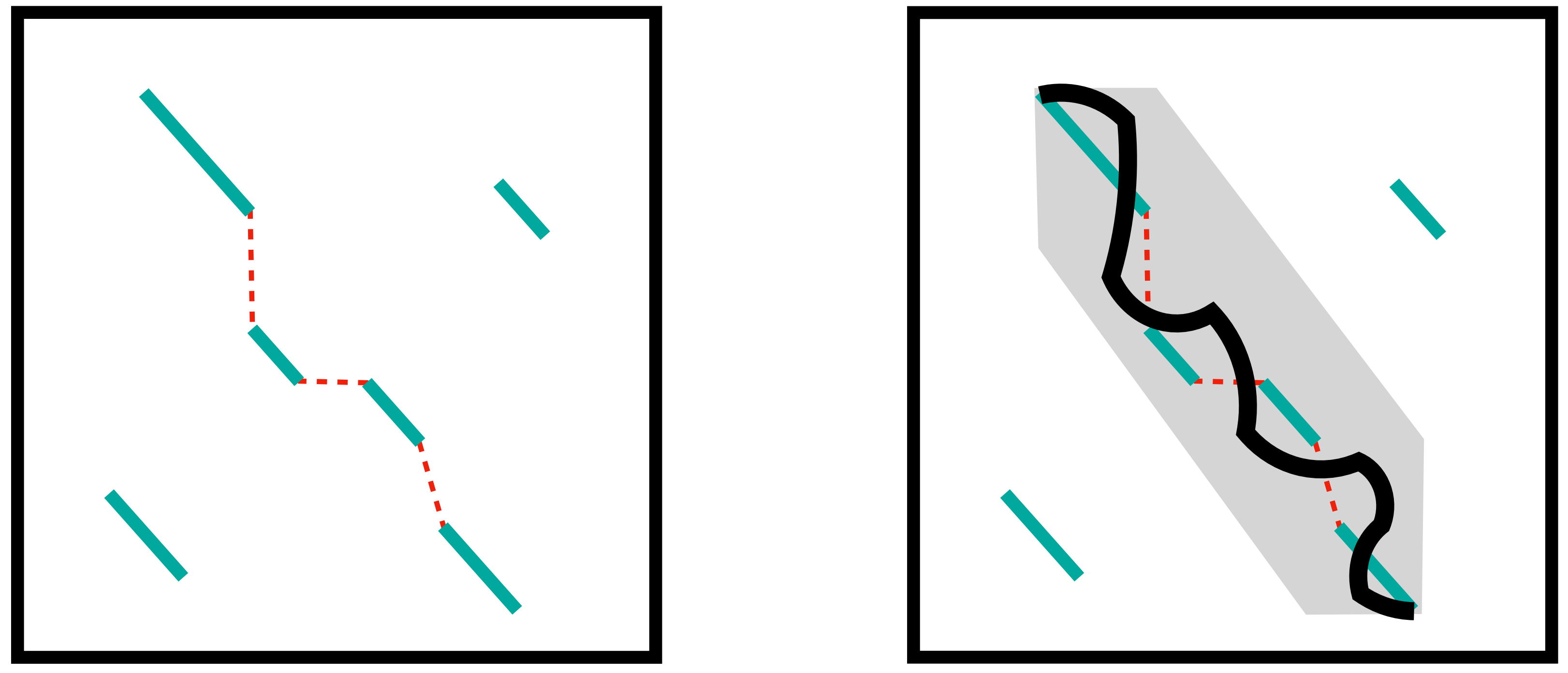
**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost





# FastA/FastP

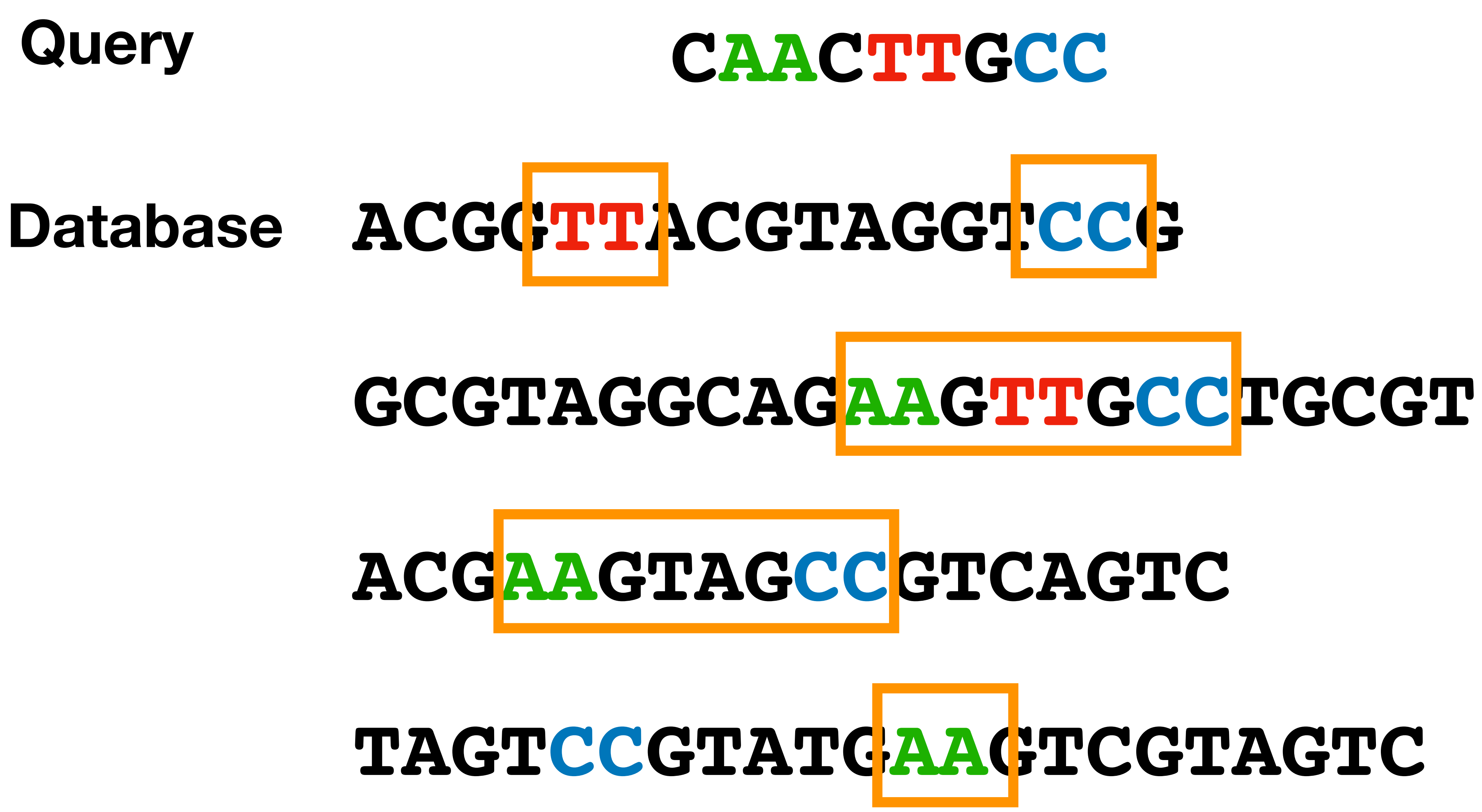
**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost



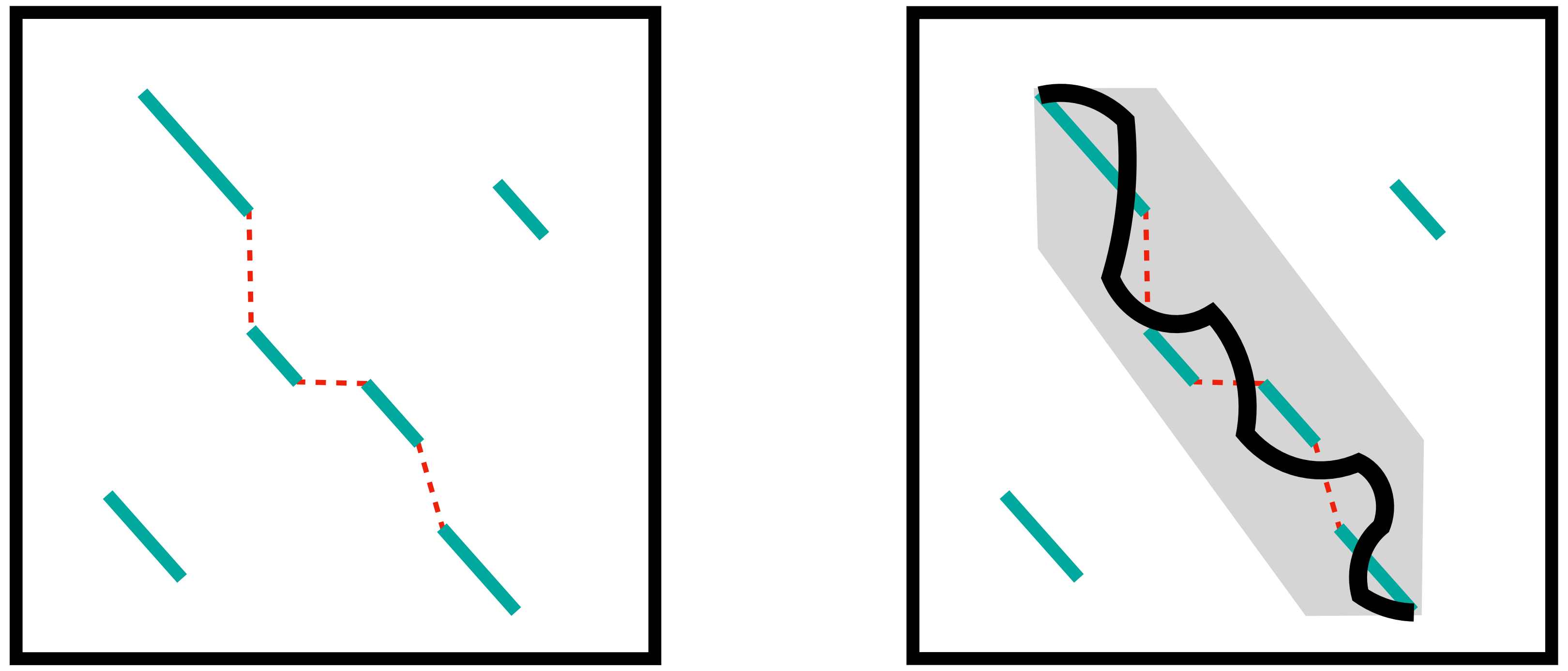
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# FastA/FastP

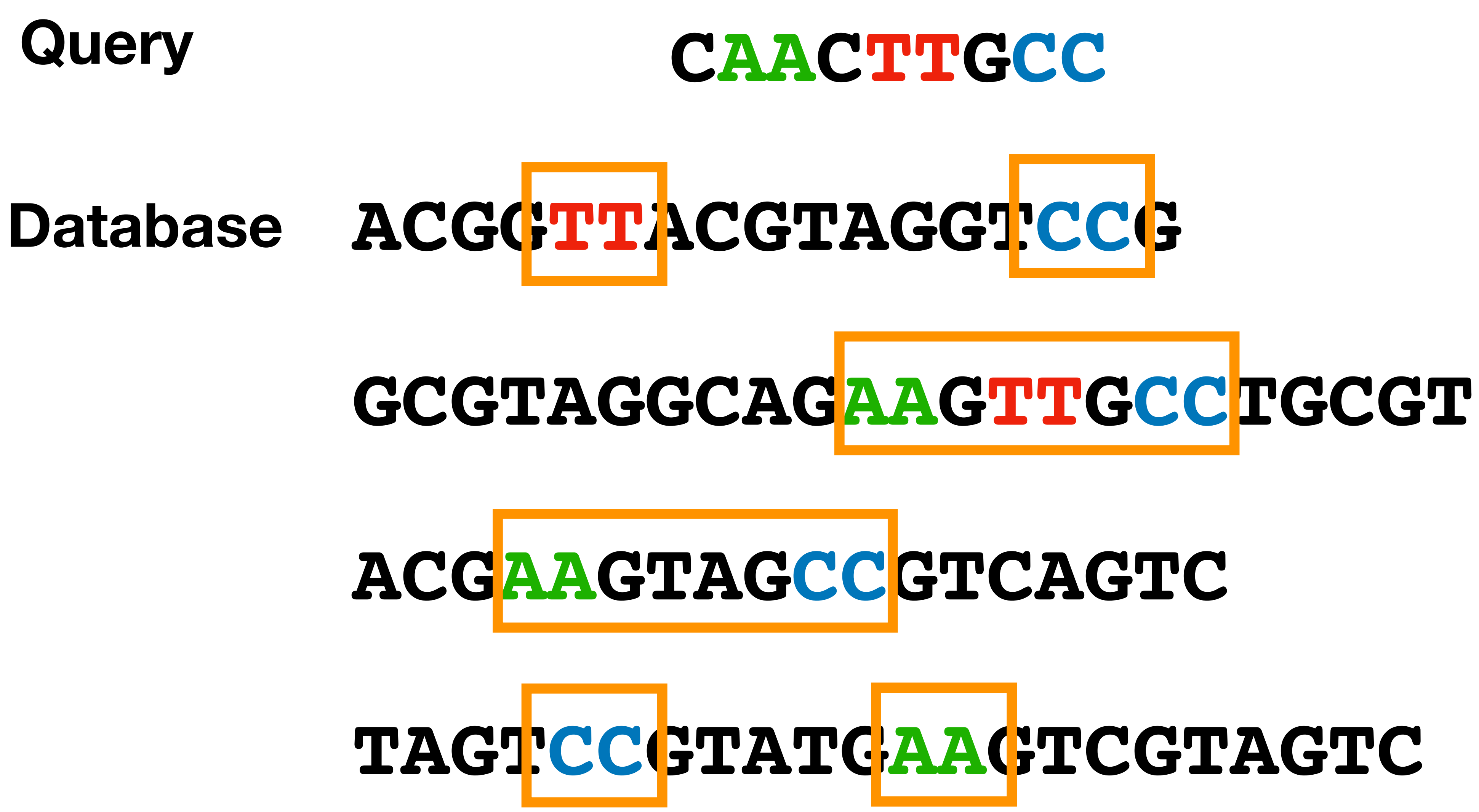
**Step 1**: Identify "hotspots" -- find *k*-mers that are shared between the query and the database using a lookup table (this table is $4^k$ for DNA and RNA, $20^k$ for Proteins)

**Step 2**: locating diagonal runs -- pairs (or larger groups) of hot spots such that the distance between the hot-spots is the same in both the query and the database sequence

**Step 3**: re-score the best diagonal runs -- rather than fixed inter-spot scores based on length, rescore the alignments using actual character matches

**Step 4 (FastA):** join diagonal runs -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost

**Step 5 (FastA):** (banded) Smith-Waterman -- using a fixed score based on the locations of the regions, join them with a fixed gap-style cost



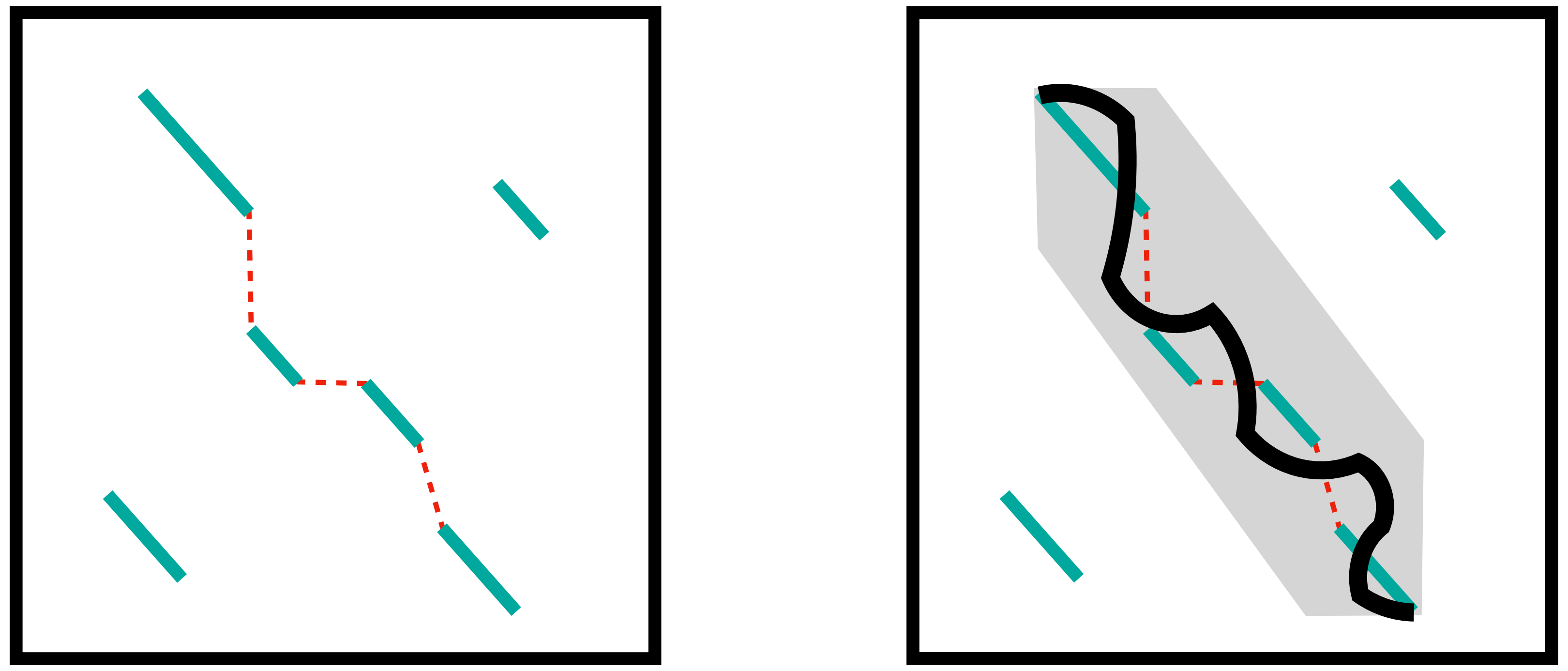
**Query** CAACTTGCC

**Database** ACGGTTACGTAGGTCCG

GCGTAGGCAGAAGTTGCCTGCGT

ACGAAGTAGCCGTCAGTC

TAGTCCGTATGAAGTCGTAGTC



# Basic Local Alignment Search Tool (BLAST)

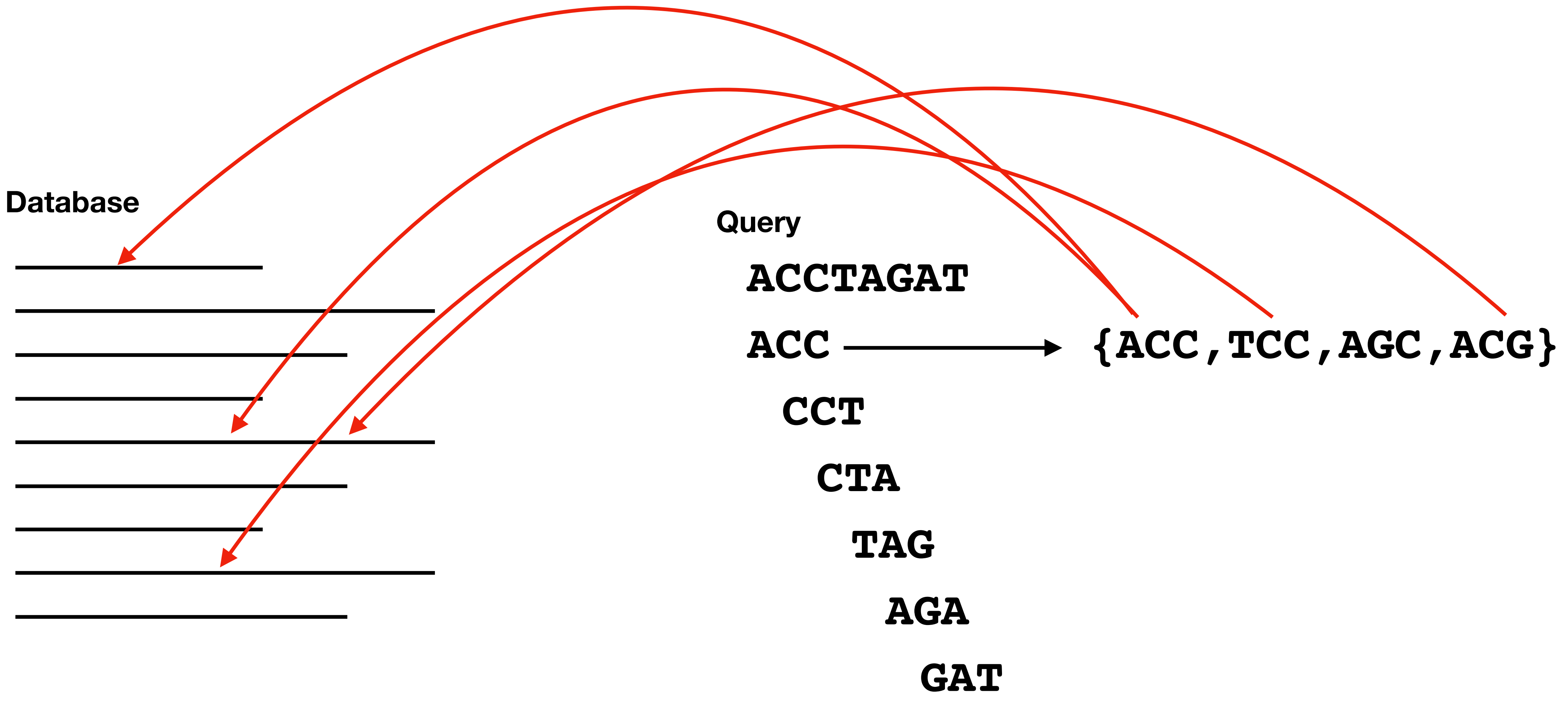
**Step 1:** Query-preprocessing:

1. split the query into *k*-mers
2. create a set of *neighbors* of each *k*-mer, other *k*-mers such that the replacement scores are not too high (this can be done with a $\Sigma^k$ lookup table)

**Step 2:** Database scanning -- label any instance of a neighbor of *Q* in any sequence *S* of *D* as a "hit", collect all of these hits

**Step 3:** Hit extension -- for any sequence *S* in *D*, with two hits (for protein, one for DNA) extend in either direction without gaps until the score drops too low

**Step 4:** Gapped extension -- run modified Smith-Waterman in each direction from the mid-point of the hits until the alignment score goes too low.



Database

Query

ACCTAGAT

ACC → {ACC,TCC,AGC,ACG}

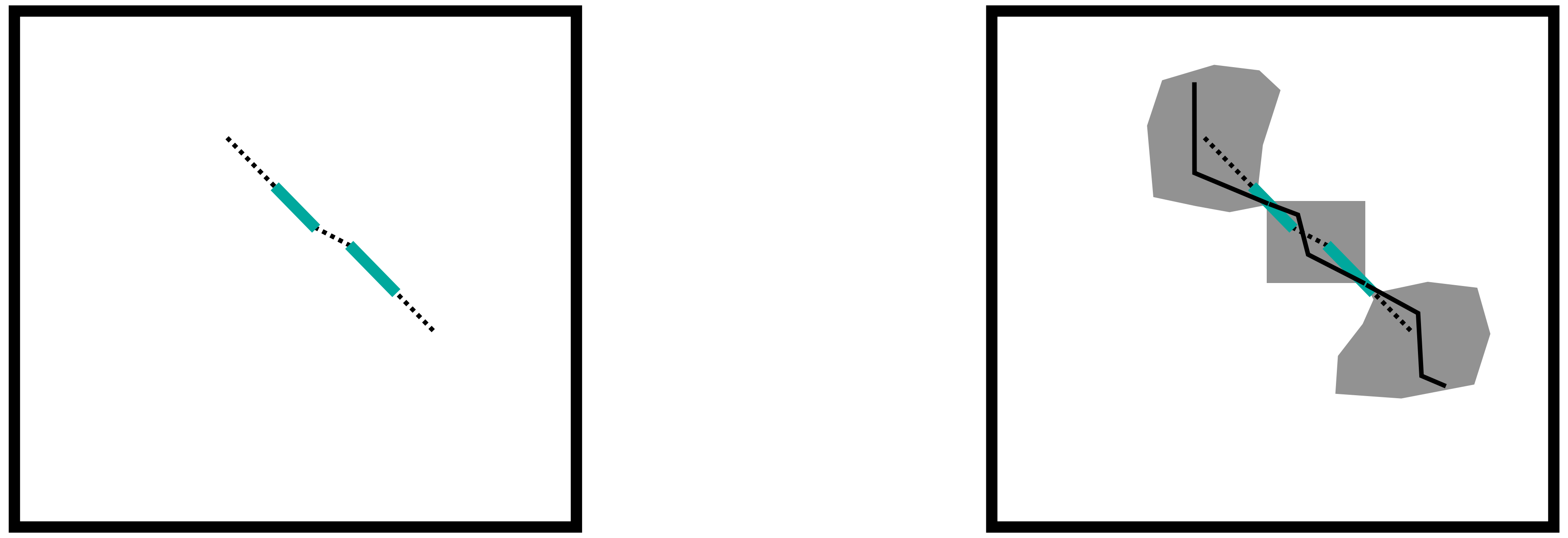
CCT

CTA

TAG

AGA

GAT



# Other Database Search Tools

MegaBLAST
- only for DNA but searches multiple sequences at once

BLAT (BLAST-Like Alignment Tool)
- only for DNA, indexes the database not the query

PatternHunter
- uses spaced-seeds rather than substings to search the database

PSI-BLAST (Position-Specific Iterated BLAST)
- updates the replacement matrix using an MSA until unchanged

QUASAR (Q-gram Alignment base on Suffix ARrays)
- uses the pigeon hole principle to find sequences in the database that are potential matches

LSH-ALL-PAIRS
- uses *k*-mer orderings to find probable matching sequences using a minimizer scheme

# Multiple Sequence Alignment Problem

Given

- A set of sequences $s_1,s_2,\ldots,s_k$ (of length $n$)
- An objective function

Find:

- an $\ell$ by $k$ matrix ($\ell \geq n$)
- where row $i$ contains the characters from sequence $s_i$ in order with inserted gap characters
- that is optimal under the objective function.

Input

AGTPNGNP
AGPGNP
AGTTPNGNP
CGTPNP
ACGTUNGNP

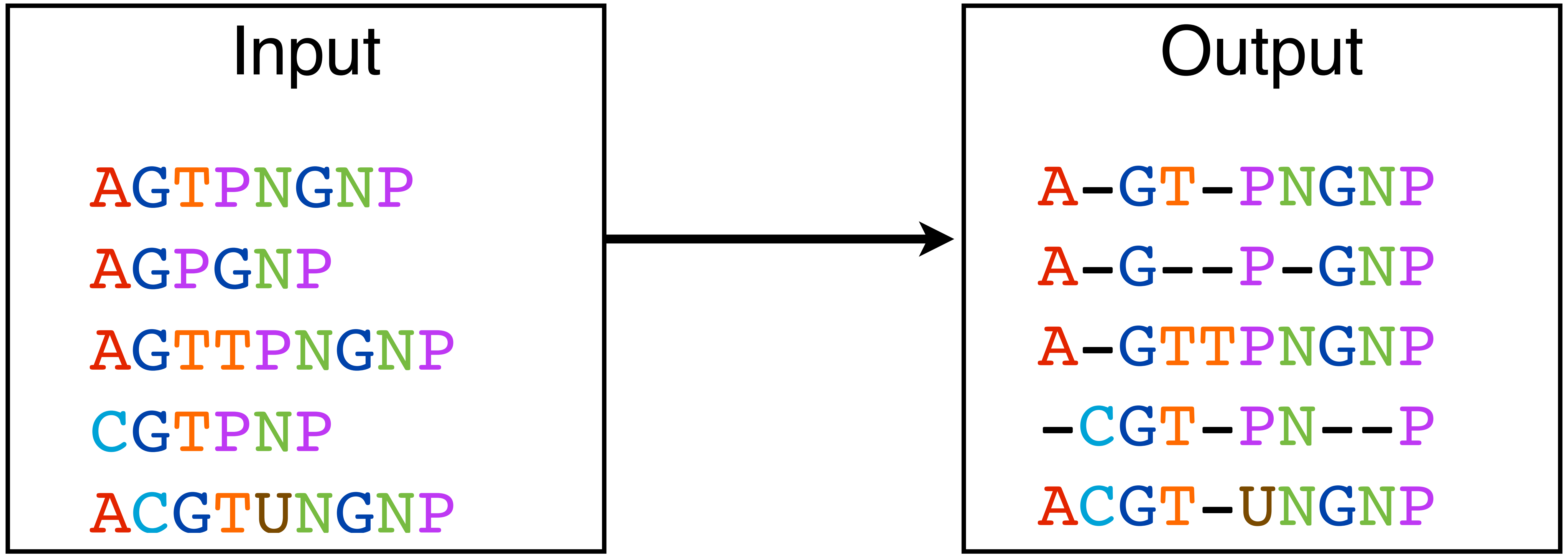
# Multiple Sequence Alignment Problem

Given

- A set of sequences $s_1, s_2, \ldots, s_k$ (of length $n$)
- An objective function

Find:

- an $\ell$ by $k$ matrix ($\ell \geq n$)
- where row $i$ contains the characters from sequence $s_i$ in order with inserted gap characters
- that is optimal under the objective function.



Input

```
AGTPNGNP
AGPGNP
AGTTPNGNP
CGTPNP
ACGTUNGNP
```

Output

```
A-GT-PNGNP
A-G--P-GNP
A-GTTPNGNP
-CGT-PN--P
ACGT-UNGNP
```

# Multiple Sequence Alignment

Whats the objective function:

- most popular -- **Sum-of-Pairs *Objective*:**
  - given some scoring function for a pairwise alignment *PairScore(*$s_1$*',*$s_2$*')* the score of the multiple alignment is:

$$SPScore(\{s'_1, s'_2, \ldots, s'_k\}) := \sum_{1 \leq i < j \leq k} PairScore(s'_i, s'_j)$$

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

- yes! we can use the same dynamic programming methods we had for pairwise alignment

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

- yes! we can use the same dynamic programming methods we had for pairwise alignment
- assume there are only 3 sequences, then the recursion is the following:

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

- yes! we can use the same dynamic programming methods we had for pairwise alignment
- assume there are only 3 sequences, then the recursion is the following:

$$V[i,j,k] = \max \begin{cases} V[i-1,j-1,k-1] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],s_3[k]) + \delta(s_1[i],s_3[k]) \\ V[i-1,j-1,k] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],'-') + \delta(s_1[i],'-') \\ V[i-1,j,k-1] & +\delta(s_1[i],'-') + \delta(s_2[j],s_3[k]) + \delta(s_1[i],s_3[k]) \\ V[i,j-1,k-1] & +\delta('-',s_2[j]) + \delta(s_2[j],s_3[k]) + \delta('-',s_3[k]) \\ V[i-1,j,k] & +2\delta(s_1[i],'-') \\ V[i,j-1,k] & +2\delta(s_2[j],'-') \\ V[i,j,k-1] & +2\delta(s_3[k],'-') \end{cases}$$

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

- yes! we can use the same dynamic programming methods we had for pairwise alignment
- assume there are only 3 sequences, then the recursion is the following:

$$V[i,j,k] = \max \begin{cases} V[i-1,j-1,k-1] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],s_3[k]) + \delta(s_1[i],s_3[k]) \\ V[i-1,j-1,k] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],'-') + \delta(s_1[i],'-') \\ V[i-1,j,k-1] & +\delta(s_1[i],'-') + \delta(s_2[j],s_3[k]) + \delta(s_1[i],s_3[k]) \\ V[i,j-1,k-1] & +\delta('-',s_2[j]) + \delta(s_2[j],s_3[k]) + \delta('-',s_3[k]) \\ V[i-1,j,k] & +2\delta(s_1[i],'-') \\ V[i,j-1,k] & +2\delta(s_2[j],'-') \\ V[i,j,k-1] & +2\delta(s_3[k],'-') \end{cases}$$

What happens with 4 sequences? How many clauses are in the max? How big is *V*?

# Finding an optimal MSA

Can we find an optimal multiple sequence alignment?

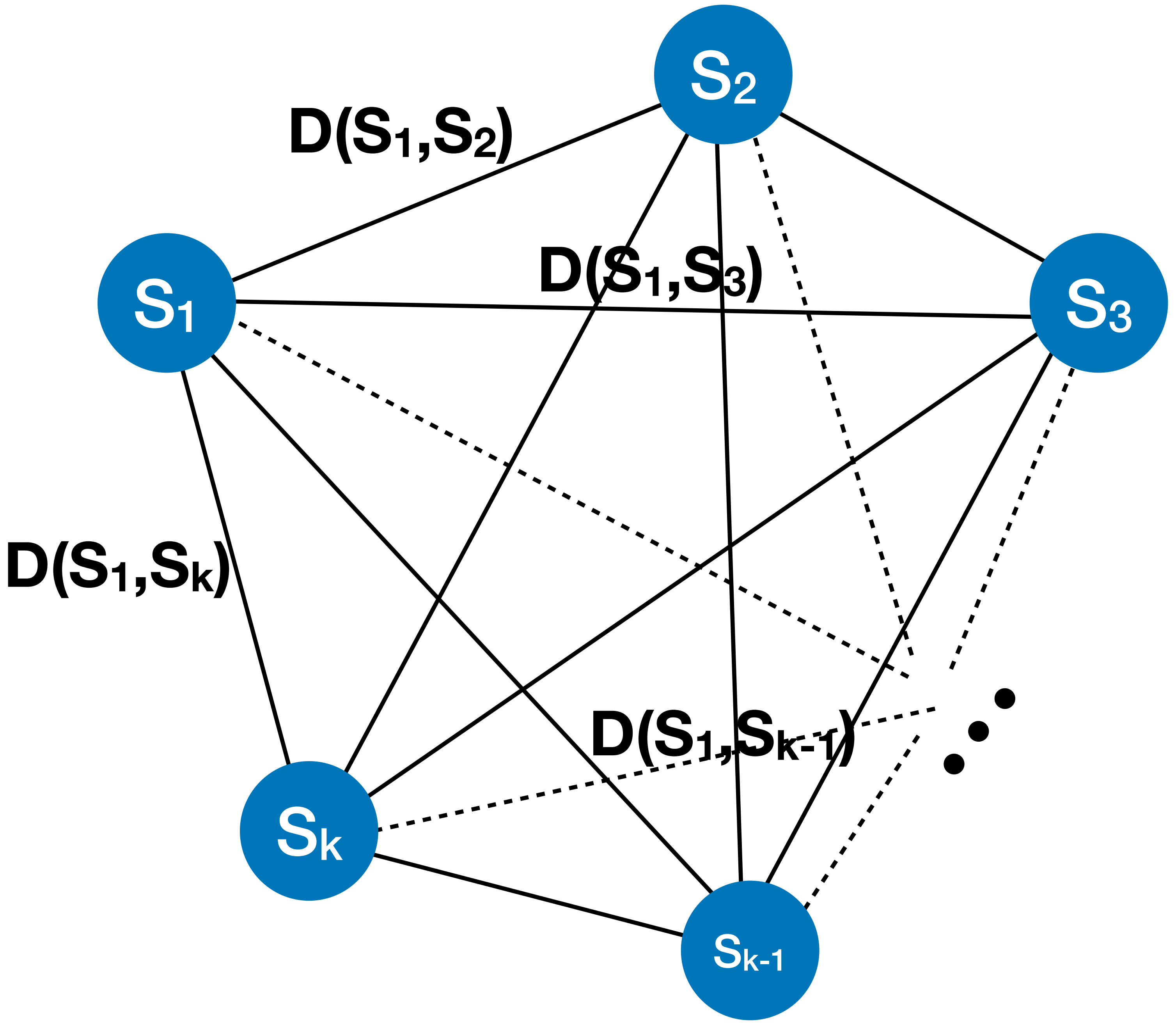
- yes! we can use the same dynamic programming methods we had for pairwise alignment
- assume there are only 3 sequences, then the recursion is the following:

$$V[i,j,k] = \max \begin{cases} V[i-1,j-1,k-1] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],s_3[k]) + \delta(s_1[i],s_3[k]) \\ V[i-1,j-1,k] & +\delta(s_1[i],s_2[j]) + \delta(s_2[j],'-') + \delta(s_1[i],'-') \\ V[i-1,j,k-1] & +\delta(s_1[i],'-') + \delta(s_2[j],s_3[k]) + \\ V[i,j-1,k-1] & +\delta('-',s_2[j]) + \delta(s_2[j],s_3[k]) + \\ V[i-1,j,k] & +2\delta(s_1[i],'-') \\ V[i,j-1,k] & +2\delta(s_2[j],'-') \\ V[i,j,k-1] & +2\delta(s_3[k],'-') \end{cases}$$

*$O(k^2 2^k n^k)$-time!!*

What happens with 4 sequences? How many clauses are in the max? How big is V

# The Center Star Method



$$S_c = \textbf{arg}\min_{1\le i\le k}\left\{\sum_{1\le j\le k} D(S_i, S_j)\right\}$$

The final step is to build an alignment so that all of the alignments between $S_c$ and $S_i$ are satisfied.

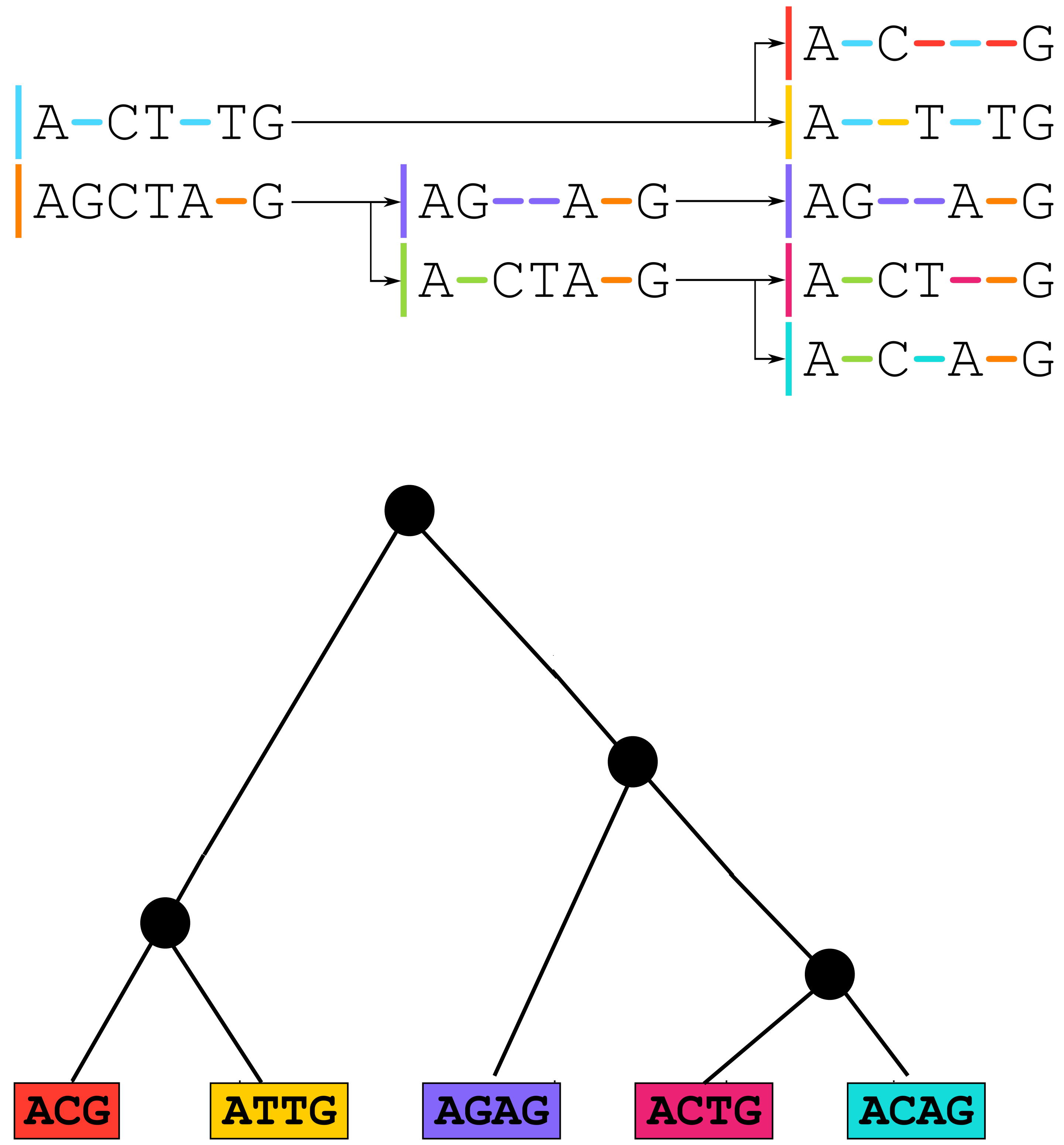
# Progressive Alignment

Similar to center star in that we use pairwise alignments to help build multiple alignments.

Introduced by Feng and Doolittle in 1987.

Basic idea:

- compute pairwise alignment scores for each pair of sequences
- generate a **guide tree** which ensures similar sequences are near to each other
- align sequences (or groups) one-by-one from the leaves of the tree



[from Balcan, *et al.* 2019, arXiv:1908.02894]

# Progressive Alignment

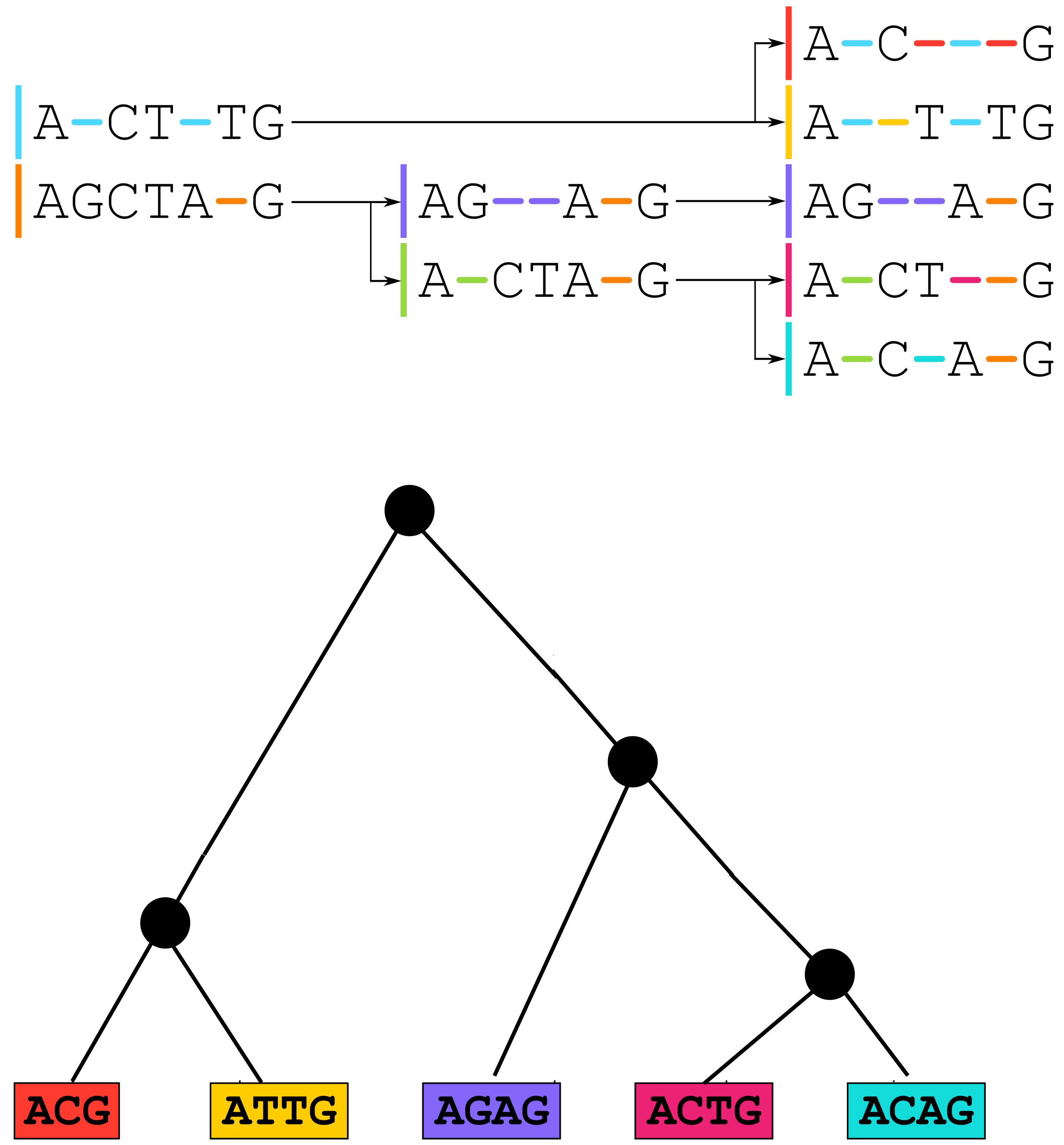
Similar to center star in that we use pairwise alignments to help build multiple alignments.

Introduced by Feng and Doolittle in 1987.

Basic idea:

- compute pairwise alignment scores for each pair of sequences
- generate a **guide tree** which ensures similar sequences are near to each other
- align sequences (or groups) one-by-one from the leaves of the tree

"Progressive alignment from consensus sequences"



[from Balcan, *et al.* 2019, arXiv:1908.02894]

# ClustalW

Algorithm

# ClustalW

Algorithm

- Calculate the $\binom{n}{2}$ pairwise alignments.

# ClustalW

Algorithm

- Calculate the $\binom{n}{2}$ pairwise alignments.
- Compute the pairwise distance between sequences as $1 - \frac{x}{y}$ where *x* is the number of gap characters, and *y* is the number of matches.

# ClustalW

Algorithm

- Calculate the $\binom{n}{2}$ pairwise alignments.
- Compute the pairwise distance between sequences as $1 - \frac{x}{y}$ where $x$ is the number of gap characters, and $y$ is the number of matches.
- Use the neighbor-joining method to create the guide tree (we will talk about the details of this later).

# ClustalW

Algorithm

- Calculate the $\binom{n}{2}$ pairwise alignments.
- Compute the pairwise distance between sequences as $1 - \frac{x}{y}$ where *x* is the number of gap characters, and *y* is the number of matches.
- Use the neighbor-joining method to create the guide tree (we will talk about the details of this later).
- From the leaves compute the alignment at each internal node

# ClustalW

Algorithm

- Calculate the $\binom{n}{2}$ pairwise alignments.
- Compute the pairwise distance between sequences as $1 - \frac{x}{y}$ where *x* is the number of gap characters, and *y* is the number of matches.
- Use the neighbor-joining method to create the guide tree (we will talk about the details of this later).
- From the leaves compute the alignment at each internal node
  - each alignment will be between either: (i) two sequences, (ii) two partial alignment, or (iii) a sequence and a partial alignment.

# MUSCLE
## (MUltiple Sequence Comparison by Log-Expectation)

Algorithm:

# MUSCLE
# (MUltiple Sequence Comparison by Log-Expectation)

Algorithm:

1. **draft progressive alignment** -- similar to ClustalW but with
    - LE score for aligning profiles,
    - a more efficient tree building algorithm, and
    - a more efficient pairwise comparison (using *k*-mer counting).

# MUSCLE
# (<u>MU</u>ltiple <u>S</u>equence <u>C</u>omparison by <u>L</u>og-<u>E</u>xpectation)

Algorithm:

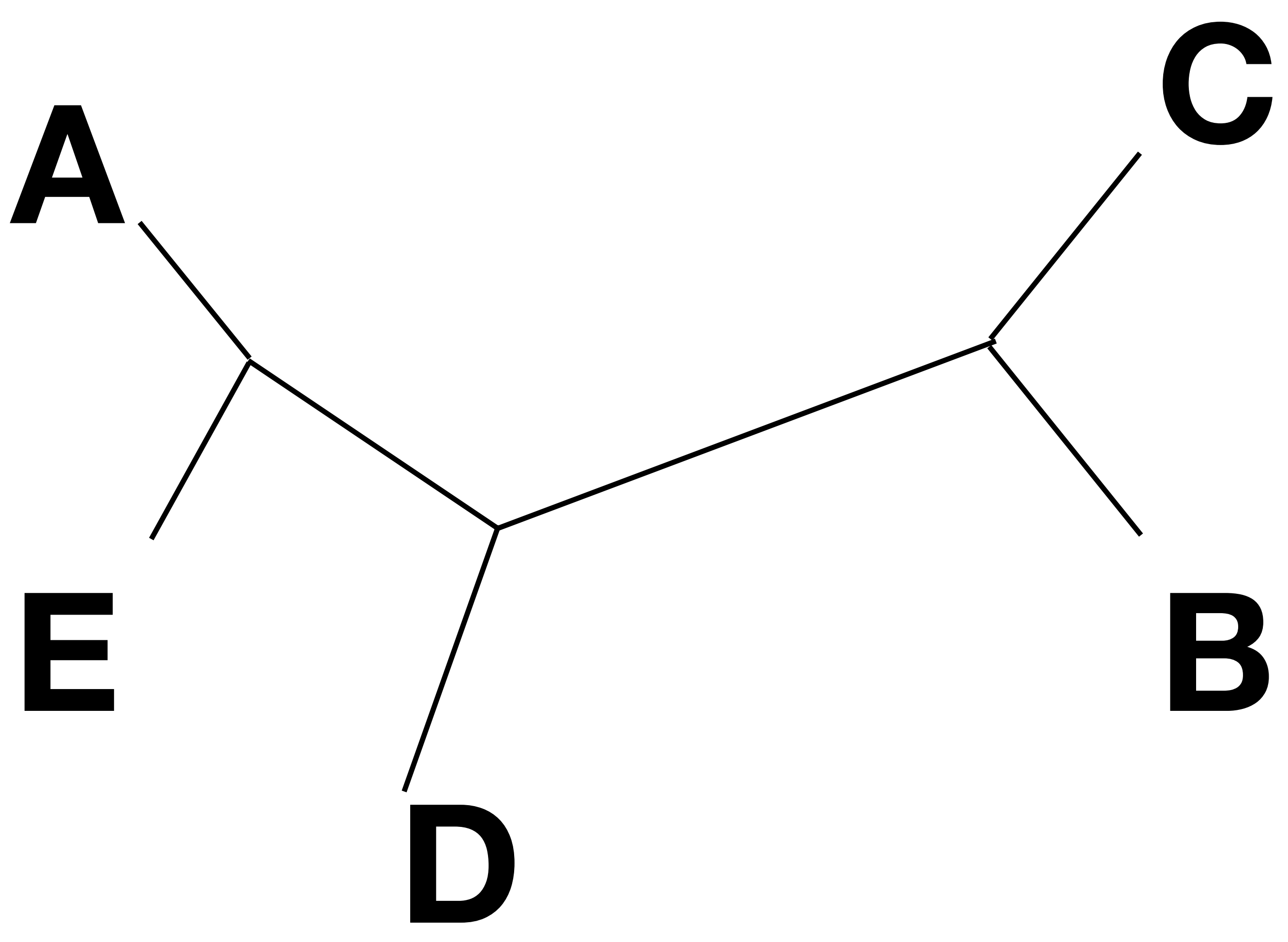
1. **draft progressive alignment** -- similar to ClustalW but with
   - LE score for aligning profiles,
   - a more efficient tree building algorithm, and
   - a more efficient pairwise comparison (using *k*-mer counting).
2. **improved progressive alignment** -- using the alignment from (1)
   - redefine the pairwise distances using the Kimura distance $-\ln\left(1 - D - \frac{D^2}{5}\right)$
   - *D* is the fraction of matches.
   - re-align.

# MUSCLE
## (MUltiple Sequence Comparison by Log-Expectation)

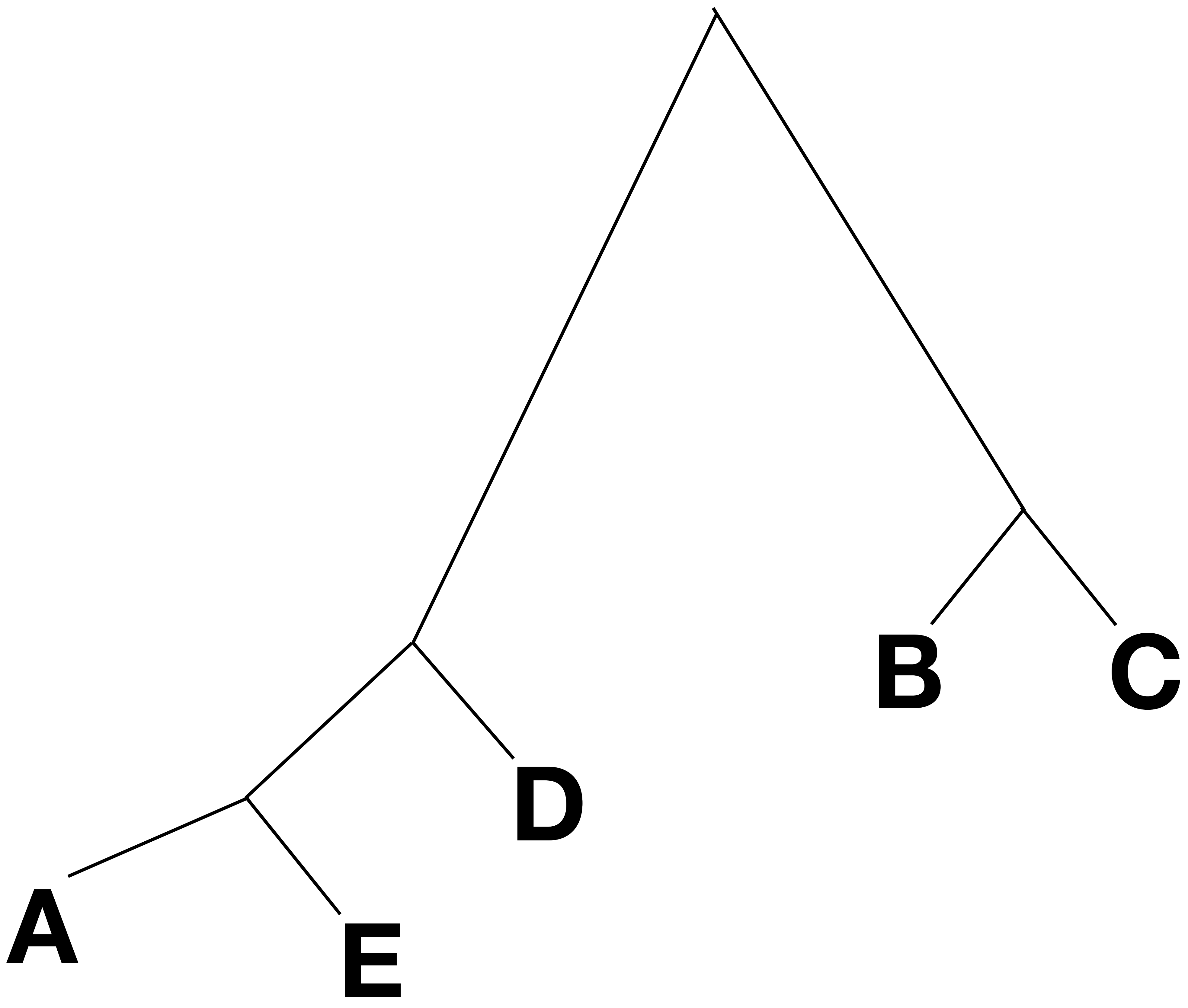
Algorithm:

1. **draft progressive alignment** -- similar to ClustalW but with
   - LE score for aligning profiles,
   - a more efficient tree building algorithm, and
   - a more efficient pairwise comparison (using *k*-mer counting).
2. **improved progressive alignment** -- using the alignment from (1)
   - redefine the pairwise distances using the Kimura distance $-\ln\left(1 - D - \frac{D^2}{5}\right)$
   - *D* is the fraction of matches.
   - re-align.
3. **refinement** -- deleting an edge in the guide tree creates two sub-groups of sequences with induced sub-alignments.
   - Extract those two sub-alignments and realign them.
   - Only keep the new alignment if the *SP* score is increased.
   - Stop when SP has not improved: in a predefined number of iterations or when all edges are visited.

# Some terminology



Unrooted



Rooted

# Tree Building *Algorithms*

Two major classes:

- **Distance-based methods**
  - for each pair of items, get some evolutionary distance (edit distance, melting temp for DNA hybridization, strength of antibody cross reactions)
  - find a tree that "agrees" with the distances either ultametric or additive
  - most cases in real life don't match this so you have to find a good approx.
- **Maximum-Parsimony methods**
  - character-based data only (not necessarily DNA/RNA/Protein data)
  - infer sequences at the internal nodes and maximize parsimony (minimize the mutations) along branches

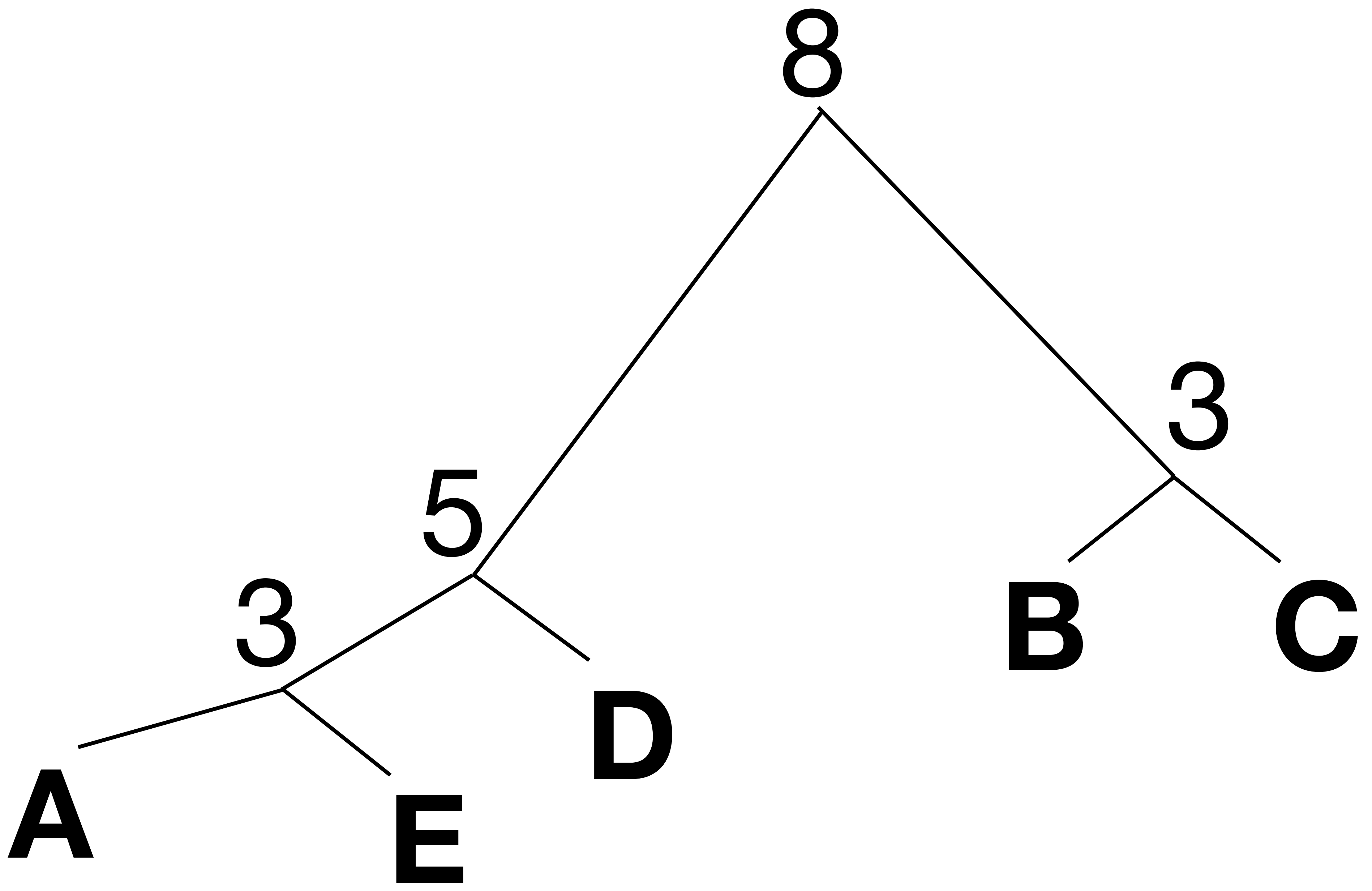
# Ultrametric Trees

Let *D* be a symmetric *nxn* matrix of real numbers. An *ultrametric* tree for *D* is a rooted tree *T* such that:

- *T* contains *n* leaves labeled by a unique row of *D*.
- Each internal node of *T* is leveled by one **entry** from *D* and has at least 2 children.
- Along any path from the root to a leaf, the numbers labeling the internal nodes are **strictly decreasing**.
- For any two leaves *i,j* of *T*, *D(i,j)* is the leavel of the least common ancestor of *i* and *j* in *T.*

Therefore, *T* (if it exists) is a compact representation of *D*

|   | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| **A** | 0 | 8 | 8 | 5 | 3 |
| **B** | 8 | 0 | 3 | 8 | 8 |
| **C** | 8 | 3 | 0 | 8 | 8 |
| **D** | 5 | 8 | 8 | 0 | 5 |
| **E** | 3 | 8 | 8 | 5 | 0 |



# Additive-distance trees

Ultrametric is the "holy grail", but when its not able to be obtained, we can use a less stringent model.

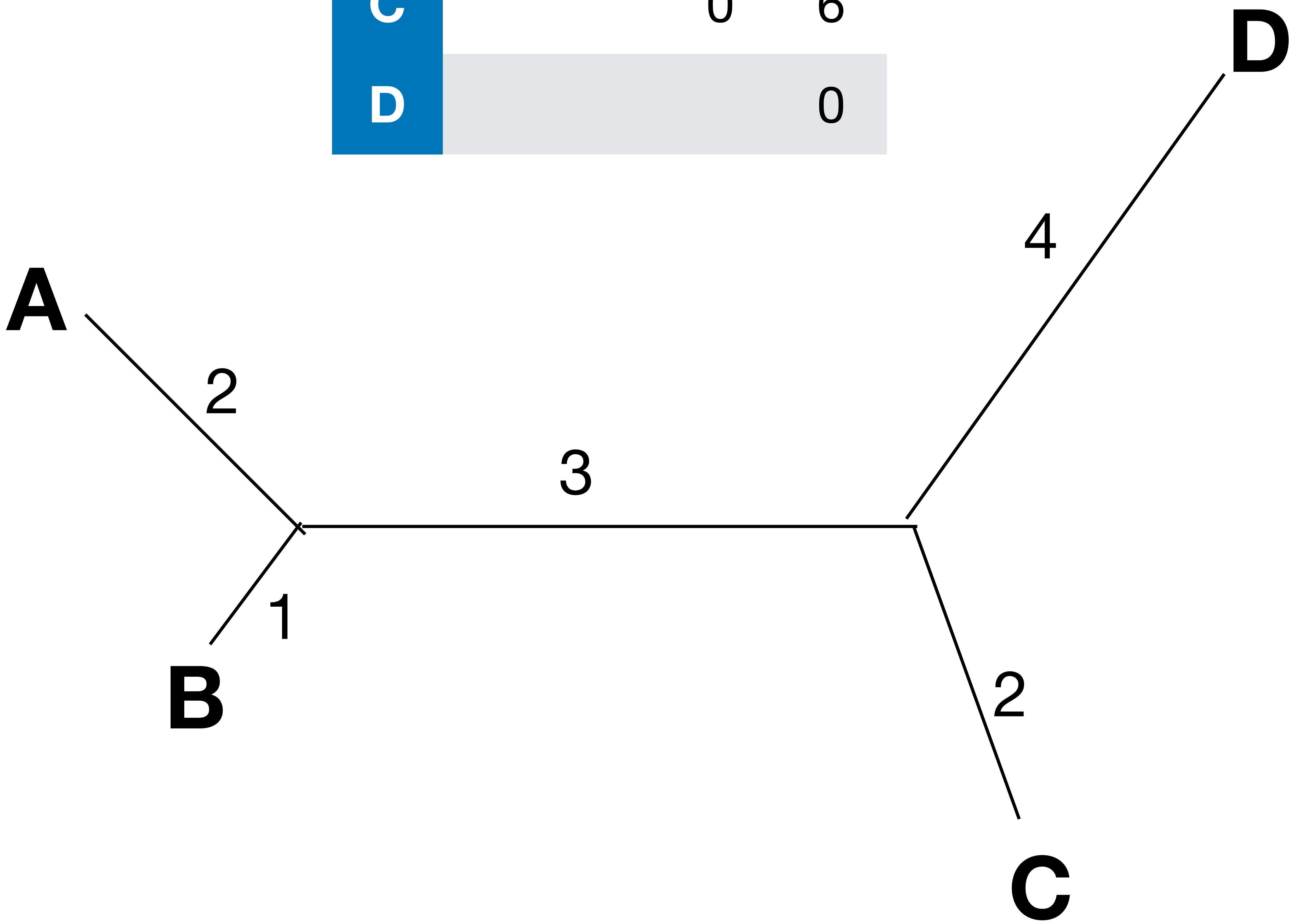
**Definition**

- Let *D* be a symmetric *n* by *n* matrix where the numbers on the diagonal are all 0, and the off-diagonal numbers are all strictly positive.
- Let *T* be an edge-weighted tree with at least *n* nodes, where *n* distinct nodes are labeled with rows of *D.*
- Tree *T* is called an *additive tree* if for every pair of *labeled* nodes (*i, j*), the path from node *i* to node *j* has total weight (or distance) exactly *D(i,j)*.

**Problem**

- Given a matrix *D* with 0s on the diagonals, and positive numbers in all other locations, find the additive tree *T* or determine that one does not exist.

| | A | B | C | D |
|---|---|---|---|---|
| **A** | 0 | 3 | 7 | 9 |
| **B** | | 0 | 6 | 8 |
| **C** | | | 0 | 6 |
| **D** | | | | 0 |



# Parsimony

Parsimony's main principle: "if there exists more than one possible answer to the question, the simpler answer is more likely to be correct" (when you hear hooves think horses not zebra).

# Parsimony

Parsimony's main principle: "if there exists more than one possible answer to the question, the simpler answer is more likely to be correct" (when you hear hooves think horses not zebra).

In sequence evolution each character in a sequence will be modified at most one time (sometimes called the *infinite sites* model).

# Parsimony

Parsimony's main principle: "if there exists more than one possible answer to the question, the simpler answer is more likely to be correct" (when you hear hooves think horses not zebra).

In sequence evolution each character in a sequence will be modified at most one time (sometimes called the *infinite sites* model).

Therefore, we can change the sequence data into a binary labeling

- 0 if the character is unchanged in this sequence
- 1 if it has already been modified

# Parsimony

Parsimony's main principle: "if there exists more than one possible answer to the question, the simpler answer is more likely to be correct" (when you hear hooves think horses not zebra).

In sequence evolution each character in a sequence will be modified at most one time (sometimes called the *infinite sites* model).

Therefore, we can change the sequence data into a binary labeling

- 0 if the character is unchanged in this sequence
- 1 if it has already been modified

**Definition** Given an $n$ by $m$ binary character matrix $M$, a *phylogenetic tree* for $M$ is a rooted tree $T$ with exactly $n$ leaves that obeys the following:

- each of the $n$ objects labels exactly 1 leaf of $T$
- each of the $m$ characters labels exactly 1 edge of $T$
- for any object $p$, the characters that label the edges along the unique path from the root to the leaf specify all of the characters of $p$ whose state is *1.*

# Maximum Parsimony

The **Maximum Parsimony Problem** (sometimes called the Large Parsimony Problem) is stated as follows:

- Given a matrix *M* for a set *S* of *n* taxa
- find the tree *T* wihch is leaf labeled by *S* and minimizes the edges that are labeled by character position changes.

This problem is *NP-Hard*

Branch and Bound

- start with a 3-leaf tree, add each leaf at each edge by breaking it and adding a new internal node
- computation tree grows exponentially

2-approximation

- find the minimum spanning tree in the leaf graph, convert into a phylogeny by adding edges
- $O(n^2m)$ time

# Neighbor Joining

**Algorithm** Given a distance matrix *M* with rows labeled *(1,2,3....n)*

- let *Z = {{1},{2},{3},..,{n}}* (* the set of initial clusters *)
- for all *{i},{j}* ∈ *Z* set $D(\{i\},\{j\})=M_{i,j}$
- while *|Z|>1*
  - define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
  - $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
  - form *C* by creating a new cluster root and connecting it to the two cluster roots with edge weights $\frac{1}{2}\left(D(A,B)+(u_A-u_B)\right)$ and $\frac{1}{2}\left(D(A,B)+(u_B-u_A)\right)$ respectively.
  - Z = Z ∪ {C} - {A,B}
  - define D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )

# Neighbor Joining

**Algorithm** Given a distance matrix $M$ with rows labeled $(1,2,3....n)$

```
O(n) • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
     • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
     • while $|Z|>1$
         • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
         • $(A,B) = \mathit{arg} \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
         • form $C$ by creating a new cluster root and connecting it to the two cluster roots
           with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
         • $Z = Z \cup \{C\} - \{A,B\}$
         • define $D(F,C) = D(C,F) = 1/2\ ( D(A,F) + D(B,F) - D(A,B) )$
```

# Neighbor Joining

**Algorithm** Given a distance matrix *M* with rows labeled *(1,2,3....n)*

```
O(n)   • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
O(n^2) • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
       • while $|Z|>1$
           • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
           • $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
           • form C by creating a new cluster root and connecting it to the two cluster roots
             with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
           • $Z = Z \cup \{C\} - \{A,B\}$
           • define $D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )$
```

# Neighbor Joining

**Algorithm** Given a distance matrix *M* with rows labeled *(1,2,3....n)*

- O(n) let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
- O(n²) for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
- while $|Z|>1$
  - O(n) define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
  - $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
  - form *C* by creating a new cluster root and connecting it to the two cluster roots with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
  - $Z = Z \cup \{C\} - \{A,B\}$
  - define $D(F,C) = D(C,F) = 1/2\ (\ D(A,F) + D(B,F) - D(A,B)\ )$

# Neighbor Joining

**Algorithm** Given a distance matrix *M* with rows labeled *(1,2,3….n)*

```
$O(n)$    • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
$O(n^2)$  • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\}) = M_{i,j}$
          • while $|Z| > 1$
$O(n)$        • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
$O(n^2)$      • $(A, B) = arg \min_{(A,B) \in Z} D(A, B) - u_A - u_B$
              • form $C$ by creating a new cluster root and connecting it to the two cluster roots
                with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
              • $Z = Z \cup \{C\} - \{A,B\}$
              • define $D(F,C) = D(C,F) = 1/2\ (\ D(A,F) + D(B,F) - D(A,B)\ )$
```

# Neighbor Joining

**Algorithm** Given a distance matrix *M* with rows labeled *(1,2,3....n)*

```
$O(n)$    • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
$O(n^2)$  • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
          • while $|Z|>1$
$O(n)$        • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
$O(n^2)$      • $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
$O(1)$        • form $C$ by creating a new cluster root and connecting it to the two cluster roots
                with edge weights $\frac{1}{2}\left(D(A,B)+(u_A-u_B)\right)$ and $\frac{1}{2}\left(D(A,B)+(u_B-u_A)\right)$ respectively.
              • $Z = Z \cup \{C\} - \{A,B\}$
              • define $D(F,C) = D(C,F) = 1/2\ (\ D(A,F) + D(B,F) - D(A,B)\ )$
```

# Neighbor Joining

```
Algorithm Given a distance matrix $M$ with rows labeled $(1,2,3....n)$
O(n)     • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
O(n²)    • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
         • while $|Z|>1$
O(n)         • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
O(n²)        • $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
O(1)         • form $C$ by creating a new cluster root and connecting it to the two cluster roots
               with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
O(1)         • $Z = Z \cup \{C\} - \{A,B\}$
             • define $D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )$
```

# Neighbor Joining

**Algorithm** Given a distance matrix $M$ with rows labeled $(1,2,3....n)$

```
• let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)    $O(n)$
• for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$    $O(n^2)$
• while $|Z|>1$
    • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$    $O(n)$
    • $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$    $O(n^2)$
    • form $C$ by creating a new cluster root and connecting it to the two cluster roots
      with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.    $O(1)$
    • $Z = Z \cup \{C\} - \{A,B\}$    $O(1)$
    • define $D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )$    $O(n)$
```

# Neighbor Joining

**Algorithm** Given a distance matrix $M$ with rows labeled $(1,2,3....n)$

```
O(n)      • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
$O(n^2)$  • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
          • while $|Z|>1$   O(n)
O(n)          • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
$O(n^2)$      • $(A,B) = arg \min_{(A,B) \in Z} D(A,B) - u_A - u_B$
O(1)          • form $C$ by creating a new cluster root and connecting it to the two cluster roots
                with edge weights $\frac{1}{2}\left(D(A,B) + (u_A - u_B)\right)$ and $\frac{1}{2}\left(D(A,B) + (u_B - u_A)\right)$ respectively.
O(1)          • $Z = Z \cup \{C\} - \{A,B\}$
O(n)          • define $D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )$
```

# Neighbor Joining

**Algorithm** Given a distance matrix $M$ with rows labeled $(1,2,3....n)$

```
$O(n)$    • let $Z = \{\{1\},\{2\},\{3\},..,\{n\}\}$ (* the set of initial clusters *)
$O(n^2)$  • for all $\{i\},\{j\} \in Z$ set $D(\{i\},\{j\})=M_{i,j}$
          • while $|Z|>1$   $O(n)$
$O(n)$        • define $u_A = 1/(n-2) * \Sigma_{F \in Z} D(A,F)$ for all $A \in Z$
$O(n^2)$      • $(A, B) = arg \min_{(A,B) \in Z} D(A, B) - u_A - u_B$
$O(1)$        • form $C$ by creating a new cluster root and connecting it to the two cluster roots
                with edge weights $\frac{1}{2}(D(A,B) + (u_A - u_B))$ and $\frac{1}{2}(D(A,B) + (u_B - u_A))$ respectively.
$O(1)$        • $Z = Z \cup \{C\} - \{A,B\}$
$O(n)$        • define $D(F,C) = D(C,F) = 1/2 ( D(A,F) + D(B,F) - D(A,B) )$
```

**$O(n^3)$ total time**

# Burrows-Wheeler Transform

Remember our old friend the suffix array?

$T$ = `mississippi$`

| $SA_T$ | | $BWT_T$ |
|---|---|---|
| 12 | `$mississipp`**`i`** | i |
| 11 | `i$mississip`**`p`** | p |
| 8 | `ippi$missis`**`s`** | s |
| 5 | `issippi$mis`**`s`** | s |
| 2 | `ississippi$`**`m`** | m |
| 1 | `mississippi`**`$`** | $ |
| 10 | `pi$mississi`**`p`** | p |
| 9 | `ppi$missis`**`i`** | i |
| 7 | `sippi$missi`**`s`** | s |
| 4 | `sissippi$mi`**`s`** | s |
| 6 | `ssippi$mis`**`i`** | i |
| 3 | `ssissippi$m`**`i`** | i |

# Burrows-Wheeler Transform

Remember our old friend the suffix array?

$T = \texttt{mississippi\$}$

| $SA_T$ | | $BWT_T$ |
|---|---|---|
| 12 | `$mississipp`**`i`** | i |
| 11 | `i$mississip`**`p`** | p |
| 8 | `ippi$missis`**`s`** | s |
| 5 | `issippi$mis`**`s`** | s |
| 2 | `ississippi$`**`m`** | m |
| 1 | `mississippi`**`$`** | $ |
| 10 | `pi$mississi`**`p`** | p |
| 9 | `ppi$missis`**`i`** | i |
| 7 | `sippi$missi`**`s`** | s |
| 4 | `sissippi$mi`**`s`** | s |
| 6 | `ssippi$miss`**`i`** | i |
| 3 | `ssissippi$m`**`i`** | i |

$$BWT_T = \begin{cases} T\left[SA_T[i] - 1\right] & \textbf{if } SA_T[i] > 1 \\ \boldsymbol{\$} & \textbf{if } SA_T[i] = 1 \end{cases}$$

# BWT Index

A **BWT Index** for a sequence *T* is a data structure with:

- the $BWT_{T\$}$ encoded as a wavelet tree; and
- the integer array *C[0...σ],* where *C[c]* stores the number of occurances of the characters less than *c* in *T$*

With the BWT Index, you can:

- construct the Suffix Array
- recover *T* in *O(log n)* per character

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

# Counting Occurrences

$i = m$

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

$i = m$

$(sp, ep) = (1, n)$

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1,n)$
while $sp \le ep$ and $i \ge 1$ do
```

# Counting Occurrences

Input

- pattern, $P = p_{1,}p_2,p_3,...,p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1,n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1,n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L,sp-1)+1$
```

# Counting Occurrences

Input

- pattern, $P = p_1,p_2,p_3,\ldots,p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1,n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L,sp-1)+1$
    $ep = C[c] + rank_c(L,ep)$
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1, n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L, sp-1)+1$
    $ep = C[c] + rank_c(L, ep)$
    $i = i - 1$
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1,n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L,sp-1)+1$
    $ep = C[c] + rank_c(L,ep)$
    $i = i - 1$
if $ep < sp$ then
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1, n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L, sp-1)+1$
    $ep = C[c] + rank_c(L, ep)$
    $i = i - 1$
if $ep < sp$ then
    return $0$
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
i = m
(sp, ep) = (1,n)
while sp ≤ ep and i ≥ 1 do
    c = p_j
    sp = C[c] + rank_c(L,sp-1)+1
    ep = C[c] + rank_c(L,ep)
    i = i - 1
if ep < sp then
    return 0
else
```

# Counting Occurrences

Input

- pattern, $P = p_1, p_2, p_3, \ldots, p_m$
- count array, $C$
- $BWT_{T\$}$, $L$

Output

- number of occurrences of $P$ in $T$

```
$i = m$
$(sp, ep) = (1, n)$
while $sp \leq ep$ and $i \geq 1$ do
    $c = p_j$
    $sp = C[c] + rank_c(L, sp-1) + 1$
    $ep = C[c] + rank_c(L, ep)$
    $i = i - 1$
if $ep < sp$ then
    return $0$
else
    return $ep - sp + 1$
```

# Bidirectional BWT

Given a string *T* a **bidirectional BWT index** is a data structure with the following operations:

# Bidirectional BWT

Given a string *T* a **bidirectional BWT index** is a data structure with the following operations:

- *isLeftMaximial(i,j)* -- 1 if $BWT_{T\$}[i...j]$ contains more than one value, 0 otherwise
- *isRightMaximial(i,j)* -- 1 if $BWT_{\underline{T}\$}[i...j]$ contains more than one value, 0 otherwise

# Bidirectional BWT

Given a string *T* a **bidirectional BWT index** is a data structure with the following operations:

- *isLeftMaximial(i,j)* -- 1 if $BWT_{T\$}[i...j]$ contains more than one value, 0 otherwise
- *isRightMaximial(i,j)* -- 1 if $BWT_{\underline{T}\$}[i...j]$ contains more than one value, 0 otherwise
- *enumerateLeft(i,j)* -- return the distinct values $BWT_{T\$}[i...j]$ in lexicographic order
- *enumerateRight(i,j)* -- return the distinct values $BWT_{\underline{T}\$}[i...j]$ in lexicographic order

# Bidirectional BWT

Given a string *T* a **bidirectional BWT index** is a data structure with the following operations:

- *isLeftMaximial(i,j)* -- 1 if $BWT_{T\$}[i...j]$ contains more than one value, 0 otherwise
- *isRightMaximial(i,j)* -- 1 if $BWT_{\underline{T}\$}[i...j]$ contains more than one value, 0 otherwise
- *enumerateLeft(i,j)* -- return the distinct values $BWT_{T\$}[i...j]$ in lexicographic order
- *enumerateRight(i,j)* -- return the distinct values $BWT_{\underline{T}\$}[i...j]$ in lexicographic order
- *extendLeft(c,**I**(W,T),**I**($\underline{W}$,$\underline{T}$))* -- returns the pair (**I**(cW,T),**I**($\underline{W}$c,$\underline{T}$))
- *extendRight(c,**I**(W,T),**I**($\underline{W}$,$\underline{T}$))* -- returns the pair (**I**(Wc,T),**I**(c$\underline{W}$,$\underline{T}$))

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T*
(interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in
the suffix tree of *T* where *~~v~~* is the
interval of *v* in the suffix array of *T$*

# Suffix Tree Traversal

*S* = empty stack

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where *~~v~~* is the interval of *v* in the suffix array of *T$*

# Suffix Tree Traversal

```
$S$ = empty stack
$S$.push(([1...n+1], [1...n+1], 0))
```

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where *~~v~~* is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs (~~*v*~~, $|\ell(v)|$) for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs ($\bar{v}$, $|\ell(v)|$) for all noes *v* in the suffix tree of *T* where $\bar{v}$ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where ~~*v*~~ is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T*
(interval *[1...n+1]* represents the root)

**Output** pairs (*~~v~~*, *|ℓ(v)|*) for all noes *v* in the suffix tree of *T* where *~~v~~* is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
```

# Suffix Tree Traversal

**Given** bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

**Output** pairs *(~~v~~, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where *~~v~~* is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

# Computational Problem

**Given**

- a reference genome $G$, and
- a set of reads $R = (r_1, r_2, r_3, \ldots, r_k) \in (\Sigma^n)^k$ where each read $r$ is a subsequence of $G$ with a small number changes

**Output**

- the semi-global alignment of $r_i$ and $G$ for all $r_i \in R$ with $<k$ changes

# Computational Problem

**Given**

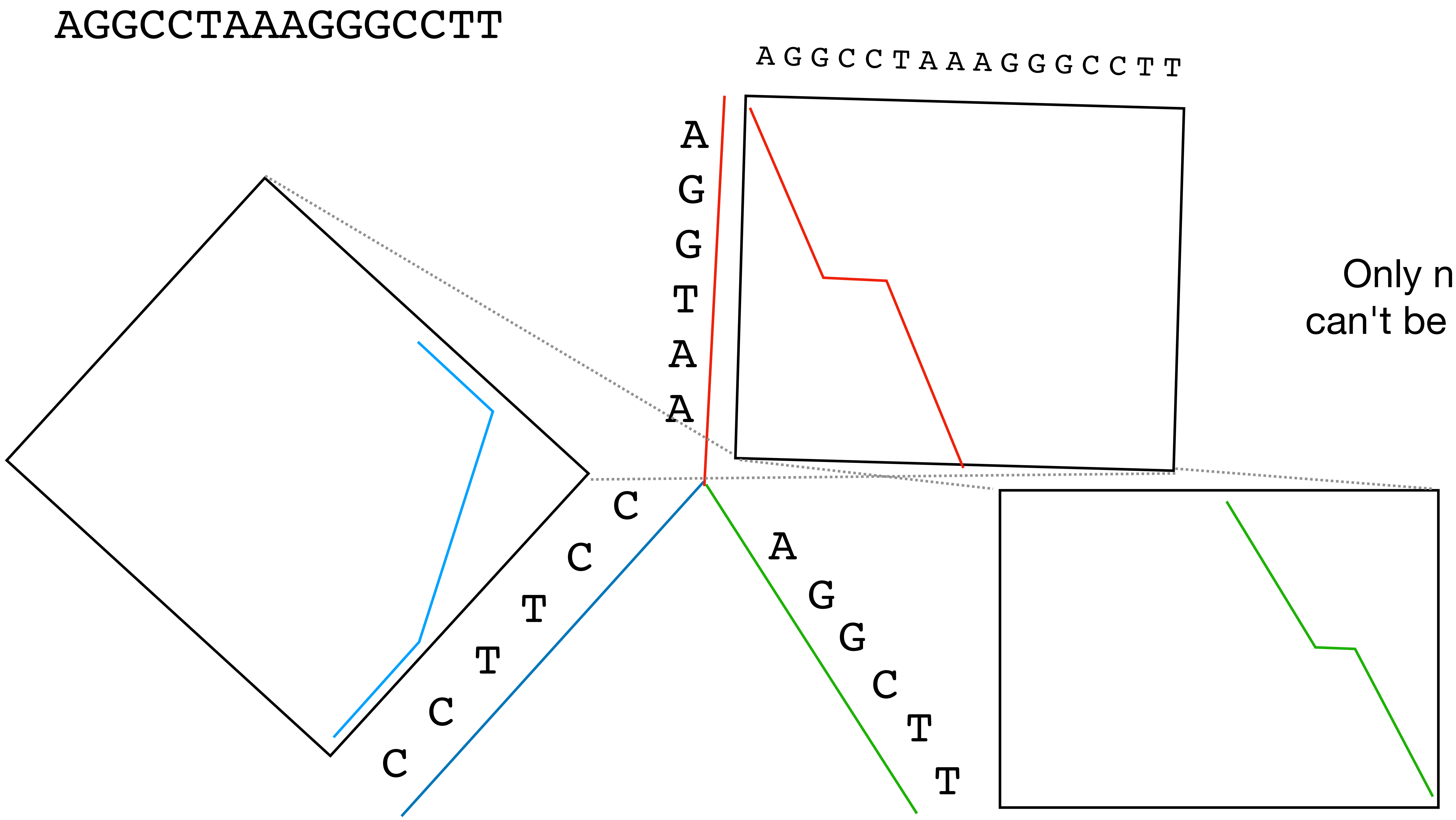
- a reference genome *G*, and
- a set of reads $R = (r_1, r_2, r_3, \ldots, r_k) \in (\Sigma^n)^k$ where each read *r* is a subsequence of *G* with a small number changes

**Output**

- the semi-global alignment of $r_i$ and *G* for all $r_i \in R$ with $<k$ changes

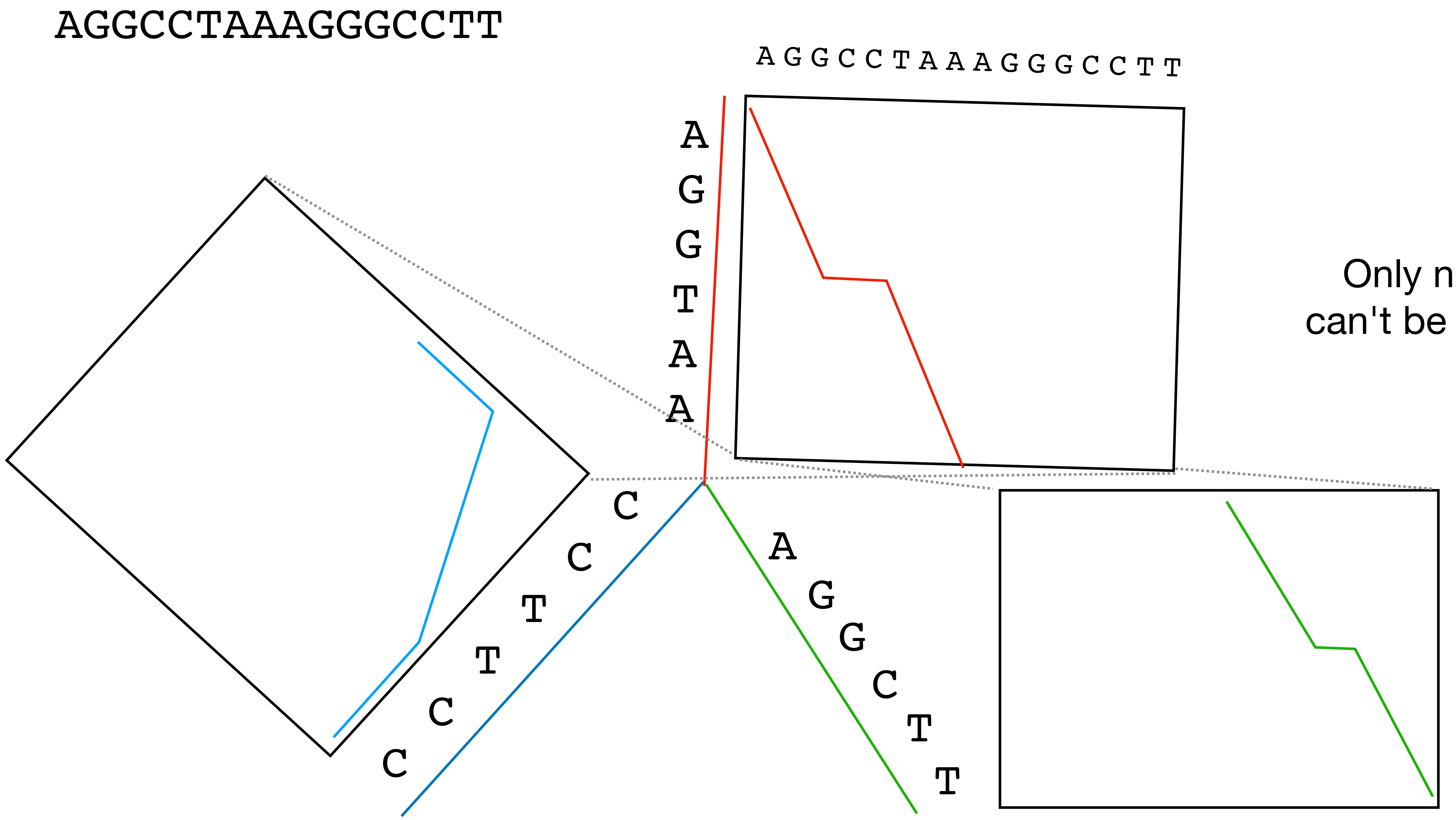
call these *k*-error mappings

# Aligning reads



Only need to go to a depth of *2m* since the best alignment can't be worse than deleting one string and inserting the other.

# Aligning reads



Only need to go to a depth of *2m* since the best alignment can't be worse than deleting one string and inserting the other.

**We don't have the suffix tree!**

# Dynamic Programming using a BWT

```
define Branch(d,[i...j]):
    for $c \in$ idx.enumerateRight(i,j) do
        process (c,d)
        if $d = 2m$ and $score > threshold$ do
            output alignment
        if $d < 2m$ do
            Branch(d+1,idx.extendRight(c, [i,j]))
```

compute the dynamic programming table row
using character *c* in row *d*

# Dynamic Programming using a BWT

```
define Branch(d,[i...j]):
    for c ∈ idx.enumerateRight(i,j) do
        process (c,d)
        if d = 2m and score > threshold do
            output alignment
        if d < 2m do
            Branch(d+1,idx.extendRight(c, [i,j]))
```

compute the dynamic programming table row using character ***c*** in row ***d***

$O(m\sigma)$-time

# Dynamic Programming using a BWT

```
define Branch(d,[i...j]):
    for $c \in$ idx.enumerateRight(i,j) do
        process (c,d)
        if $d = 2m$ and score > threshold do
            output alignment
        if $d < 2m$ do
            Branch(d+1,idx.extendRight(c, [i,j]))
```

**compute the dynamic programming table row using character *c* in row *d***

$O(m\sigma)$-time

$O(m^2+m\sigma)$-space

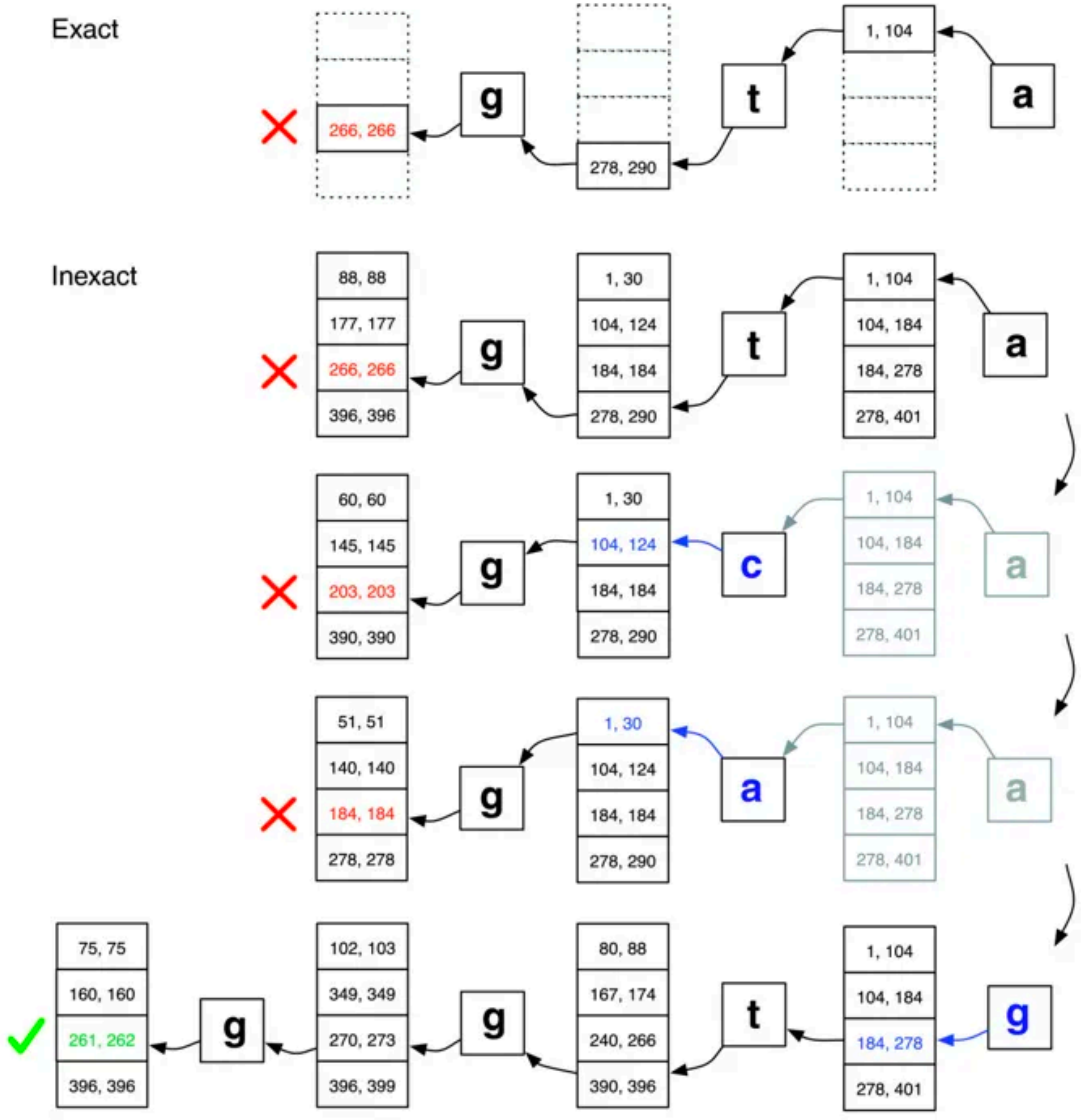
# Backtracking

Start by matching the exact sequence

If the algorithm reaches a point with no matches swap out characters already matched and restart search from that there

When ties occur, start with the character with the lowest quality score, keep the rest in a stack

Keep track of how many changes are made



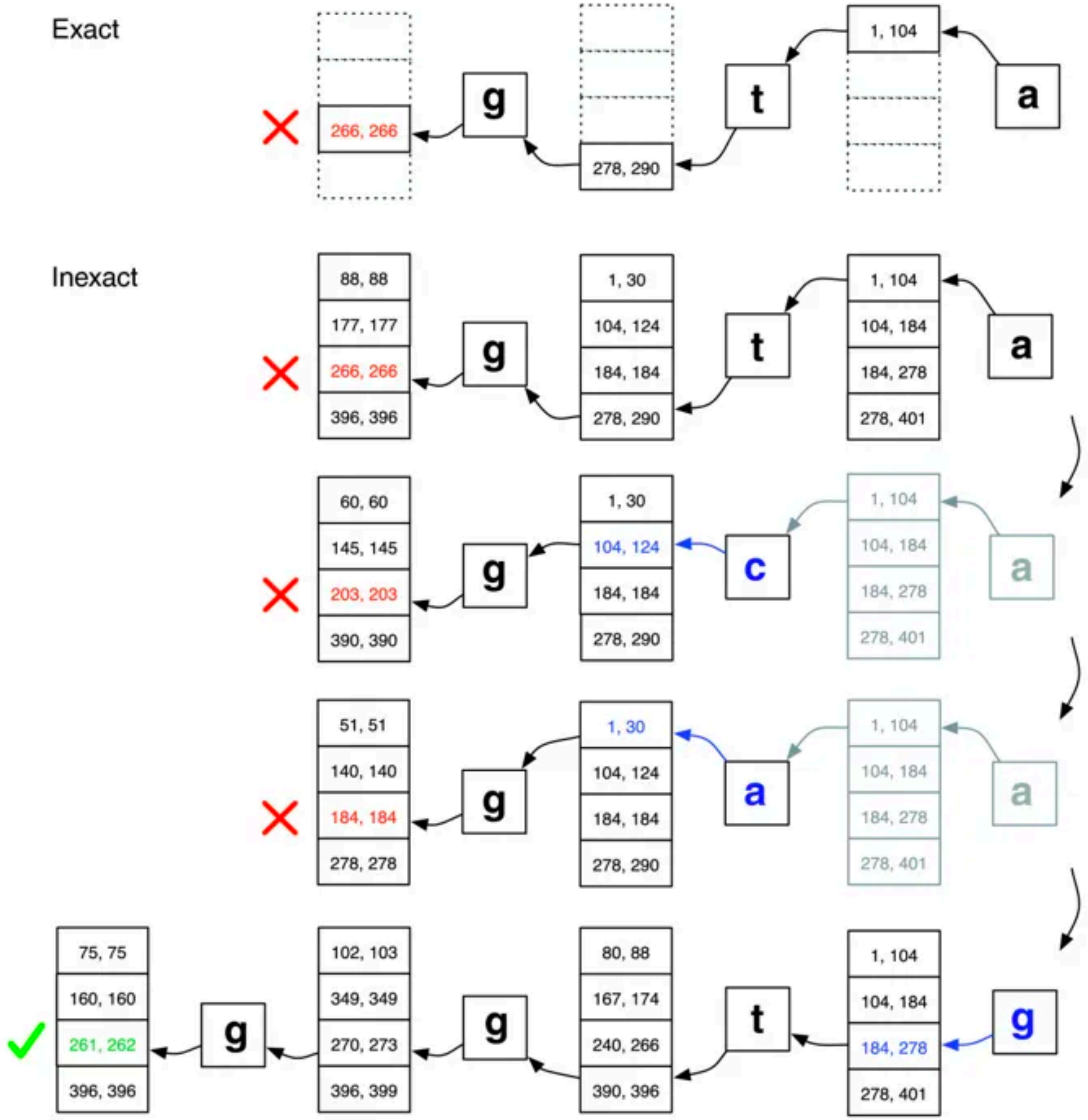
# Backtracking

Start by matching the exact sequence

If the algorithm reaches a point with no matches swap out characters already matched and restart search from that there

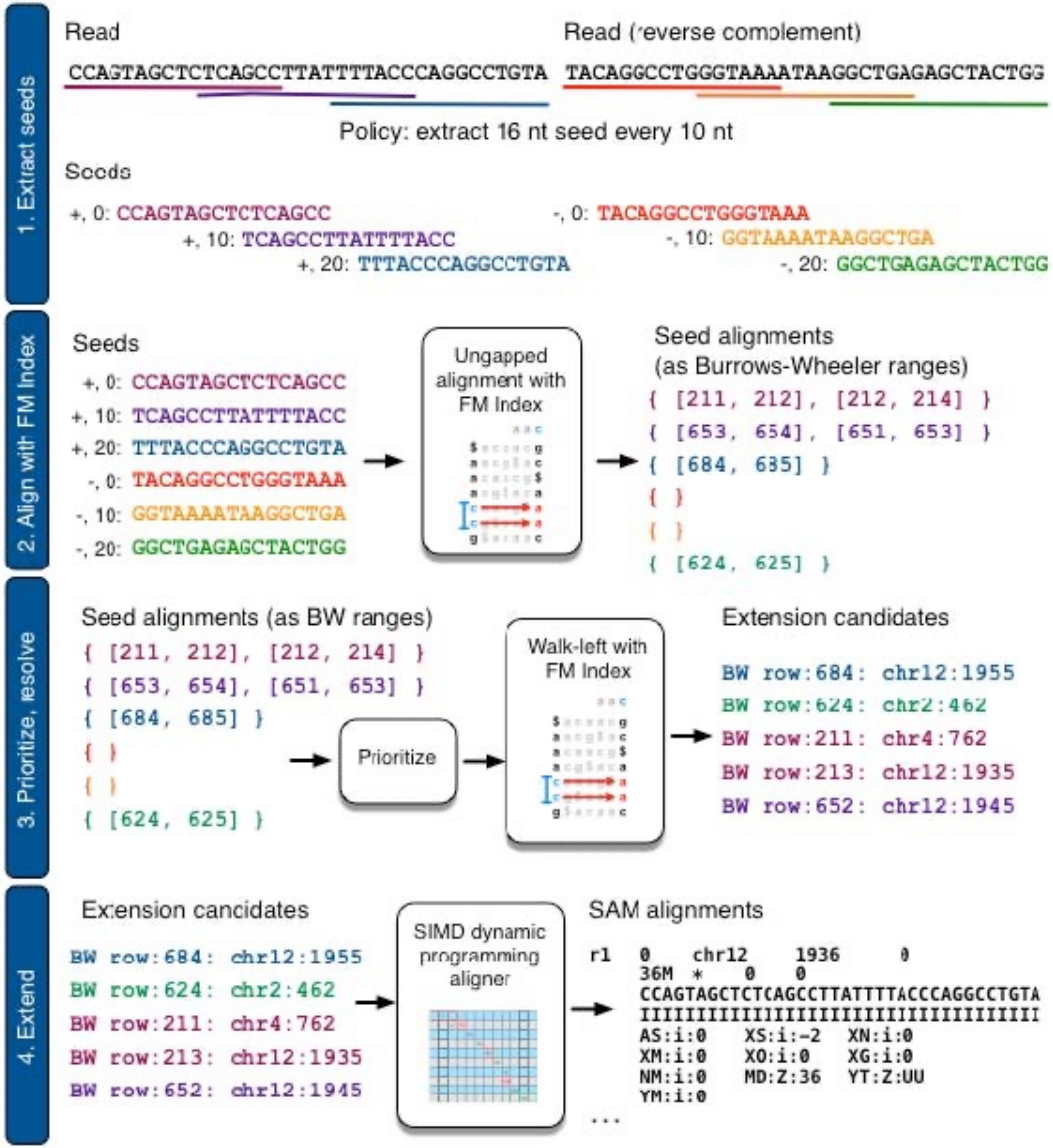
When ties occur, start with the character with the lowest quality score, keep the rest in a stack

Keep track of how many changes are made



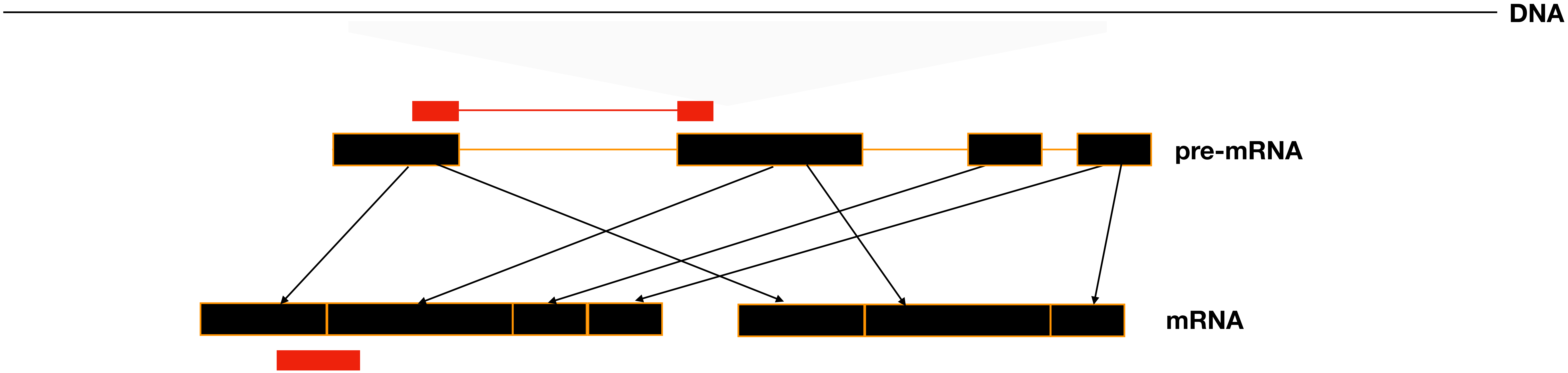
*"Bowtie conducts a quality-aware, greedy, randomized, depth-first search through the space of possible alignments."*

# Bowtie2



```
r1  0    chr12    1936     0
    36M  *    0    0
    CCAGTAGCTCTCAGCCTTATTTTACCCAGGCCTGTA
    IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
    AS:i:0    XS:i:-2   XN:i:0
    XM:i:0    XO:i:0    XG:i:0
    NM:i:0    MD:Z:36   YT:Z:UU
    YM:i:0
...
```

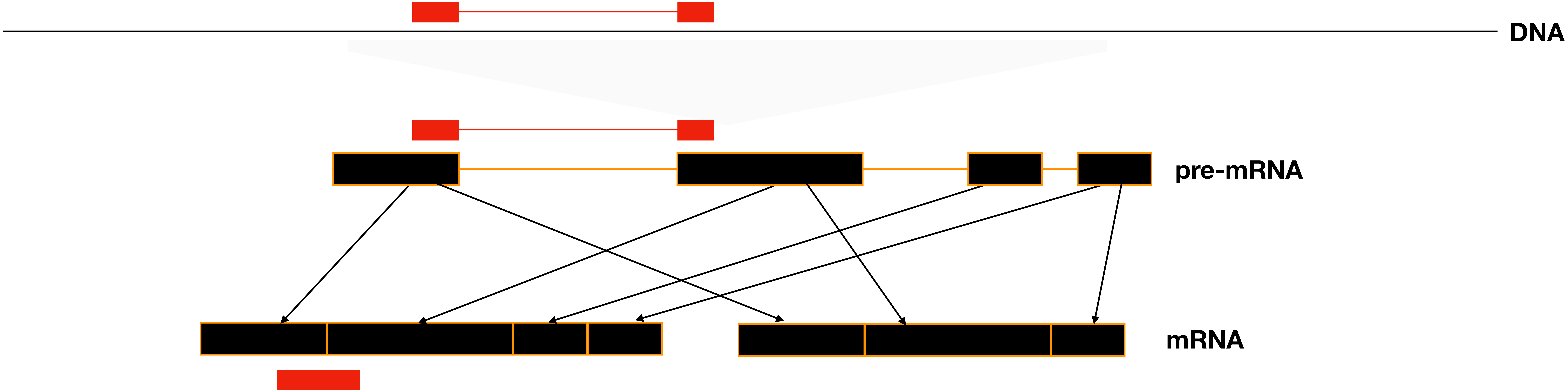
# Sequencing Applications



RNA sequencing

adapted from figure 1.2 in Mäkinen, *et al.* 2015

# Sequencing Applications



RNA sequencing

adapted from figure 1.2 in Mäkinen, *et al.* 2015

# Seed Searching

Maximal Mappable Prefix (*MMP*) for read *R,* read start location *i*, and genome *G*:

- the longest substring $R[i \ldots (i + MML - 1)]$
- such that there exists some set $J = \{j_1, j_2, \ldots, j_n\}$ where for all $j_k \in J$

$$R[i \ldots (i+MML-1)] = G[j \ldots (j_k+MML-1)]$$

- where *MML* is the Maximal Mapping Length

The basic algorithm is

- map from the start of the read as far as possible
- restart searching from the next position to the right

# Seed Searching

Maximal Mappable Prefix (*MMP*) for read *R,* read start location *i*, and genome *G*:

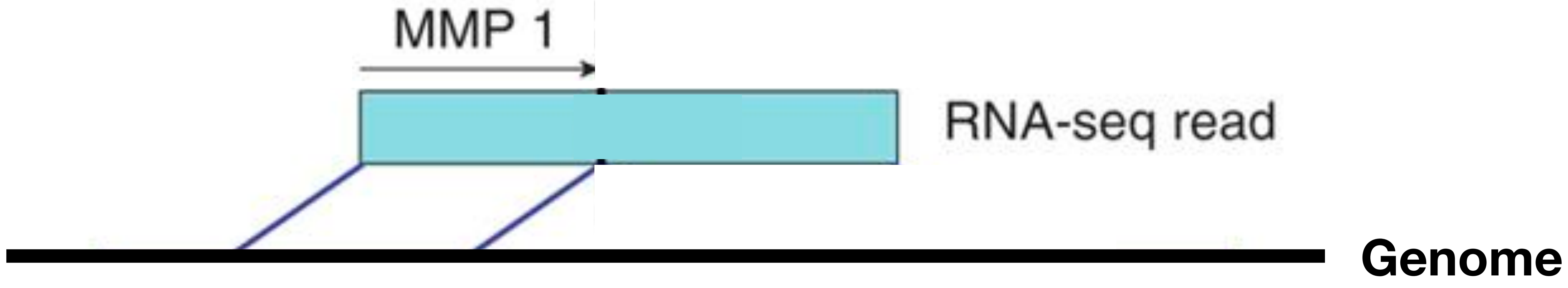
- the longest substring $R[i \ldots (i + MML - 1)]$
- such that there exists some set $J = \{j_1, j_2, \ldots, j_n\}$ where for all $j_k \in J$

$$R[i \ldots (i+MML-1)] = G[j \ldots (j_k+MML-1)]$$

- where *MML* is the Maximal Mapping Length

The basic algorithm is

- map from the start of the read as far as possible
- restart searching from the next position to the right



# Seed Searching

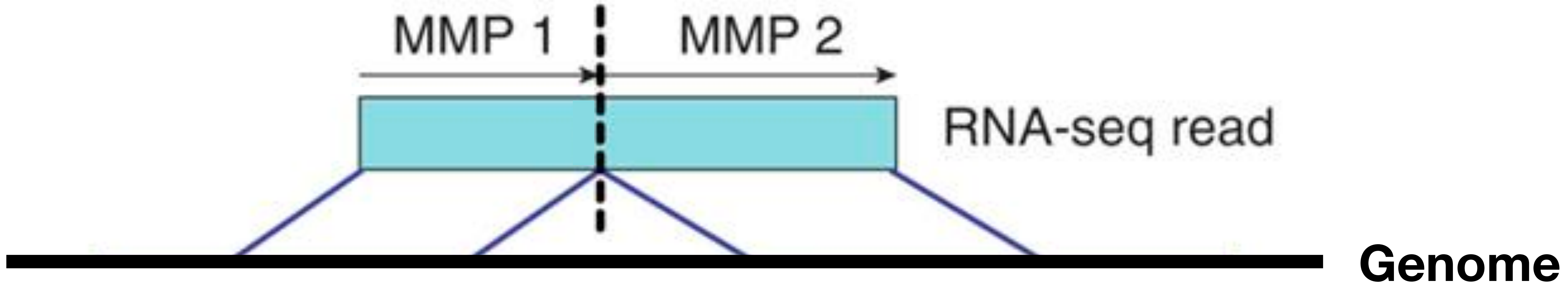
Maximal Mappable Prefix (*MMP*) for read *R,* read start location *i*, and genome *G*:

- the longest substring $R[i \ldots (i + MML - 1)]$
- such that there exists some set $J = \{j_1, j_2, \ldots, j_n\}$ where for all $j_k \in J$

  $R[i \ldots (i+MML-1)] = G[j \ldots (j_k+MML-1)]$
- where *MML* is the Maximal Mapping Length

The basic algorithm is

- map from the start of the read as far as possible
- restart searching from the next position to the right



# Seed Searching

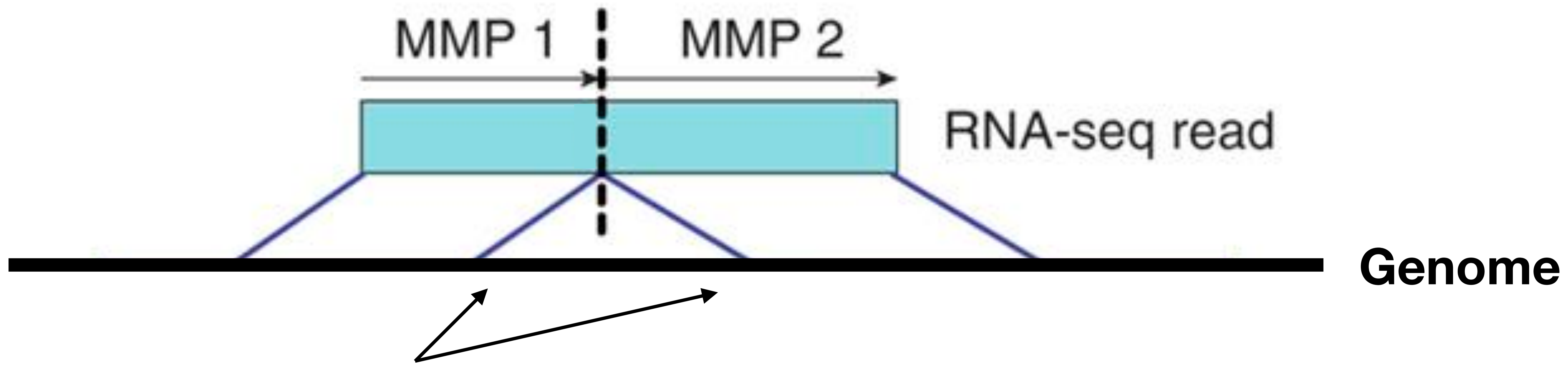
Maximal Mappable Prefix (*MMP*) for read *R,* read start location *i*, and genome *G*:

- the longest substring $R[i \ldots (i + MML - 1)]$
- such that there exists some set $J = \{j_1, j_2, \ldots, j_n\}$ where for all $j_k \in J$

  $$R[i \ldots (i+MML-1)] = G[j \ldots (j_k+MML-1)]$$
- where *MML* is the Maximal Mapping Length

The basic algorithm is

- map from the start of the read as far as possible
- restart searching from the next position to the right



**the "splice junctions" are inferred from the alignment**

# Seed Searching

Maximal Mappable Prefix (*MMP*) for read $R$, read start location $i$, and genome $G$:

- the longest substring $R[i \ldots (i + MML - 1)]$
- such that there exists some set $J = \{j_1, j_2, \ldots, j_n\}$ where for all $j_k \in J$

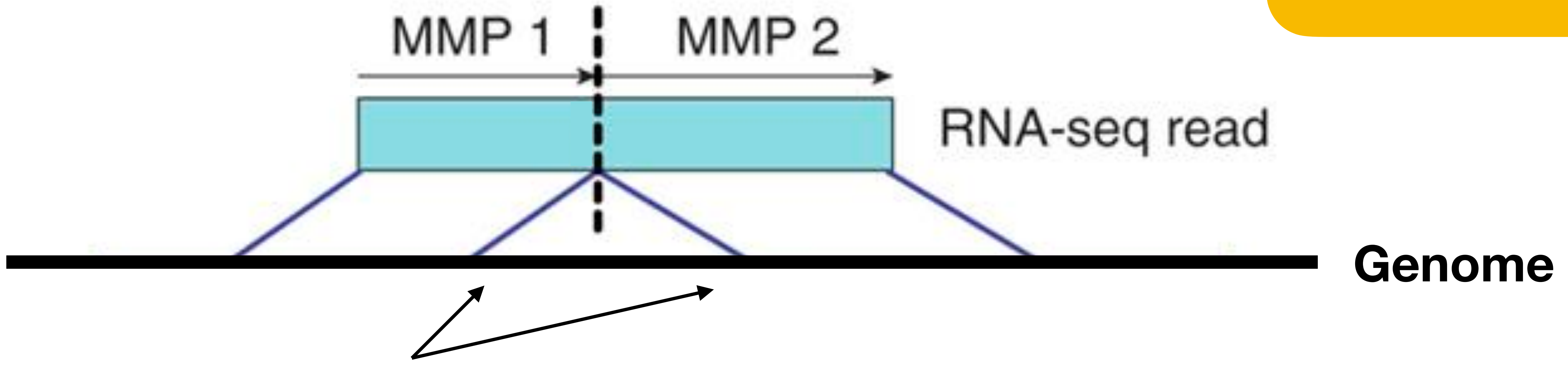
$$R[i \ldots (i+MML-1)] = G[j \ldots (j_k+MML-1)]$$

- where *MML* is the Maximal Mapping Length

The basic algorithm is

- map from the start of the read as far as possible
- restart searching from the next position to the right

The key is that the re-mapping only happens from the end of MMP1 rather than finding all maximal matchings then stitching



**the "splice junctions" are inferred from the alignment**

# Take Aways for STAR

Non-contiguous alignment for RNA-Seq is not a totally solved problem

STAR is specifically designed to take introns into account during alignment

Algorithm is extendable to longer read lengths since it can ignore poor quality regions and chimeric reads

Large memory consumption, but fast due to the use of uncompressed SAs

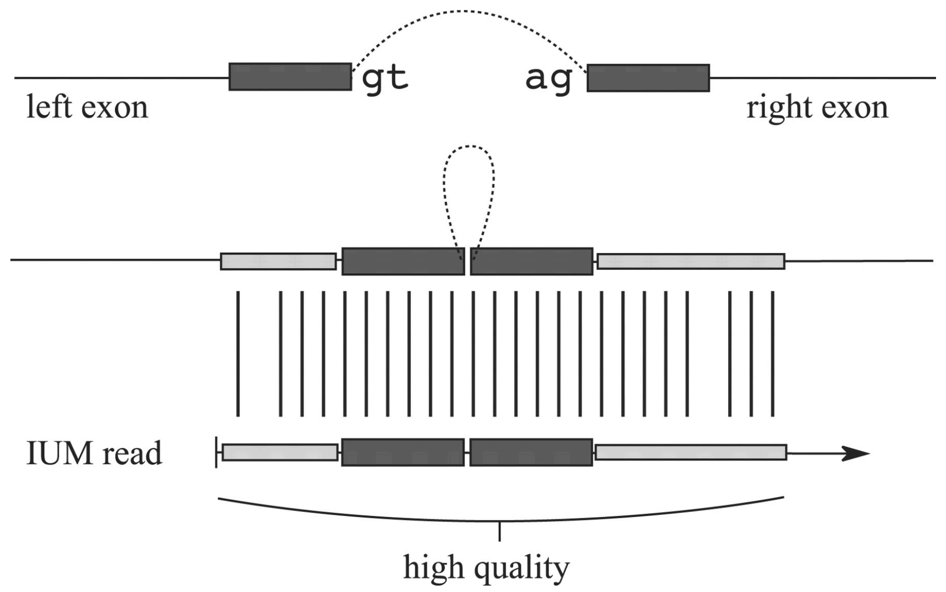
# TopHat

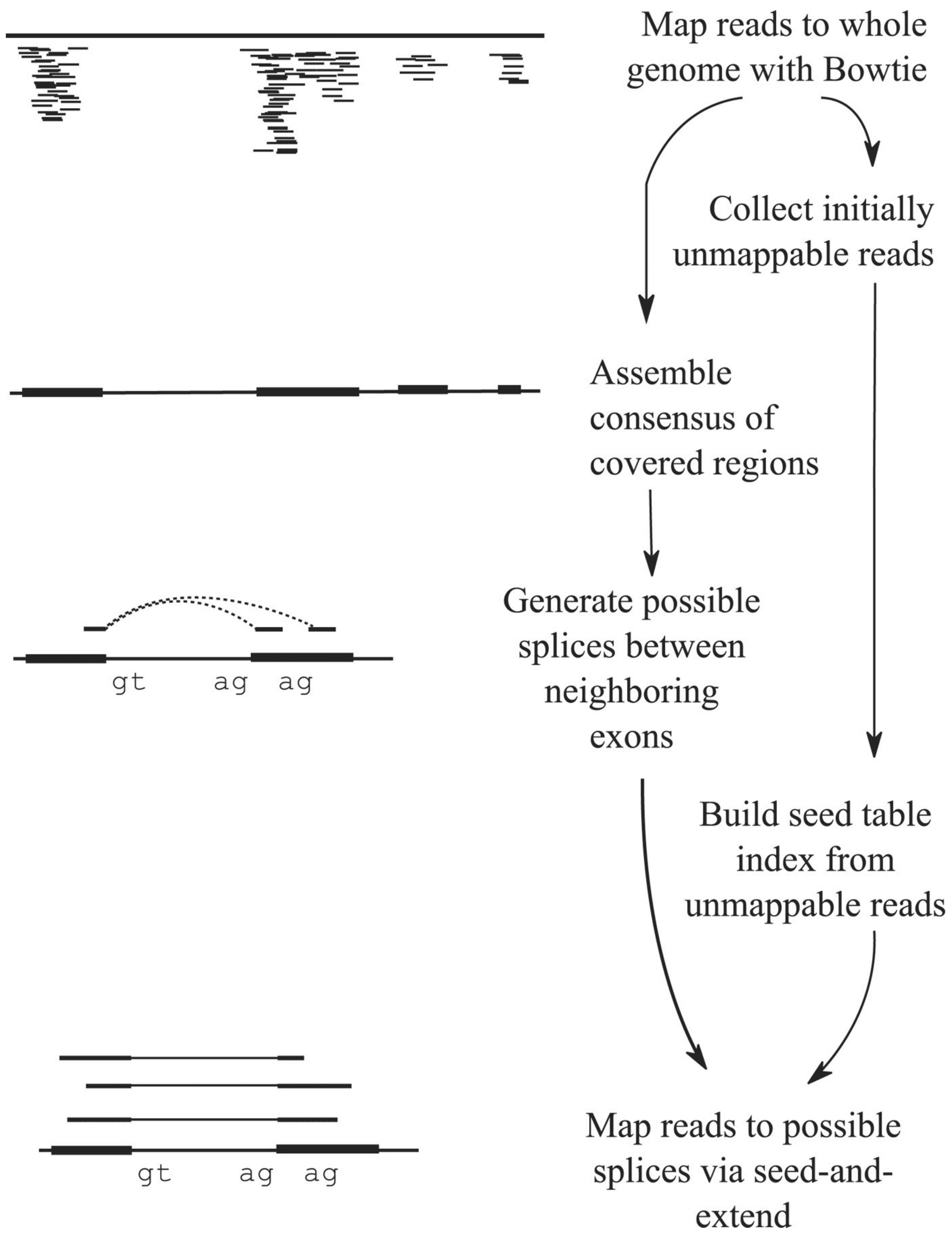
Using strict alignment critera, TopHat uses Bowtie to align reads to the whole genome

Construct the set of mapped sequences
- the "islands" of sequence that map to the genome
- using the assemble functionality of MAQ

Splice junctions usually happen with predictable bases
- consider all possible pairs as potential splice locations
- create a set of new sequences
- store the *k*-mer surrounding such locations as a seed for mapping

For each unmapped read
- extract all unique *k*-mers from the "high quality" region
- here *k~10*





# TopHat

Using strict alignment critera, TopHat uses Bowtie to align reads to the whole genome

Construct the set of mapped sequences

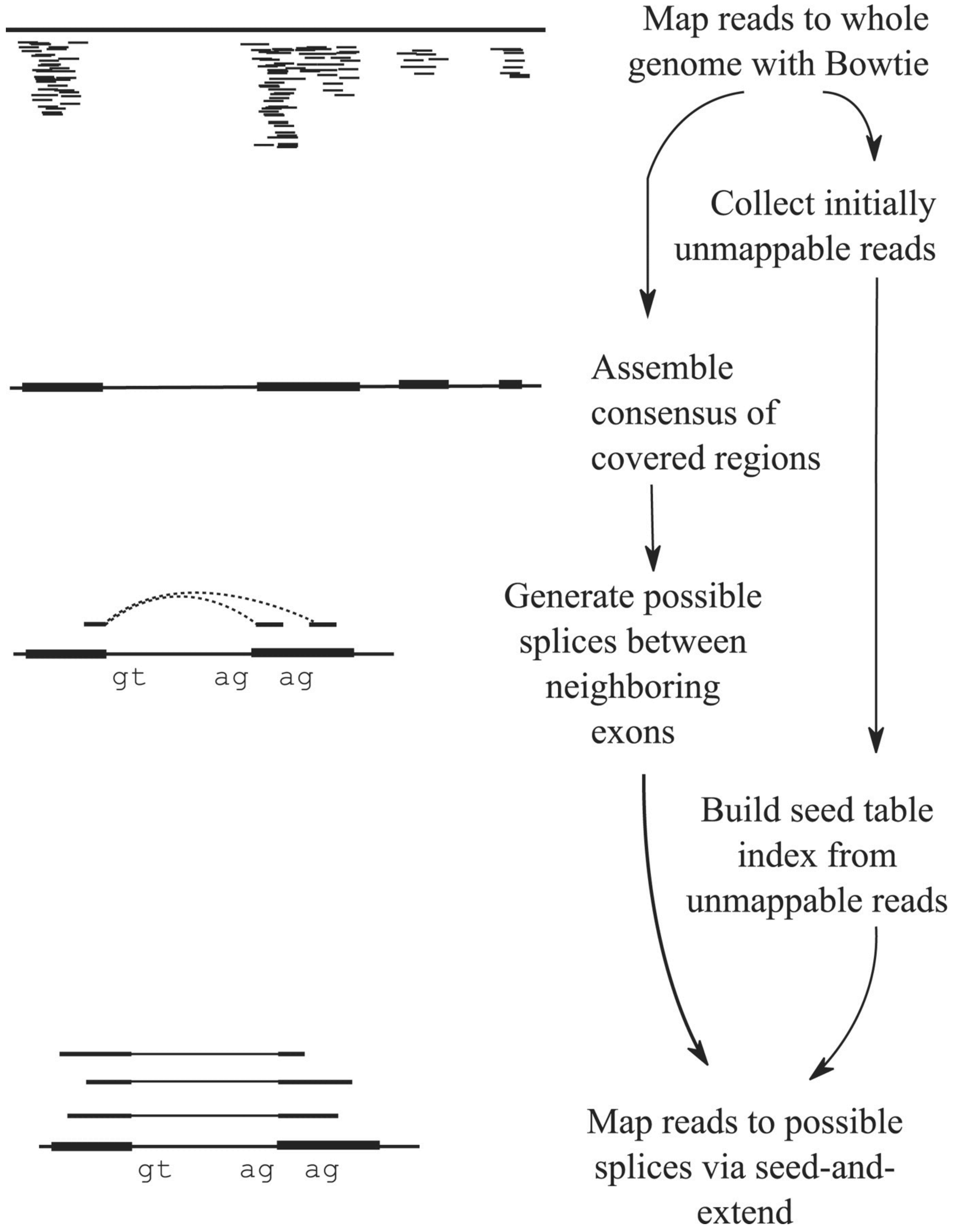
- the "islands" of sequence that map to the genome
- using the assemble functionality of MAQ

Splice junctions usually happen with predictable bases

- consider all possible pairs as potential splice locations
- create a set of new sequences
- store the *k*-mer surrounding such locations as a seed for mapping

For each unmapped read

- extract all unique *k*-mers from the "high quality" region
- here *k~10*



Map reads to whole genome with Bowtie

Collect initially unmappable reads

Assemble consensus of covered regions

Generate possible splices between neighboring exons

Build seed table index from unmappable reads

Map reads to possible splices via seed-and-extend

# Take Aways from TopHat

Uses existing software to do some of the heavy lifting

Strict parameters on the splice junctions make the algorithm fast

Limited in the splice junction sequence

# De Brujin Graphs

**Definition** a *k*-order de Brujin Graph (DBG) $D = (V,E)$ has:

- $V = \Sigma^k$ -- there is a vertex for each possible *k*-mer
- $E = \{ax \rightarrow xb \mid a,b \in \Sigma,\ x \in \Sigma^{(k-1)}\}$ -- for each $(k+1)$-mer *axb*, there is an edge from the *k*-mer *ax* to the *k*-mer *xb*

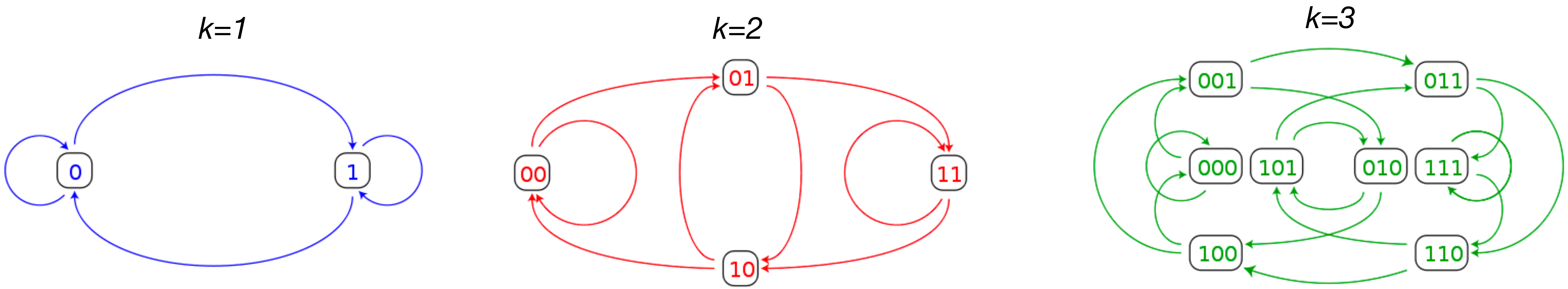


Image courtesy commons.wikimedia.org/wiki/File:DeBruijn-as-line-digraph.svg

# *Sequence* de Brujin Graphs

What is most commonly used in practice for genome assembly is a subset of the DBG based on a given sequence

This is sometimes in literature referred to as simply a de Brujin Graph

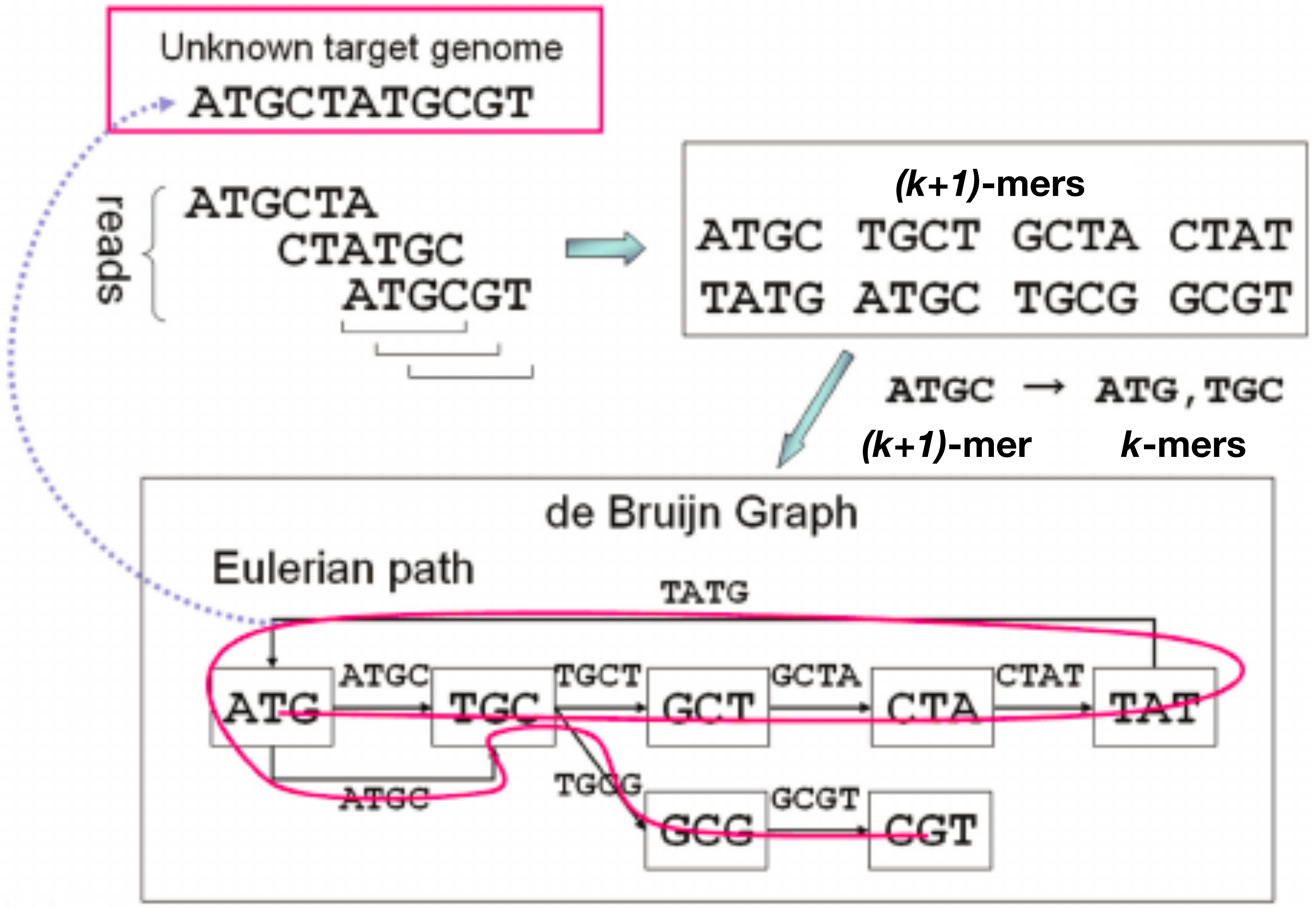
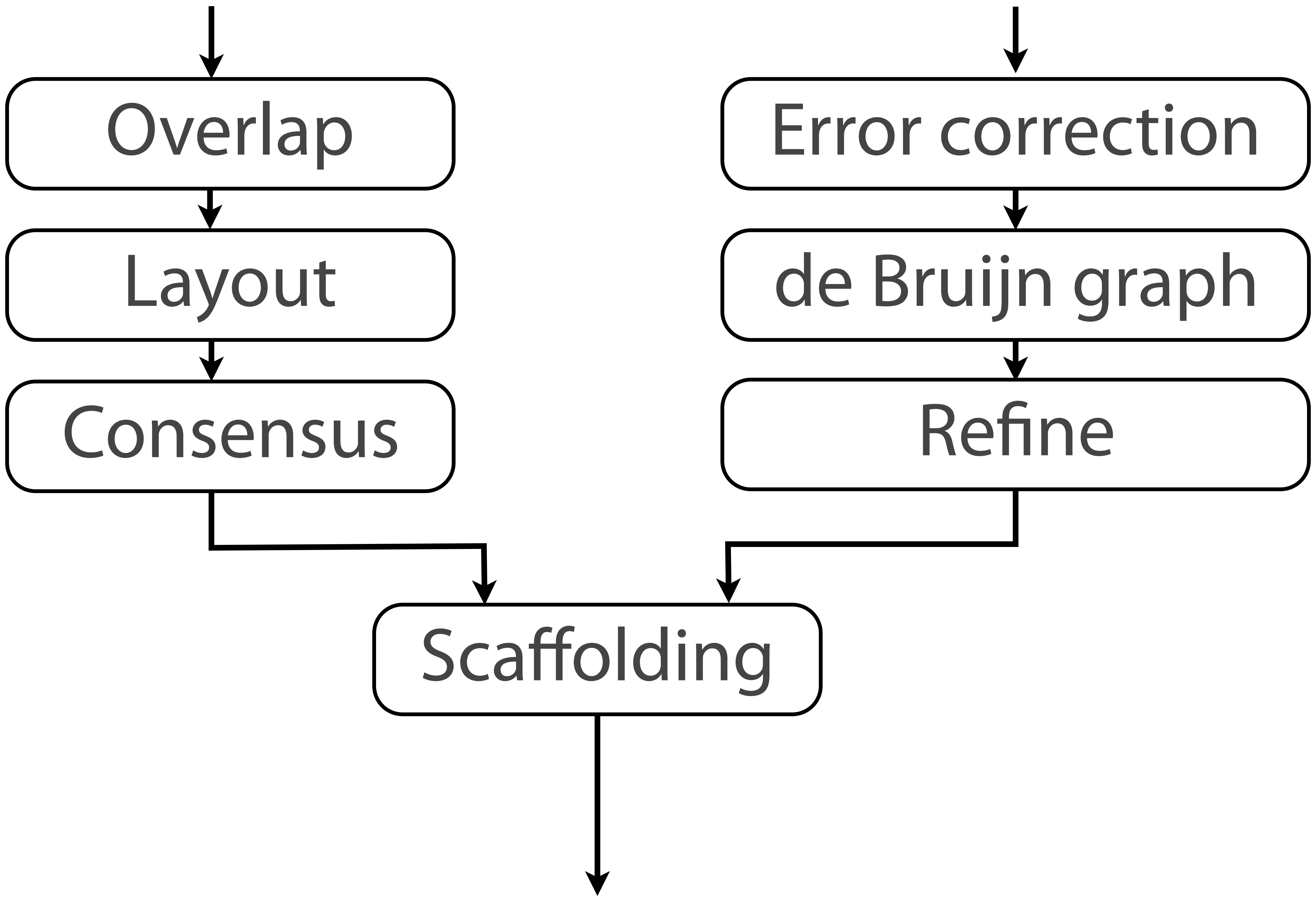


Image courtesy 10.1093/nar/gks678

# Assembly alternatives

Alternative 1: Overlap-Layout-Consensus (OLC) assembly

Alternative 2: De Bruijn graph (DBG) assembly



Overlap

Layout

Consensus

Error correction

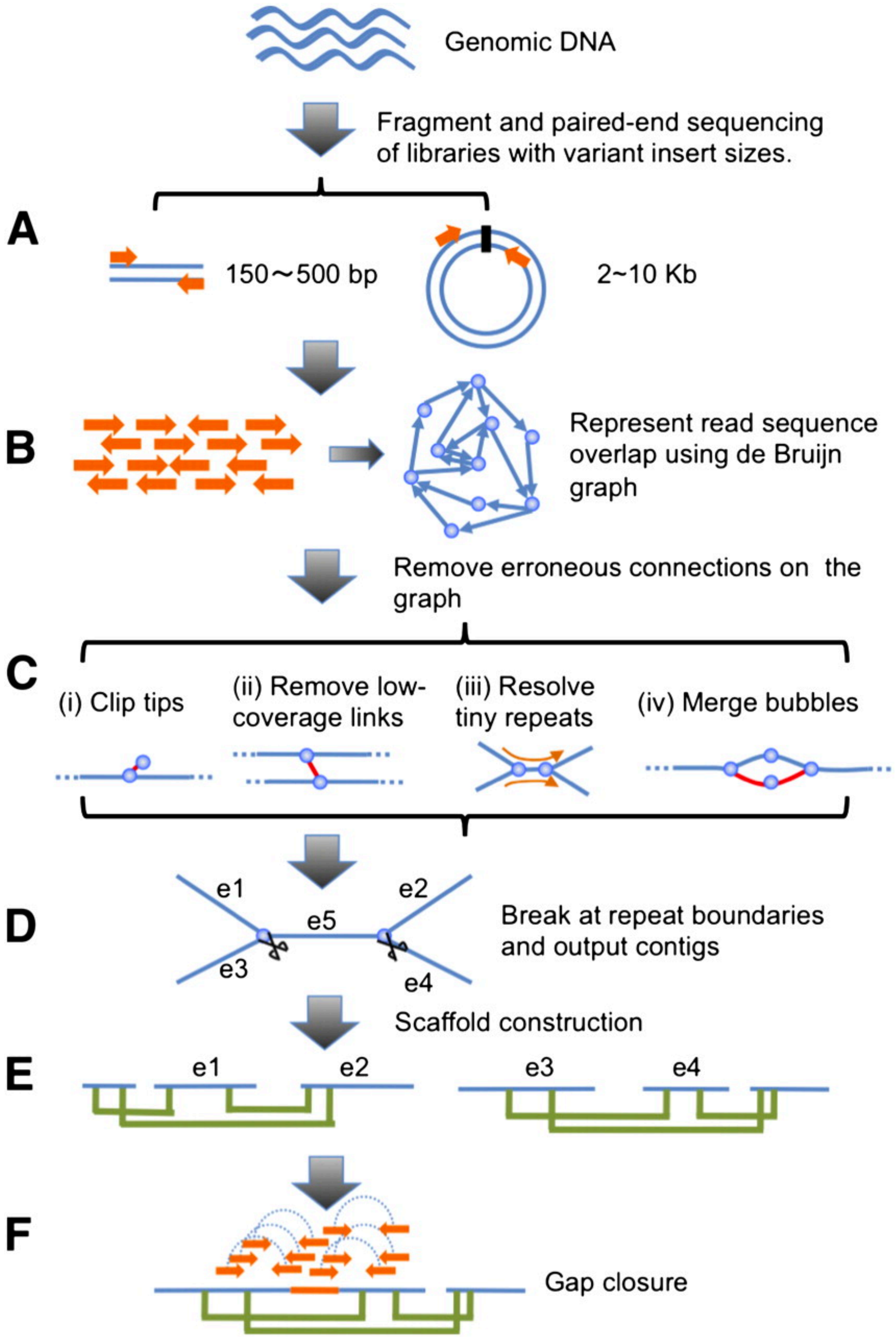
de Bruijn graph

Refine

Scaffolding

Slide courtesy of Ben Langmead: "Assembly in practice: OLC" langmead-lab.org/teaching-materials/

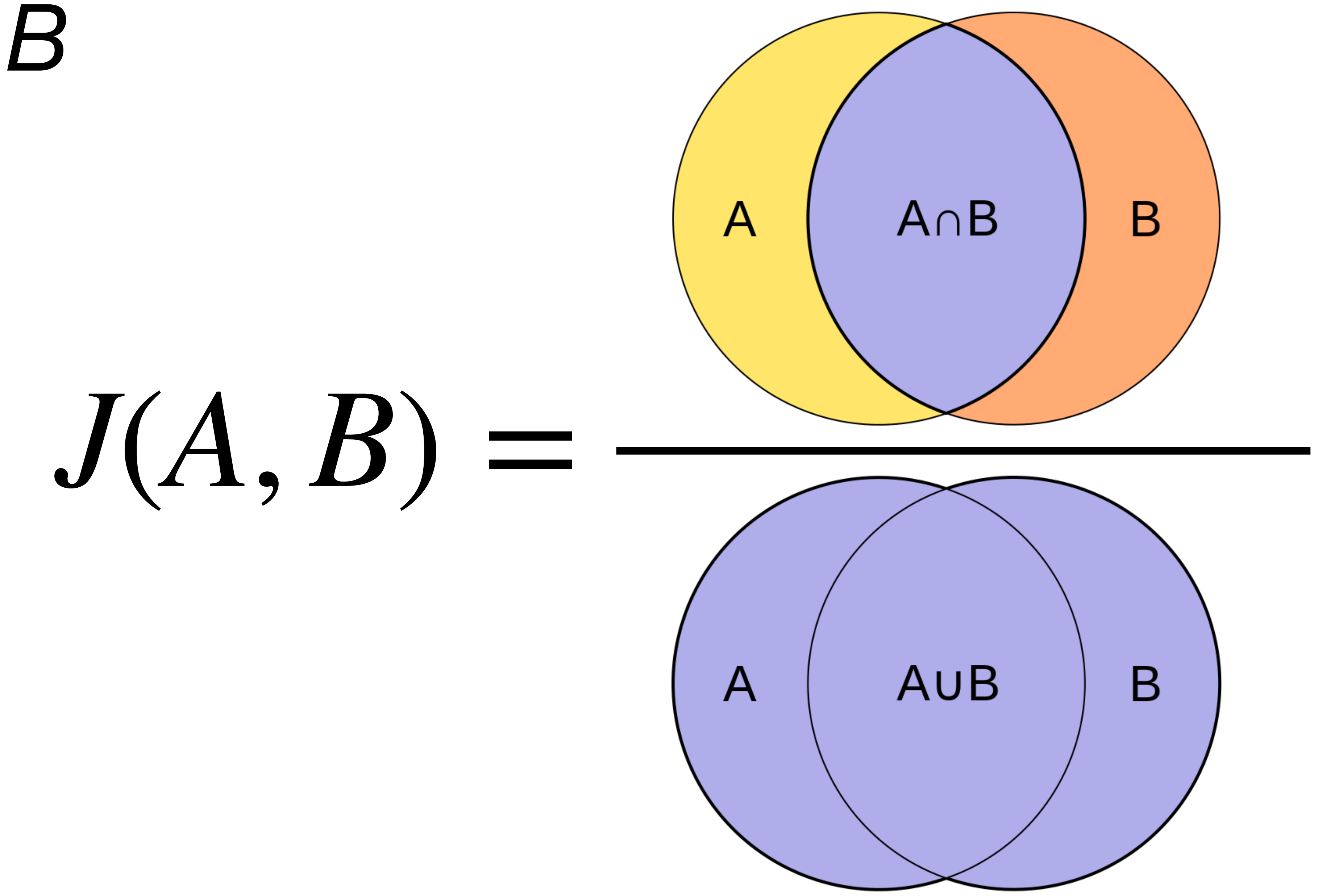
# SOAPdenovo


Genomic DNA
Fragment and paired-end sequencing of libraries with variant insert sizes.
A
150～500 bp
2~10 Kb
B
Represent read sequence overlap using de Bruijn graph
Remove erroneous connections on the graph
C
(i) Clip tips
(ii) Remove low-coverage links
(iii) Resolve tiny repeats
(iv) Merge bubbles
D
e1
e2
e5
e3
e4
Break at repeat boundaries and output contigs
Scaffold construction
E
e1
e2
e3
e4
F
Gap closure

# Jaccard Similarity

Measures the similarity of two sets of items *A* and *B* as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

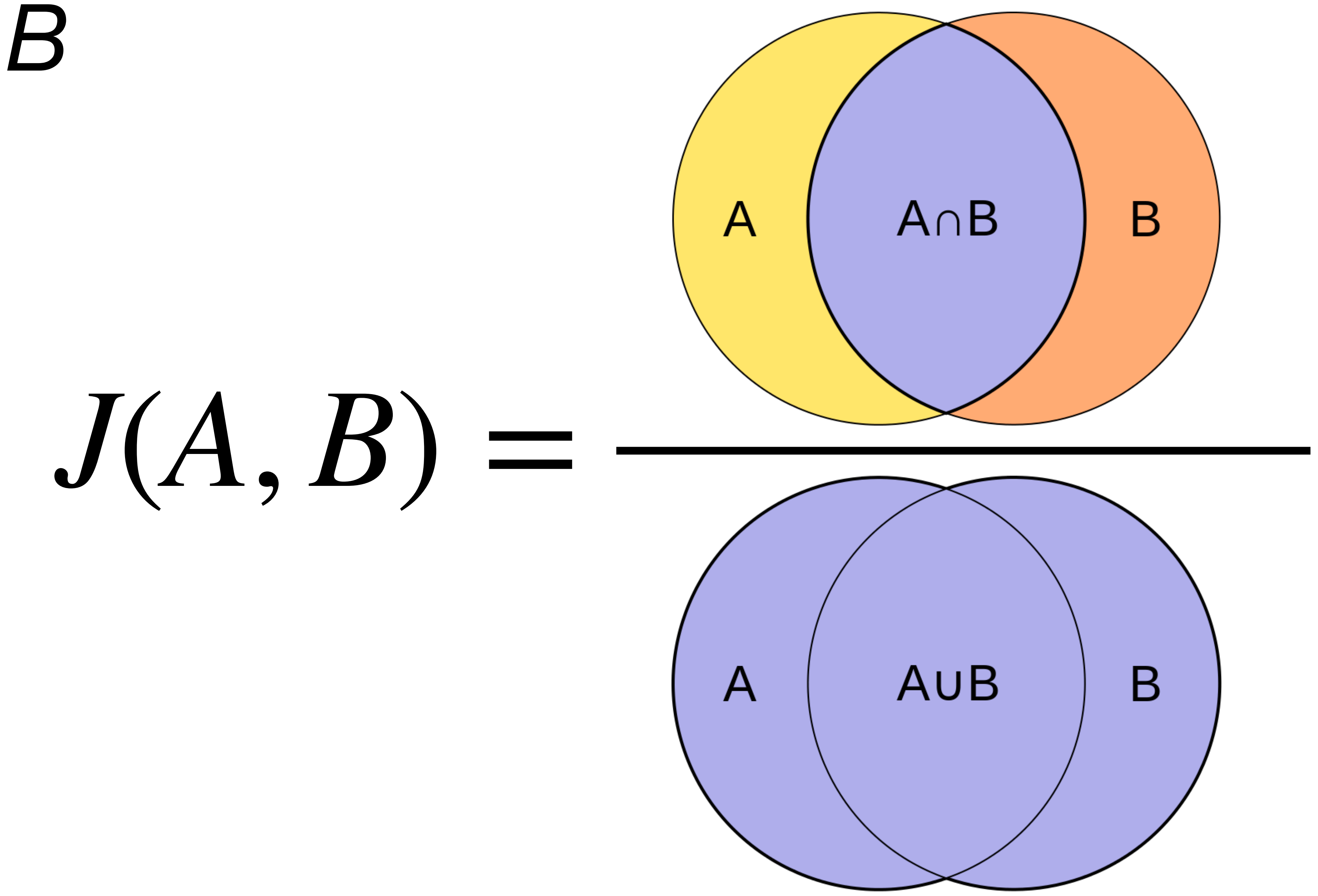


Images courtesy of Wikimedia Commons.

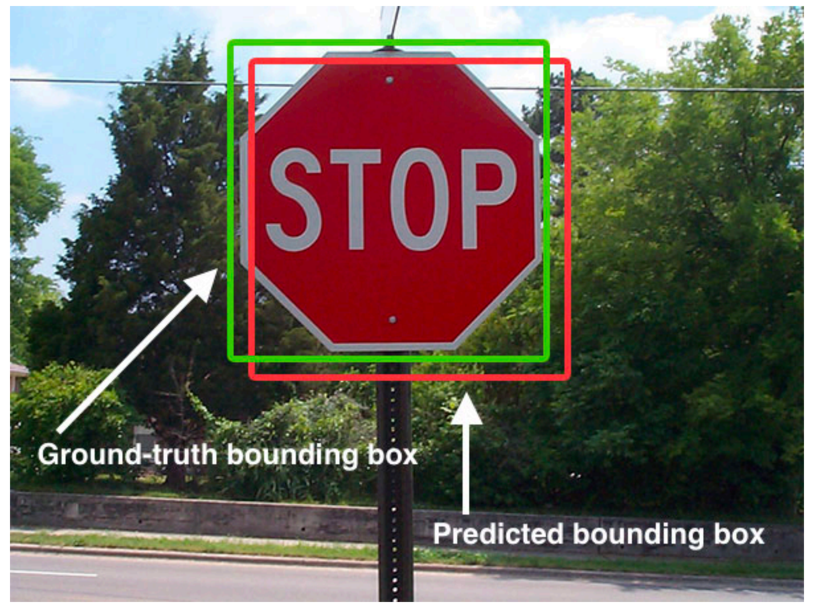
# Jaccard Similarity

Measures the similarity of two sets of items *A* and *B* as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$



Used also used in computer vision, sometimes called the "Intersection over Union" (IoU) metric

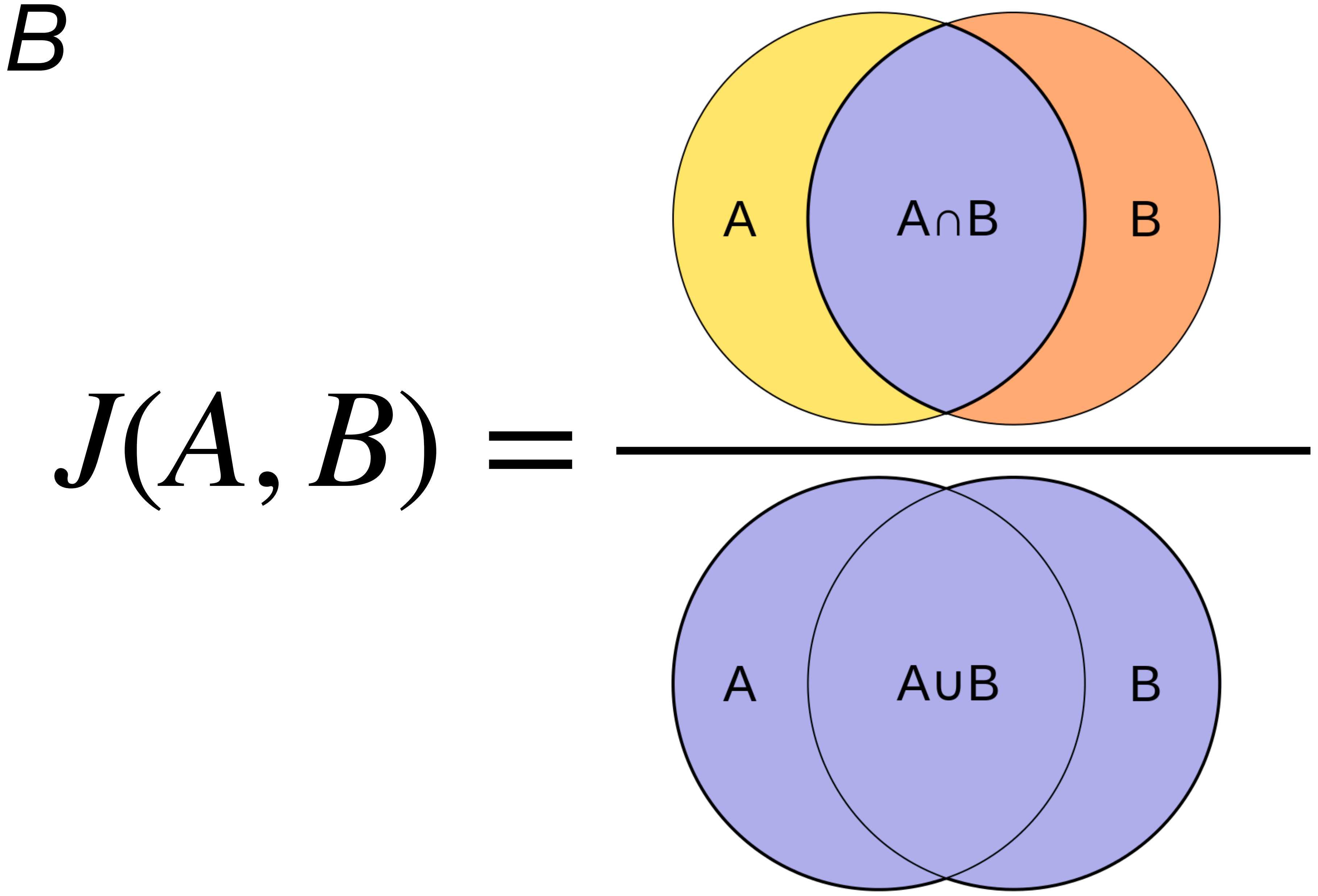


Images courtesy of Wikimedia Commons.

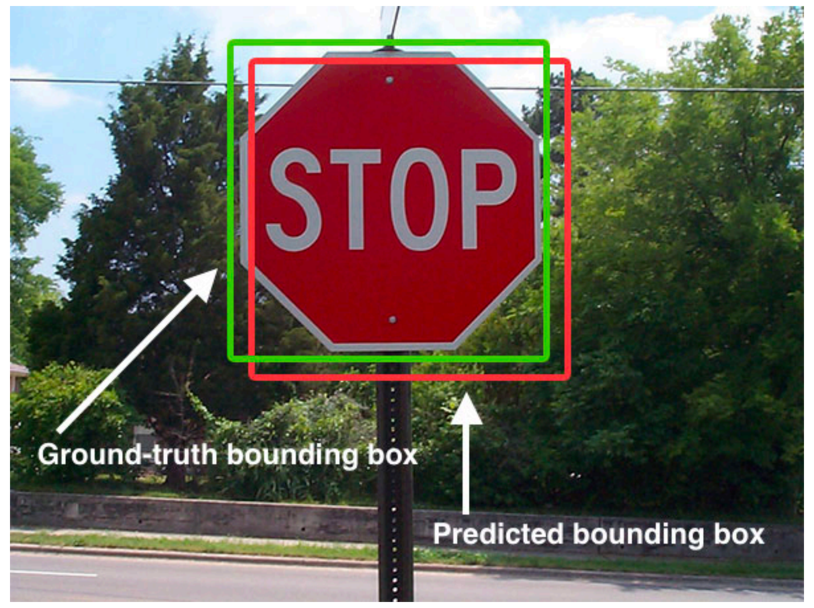
# Jaccard Similarity

Measures the similarity of two sets of items *A* and *B* as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$



Used also used in computer vision, sometimes called the "Intersection over Union" (IoU) metric

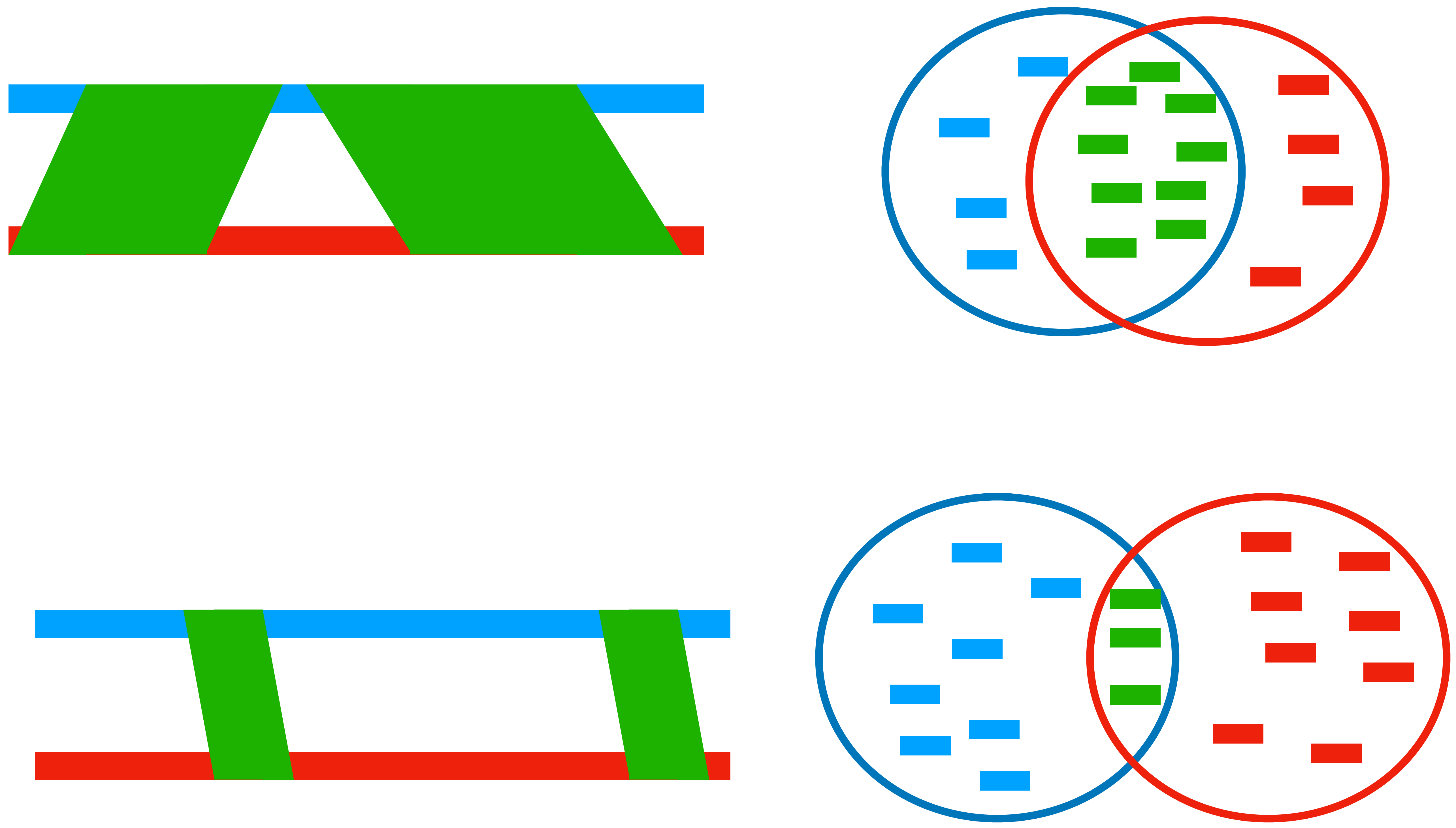


How would we use Jaccard for sequences?

Images courtesy of Wikimedia Commons.

# Jaccard Similarity

In sequence analysis we construct a sets of *k*-mers for each of the strings being compared



# Min-Hash Sketch

Calculating the union and intersection of a set of anything (in particular $k$-mers) can be time consuming (*O(n)* time)

Can we calculate it faster?

# Min-Hash Sketch

Calculating the union and intersection of a set of anything (in particular $k$-mers) can be time consuming (*O(n)* time)

Can we calculate it faster?

Consider the following scenario:

- given a hash function on *k*-mers $h: \Sigma^k \rightarrow \mathbb{Z}^+$
- and the sets of *k*-mers for two string *A* and *B,*
- What is the probability that $min_{c \in A}\{h(c)\} = min_{c \in B}\{h(c)\}$?

# Min-Hash Sketch

Calculating the union and intersection of a set of anything (in particular $k$-mers) can be time consuming (*O(n)* time)

Can we calculate it faster?

Consider the following scenario:

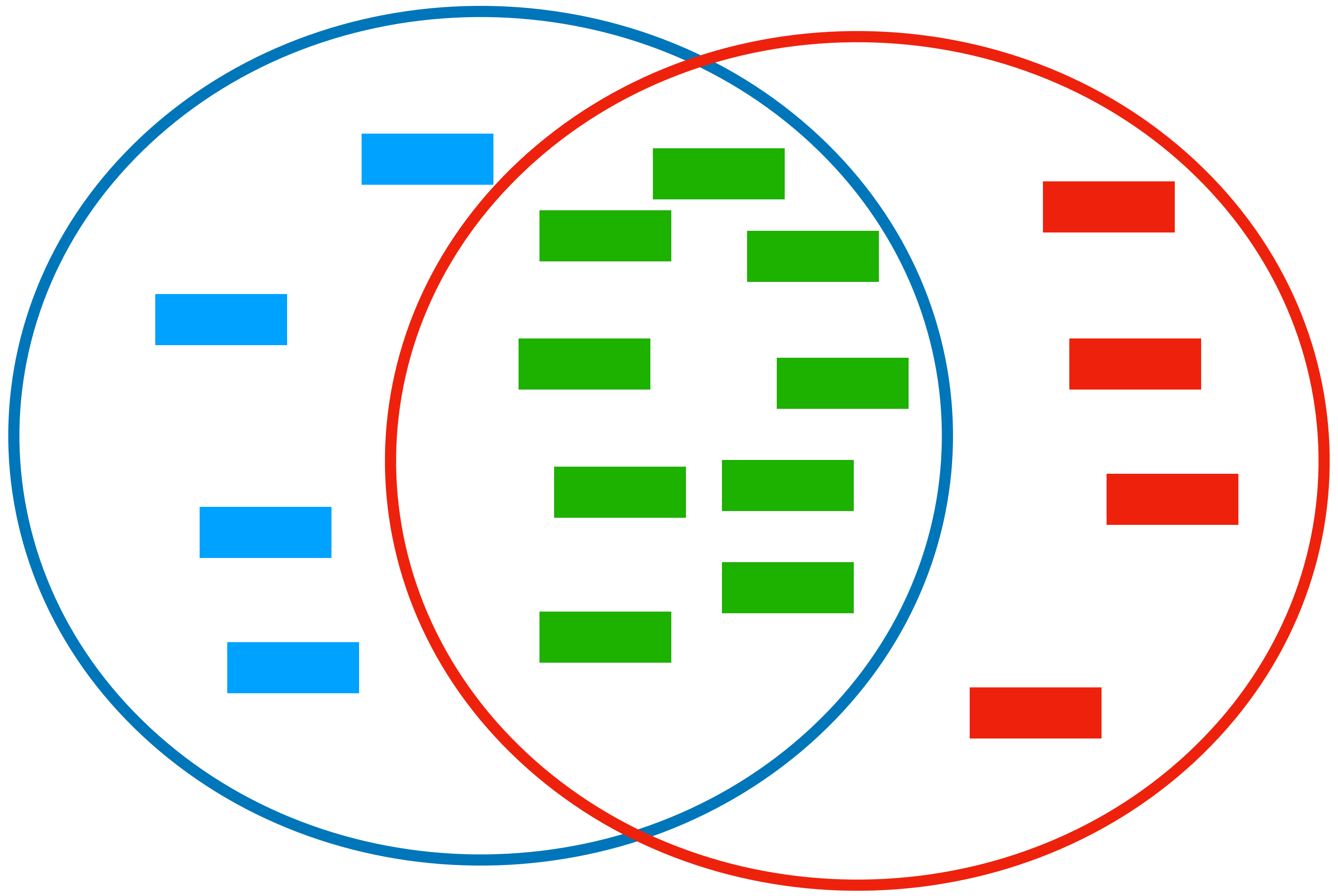
- given a hash function on *k*-mers $h: \Sigma^k \rightarrow \mathbf{Z}^+$
- and the sets of *k*-mers for two string *A* and *B,*
- What is the probability that $min_{c \in A}\{h(c)\} = min_{c \in B}\{h(c)\}$?

Turns out that

$$Pr_h\left[min_{c \in A}\{h(c)\} = min_{c \in B}\{h(c)\}\right] = J(A, B)$$

# Min-Hash Sketch

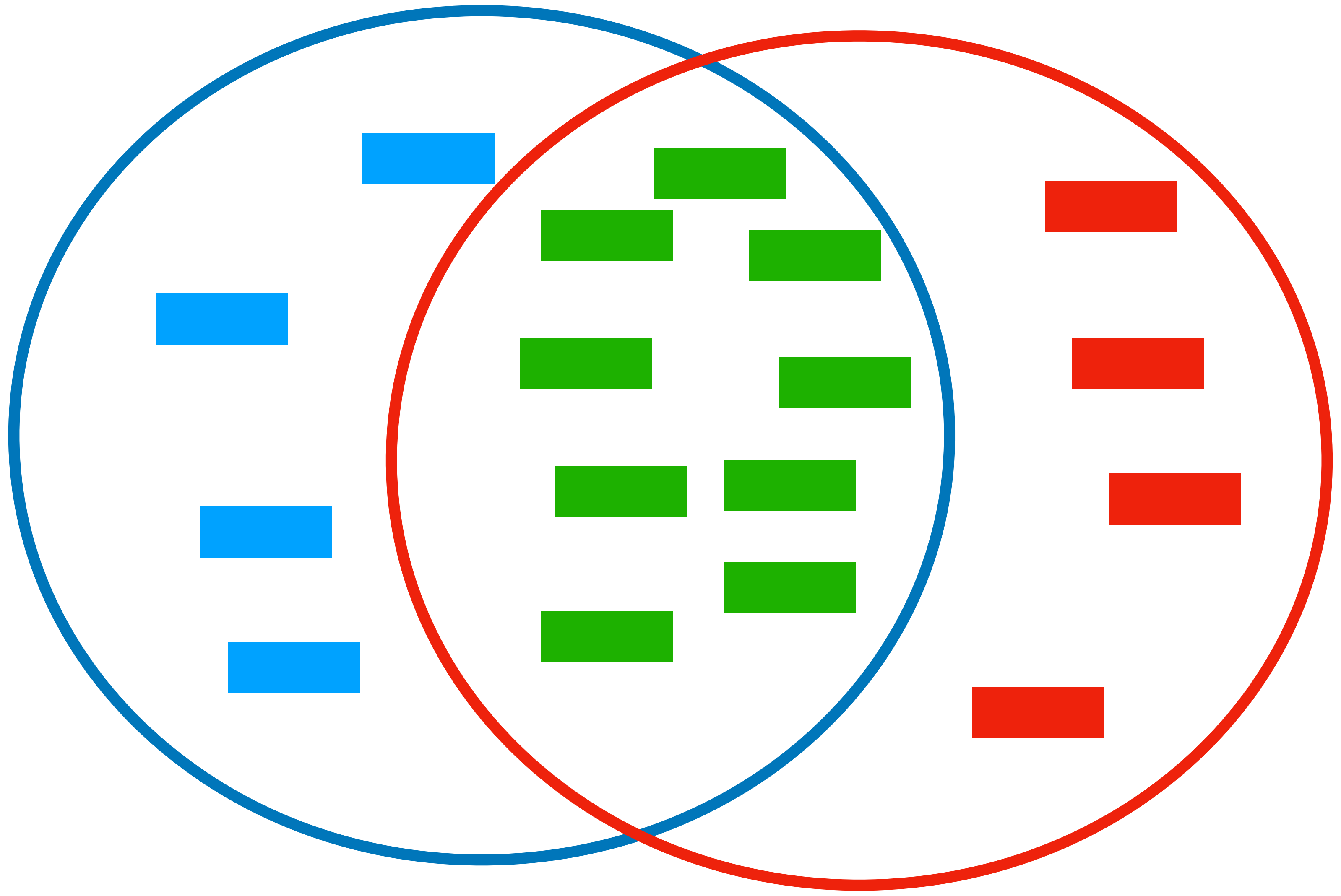
Why is $Pr_h\left[min_{c\in A}\{h(c)\} = min_{c\in B}\{h(c)\}\right] = J(A,B)$?



# Min-Hash Sketch

Why is $Pr_h\left[min_{c\in A}\{h(c)\} = min_{c\in B}\{h(c)\}\right] = J(A,B)$?

**Think of *h* as applying a randomized ordering on the *k*-mers.**

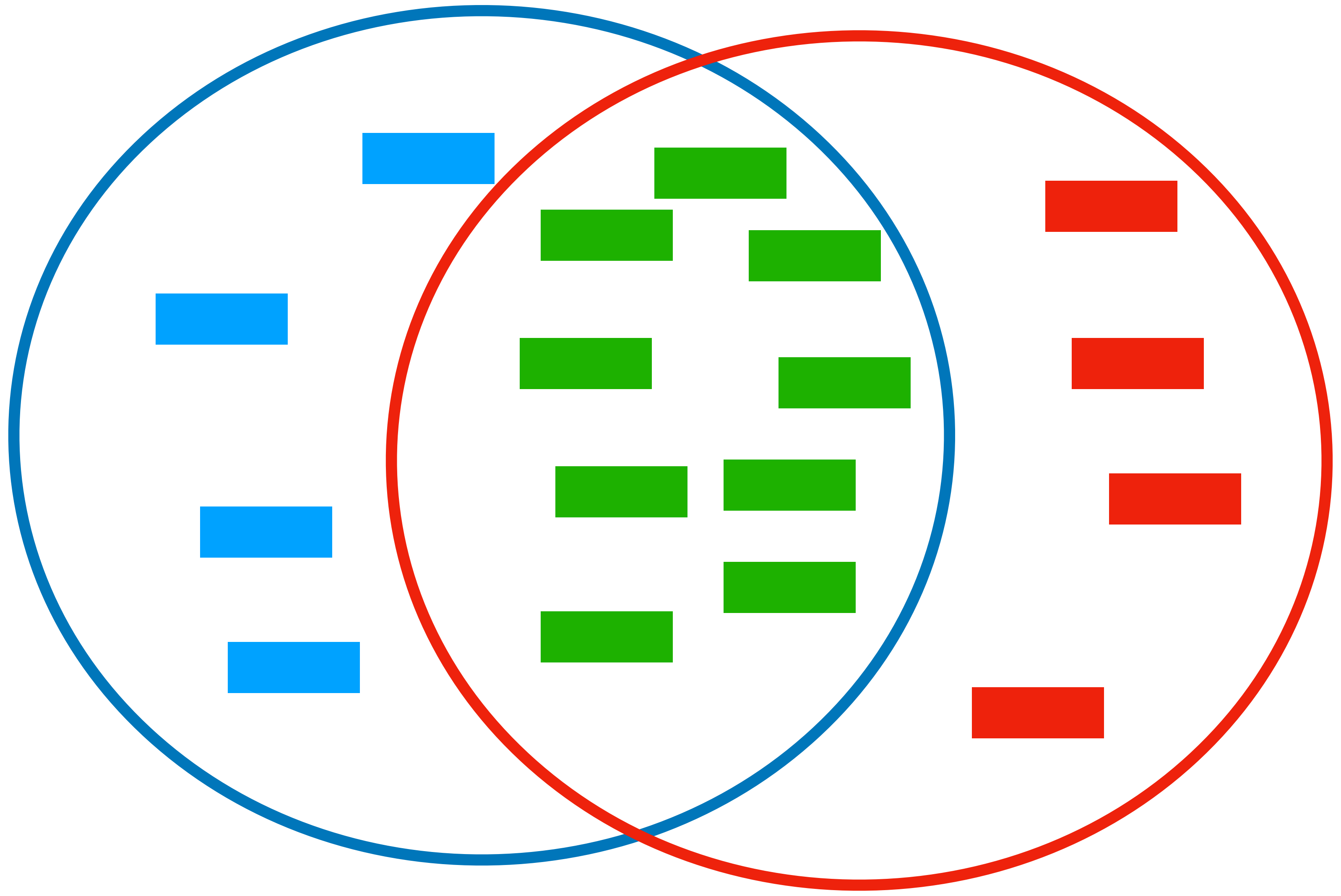


# Min-Hash Sketch

Why is $Pr_h \left[ min_{c \in A} \{h(c)\} = min_{c \in B} \{h(c)\} \right] = J(A, B)$?

**Think of *h* as applying a randomized ordering on the *k*-mers.**

**If the minimum *k*-mer from the union is in the intersection, it will be minimum for both *A* and *B.***



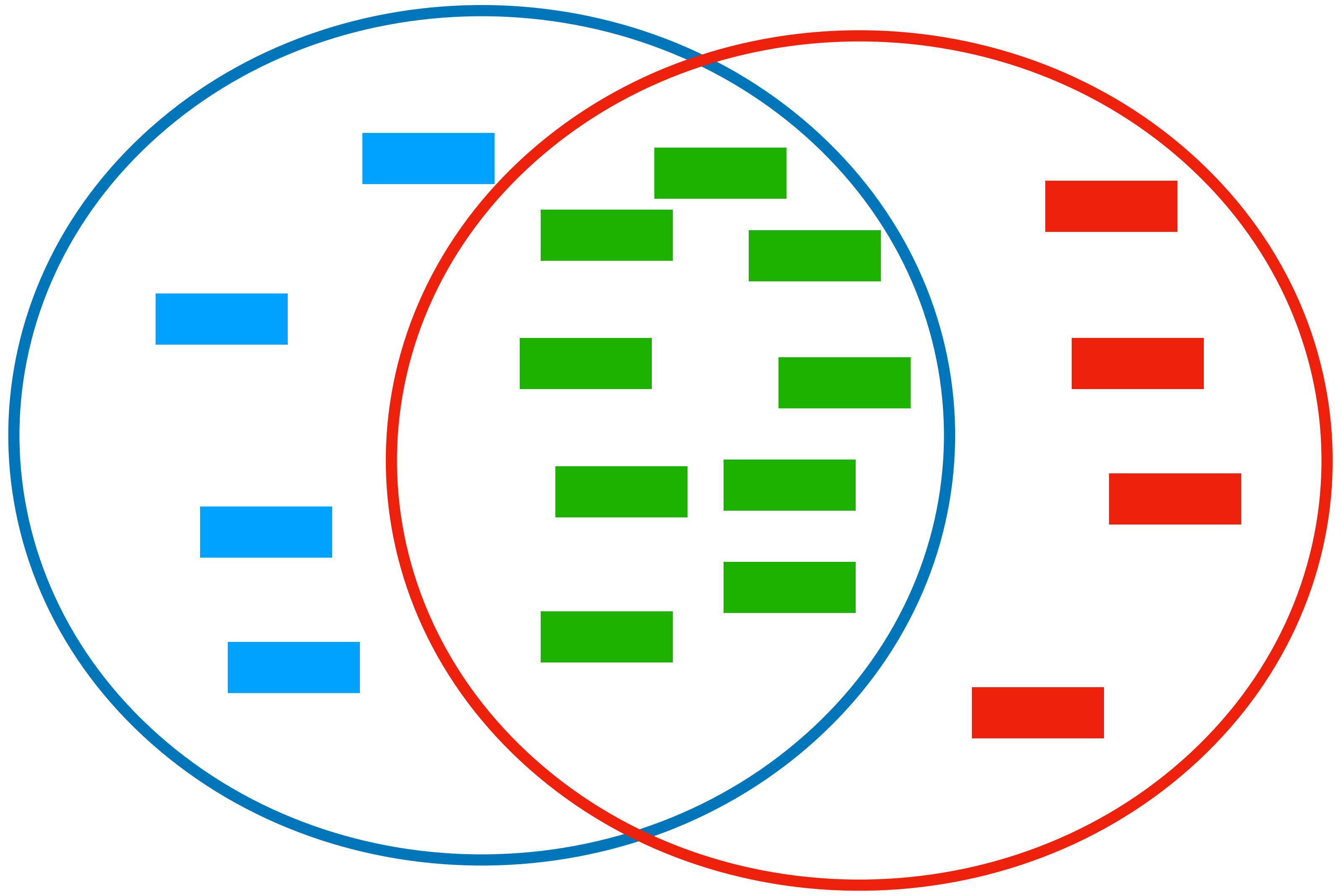
# Min-Hash Sketch

Why is $Pr_h\left[min_{c\in A}\{h(c)\} = min_{c\in B}\{h(c)\}\right] = J(A, B)$?

**Think of *h* as applying a randomized ordering on the *k*-mers.**

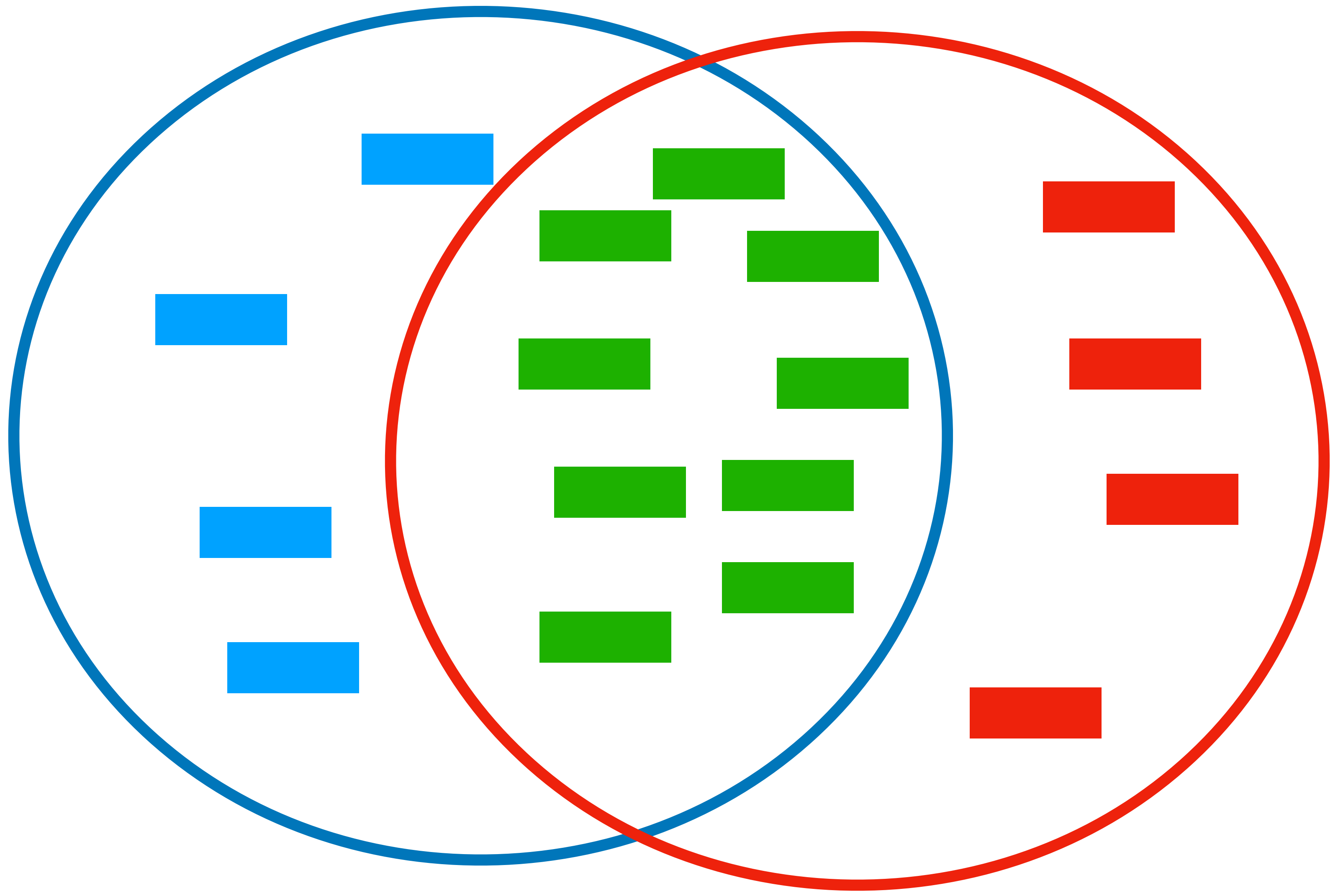
**If the minimum *k*-mer from the union is in the intersection, it will be minimum for both *A* and *B.***

**How many minimum *k*-mers from the union can we choose?**



# Min-Hash Sketch

Why is $Pr_h\left[min_{c\in A}\{h(c)\} = min_{c\in B}\{h(c)\}\right] = J(A,B)$?



**Think of *h* as applying a randomized ordering on the *k*-mers.**

**If the minimum *k*-mer from the union is in the intersection, it will be minimum for both *A* and *B.***

**How many minimum *k*-mers from the union can we choose?**

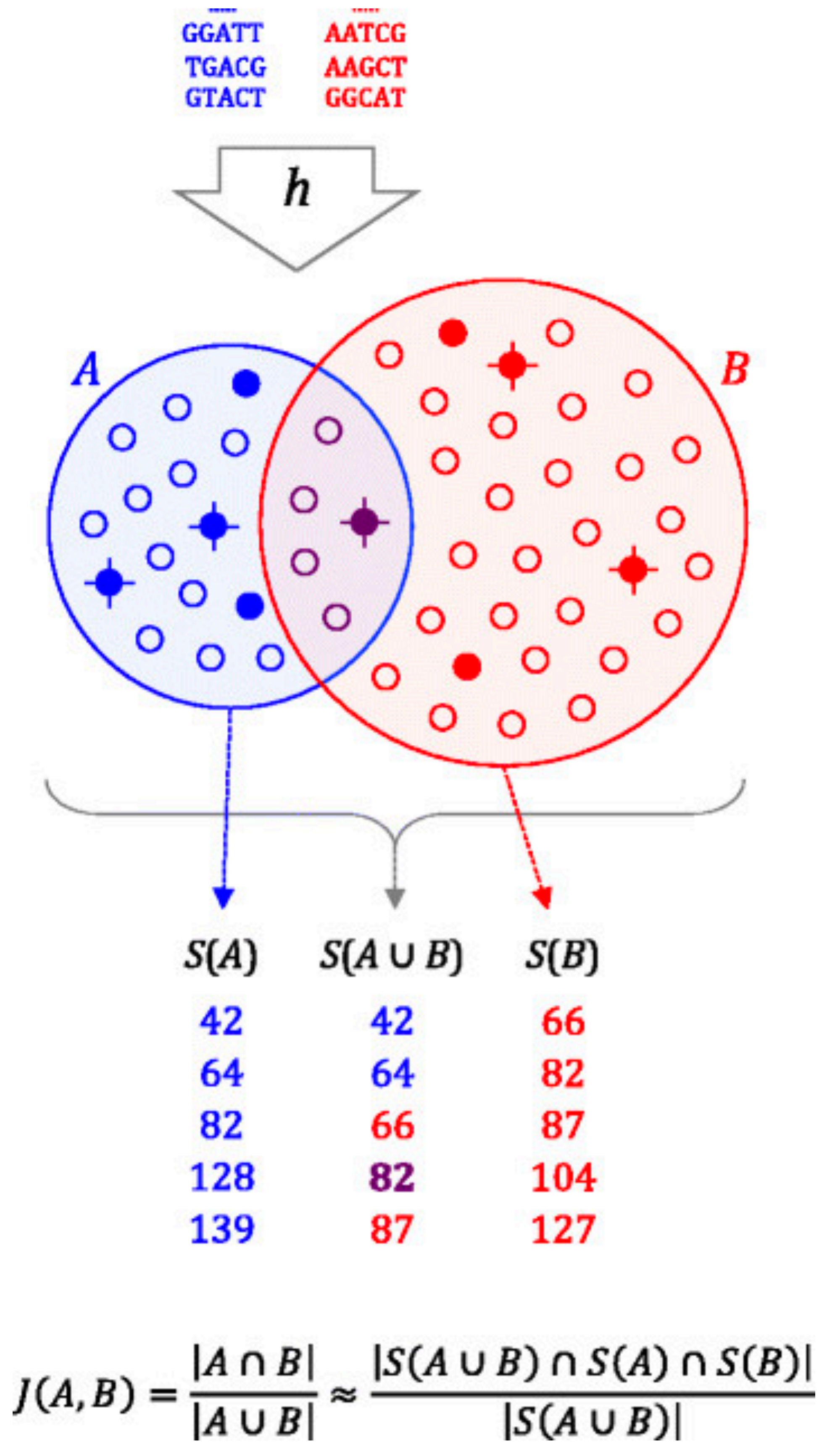
**What fraction of those are in the intersection?**

# Min Hash Sketch with 1 Hash

The idea is that you choose the minimum *n* elements according to the hash *h*, and compute jaccard on these subsets

This subset of *k*-mers is called a "sketch"

Sometimes called "MinHash bottom sketching"



| S(A) | S(A ∪ B) | S(B) |
|---|---|---|
| 42 | 42 | 66 |
| 64 | 64 | 82 |
| 82 | 66 | 87 |
| 128 | **82** | 104 |
| 139 | 87 | 127 |

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$$

Image credit: Ondov, et al. (2016) *Mash: Fast genome and metagenome distance estimation using MinHash.* **Genome Biology.**

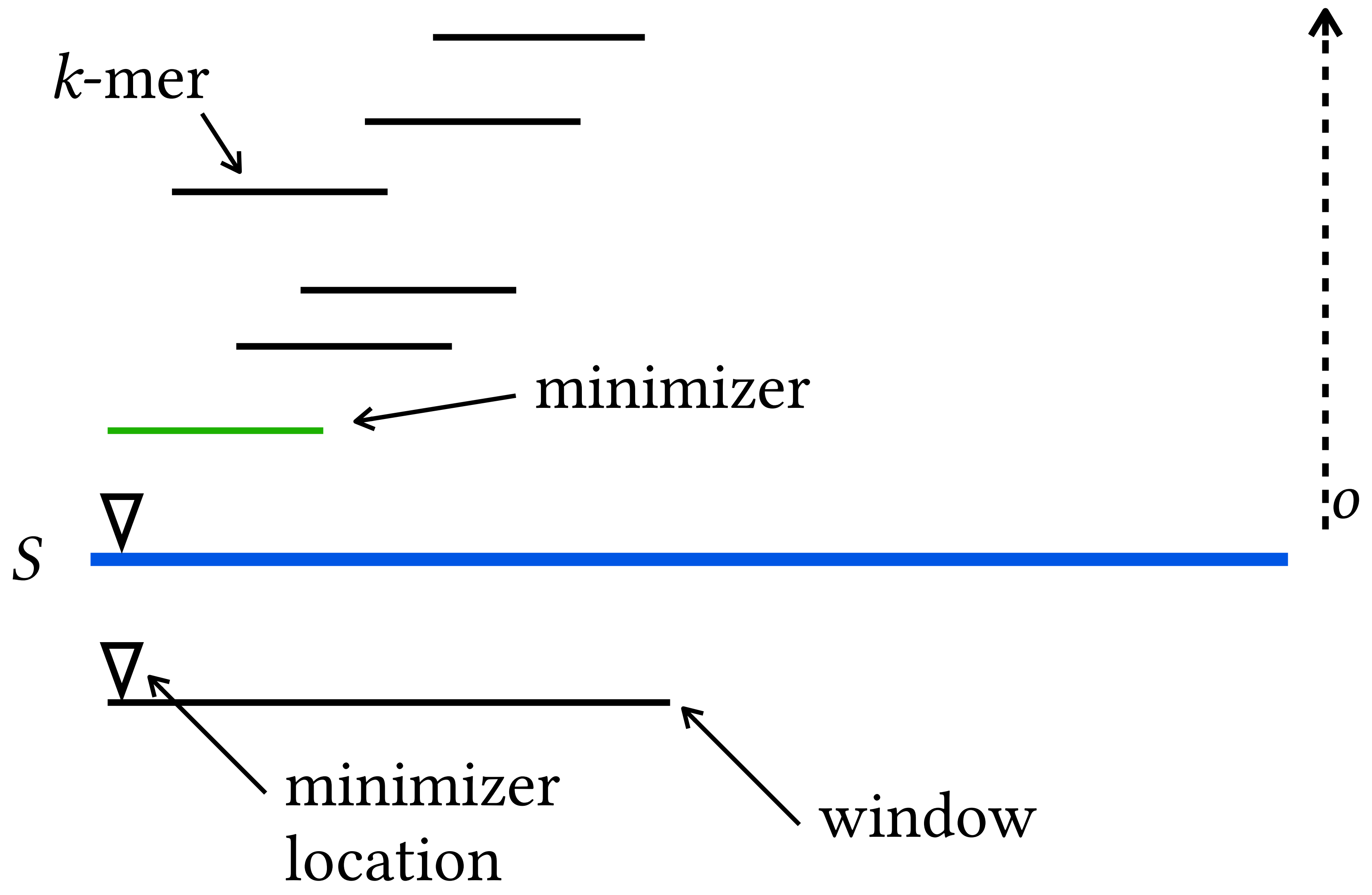
# Minimizer Schemes

For a windows of $w$ consecutive $k$-mers from a sequence $S$, a minimizer scheme selects the minimum according to an ordering $o$ as a representative

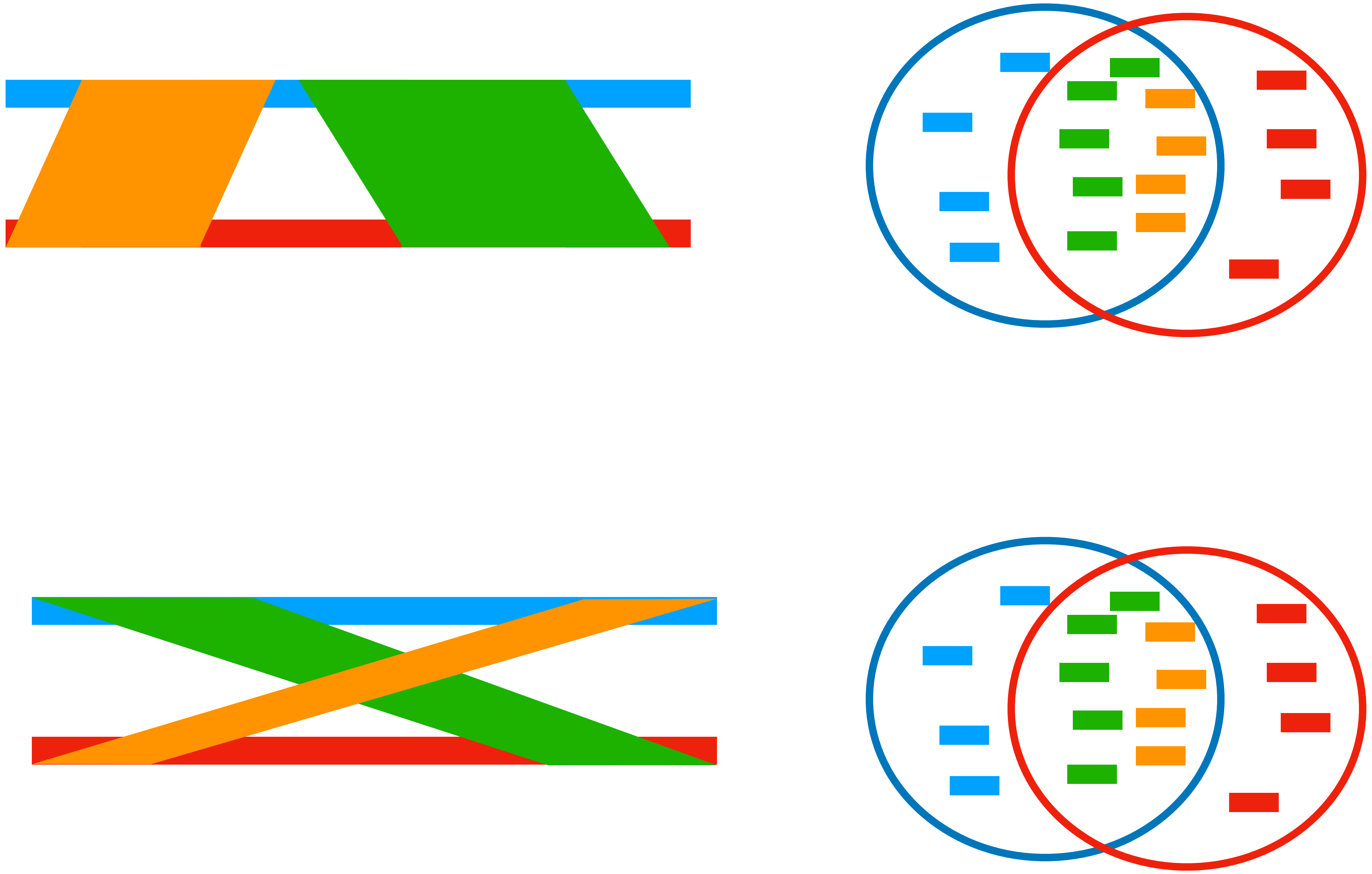
Minimizer schemes have two special properties:

- two sequences with a long exact match must select the same $k$-mers
- there are no large gap between selected $k$-mers

Use in $k$-mer counting, *de Brujin* graph construction, data structure sparsification, etc.



# Problems with Jaccard



# Problem Formulation

Given:

- a read *A*,
- a maximum per-base error rate, $\varepsilon_{max}$, and
- a reference genome, *B.*

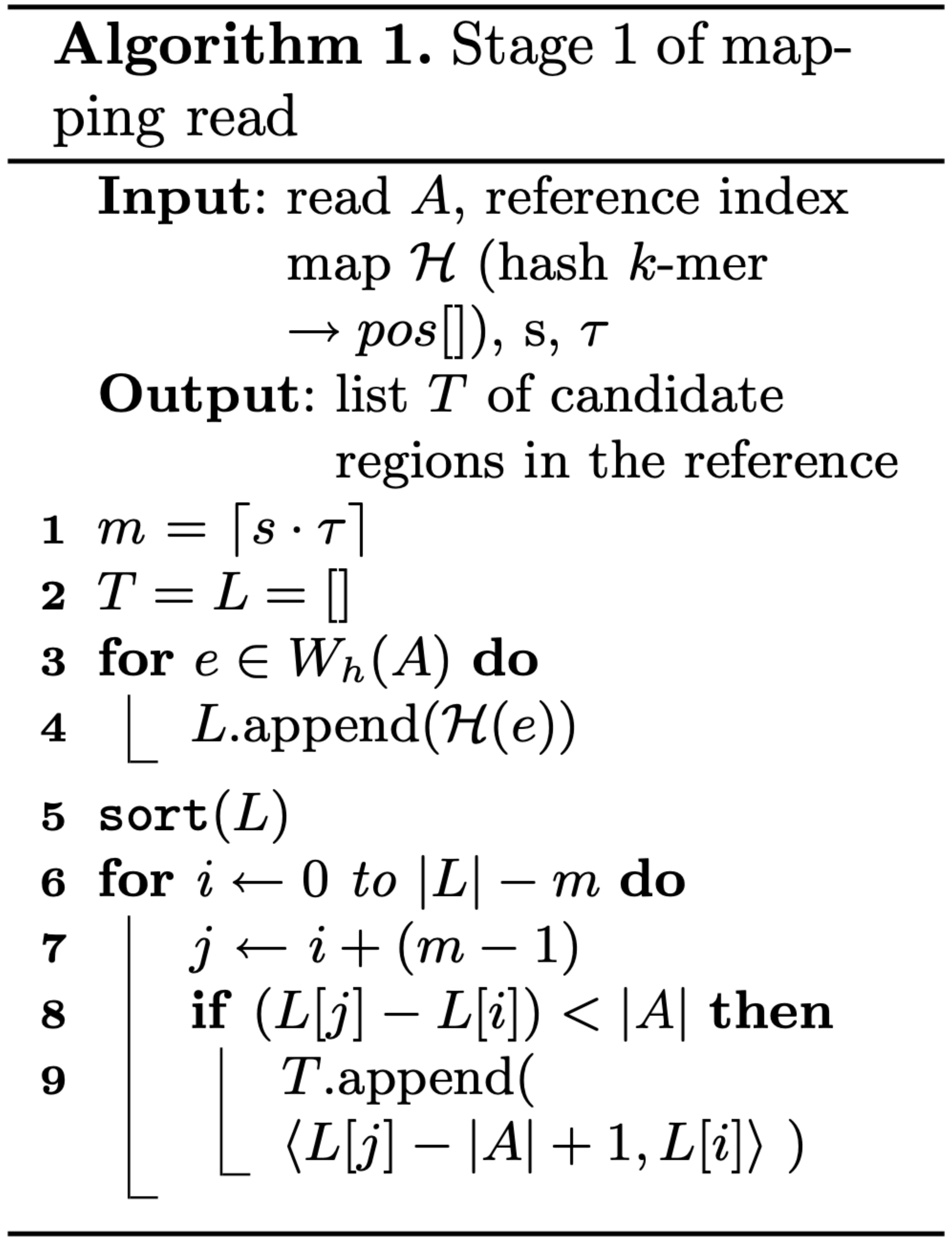
Goal:

- ~~identify target positions in *B* where *A* will map with $\leq \varepsilon_{max}|A|$ errors~~
- identify target positions $B_i$ where:

$$J(A, B_i) \geq \mathscr{G}(\varepsilon_{max}, k) - \delta$$

$\mathbb{E}\left(J\left(A, B_i\right)\right) \geq \mathscr{G}(\varepsilon_{max}, k)$ but only in expectation, so $\delta$ = (90% confidence interval) is subtracted to account for variance in the estimate

# Stage 1



**Algorithm 1.** Stage 1 of mapping read

```
Input: read $A$, reference index map $\mathcal{H}$ (hash $k$-mer $\rightarrow pos[]$), s, $\tau$
Output: list $T$ of candidate regions in the reference
$m = \lceil s \cdot \tau \rceil$
$T = L = []$
for $e \in W_h(A)$ do
    $L$.append($\mathcal{H}(e)$)
sort($L$)
for $i \leftarrow 0$ to $|L| - m$ do
    $j \leftarrow i + (m - 1)$
    if $(L[j] - L[i]) < |A|$ then
        $T$.append( $\langle L[j] - |A| + 1, L[i] \rangle$ )
```

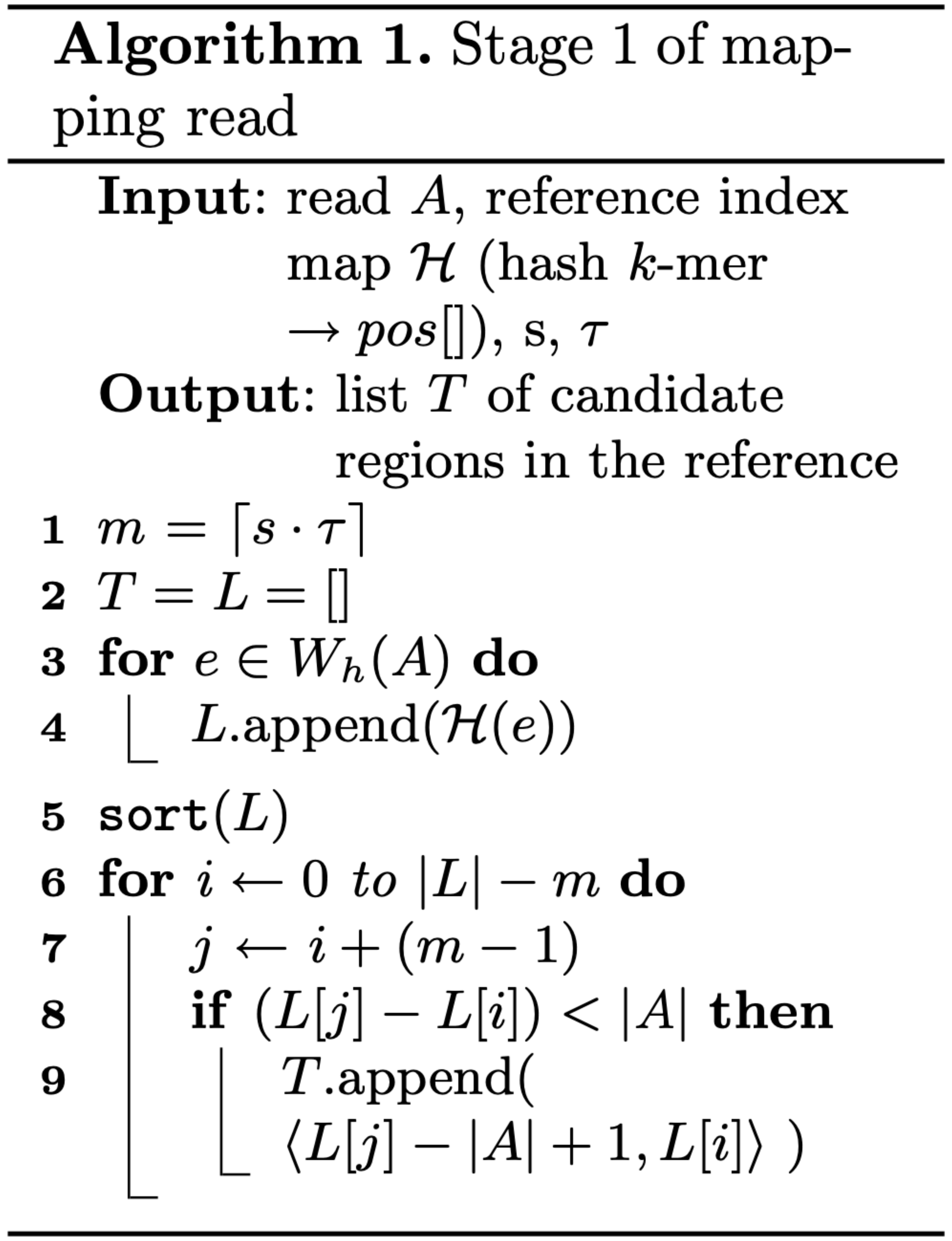
Find all ranges in *B* that *could* be a match to *A*

- they have ≥ $s\tau = m$ number of matching *k*-mers

This is actually performed somewhat in reverse

- first find all matching minimizers
- sort them by location
- in each range of *m* matches
  - ask if they are they condensed enough

# Stage 1



**Algorithm 1.** Stage 1 of mapping read

```
Input: read $A$, reference index map $\mathcal{H}$ (hash $k$-mer $\to pos[]$), s, $\tau$
Output: list $T$ of candidate regions in the reference
$m = \lceil s \cdot \tau \rceil$
$T = L = []$
for $e \in W_h(A)$ do
    $L$.append($\mathcal{H}(e)$)
sort($L$)
for $i \leftarrow 0$ to $|L| - m$ do
    $j \leftarrow i + (m - 1)$
    if $(L[j] - L[i]) < |A|$ then
        $T$.append( $\langle L[j] - |A| + 1, L[i] \rangle$ )
```

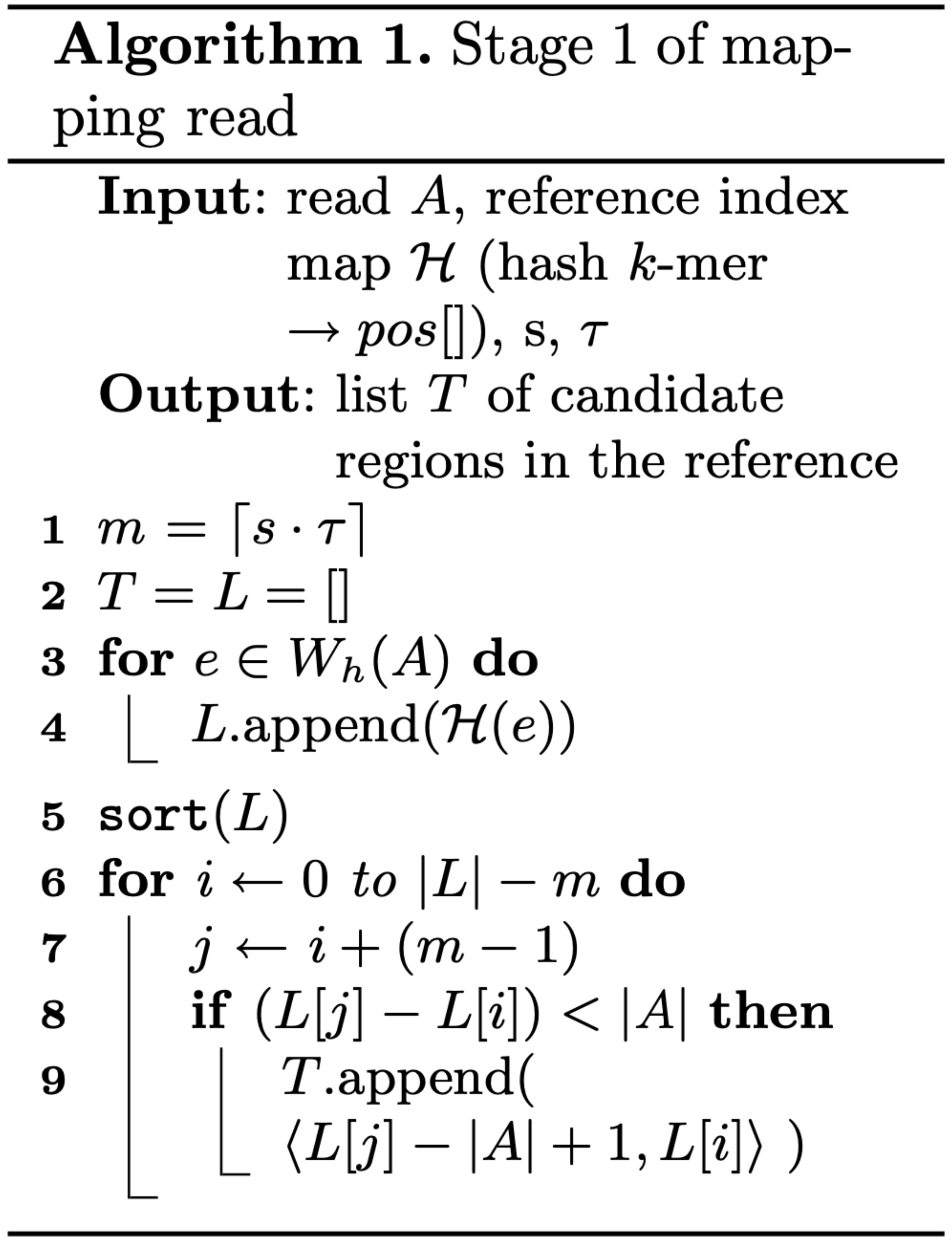
Find all ranges in *B* that *could* be a match to *A*

- they have ≥ $s\tau = m$ number of matching *k*-mers

This is actually performed somewhat in reverse

- first find all matching minimizers
- sort them by location
- in each range of *m* matches
  - ask if they are they condensed enough

# Stage 1



**Algorithm 1.** Stage 1 of mapping read

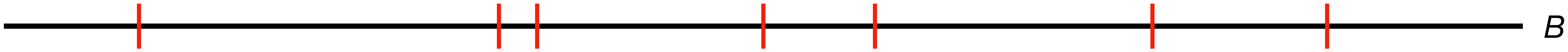
```
Input: read $A$, reference index map $\mathcal{H}$ (hash $k$-mer $\to pos[]$), s, $\tau$
Output: list $T$ of candidate regions in the reference
$m = \lceil s \cdot \tau \rceil$
$T = L = []$
for $e \in W_h(A)$ do
    $L$.append($\mathcal{H}(e)$)
sort($L$)
for $i \leftarrow 0$ to $|L| - m$ do
    $j \leftarrow i + (m - 1)$
    if $(L[j] - L[i]) < |A|$ then
        $T$.append( $\langle L[j] - |A| + 1, L[i] \rangle$ )
```

Find all ranges in *B* that *could* be a match to *A*

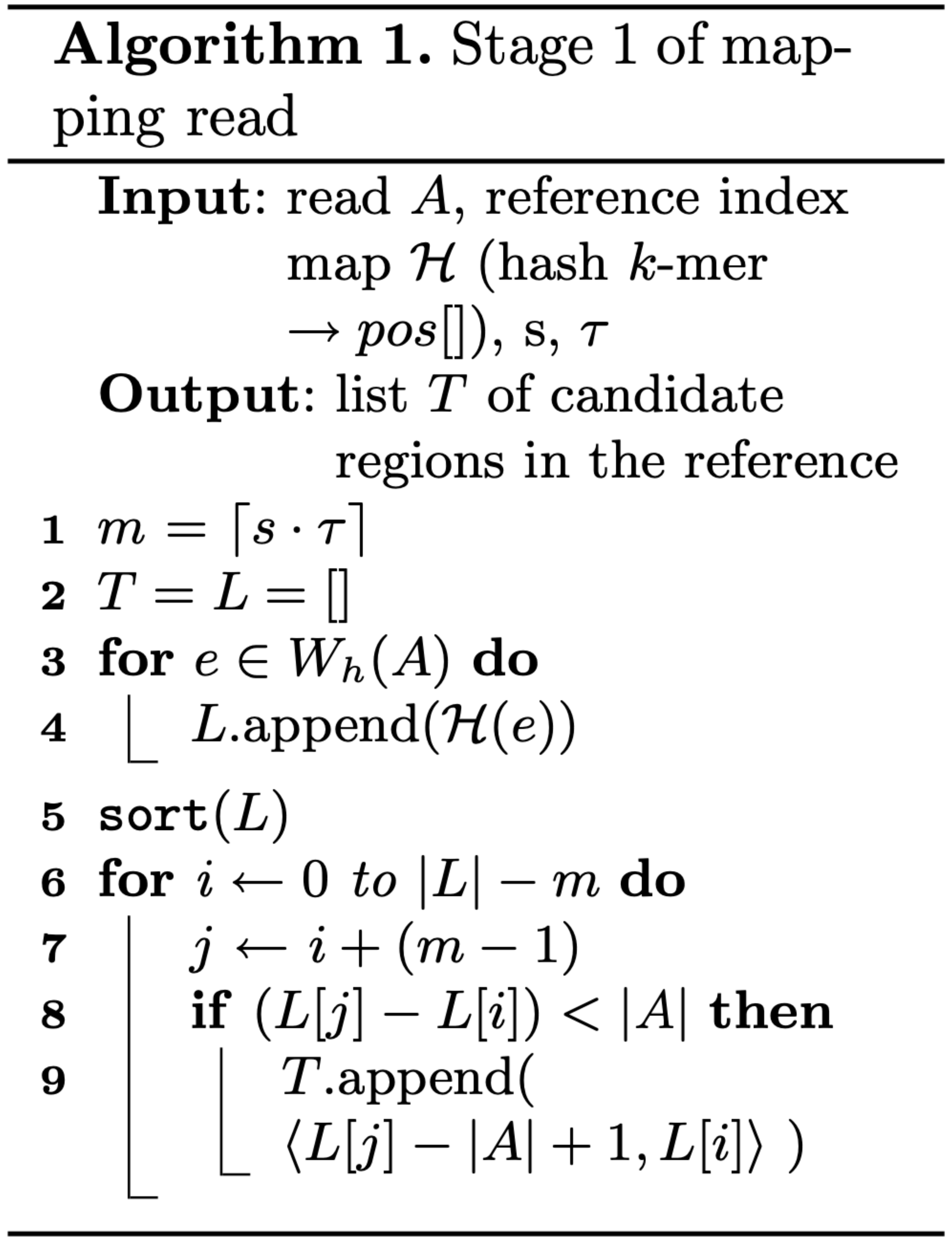
- they have ≥ $s\tau = m$ number of matching *k*-mers

This is actually performed somewhat in reverse

- first find all matching minimizers
- sort them by location
- in each range of *m* matches
  - ask if they are they condensed enough



# Stage 1



**Algorithm 1.** Stage 1 of mapping read

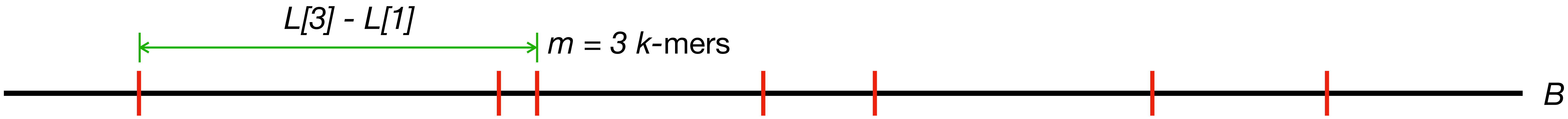
```
Input: read $A$, reference index map $\mathcal{H}$ (hash $k$-mer $\to pos[]$), s, $\tau$
Output: list $T$ of candidate regions in the reference
$m = \lceil s \cdot \tau \rceil$
$T = L = []$
for $e \in W_h(A)$ do
    $L$.append($\mathcal{H}(e)$)
sort($L$)
for $i \leftarrow 0$ to $|L| - m$ do
    $j \leftarrow i + (m - 1)$
    if $(L[j] - L[i]) < |A|$ then
        $T$.append( $\langle L[j] - |A| + 1, L[i] \rangle$ )
```

Find all ranges in *B* that *could* be a match to *A*

- they have ≥ $s\tau = m$ number of matching *k*-mers

This is actually performed somewhat in reverse

- first find all matching minimizers
- sort them by location
- in each range of *m* matches
    - ask if they are they condensed enough



# Stage 2

**Algorithm 2:** Stage 2 of mapping a read

```
Input: index $\mathcal{M}$, stage 1 output $T$, $s$, $\tau$
Output: $\mathcal{P}$
$\mathcal{L}_0 = \{\}$;
$\mathcal{L}_0$.insert($W_h(A)$);
for $\langle x, y \rangle \in T$ do
    $i \leftarrow x$;
    $j \leftarrow x + |A|$;
    $\mathcal{L} \leftarrow \mathcal{L}_0$;
    $\mathcal{L}$.insert(getMinimizers($i, j$));
    $\mathcal{J}$ = solveJaccard($\mathcal{L}$);
    if $\mathcal{J} \geq \tau$ then
        $\mathcal{P}$.append($\langle i, \mathcal{J} \rangle$);
    while $i \leq y$ do
        $\mathcal{L}$.delete(getMinimizers(i,i+1));
        $\mathcal{L}$.insert(getMinimizers(j,j+1));
        $\mathcal{J}$ = solveJaccard($\mathcal{L}$);
        if $\mathcal{J} \geq \tau$ then
            $\mathcal{P}$.append($\langle i, \mathcal{J} \rangle$);
        $i$++;
        $j$++;
Function getMinimizers($p$,$q$):
    return $\{h : \langle h, pos \rangle \in W(B), p \leq pos \leq q\}$;
Function solveJaccard($\mathcal{L}$):
    return $\frac{\sum_{0 \leq k \leq s-1} \mathcal{L}[k]}{s}$;
```

For every $B_i$ in all potential places identified in stage 1

- estimate the jaccard using the winnowed sketch
- retain it as a match if its larger than $\tau$

# CANU

Follows one of the same basic procedure we saw for short read assembly:

- calculate the **overlaps** between reads
- decide on a **layout** for the reads
- construct contigs using the **consensus** sequences

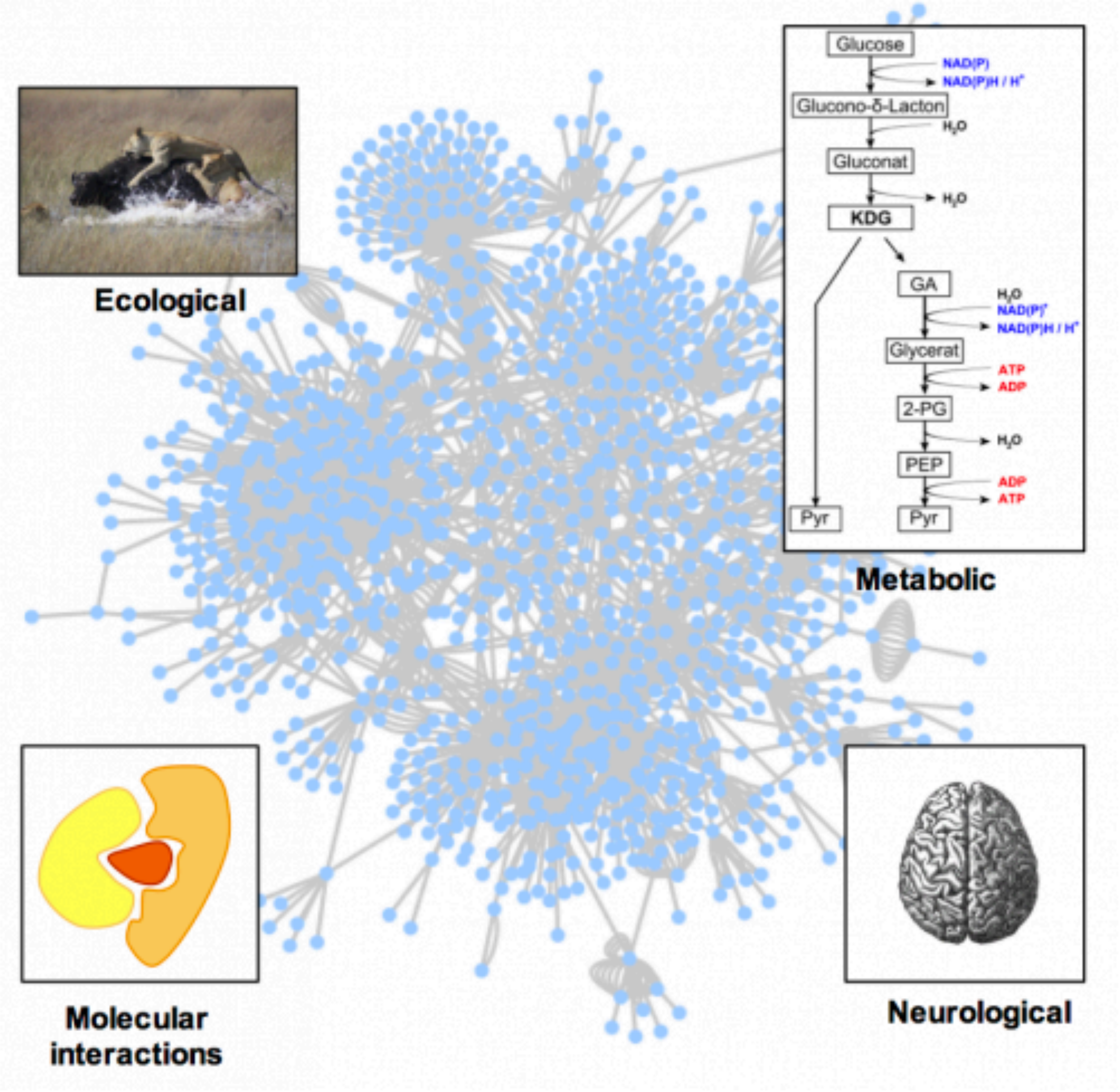
Uses an adaptation of *MHAP* for overlaps which is an extension of MinHash

- frequent *k*-mers like those in loops can sometimes interfere with overlap prediction
- they use *tf-idf* (term frequency–inverse document frequency) weights to bias the hashes used

# Networks in Biology

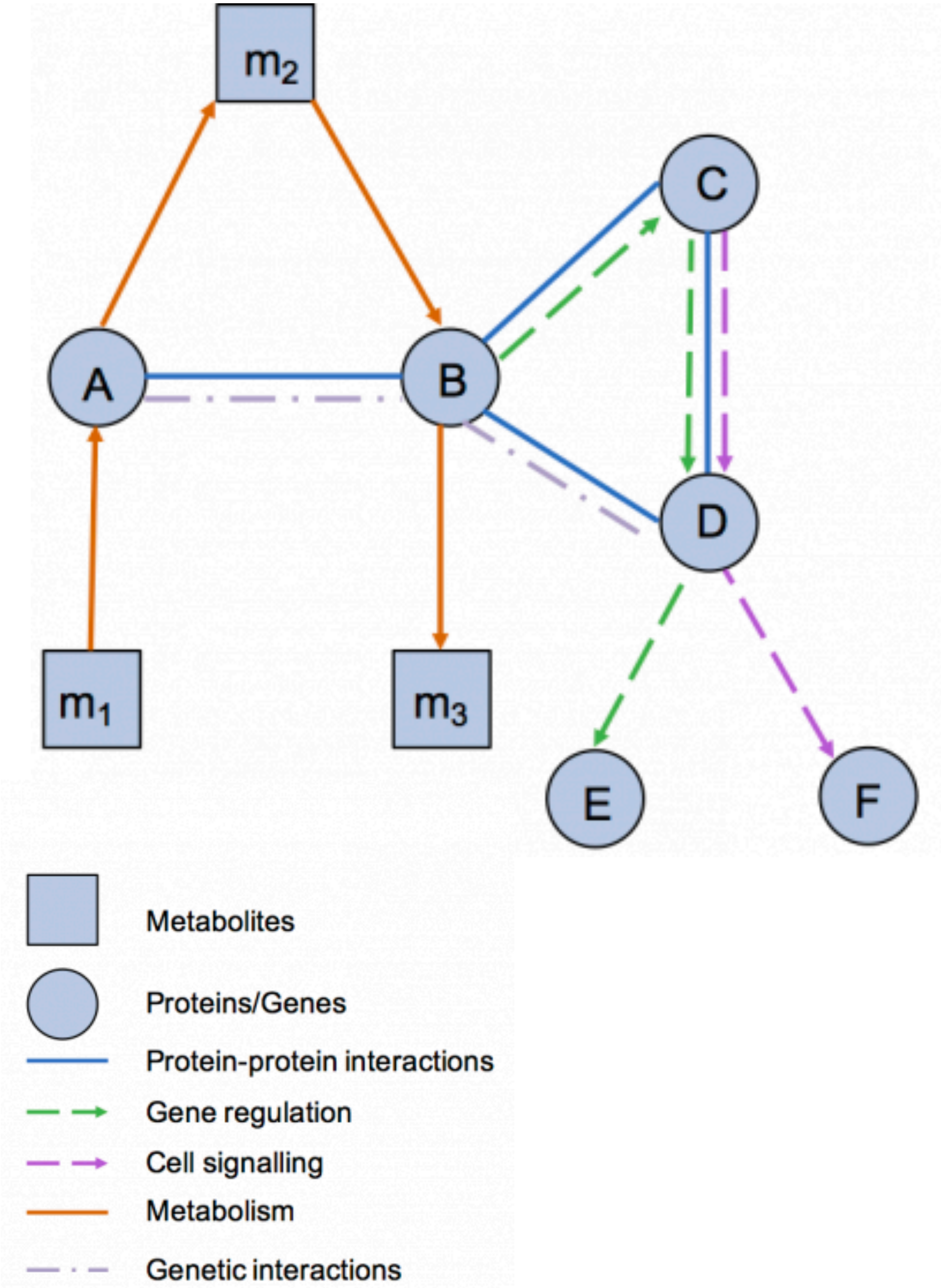
So far we have only talked about sequences

- Many *interactions* in biology are not captured in sequences
- We use graph theory to make biological conclusions



# Combined Networks

The meaning of the nodes and edges used in a network representation depends on the type of data used to build the network and this should be taken into account when analysing it.

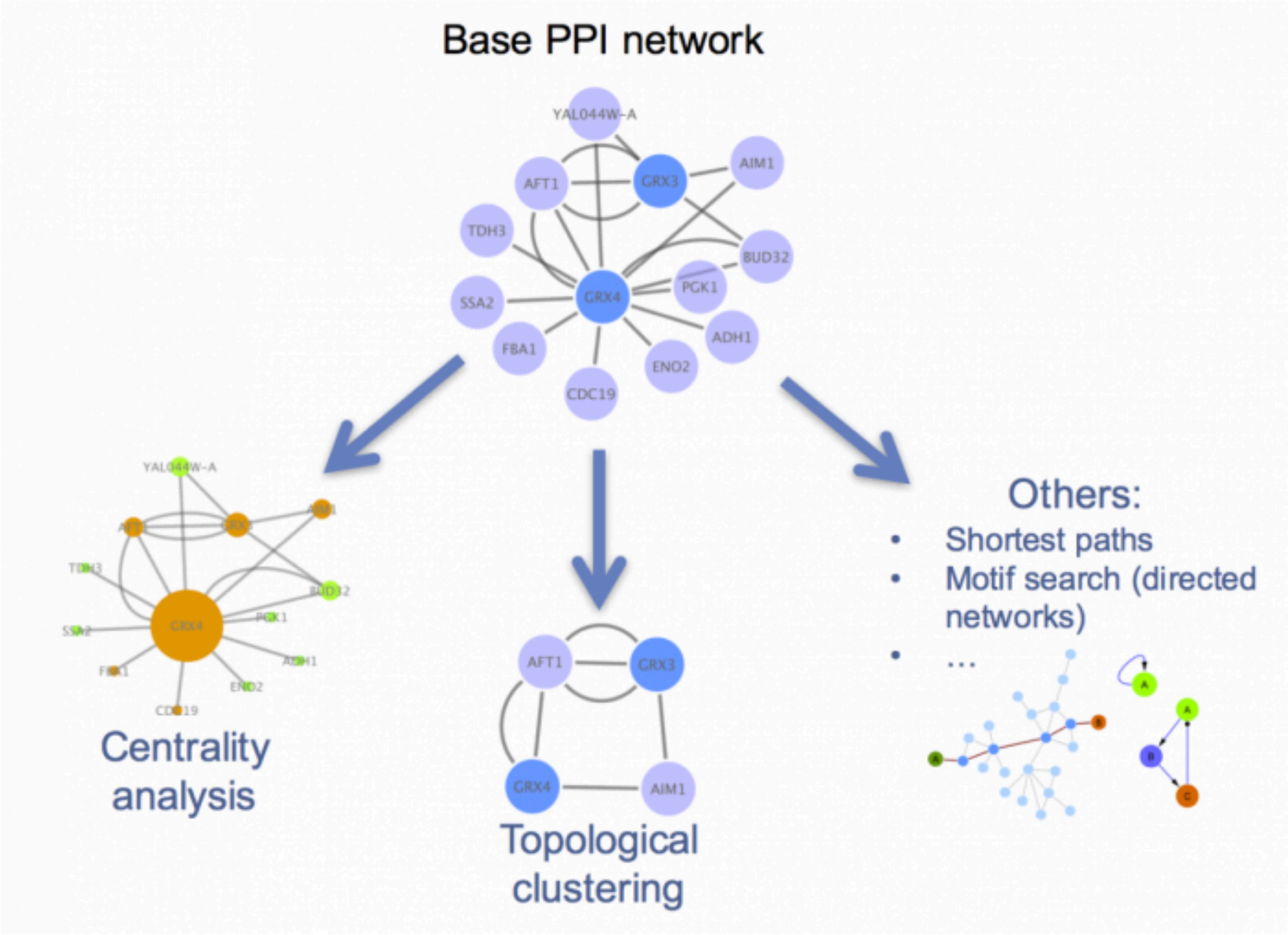


# Topology Analysis

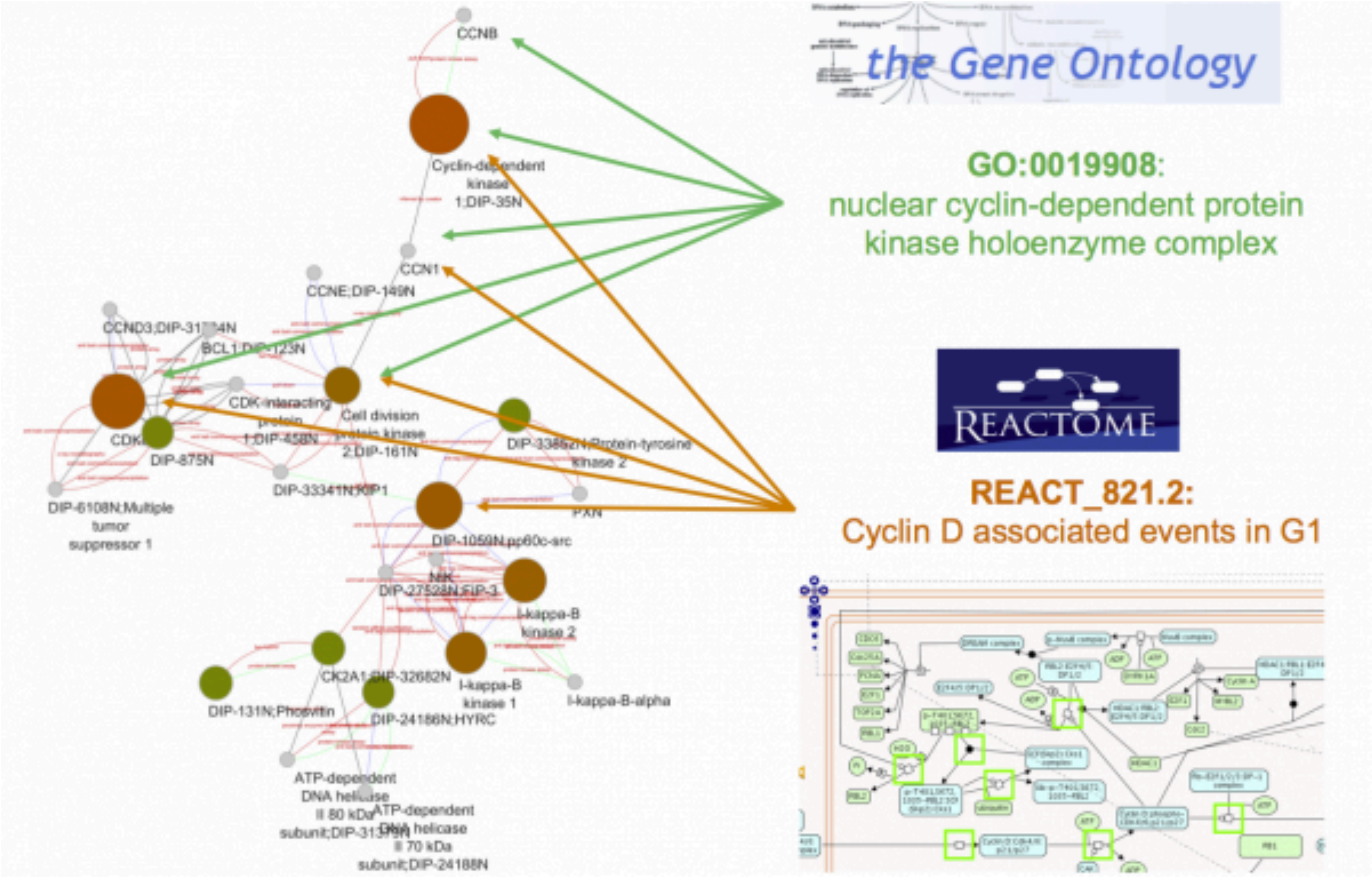
Analyzing the topological features of a network is a useful way of identifying relevant participants and substructures that may be of biological significance.

Some methods

- centrality analysis
- topological clustering
- search for shortest paths
- motifs that are more often applied to networks with directionality



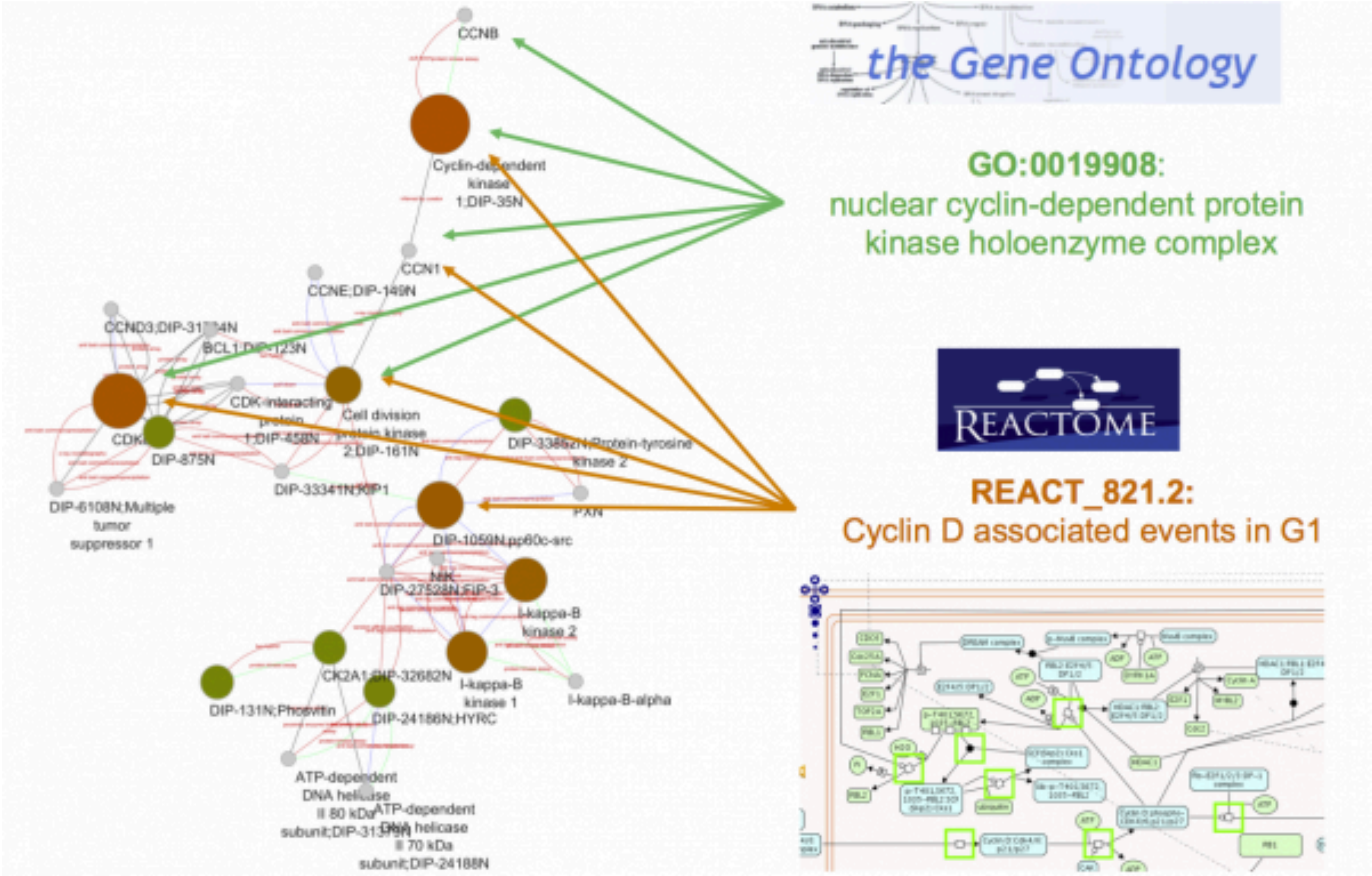
# Annotation enrichment analysis



# Annotation enrichment analysis

Annotation enrichment analysis uses gene/protein annotations to infer which annotations are over-represented in a list of genes/proteins taken from a network.

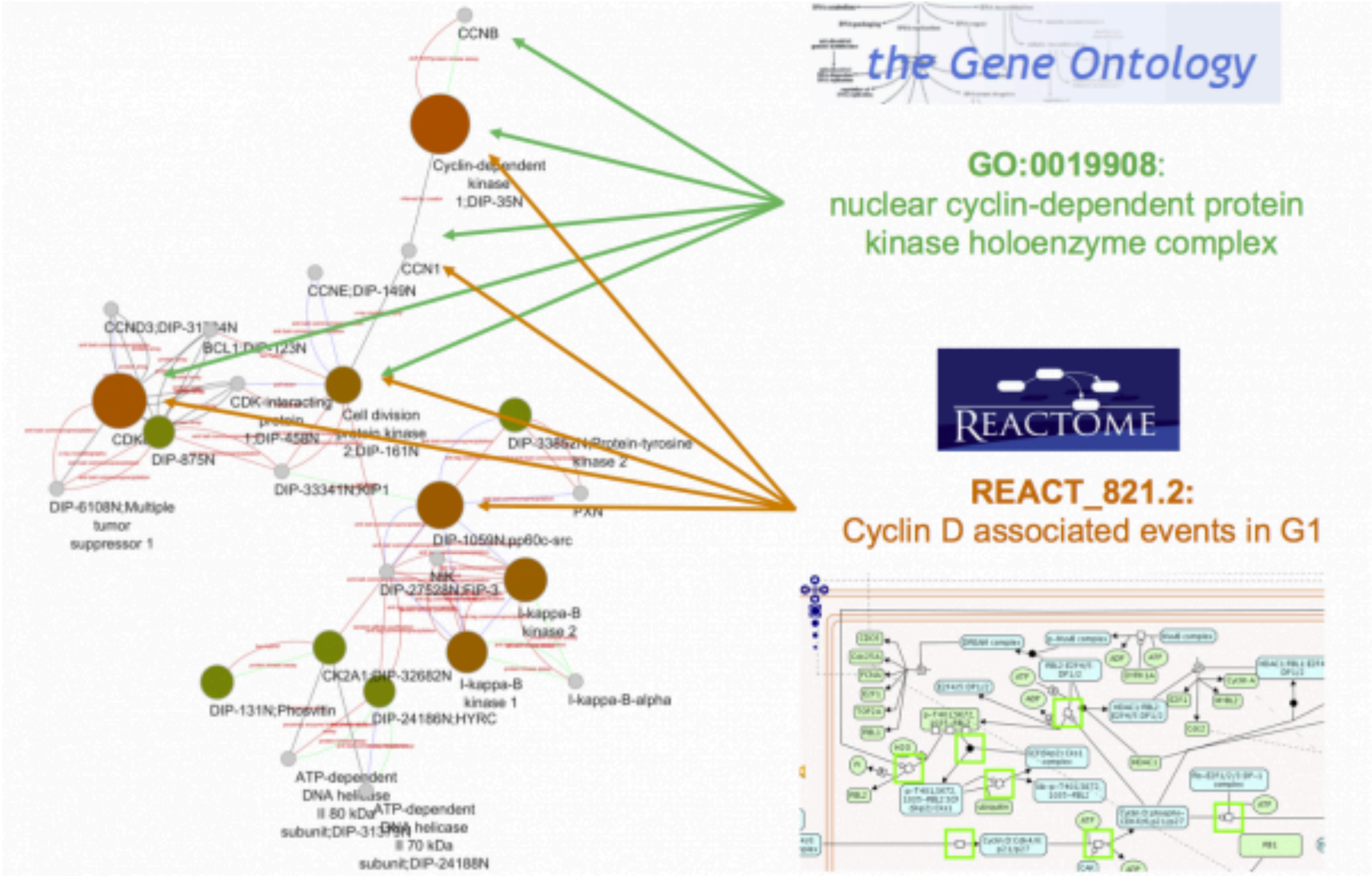
- Annotation tools perform statistical test tries to that answer:
  - *When sampling X proteins (test set) out of N proteins (reference set; graph or annotation), what is the probability that x, or more, of these proteins belong to a functional category C shared by n of the N proteins in the reference set.*
- The result of this test provides us with a list of terms that describe the list/network, or rather a part of it, as a whole.



# Annotation enrichment analysis

Annotation enrichment analysis uses gene/protein annotations to infer which annotations are over-represented in a list of genes/proteins taken from a network.

- Annotation tools perform statistical test tries to that answer:
  - *When sampling X proteins (test set) out of N proteins (reference set; graph or annotation), what is the probability that x, or more, of these proteins belong to a functional category C shared by n of the N proteins in the reference set.*
- The result of this test provides us with a list of terms that describe the list/network, or rather a part of it, as a whole.



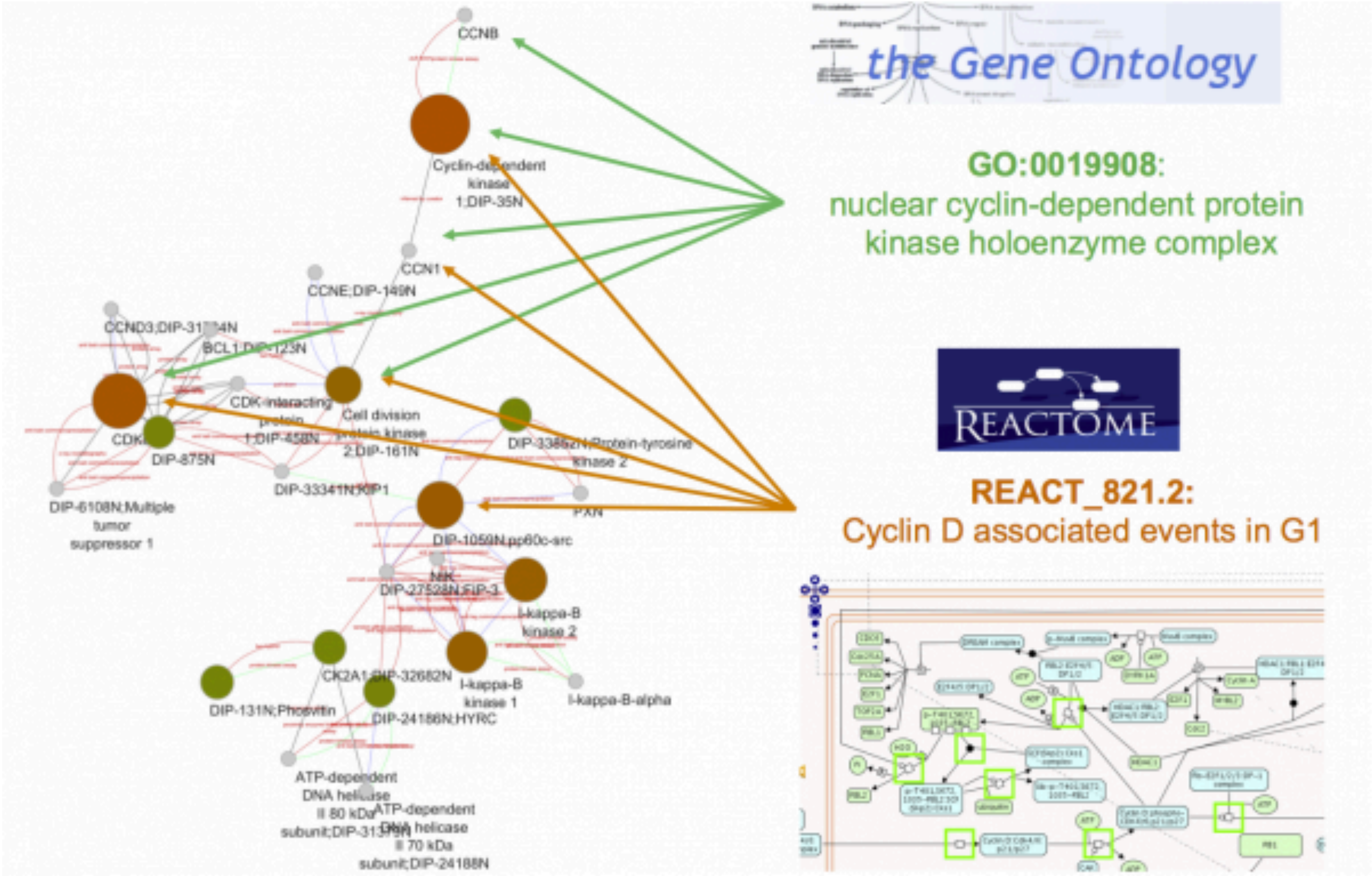
This analysis is most frequently performed using GO annotation as a reference.

- This is a widely used technique that helps characterize the network as a whole or sub-sets of it, such as inter-connected communities found through topological clustering analysis.

# Annotation enrichment analysis

Annotation enrichment analysis uses gene/protein annotations to infer which annotations are over-represented in a list of genes/proteins taken from a network.

- Annotation tools perform statistical test tries to that answer:
  - *When sampling X proteins (test set) out of N proteins (reference set; graph or annotation), what is the probability that x, or more, of these proteins belong to a functional category C shared by n of the N proteins in the reference set.*
- The result of this test provides us with a list of terms that describe the list/network, or rather a part of it, as a whole.



This analysis is most frequently performed using GO annotation as a reference.

- This is a widely used technique that helps characterize the network as a whole or sub-sets of it, such as inter-connected communities found through topological clustering analysis.

More complex versions of this technique can factor in continuous variables such as expression fold change.

# Pathway reconstruction problem

**Given**

- weighted, directed interactome, *G*, with physical & regulatory interactions
- receptors, *S*, in a signaling pathway of interest
- transcriptional regulators (TRs), *T*, in the same pathway
- a parameter *k*

**Find**

- the *k* highest scoring loopless paths that begin at any receptor in *S* and end at any TR in *T*
- the score of the path is the product of the edge weights (all in *[0,1]*)

# Method Setup

**Modify the graph**

- Add an extra source node *s* and an extra sink node
- add edges *(s,x)* for $x \in S$
- add edges *(y,t)* for $y \in T$
- assign the following *costs* to each edge *(u,v)*

$$c_{uv} = \begin{cases} -log(w_{uv}) & \text{if } u, v \in V \backslash \{s, t\} \\ 0 & \text{if } u = s \text{ or } v = t \end{cases}$$

- Let the cost of a path be the sum of the edges on the path.

# Method Setup

**Modify the graph**

- Add an extra source node *s* and an extra sink node
- add edges *(s,x)* for *x* ∈ *S*
- add edges *(y,t)* for *y* ∈ *T*
- assign the following *costs* to each edge *(u,v)*

$$c_{uv} = \begin{cases} -log(w_{uv}) & \text{if } u, v \in V \backslash \{s, t\} \\ 0 & \text{if } u = s \text{ or } v = t \end{cases}$$

- Let the cost of a path be the sum of the edges on the path.

The least costly *s*→*t* path will be the highest weight *s*→*t* path

# PathLinker

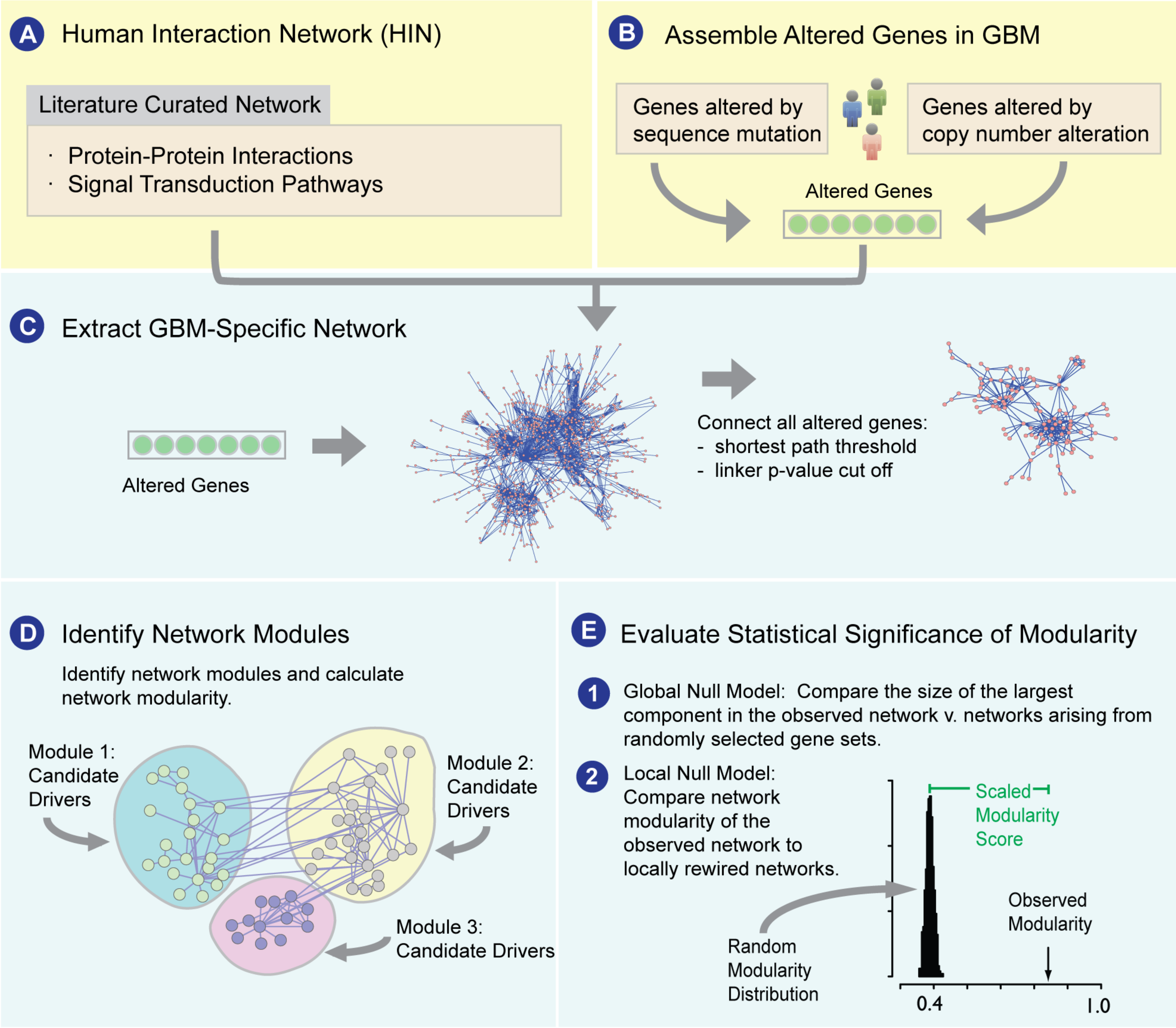
**Algorithm**

- Find the set of *k* highest scoring paths $P_1, P_2, \ldots, P_k$ where each $P_i = (V_i, E_i)$
- Return $G_k = \left( \cup_{1 \le i \le k} V_i, \cup_{1 \le i \le k} E_i \right)$

# NetBox

## Basic Algorithm

- **A.** create human interactome (both interaction and pathway information)
- **B.** find mutated or copy number variant genes for condition in question
- **C.** extract these genes and their neighbors from the interactome
- **D.** run the Newman-Girvan algorithm to find modules
- **E.** analyze statistical significance



# MashMap Idea

First find the winnowed representation of a read

Run the MinHash Sketch on this representation

Reduces the space the hash considers and speeds up computation

They define the **winnowed-minhash** estimate:

$$\mathscr{J}(A, B_i) = \frac{\left| S\Big(W(A) \cup W\left(B_i\right)\Big) \cap S\left(W(A)\right) \cap S\Big(W\left(B_i\right)\Big) \right|}{\left| S\Big(W(A) \cup W\left(B_i\right)\Big) \right|}$$