Building large de Bruijn graphs

Giulio Ermanno Pibiri
Ca’ Foscari University of Venice

Venice, Italy, 2 December 2025

Agenda

1. Context, motivations, and problem definition
2. A simple algorithm
3. Refined algorithms:
    - **BCALM** — minimizers and union-find
    - **GGCAT** — super-kmers and randomization

1. Context, motivations, and problem definition

Massive DNA Collections

ENA European Nucleotide Archive

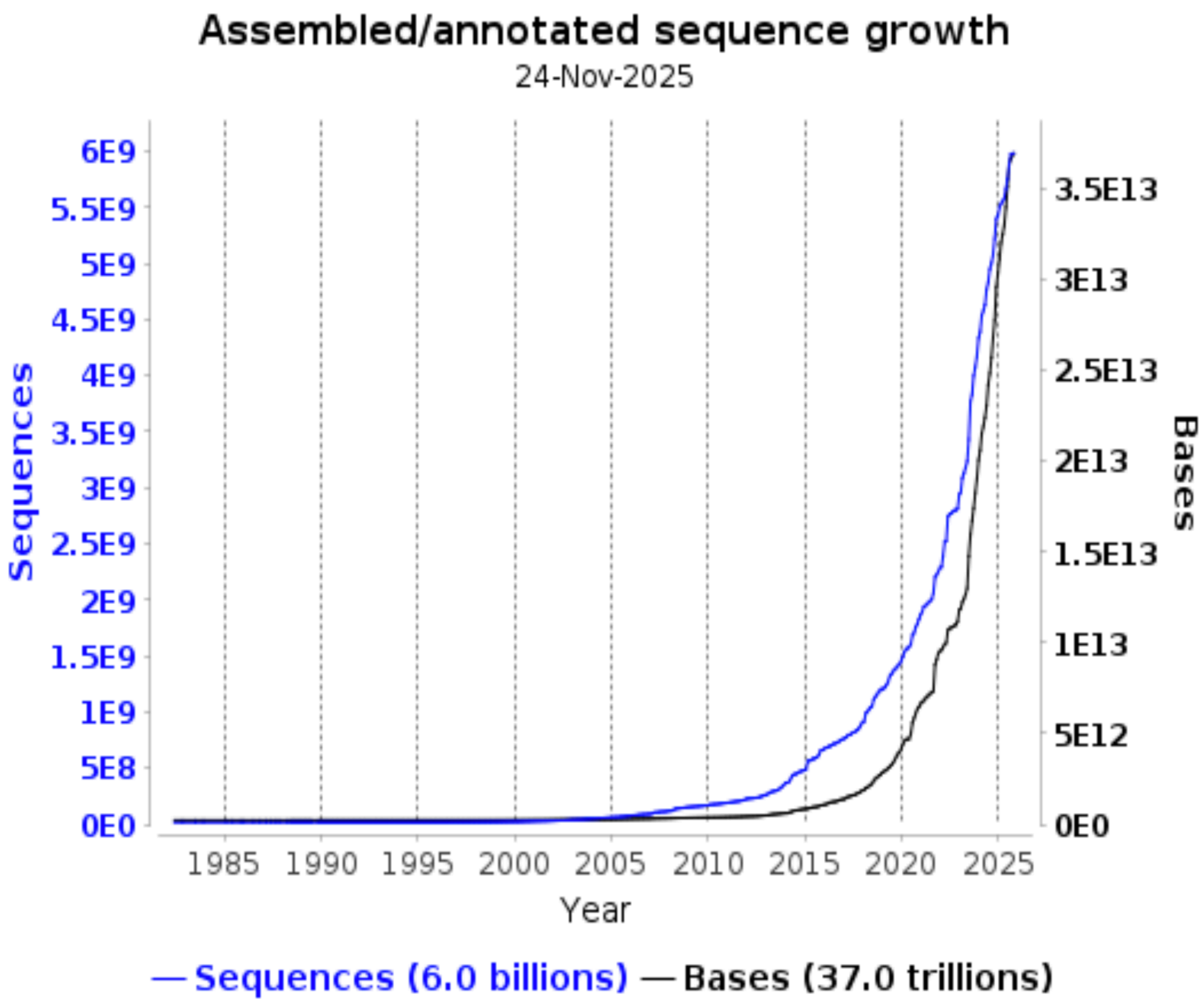
Ensembl

NIH National Library of Medicine
National Center for Biotechnology Information

- **Peta bytes** of data available:

  - ENA (European Nucleotide Archive)
  - SRA (Sequence Read Archive)
  - RefSeq (Reference Sequence Database)
  - Ensembl

- These collections are paving the way to answer fundamental questions regarding biology and evolution.



https://www.ebi.ac.uk/ena/browser/about/statistics

k-mers

- **Q.** But how do we exploit such potential?

  We need efficient methods to index and search data at this scale.

- One popular strategy: transform a DNA sequence into a **set** of short substrings of fixed length k — the so-called **k-mers**.

```
ACGGTAGAACCGATTCAAATTCGACGTAGC...
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
       AACCGATTCAAAT
        ...
```

Example for k=13.

k-mers applications

- Software tools based on k-mers are predominant in bioinformatics.
- Many applications:
  - genome assembly
  - variant calling
  - sequence comparison/alignment
  - pan-genome analysis
  - meta-genomics

  …
- But we will not talk about applications in the following…

Collapsing the redundancy in large k-mer sets

- Note that, given a set S of long strings, the corresponding k-mer set is **highly redundant** (for a suitable value of k).

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.

Set of k-mers:

```
ACGT CGTT
GTTA TTAC
TACG ACGA
CGAA GAAA
CGAC GACA
ACAA CAAA
AAAT AATT
```

Collapsing the redundancy in large k-mer sets

- Note that, given a set S of long strings, the corresponding k-mer set is **highly redundant** (for a suitable value of k).

- Example for S = { "`ACGTTACGTTAC`", "`ACGTTACGAAA`", "`ACGACAAATT`" } and k = 4.

Set of k-mers:

```
ACGT  CGTT
GTTA  TTAC
TACG  ACGA
CGAA  GAAA
CGAC  GACA
ACAA  CAAA
AAAT  AATT
```

- S alone would cost $12 + 11 + 10 + (2) = 35$ chars.

- But its 14 distinct k-mers cost $14 \cdot (4 + 1) = 70$ chars! Twice as much. This is due to **(k-1)-length overlaps** being represented redundantly.

A special char to distinguish the strings

Collapsing the redundancy in large k-mer sets

- Note that, given a set S of long strings, the corresponding k-mer set is **highly redundant** (for a suitable value of k).

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.

Set of k-mers:

ACGT	CGTT
GTTA	TTAC
TACG	ACGA
CGAA	GAAA
CGAC	GACA
ACAA	CAAA
AAAT	AATT

- S alone would cost $12 + 11 + 10 + \boxed{(2)} = 35$ chars.

A special char to distinguish the strings

- But its 14 distinct k-mers cost $14 \cdot (4 + \boxed{1}) = 70$ chars! Twice as much. This is due to **(k-1)-length overlaps** being represented redundantly.

- **Q.** How to collapse (i.e., reduce) this redundancy?

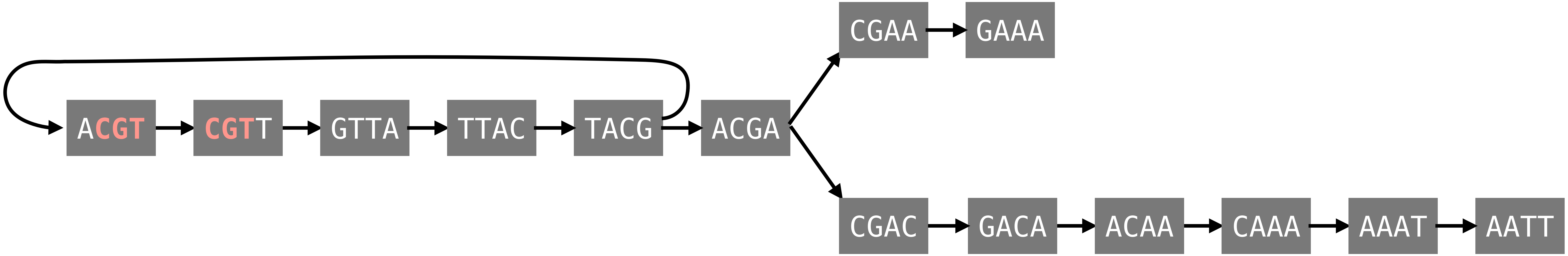
de Bruijn graphs

- **de Bruijn graph.** A de Bruijn graph (dBG) of order $k > 0$ for a set S of strings is a directed graph $G_k(S)$ where nodes are the distinct k-mers of S and there exists a directed edge from x to y if $x[2..k] = y[1..k-1]$ and x "glued" with y, i.e., $x + y[k]$, is a (k+1)-mer of S.

string concatenation

- A path in the graph spells a string obtained by glueing all its k-mers.
With a little abuse of notation we refer to paths and their spelled strings interchangeably.

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.



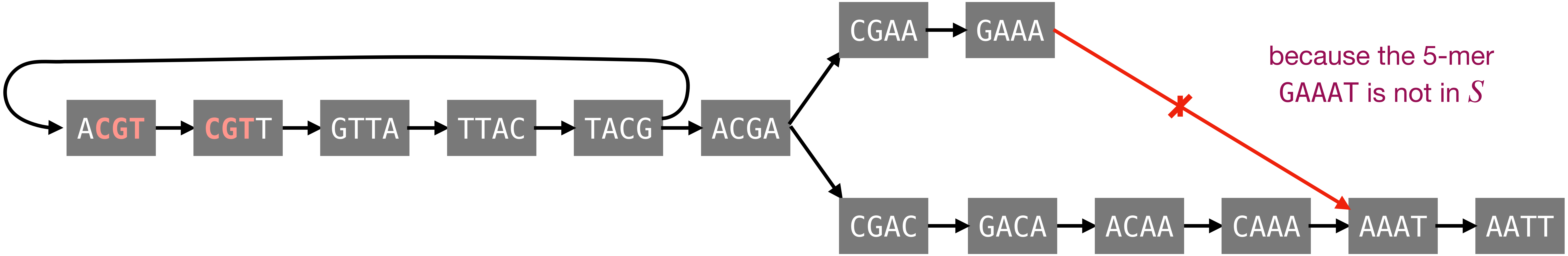
de Bruijn graphs

- **de Bruijn graph.** A de Bruijn graph (dBG) of order $k > 0$ for a set S of strings is a directed graph $G_k(S)$ where nodes are the distinct k-mers of S and there exists a directed edge from x to y if $x[2..k] = y[1..k-1]$ and x "glued" with y, i.e., $x + y[k]$, is a (k+1)-mer of S.

  string concatenation

- A path in the graph spells a string obtained by glueing all its k-mers.
  With a little abuse of notation we refer to paths and their spelled strings interchangeably.

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.



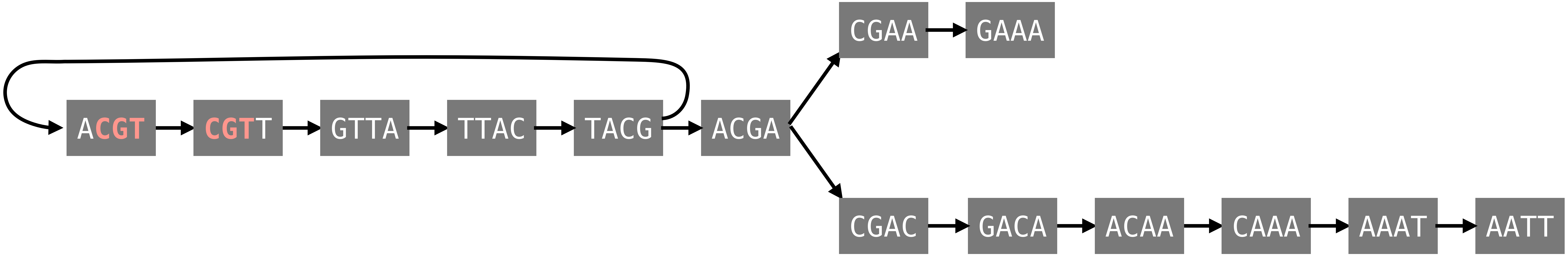
de Bruijn graphs

- **de Bruijn graph.** A de Bruijn graph (dBG) of order $k > 0$ for a set S of strings is a directed graph $G_k(S)$ where nodes are the distinct k-mers of S and there exists a directed edge from x to y if $x[2..k] = y[1..k-1]$ and x "glued" with y, i.e., $x + y[k]$, is a (k+1)-mer of S.

string concatenation

- A path in the graph spells a string obtained by glueing all its k-mers.
  With a little abuse of notation we refer to paths and their spelled strings interchangeably.

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.



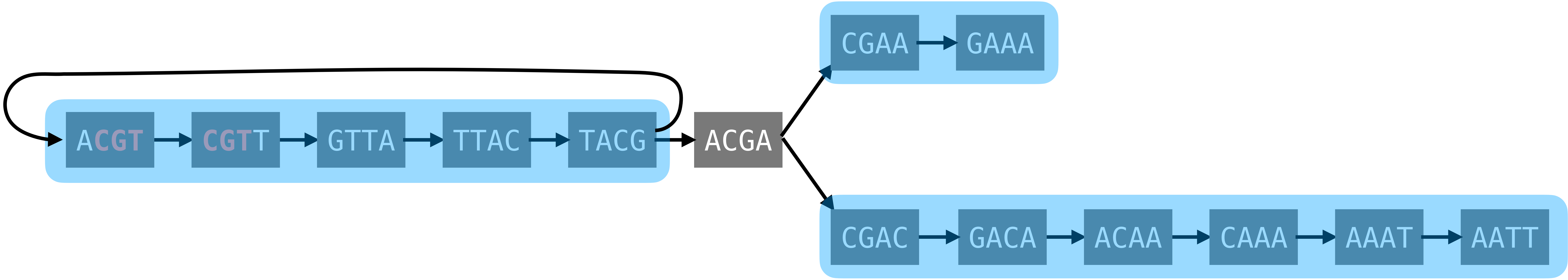
de Bruijn graphs

- **de Bruijn graph.** A de Bruijn graph (dBG) of order $k > 0$ for a set S of strings is a directed graph $G_k(S)$ where nodes are the distinct k-mers of S and there exists a directed edge from x to y if $x[2..k] = y[1..k-1]$ and x "glued" with y, i.e., $x + y[k]$, is a (k+1)-mer of S.

string concatenation

- A path in the graph spells a string obtained by glueing all its k-mers.
  With a little abuse of notation we refer to paths and their spelled strings interchangeably.

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.

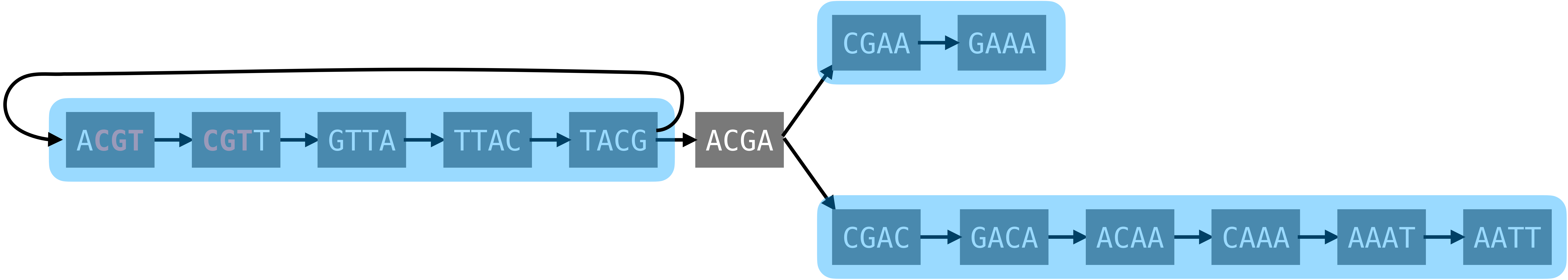


Compacted de Bruijn graphs

- **Unitig and maximal unitig.** A unitig in $G_k(S)$ is a path where all inner nodes have in/out degree 1. A maximal unitig is a unitig that cannot be extended without loosing the property of being a unitig. Let U be the set of all maximal unitigs of $G_k(S)$.

- **Compacted de Bruijn graph.** The compacted dBG of $G_k(S)$ is a directed graph where nodes are the strings of U and there exists a directed edge from x to y if $x[|x|-k+2..|x|] = y[1..k-1]$ and $x[|x|-k+1..|x|] + y[k]$ is a (k+1)-mer in S.

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.

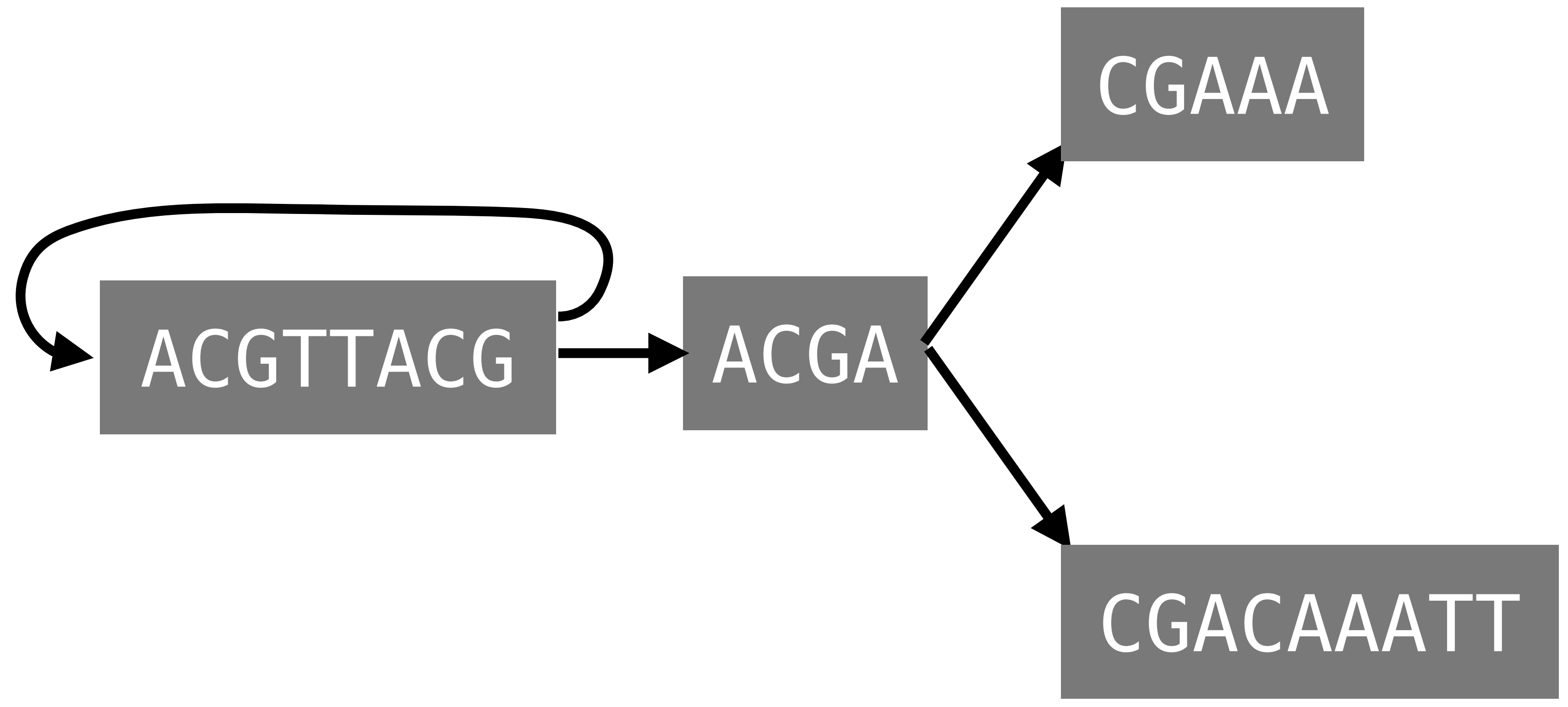


Compacted de Bruijn graphs

- **Unitig and maximal unitig.** A unitig in $G_k(S)$ is a path where all inner nodes have in/out degree 1. A maximal unitig is a unitig that cannot be extended without loosing the property of being a unitig. Let U be the set of all maximal unitigs of $G_k(S)$.

- **Compacted de Bruijn graph.** The compacted dBG of $G_k(S)$ is a directed graph where nodes are the strings of U and there exists a directed edge from x to y if $x[\,|x|-k+2..|x|\,] = y[1..k-1]$ and $x[\,|x|-k+1..|x|\,]+y[k]$ is a (k+1)-mer in S.

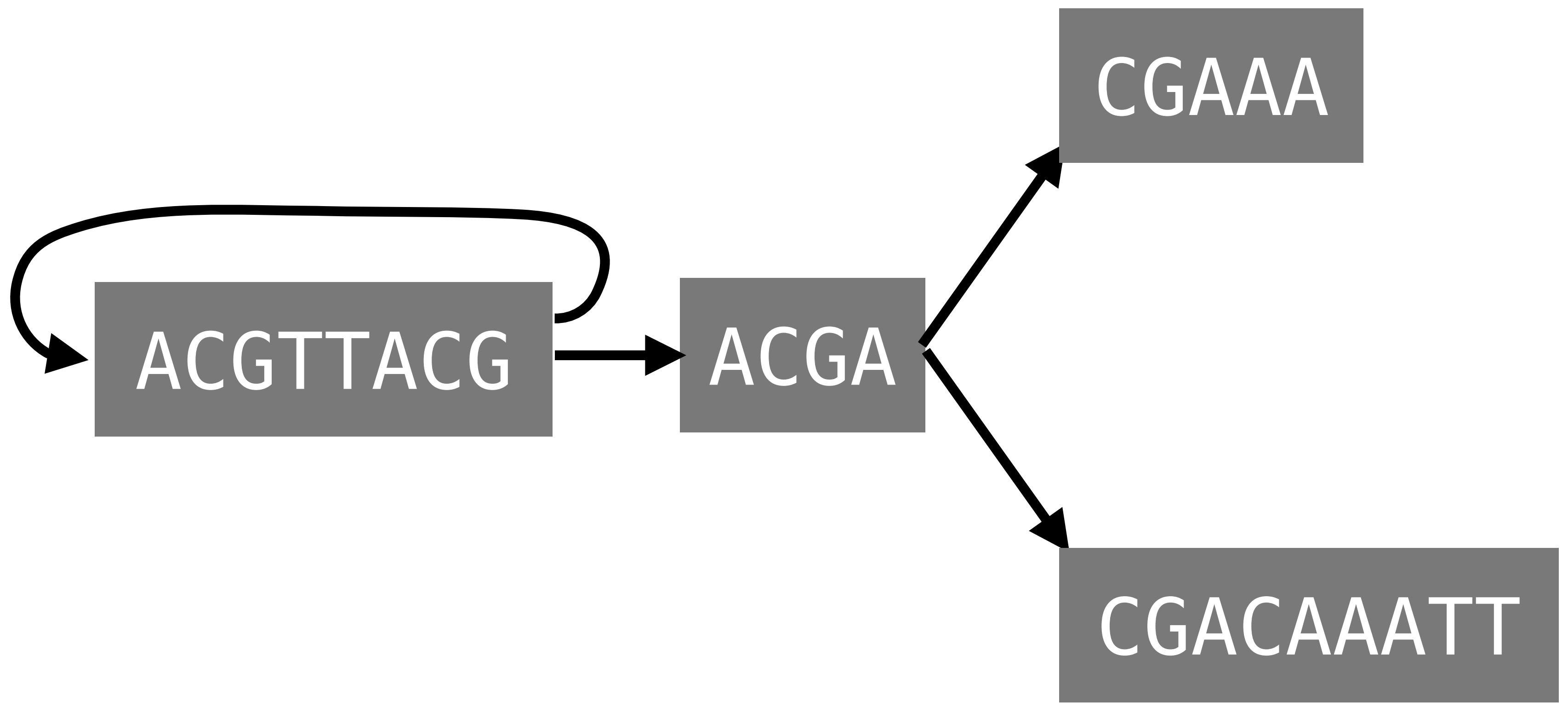
- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.



U = { "ACGTTACG", "ACGA", "CGAAA", "CGACAAATT" }

Compacted de Bruijn graphs

- Example for S = { “ACGTTACGTTAC”, “ACGTTACGAAA”, “ACGACAAATT” } and k = 4.

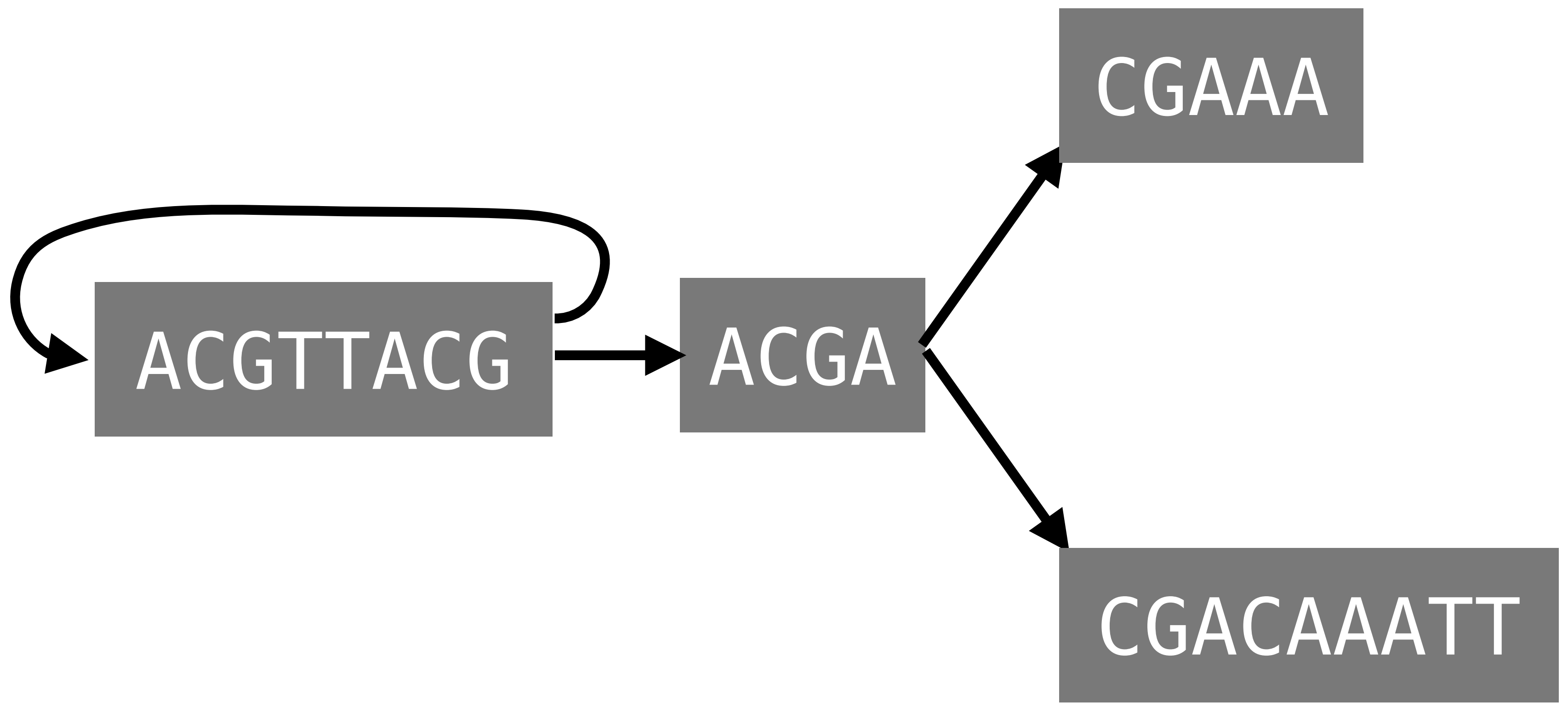


U = { “ACGTTACG”, “ACGA”,
“CGAAA”, “CGACAAATT” }

- **Two remarks**:

Compacted de Bruijn graphs

- Example for S = { "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" } and k = 4.



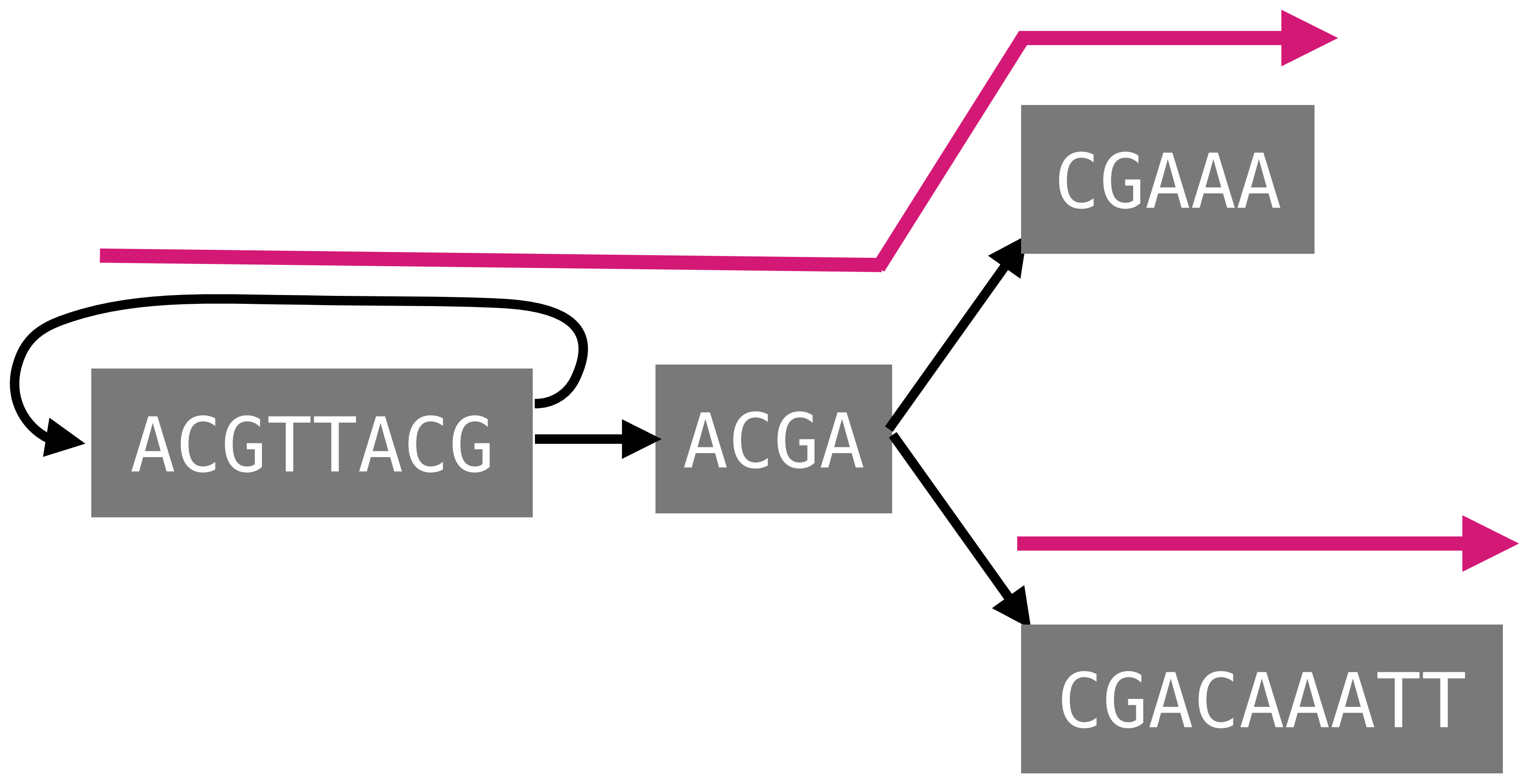
U = { "ACGTTACG", "ACGA", "CGAAA", "CGACAAATT" }

- **Two remarks**:

  1. To store U we need 26+3=29 chars, much better than the 70 chars for the "plain" k-mer set.

Compacted de Bruijn graphs

- Example for S = { “ACGTTACGTTAC”, “ACGTTACGAAA”, “ACGACAAATT” } and k = 4.



U = { “ACGTTACG”, “ACGA”, “CGAAA”, “CGACAAATT” }

U^* = { “ACGTTACGAAA”, “CGACAAATT” }

- **Two remarks**:

  1. To store U we need 26+3=29 chars, much better than the 70 chars for the “plain” k-mer set.
  2. We could do even better by glueing some unitigs via a (smallest, i.e., with minimum number of paths) **disjoint-node path cover** U^* (just 20 chars!).

Is it relevant/useful?

- Example.

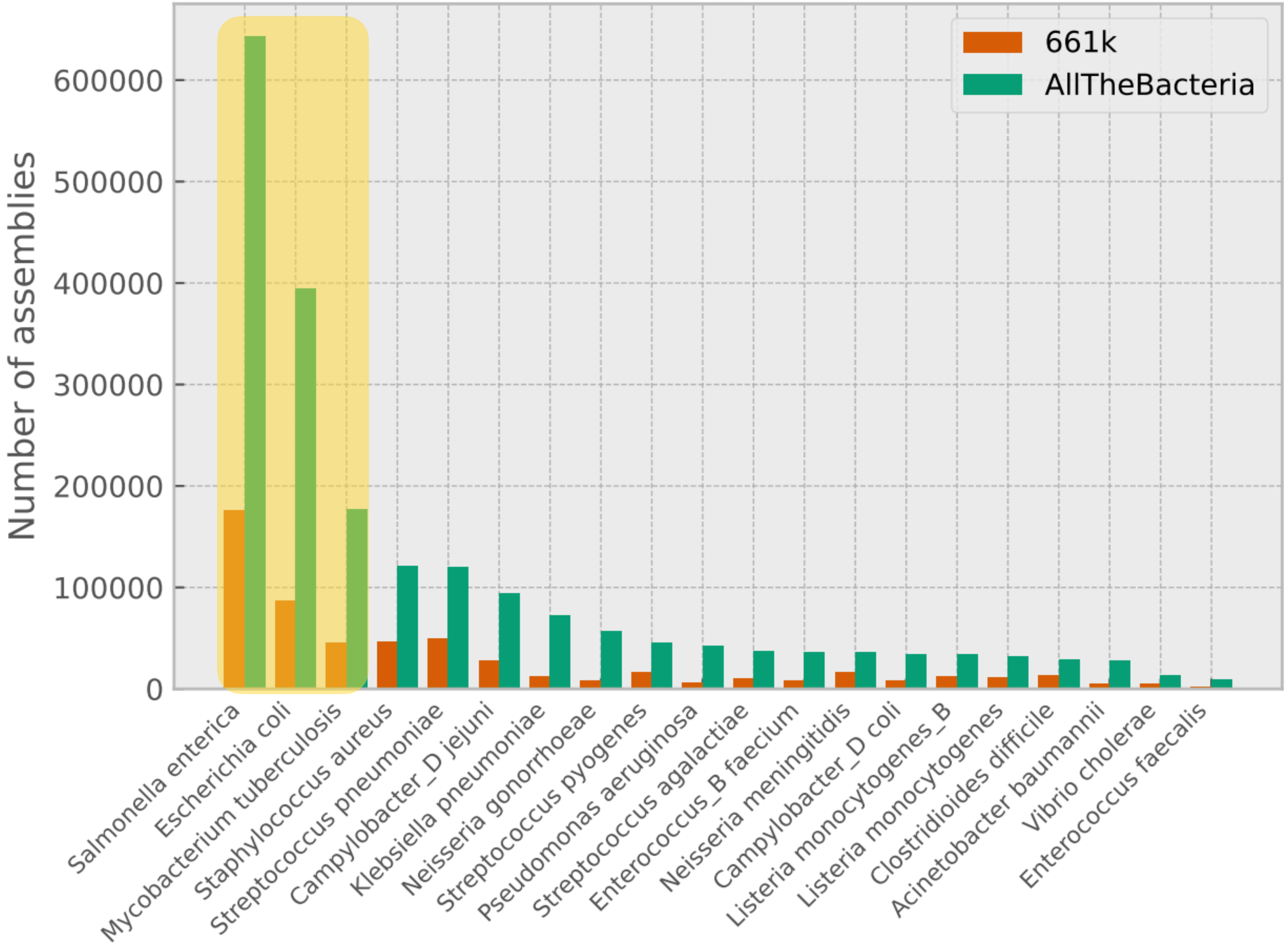
  The **top-3** species in the "AllTheBacteria" collection take **1.3 TB of gzipped files**.

  The maximal unitigs of the corresponding dBG take **< 9 GB**.

  Hence, a $\approx 145\times$ reduction.

- Save storage space and speed up applications involving large k-mer sets.

"AllTheBacteria", https://allthebacteria.org



Taken from https://www.biorxiv.org/content/10.1101/2024.03.08.584059v3.full

Problem definition

- **Problem.** Given a set S of strings and an integer $k > 0$, we want to compute the set U of maximal unitigs of $G_k(S)$ quickly and using little computer memory.

- Usually S is very large (e.g., hundreds of thousands or millions of genomes) and $k \in \{31,\ldots,63\}$.

- An intensively studied problem. A lot of research effort is actively spent on it.

  BCALM [Chikhi et al., **2015**]
  BCALM v2 [Chikhi, Limasset, and Medvedev, **2016**]
  TwoPaCo [Minkin, Pham, and Medvedev, **2016**]
  Cuttlefish [Khan and Patro, **2021**]
  Cuttlefish v2 [Khan et al., **2022**]
  GGCAT [Cracco and Tomescu, **2023**] ← **state of the art**

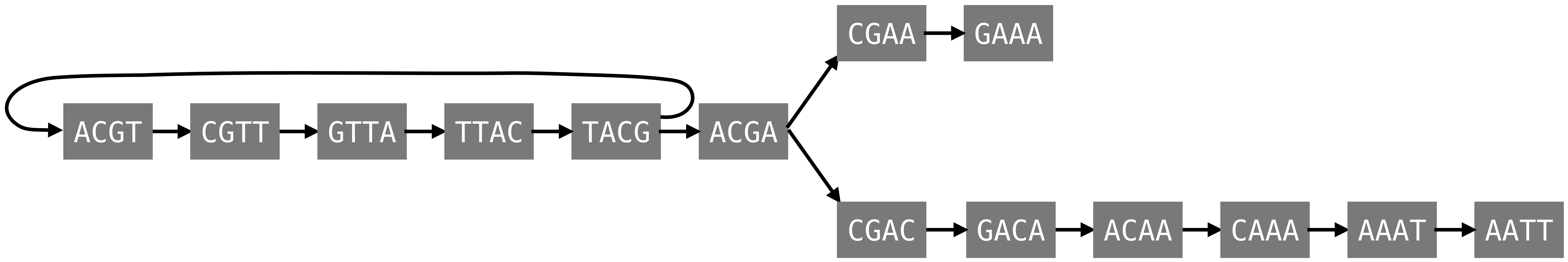
2. A simple algorithm

A simple algorithm

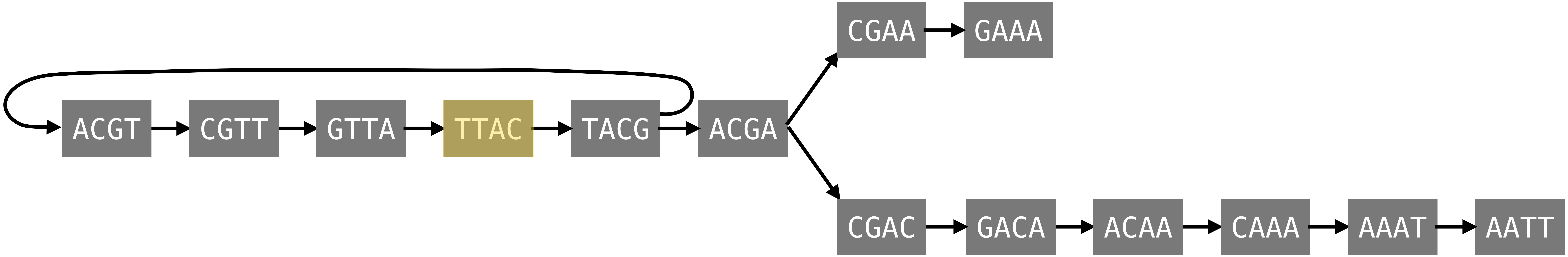
- **Idea.** Build $G_k(S)$ and visit it.
- We start a new path from a randomly chosen k-mer $x \in S$, and extend it as much as possible forward and backward **as long as the path is a unitig** (i.e., inner nodes have one predecessor and one successor only).

  The resulting path is a **maximal** unitig.
- During the visit, we keep track of visited nodes.
- We repeat this process until all nodes have been visited.

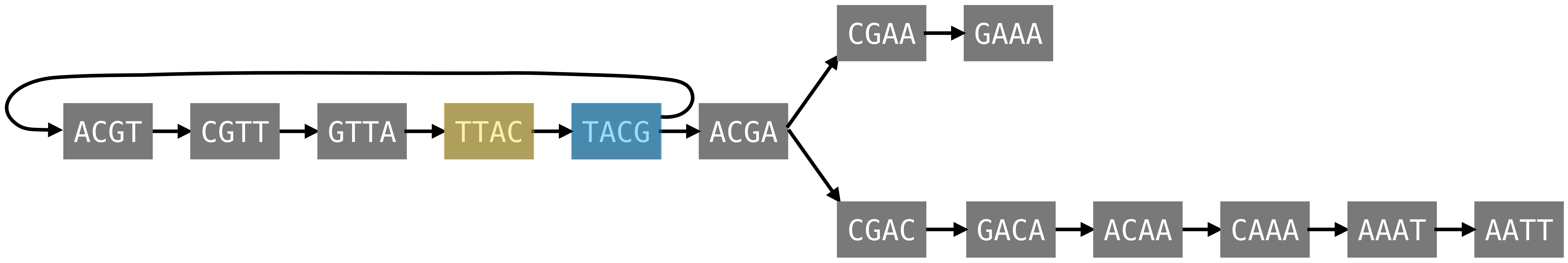
A simple algorithm — Demo



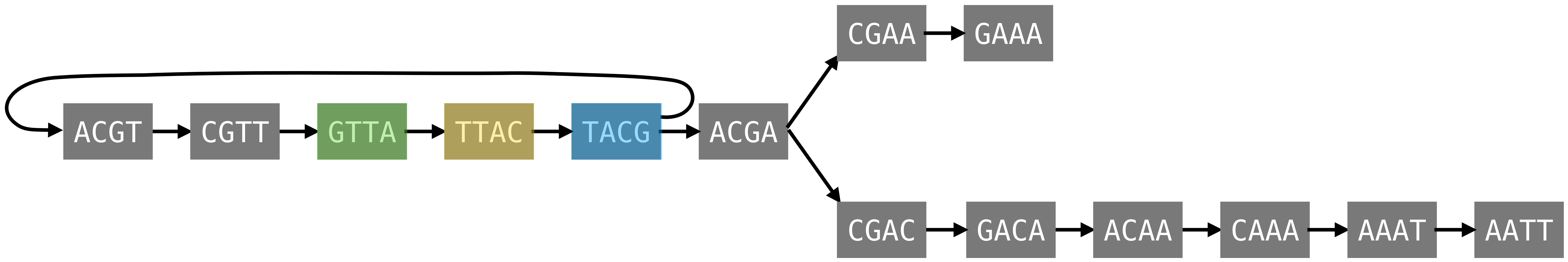
A simple algorithm — Demo



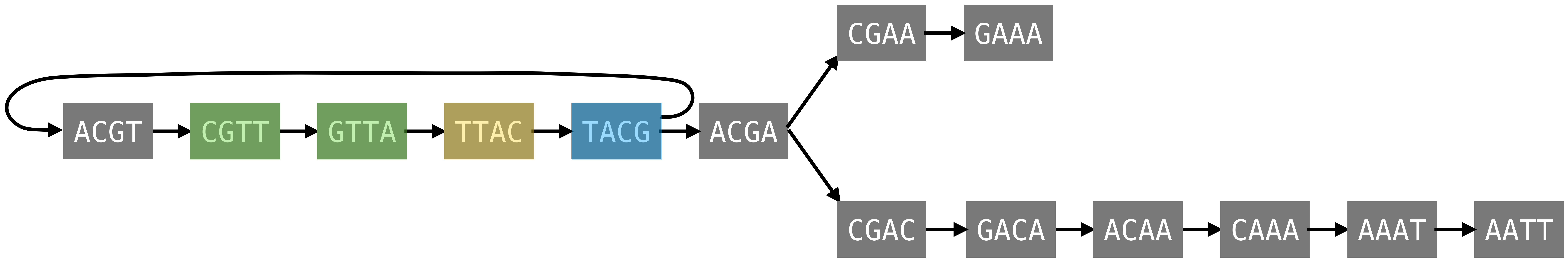
A simple algorithm — Demo



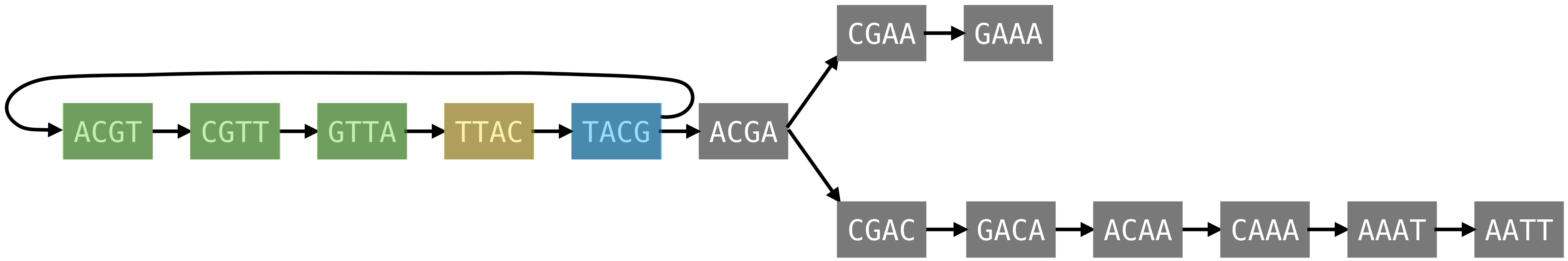
A simple algorithm — Demo



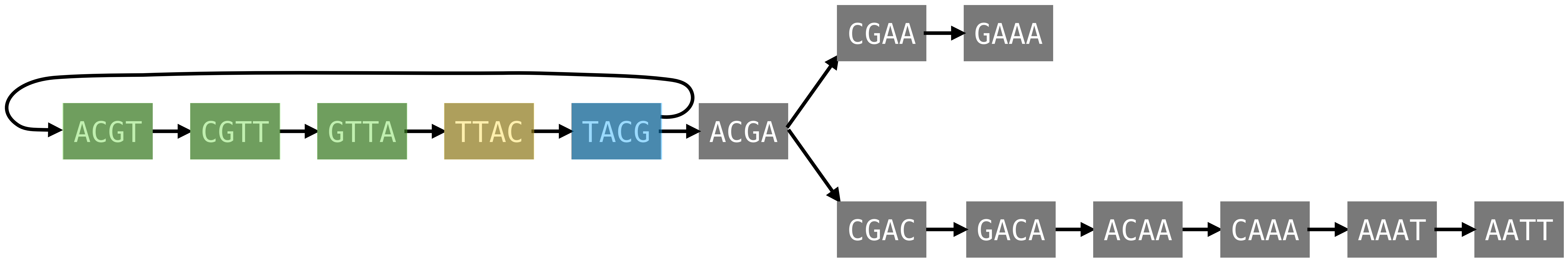
A simple algorithm — Demo



A simple algorithm — Demo

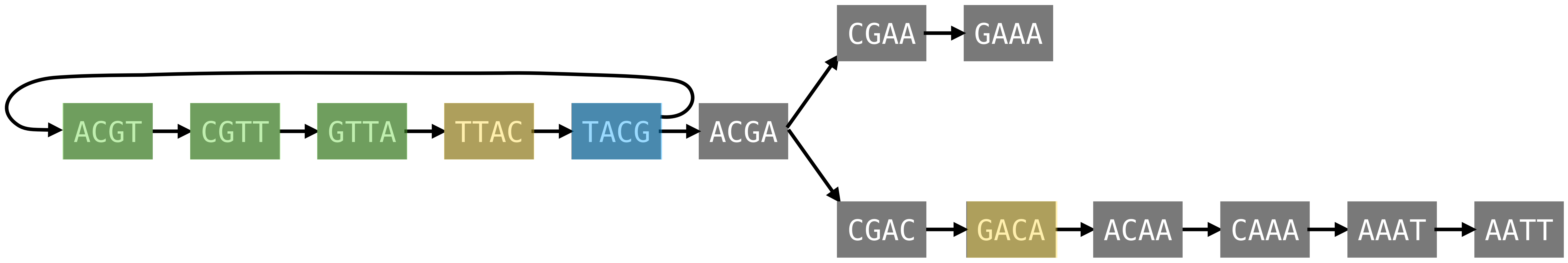


A simple algorithm — Demo



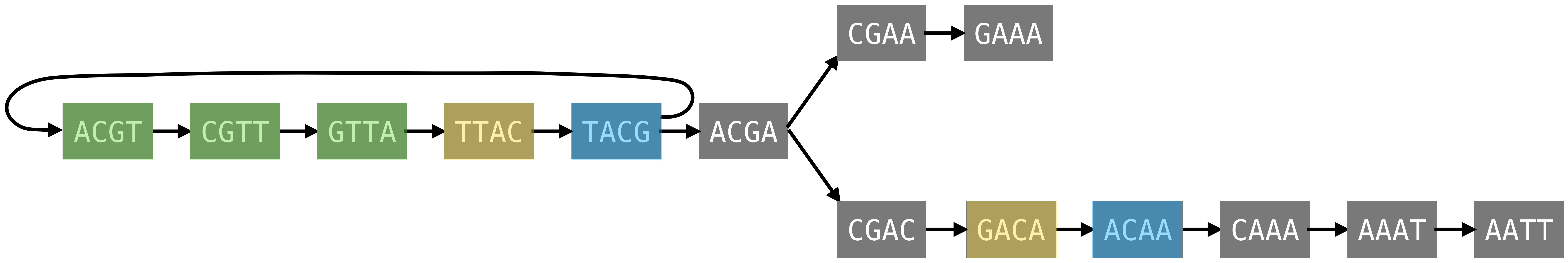
ACG–TTAC–G

A simple algorithm — Demo



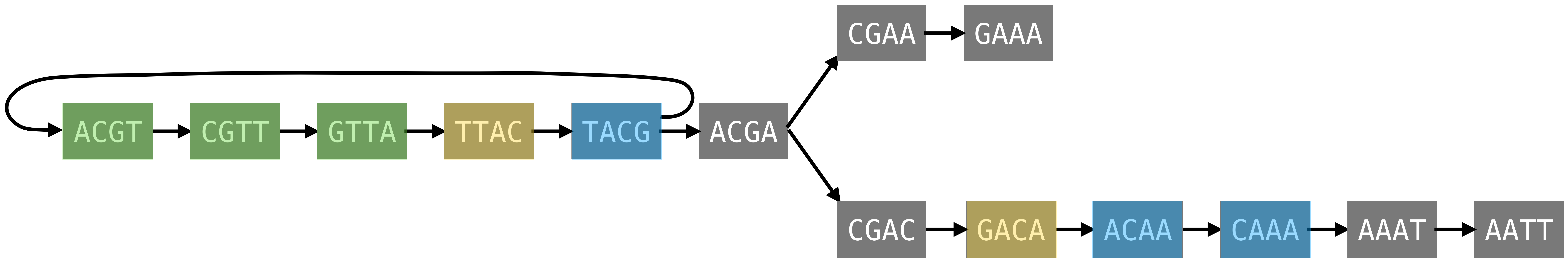
ACG–TTAC–G

A simple algorithm — Demo



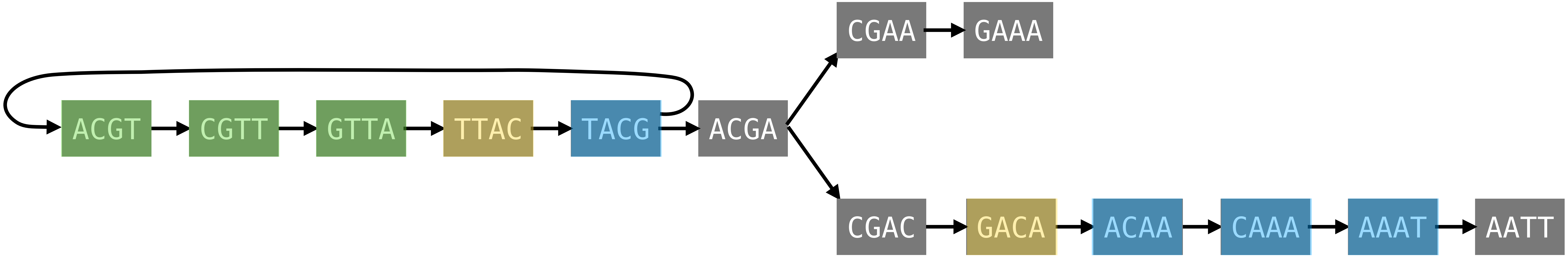
ACG–TTAC–G

A simple algorithm — Demo



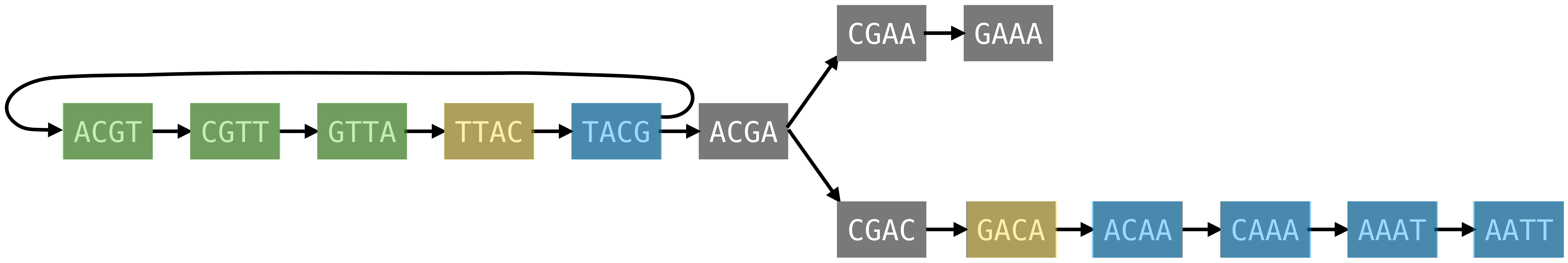
ACG–TTAC–G

A simple algorithm — Demo



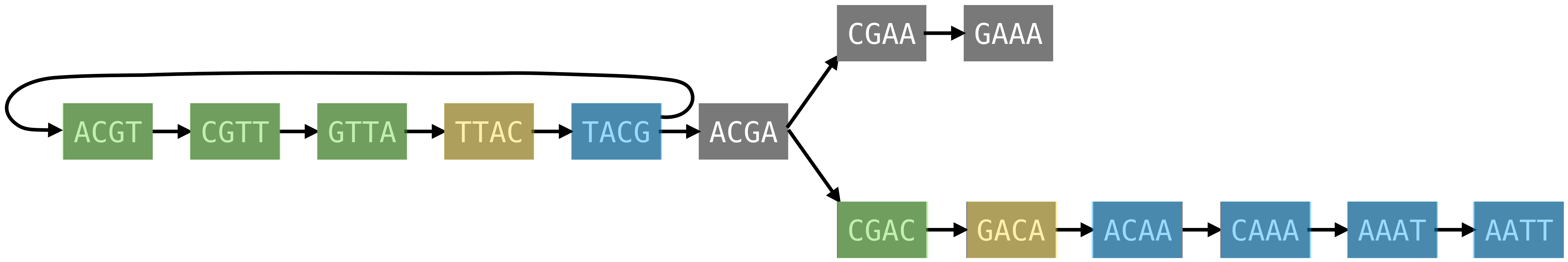
ACG–TTAC–G

A simple algorithm — Demo



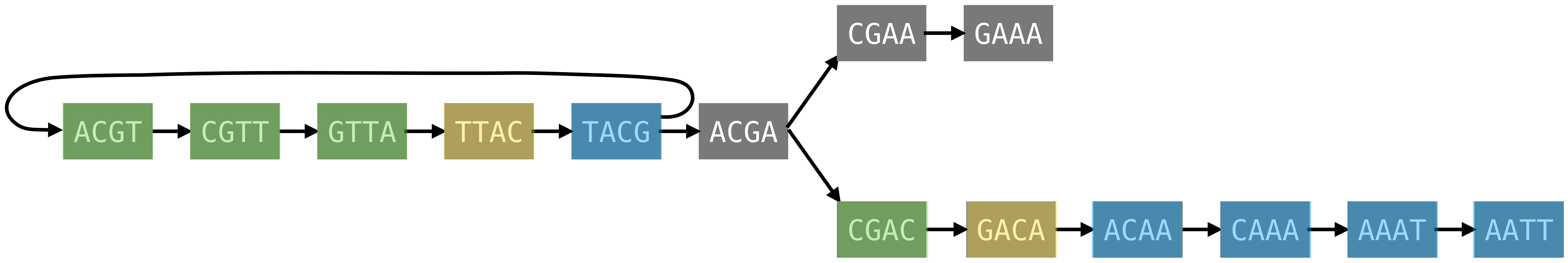
ACG–TTAC–G

A simple algorithm — Demo



ACG–TTAC–G

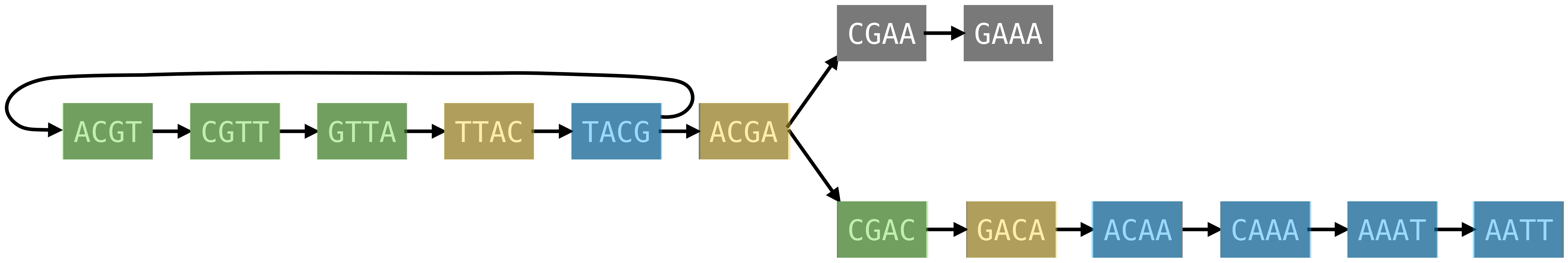
A simple algorithm — Demo



ACG–TTAC–G

C–GACA–AATT

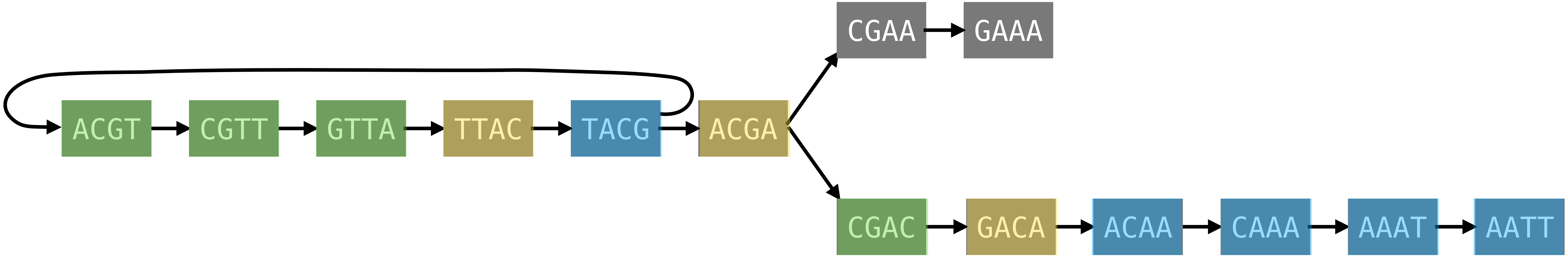
A simple algorithm — Demo



ACG–TTAC–G

C–GACA–AATT

A simple algorithm — Demo

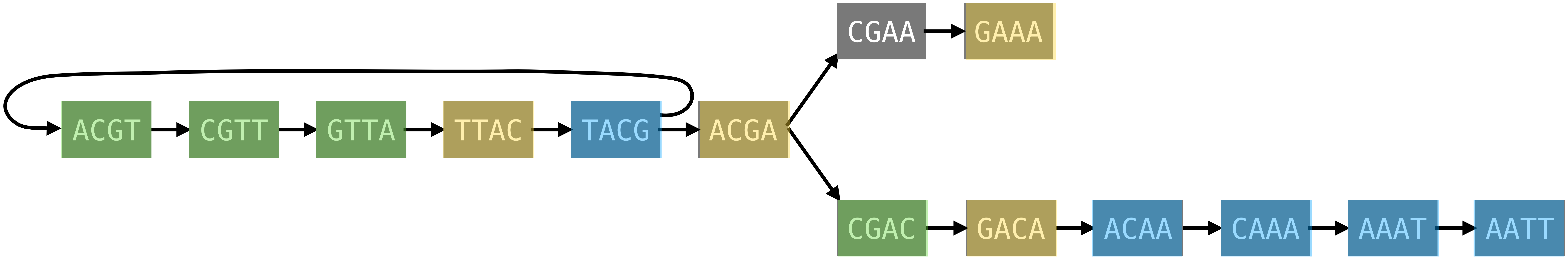


ACG–TTAC–G

C–GACA–AATT

ACGA

A simple algorithm — Demo

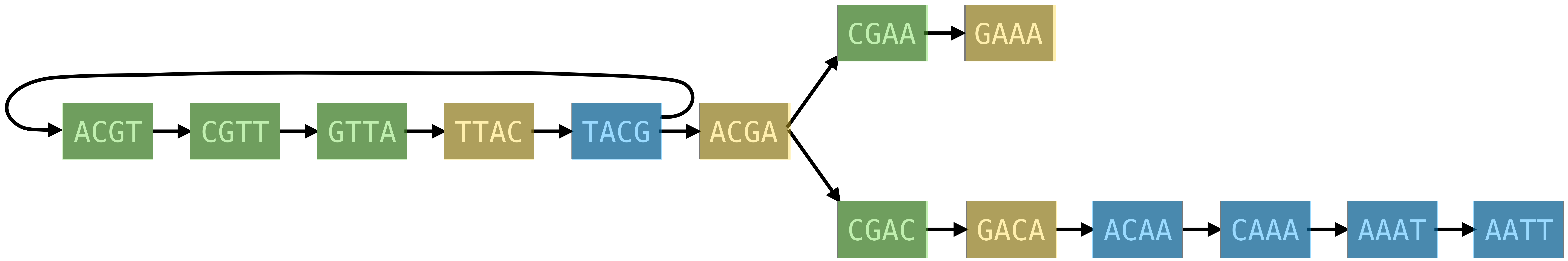


ACG–TTAC–G

C–GACA–AATT

ACGA

A simple algorithm — Demo

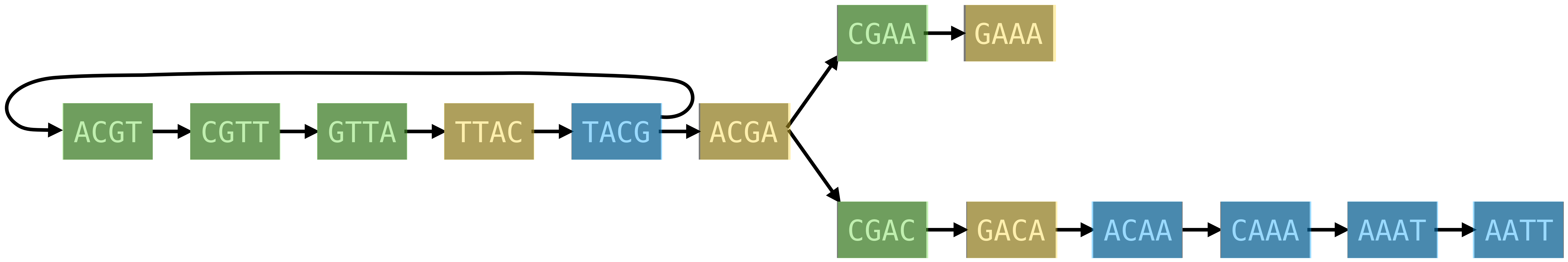


ACG–TTAC–G

C–GACA–AATT

ACGA

A simple algorithm — Demo



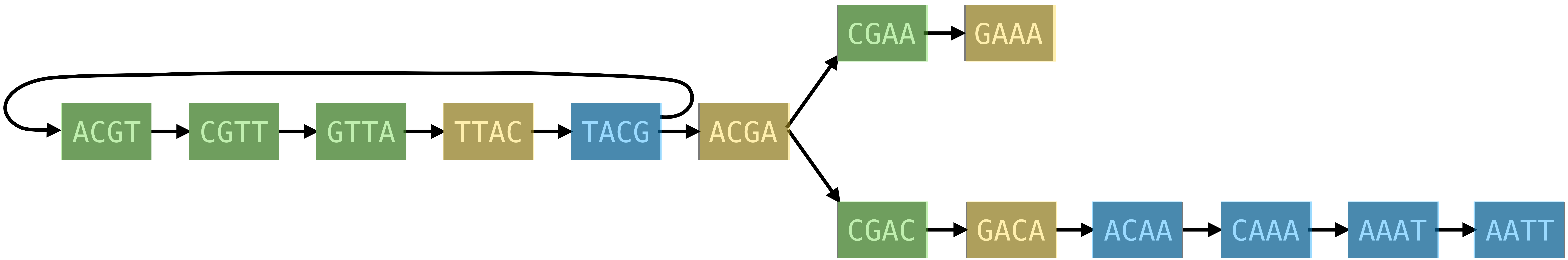
ACG–TTAC–G

C–GACA–AATT

ACGA

C–GAAA

A simple algorithm — Demo



ACG–TTAC–G

C–GACA–AATT

ACGA

C–GAAA

Q. How would you implement it?

Representing the graph

- We create e hash table G such that, for each distinct k-mer $x \in S$, $G[x] = (pred(x), succ(x))$

  where

  $pred(x) = \{c \,|\, c + x[1..k-1]$ is a k-mer of $S\}$
  $succ(x) = \{c \,|\, x[2..k] + c$ is a k-mer of $S\}$

```python
def dbg(S, k):
    G = {}
    for s in S:
        n = len(s)
        for i in range(n - k + 1):
            x = s[i:i+k]
            pred, succ = G.setdefault(x, (set(), set()))
            if i > 0:
                pred.add(s[i-1])
            if i + k < n:
                succ.add(s[i+k])
    return G
```

Representing the graph

- We create e hash table G such that, for each distinct k-mer $x \in S$, $G[x] = (pred(x), succ(x))$

  where

  $pred(x) = \{c \mid c + x[1..k-1]$ is a k-mer of $S\}$
  $succ(x) = \{c \mid x[2..k] + c$ is a k-mer of $S\}$

```python
def dbg(S, k):
    G = {}
    for s in S:
        n = len(s)
        for i in range(n - k + 1):
            x = s[i:i+k]
            pred, succ = G.setdefault(x, (set(), set()))
            if i > 0:
                pred.add(s[i-1])
            if i + k < n:
                succ.add(s[i+k])
    return G
```

- Let:

  N = the total number of k-mers in S
  n = the number of distinct k-mers in S
  (nodes of the dBG)
  m = number of edges in the dBG

- G is built in $\Theta(N)$ time and takes

  $O(n + \sum_{x \in G} (|pred(x)| + |succ(x)|)) = O(n + m)$ space, because $\sum_{x \in G} (|pred(x)| + |succ(x)|) = 2m$.

Visiting the graph

```
def compact(G, k):
    visited = set()
    U = []
    for x in G:
        if x in visited:
            continue
        visited.add(x);
        fwd = extend_fwd(x, k, G, visited)
        bwd = extend_bwd(x, k, G, visited)
        u = bwd + x + fwd
        U.append(u)
    return U
```

- $\Theta(n + m)$ time to visit the graph.

```
def extend_fwd(x, k, G, visited):
    path = ''
    while True:
        _, succ = G[x]
        if len(succ) != 1:
            break
        nuc = next(iter(succ))
        nxt = x[1:] + nuc
        pred, _ = G[nxt]
        if len(pred) != 1 or nxt in visited:
            break
        visited.add(nxt)
        path = path + nuc
        x = nxt
    return path
```

```
def extend_bwd(x, k, G, visited):
    path = ''
    while True:
        pred, _ = G[x]
        if len(pred) != 1:
            break
        nuc = next(iter(pred))
        prv = nuc + x[:k-1]
        _, succ = G[prv]
        if len(succ) != 1 or prv in visited:
            break
        visited.add(prv)
        path = nuc + path
        x = prv
    return path
```

Putting all together

```
k = 4
S = [ "ACGTTACGTTAC", "ACGTTACGAAA", "ACGACAAATT" ]

G = dbg(S, k)
U = compact(G, k)
print(U) # ['ACGTTACG', 'ACGA', 'CGAAA', 'CGACAAATT']
```

- Summary: complexity (in both time and space) is **linear** in the size of the graph.
- The problem here is the **space**!
- Let’s call this algorithm SIMPLE in the following.

3. Refined algorithms

Bird’s eye view

- **Divide and conquer.** Split the k-mer set into buckets and work on them independently in parallel using SIMPLE.
- Note: There are many engineering aspects (e.g., multithreading, compressed I/Os, disk pipelining) that we will not discuss but contribute significantly to keep the overall running time low.

  Implementation matters a lot!

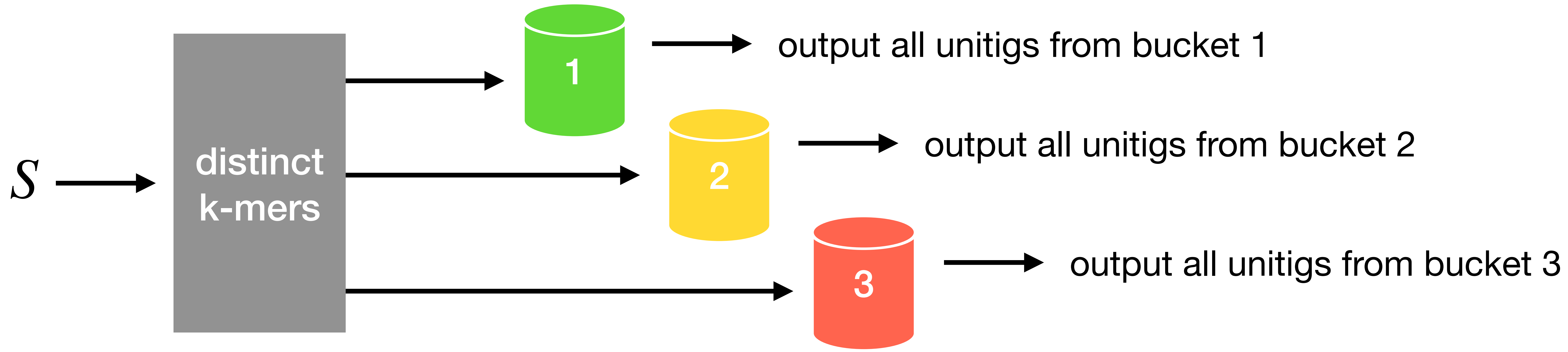
BCALM

Chikhi, Limasset, and Medvedev, 2016

- **Idea.** Distribute k-mers into buckets and process the buckets (sub-graphs) independently in parallel using SIMPLE. A bucket is a file on disk.

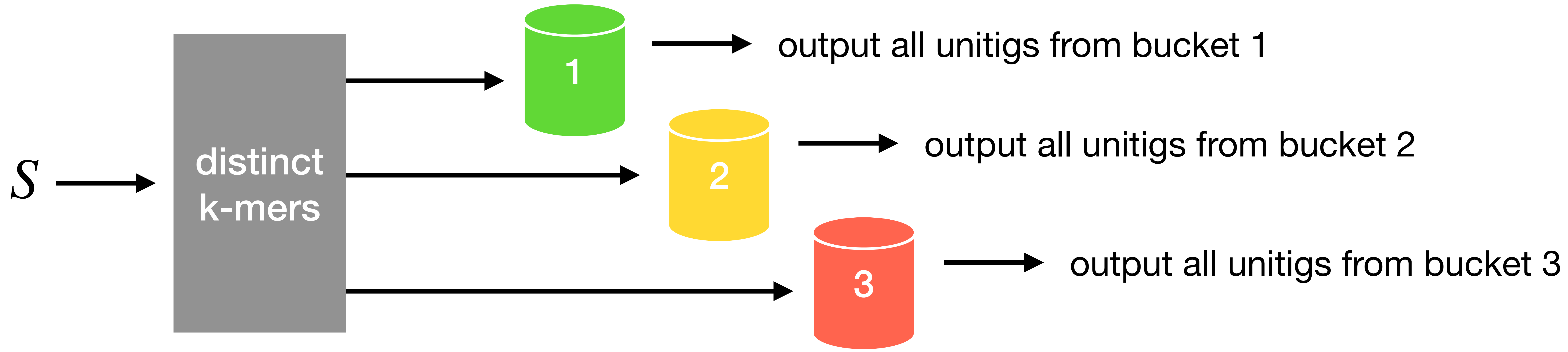
- Not all buckets are loaded into main memory at the same time to run SIMPLE, hence allowing to scale to very large datasets using a prescribed amount of RAM.

- The algorithm takes as input a set of kmers, hence a first preprocessing step of k-mer counting is performed.



Challenges

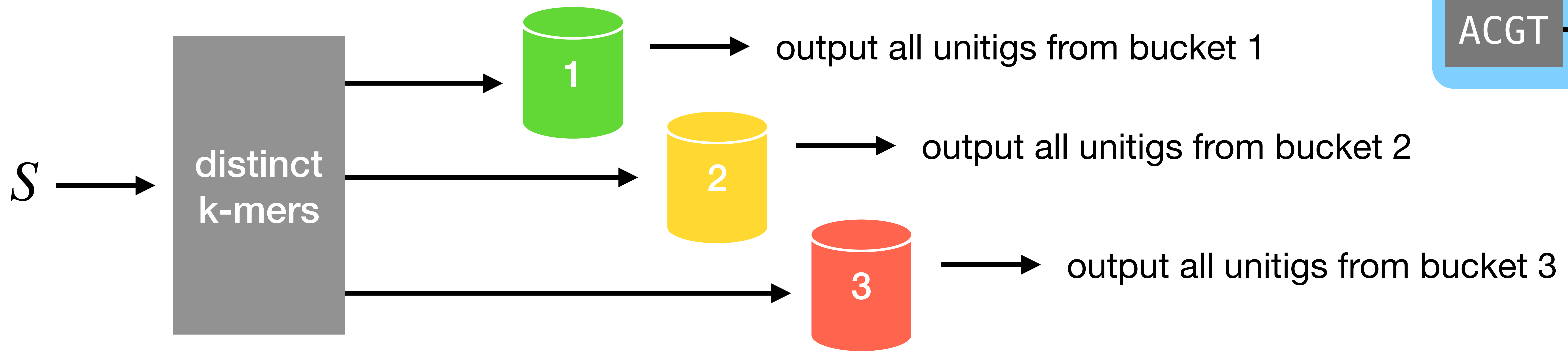
1. **How to guarantee that two k-mers sharing an overlap are placed in the same bucket?** In principle, we could define one bucket per (k-1)-mer but there would be too many for typical values of k.

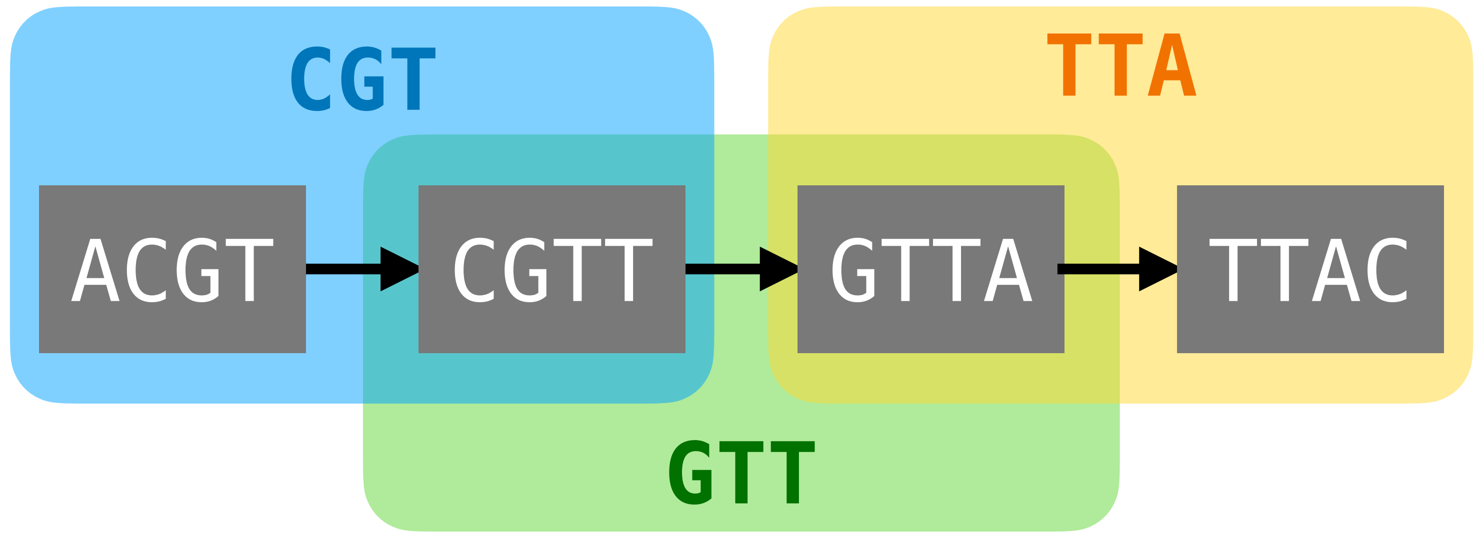


Challenges

1. **How to guarantee that two k-mers sharing an overlap are placed in the same bucket?**
   In principle, we could define one bucket per (k-1)-mer but there would be too many for typical values of k.

2. **In general, a maximal unitig can span more than one bucket.**
   Some unitigs computed in different buckets must be further glued into a maximal unitig.



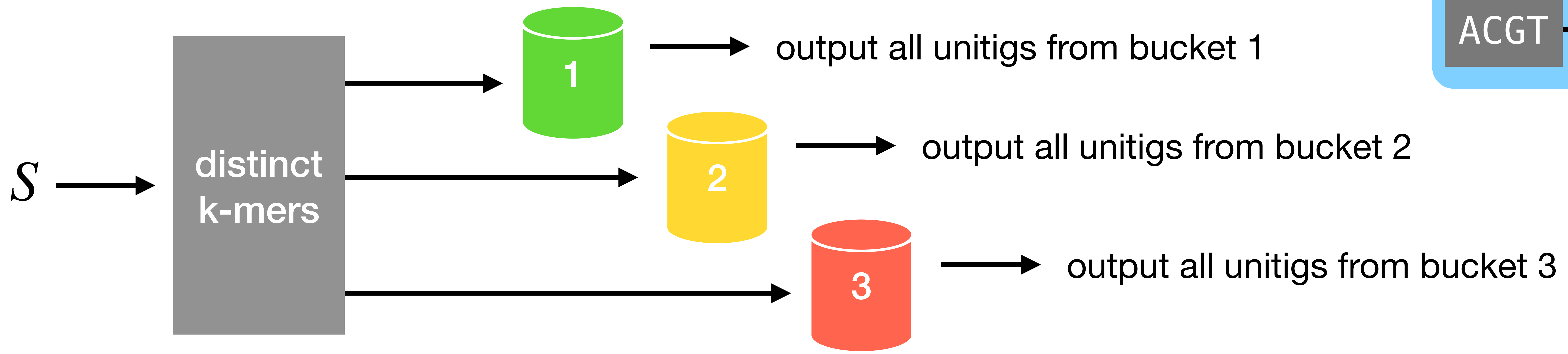


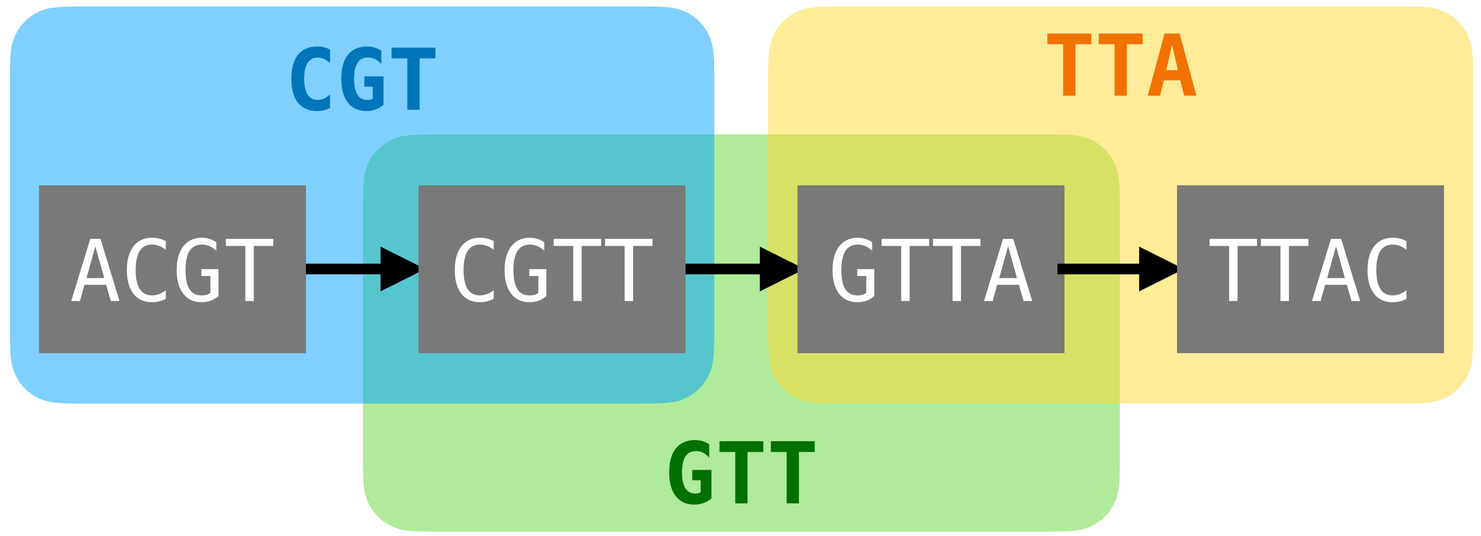
Challenges

1. **How to guarantee that two k-mers sharing an overlap are placed in the same bucket?**
   In principle, we could define one bucket per (k-1)-mer but there would be too many for typical values of k.

2. **In general, a maximal unitig can span more than one bucket.**
   Some unitigs computed in different buckets must be further glued into a maximal unitig.

3. **Load-balancing: buckets should have approx. the same size.** (We will not talk about this.)





Minimizers

- **Minimizer.** Given a k-mer x and an order $\mathcal{O}$ over all ℓ-mers, the minimizer of length $\ell \leq k$ of x is the (leftmost) smallest ℓ-mer of x according to $\mathcal{O}$.

- Example for $x =$ `TCGATAGAAC` $(k = 10)$, $\ell = 4$, and using $\mathcal{O}$ = lexicographic order.

```
TCGATAGAAC
TCGA
 CGAT
  GATA
   ATAG
    TAGA
 →   AGAA
      GAAC
```

Minimizers

- **Minimizer.** Given a k-mer x and an order $\mathcal{O}$ over all ℓ-mers, the minimizer of length $\ell \leq k$ of x is the (leftmost) smallest ℓ-mer of x according to $\mathcal{O}$.

- Example for $x = \texttt{TCGATAGAAC}$ $(k = 10)$, $\ell = 4$, and using $\mathcal{O}$ = lexicographic order.

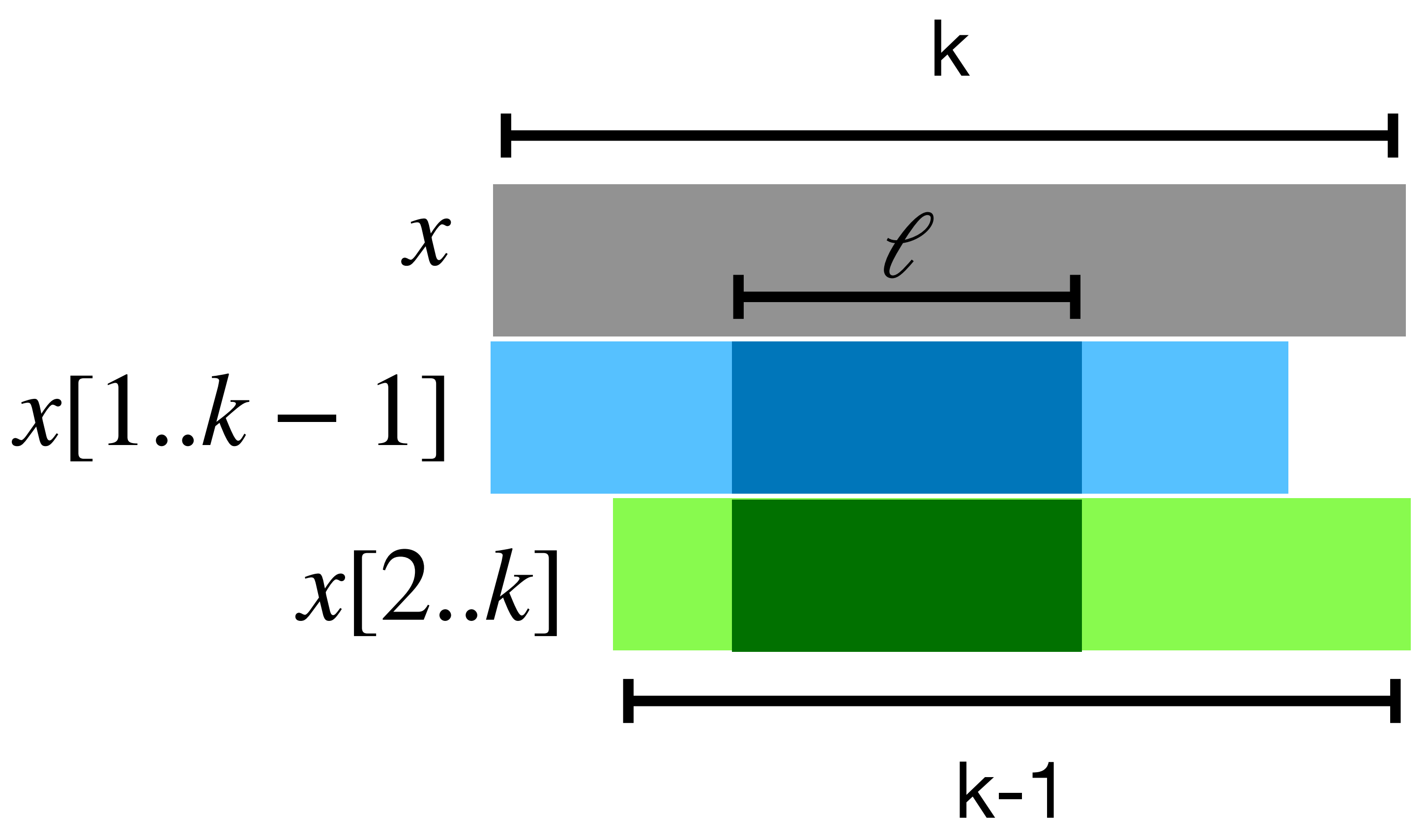
```
TCGA
 CGAT
  GATA
   ATAG
    TAGA
→    AGAA
      GAAC
```

- **Property.** Consecutive k-mers tend to share the same minimizer.

```
TCGATAGAACCGATTCAAATTCGAT...
TCGATAGAAC
 CGATAGAACC
  GATAGAACCG
   ATAGAACCGA
    TAGAACCGAT
     AGAACCGATT
      GAACCGATTC
       AACCGATTCA
         ...
```

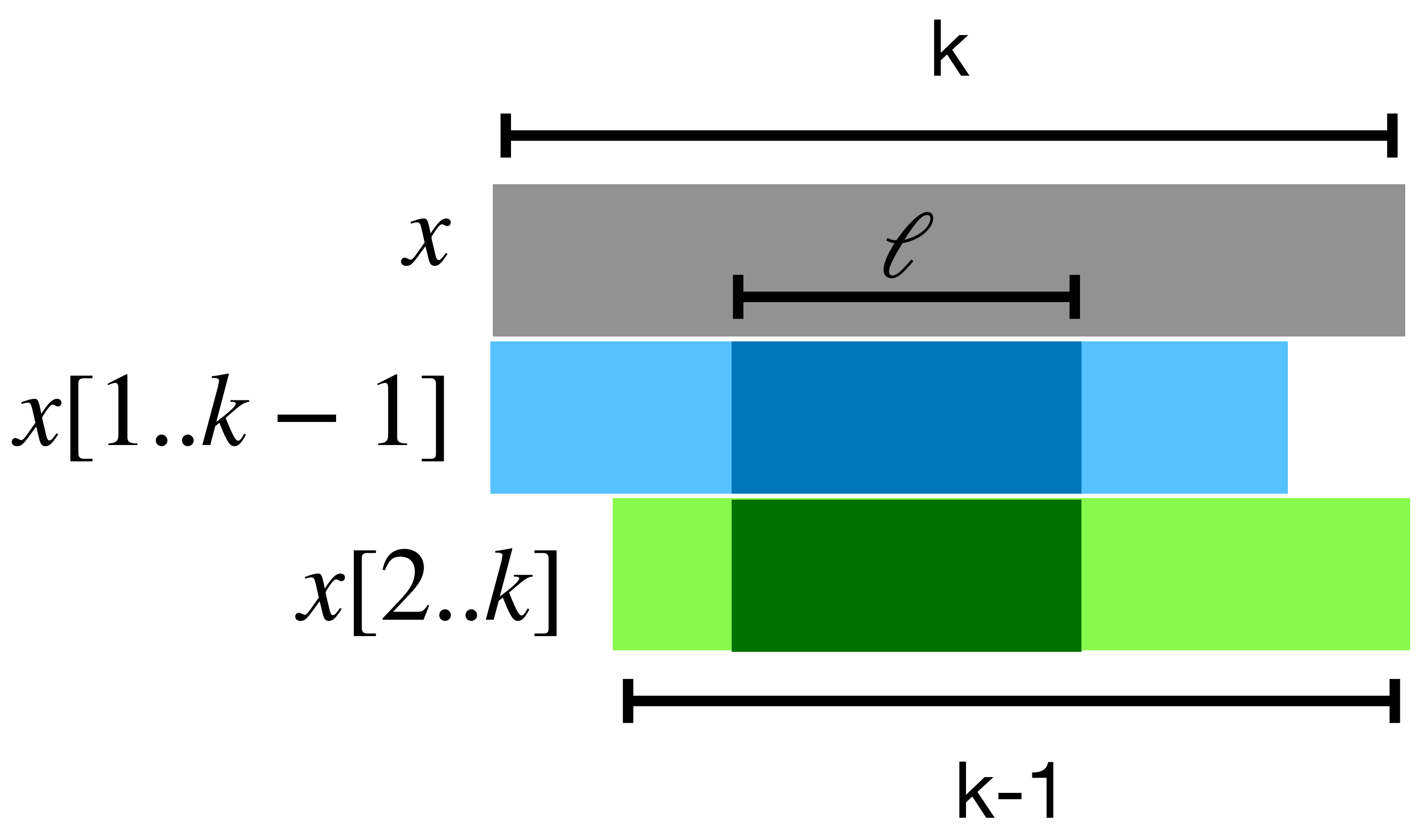
Left/right minimizers

- **Left/right minimizer.** For a k-mer x, let $lm(x)$ and $rm(x)$ be the left and right minimizer of x, defined as the minimizer of length $\ell \leq k-1$ of the (k-1)-length prefix/suffix of x, $x[1..k-1]$ and $x[2..k]$ respectively.



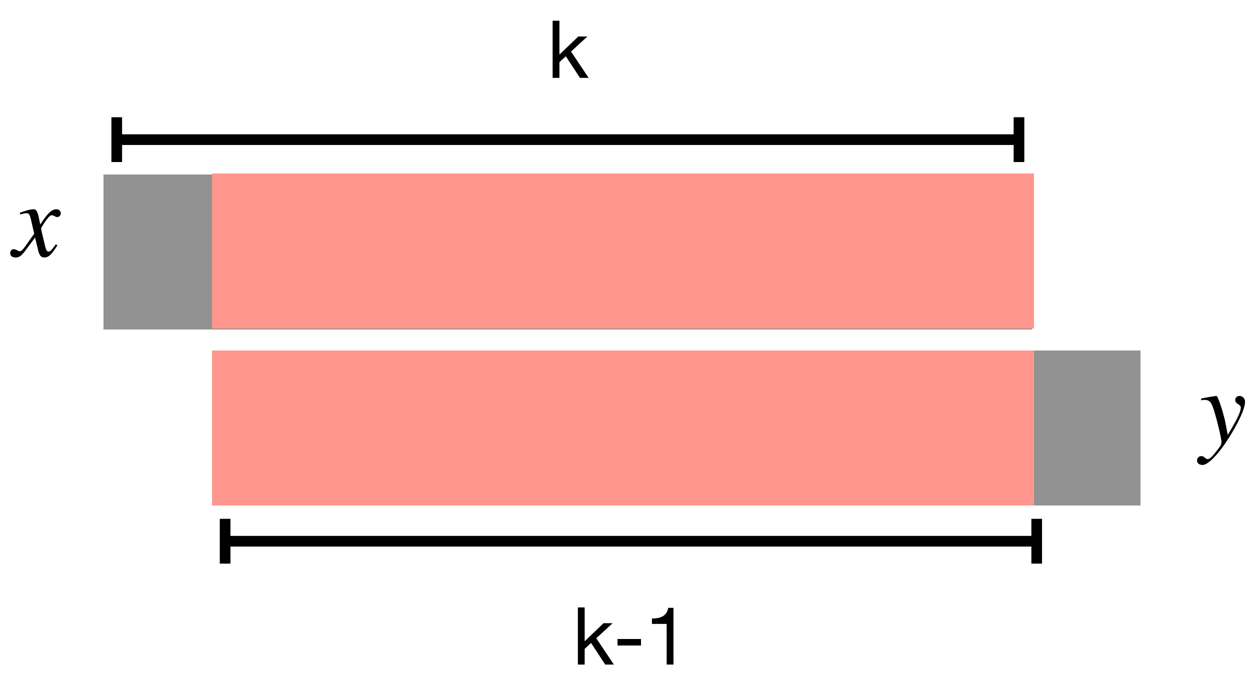
Left/right minimizers

- **Left/right minimizer.** For a k-mer x, let $lm(x)$ and $rm(x)$ be the left and right minimizer of x, defined as the minimizer of length $\ell \leq k-1$ of the (k-1)-length prefix/suffix of x, $x[1..k-1]$ and $x[2..k]$ respectively.



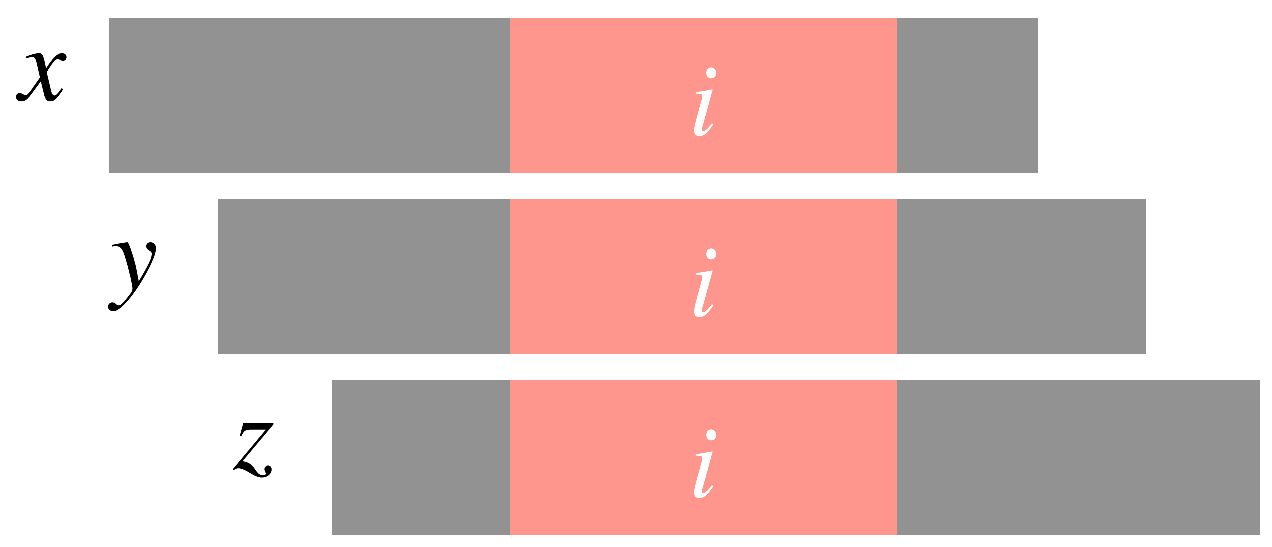
- Now, if x and y have a (k-1)-length overlap, they have at least a minimizer in common. (The viceversa is not true in general.)

- Here, surely $rm(x) = lm(y)$.



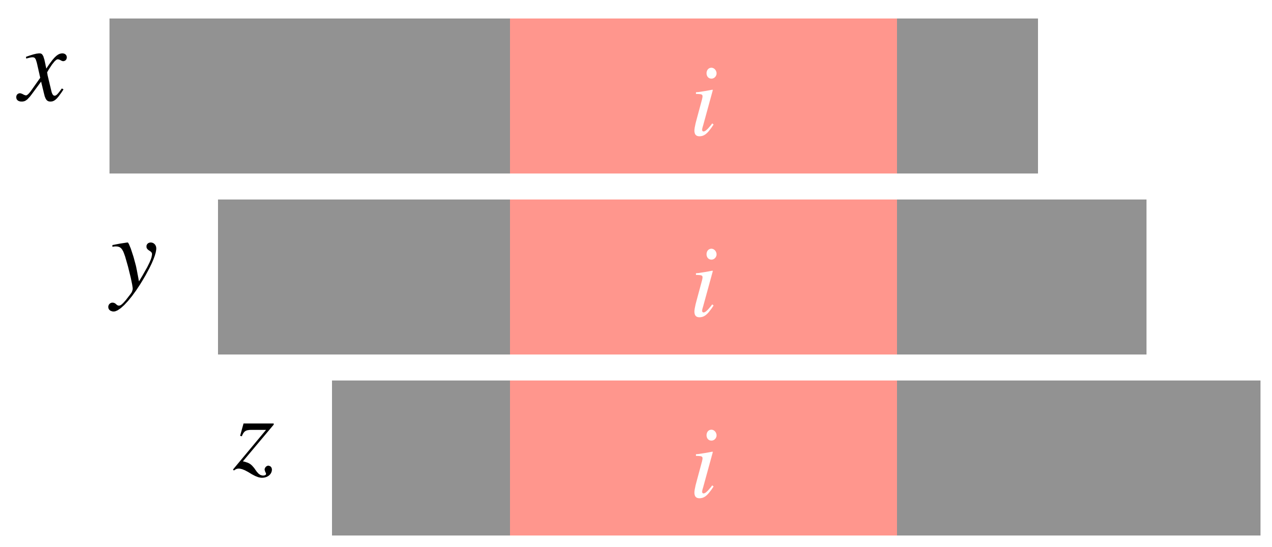
Distribution via left/right minimizers

- BCALM places k-mer x in bucket $i \in \{1..4^{\ell}\}$ if $i = lm(x)$, for a small $\ell > 0$ (like $\ell = 8$). If $j = rm(x)$, then x **also** goes to bucket j.
- The intuition is that, since consecutive k-mers tend to share the same minimizer, a bucket is likely to hold k-mers that are part of the same unitig.
- The parameter ℓ controls the size of the buckets.

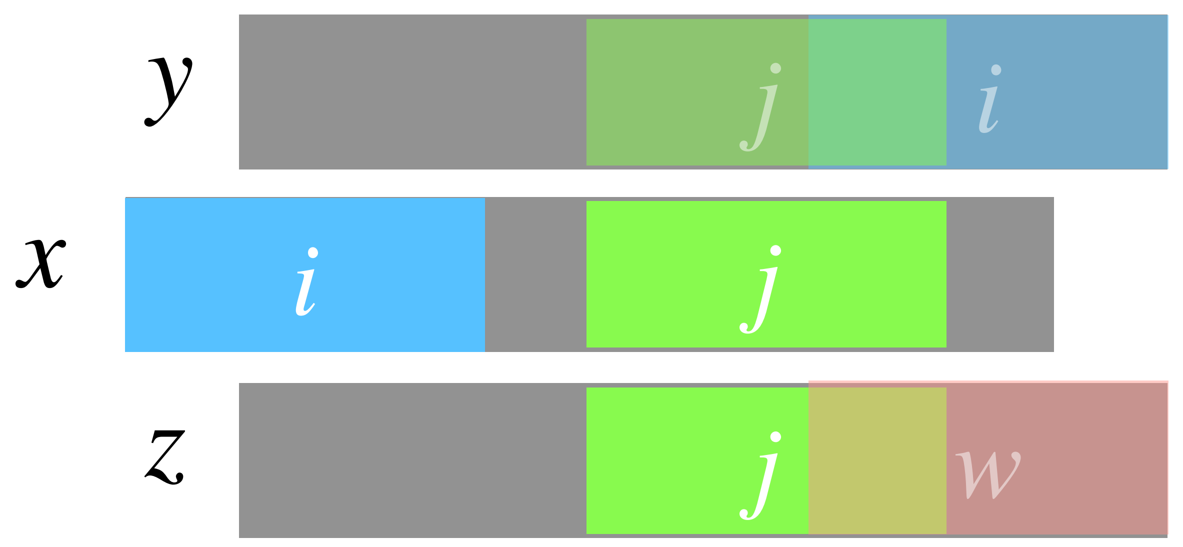


Distribution via left/right minimizers

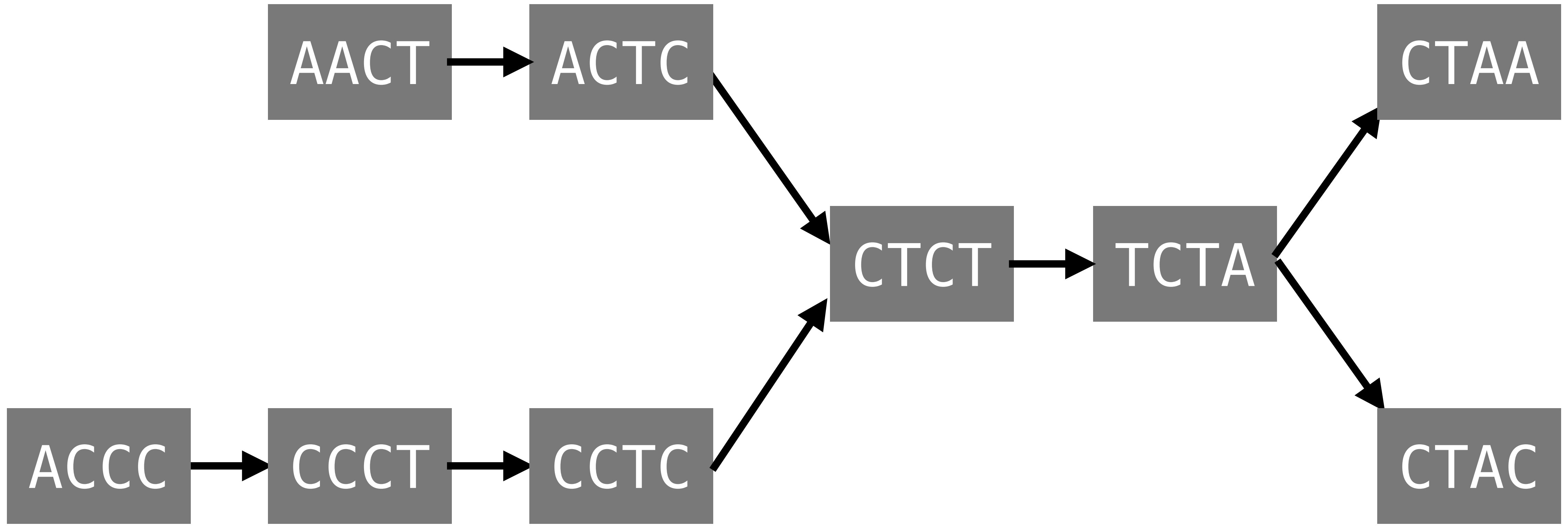
- BCALM places k-mer x in bucket $i \in \{1..4^\ell\}$ if $i = lm(x)$, for a small $\ell > 0$ (like $\ell = 8$). If $j = rm(x)$, then x **also** goes to bucket j.
- The intuition is that, since consecutive k-mers tend to share the same minimizer, a bucket is likely to hold k-mers that are part of the same unitig.
- The parameter ℓ controls the size of the buckets.



- Bucket i also contains k-mers x for which $rm(x) = j$. Such k-mers might potentially induce a **branch** with the k-mers belonging to bucket j (e.g., the k-mer z).
- We therefore only compact k-mers x and y in bucket i if they share an overlap **and** $i = rm(x) = lm(y)$. We do **not** compact x and y in the example. The k-mers x for which $lm(x) \neq rm(x)$ **will be at the end of a unitig**.

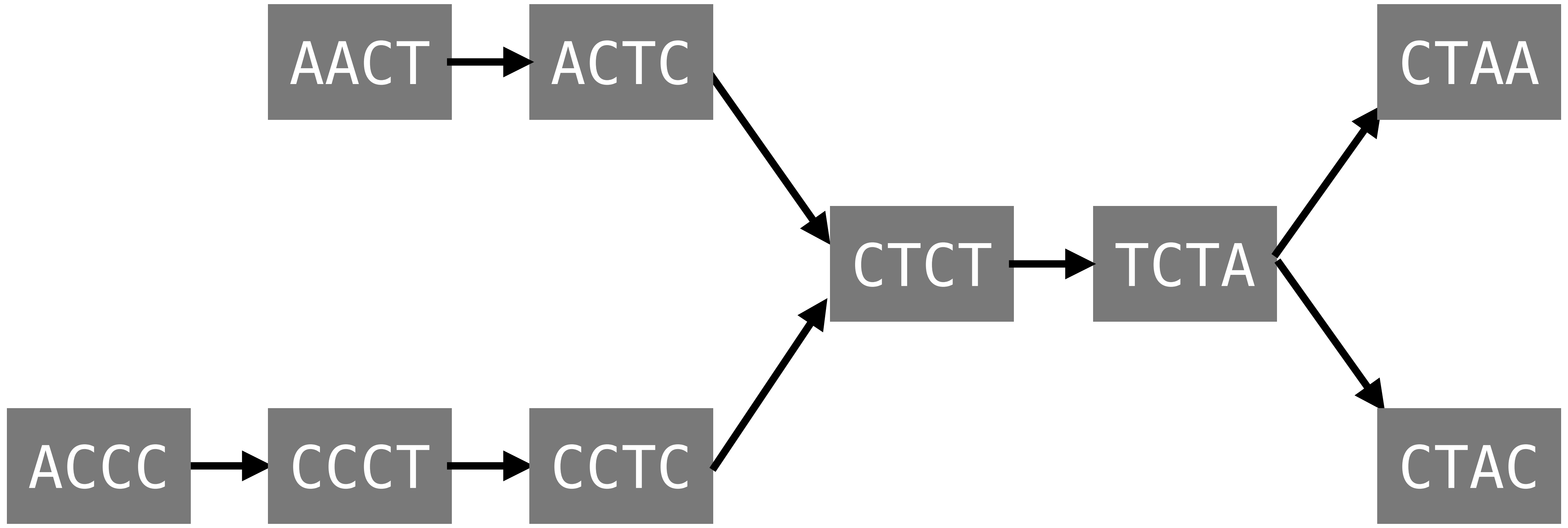


Distribution and compaction — Example



- Example for $k = 4$ and $\ell = 2$.
- (Empty buckets omitted.)

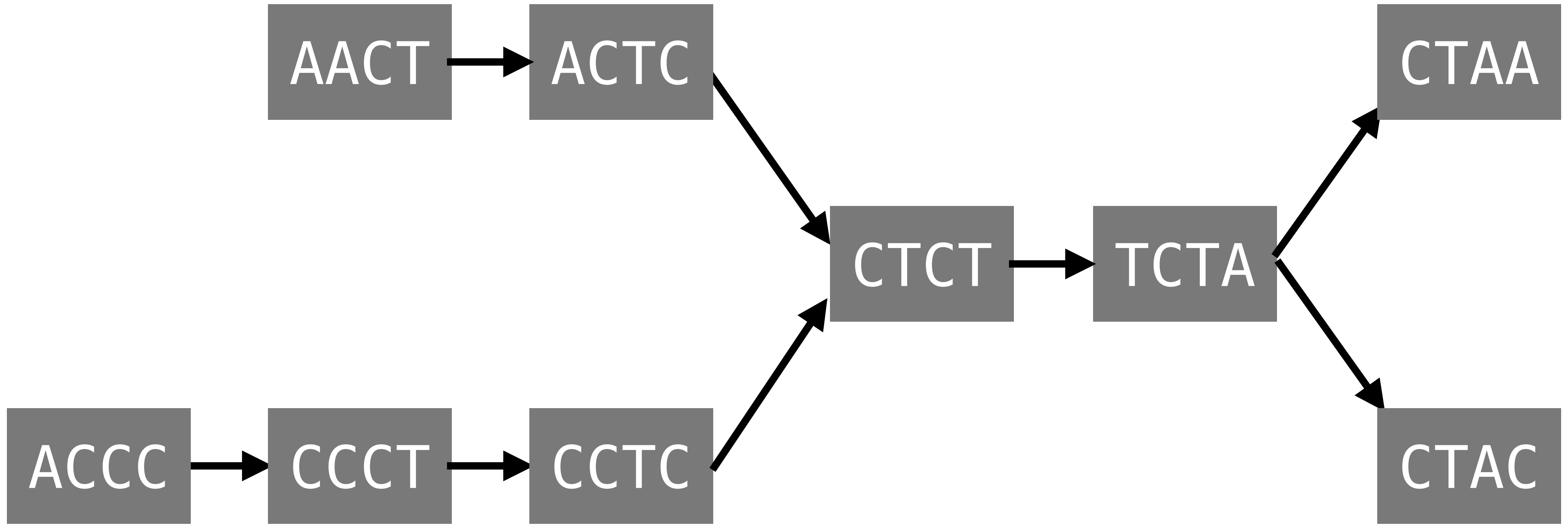
Distribution and compaction — Example



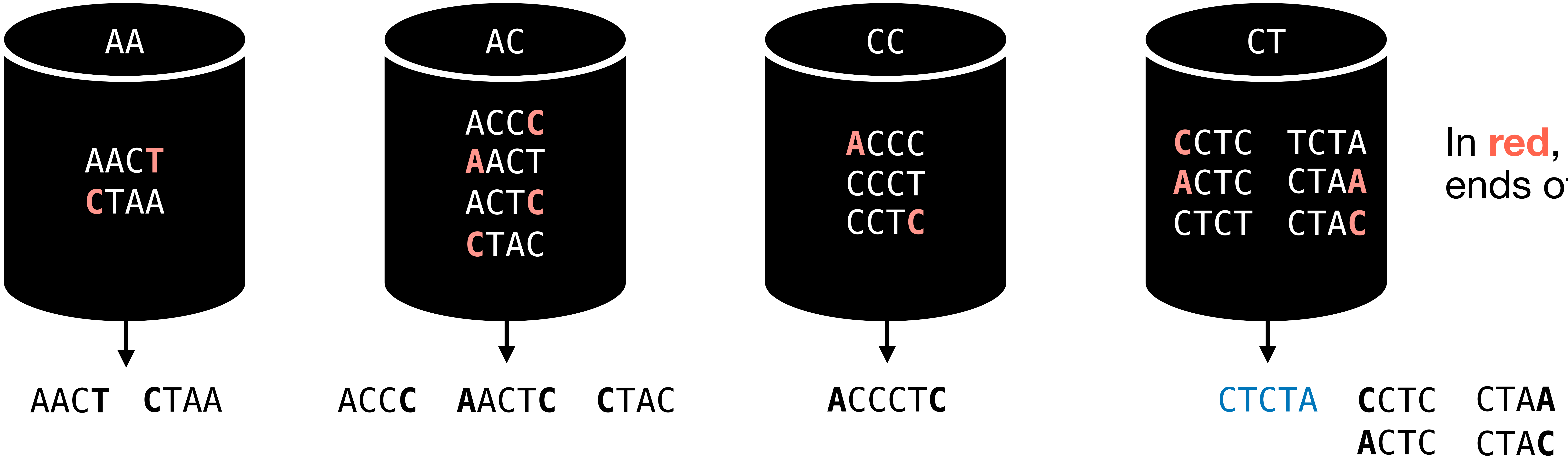
- Example for $k = 4$ and $\ell = 2$.
- (Empty buckets omitted.)

In red, we mark the ends of unitigs.

Distribution and compaction — Example



- Example for $k = 4$ and $\ell = 2$.
- (Empty buckets omitted.)



In red, we mark the ends of unitigs.

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

1	2	3	4	5	6
AACT	CTAA	ACCC	ACTC	CTAC	CCTC

AAC**T** **C**TAA	ACC**C** **A**ACT**C** **C**TAC	**A**CCCT**C**	**C**CTC **A**CTC	CTA**A** CTA**C**

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

1	2	3	4	5	6
AACT	CTAA	ACCC	ACTC	CTAC	CCTC

AAC**T**
CTAA

ACC**C**
AACT**C**
CTAC

ACCCT**C**

CCTC
ACTC

CTA**A**
CTA**C**

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

1	2	3	1	5	6
AACT	CTAA	ACCC	ACTC	CTAC	CCTC

AACT CTAA	ACCC AACTC CTAC	ACCCTC	CCTC ACTC	CTAA CTAC

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

1	2	3	1	5	6
AACT	CTAA	ACCC	ACTC	CTAC	CCTC

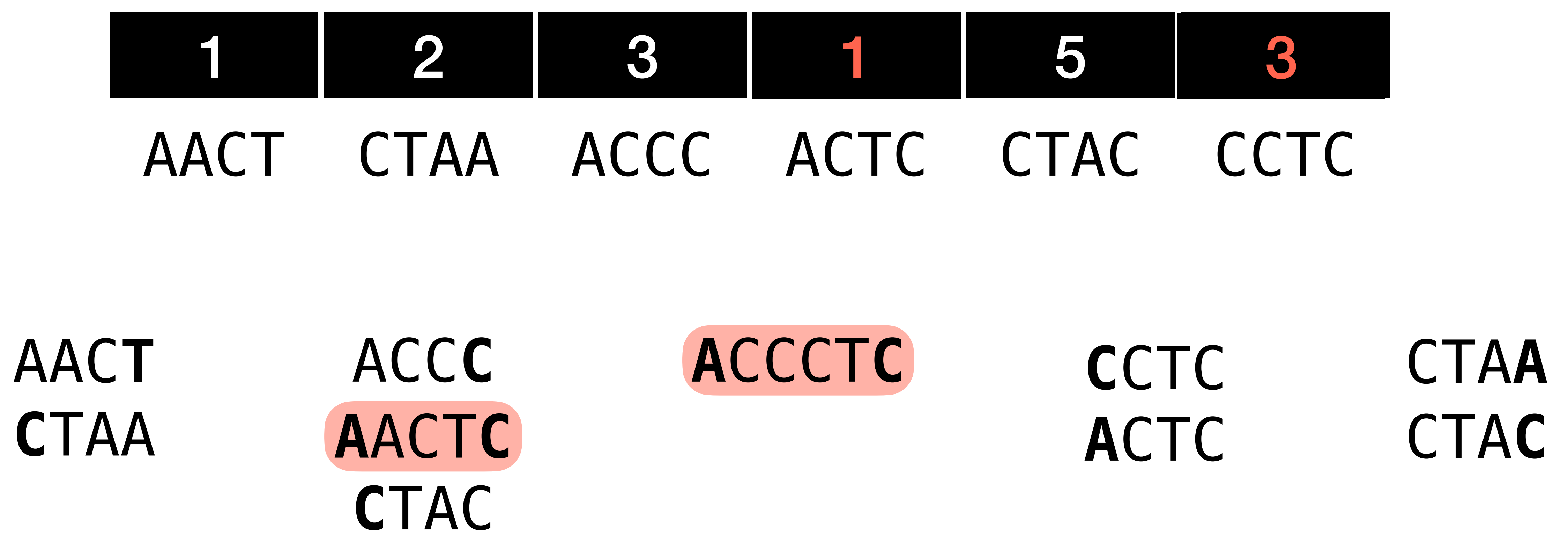
AACT	ACCC	ACCCTC	CCTC	CTAA
CTAA	AACTC		ACTC	CTAC
	CTAC			

Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.

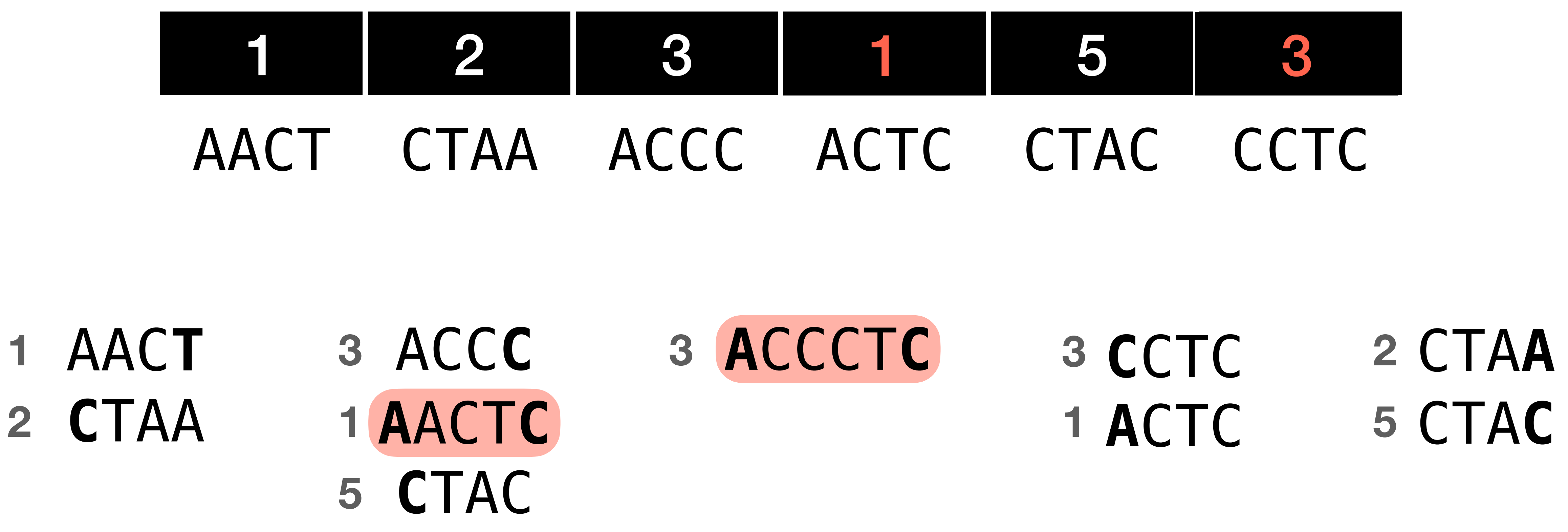
- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.

- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.

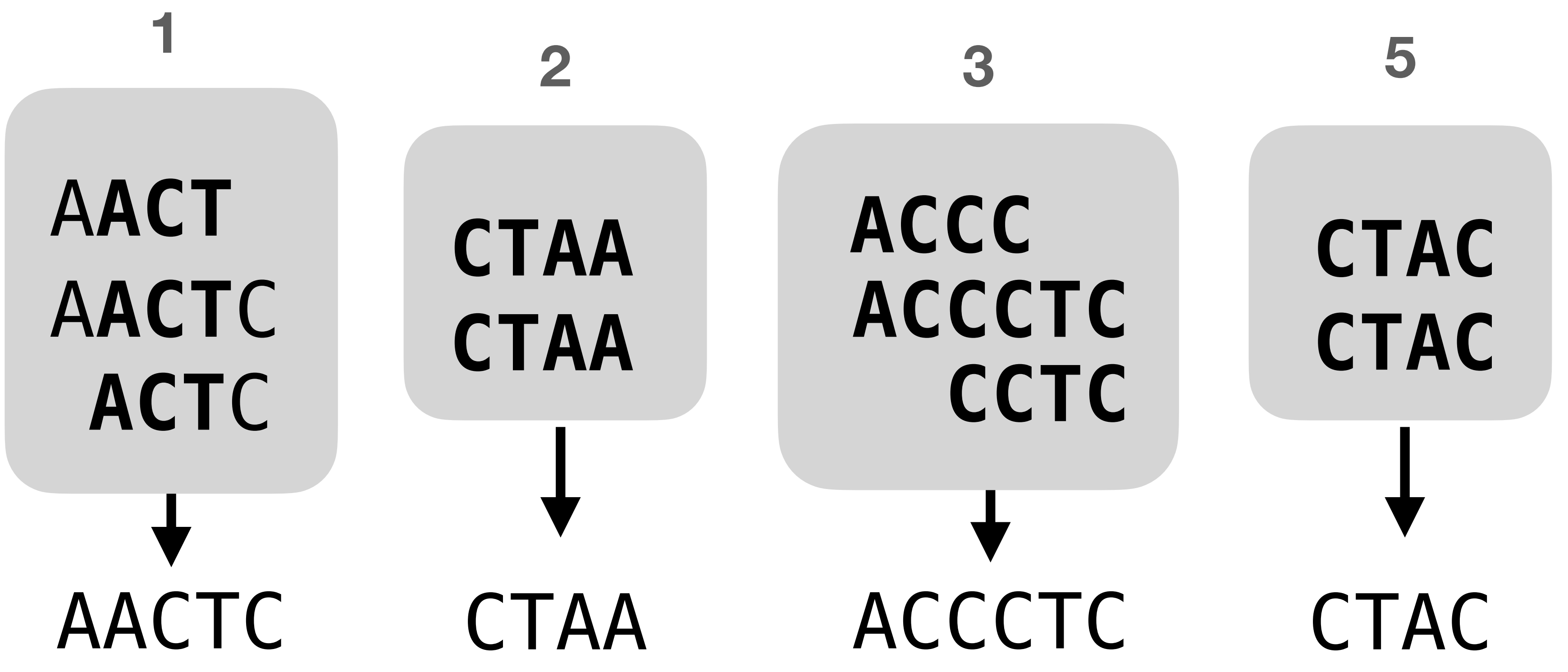


Linking non-maximal unitigs: union-find

- k-mers x for which $lm(x) \neq rm(x)$ exist in two copies, in bucket $lm(x)$ and in bucket $rm(x)$. They appear at the end of some unitig. These k-mers must be deduplicated to obtain the final maximal unitigs.
- The goal is to partition the unitigs into groups, each group containing the ones that should be compacted. Clearly, each group can be compacted independently in parallel.
- To achieve the partitioning, BCALM uses a **union-find data structure**: operation union is performed between the first and last k-mer of a unitig.



So we obtained 4 groups: 1, 2, 3, 5



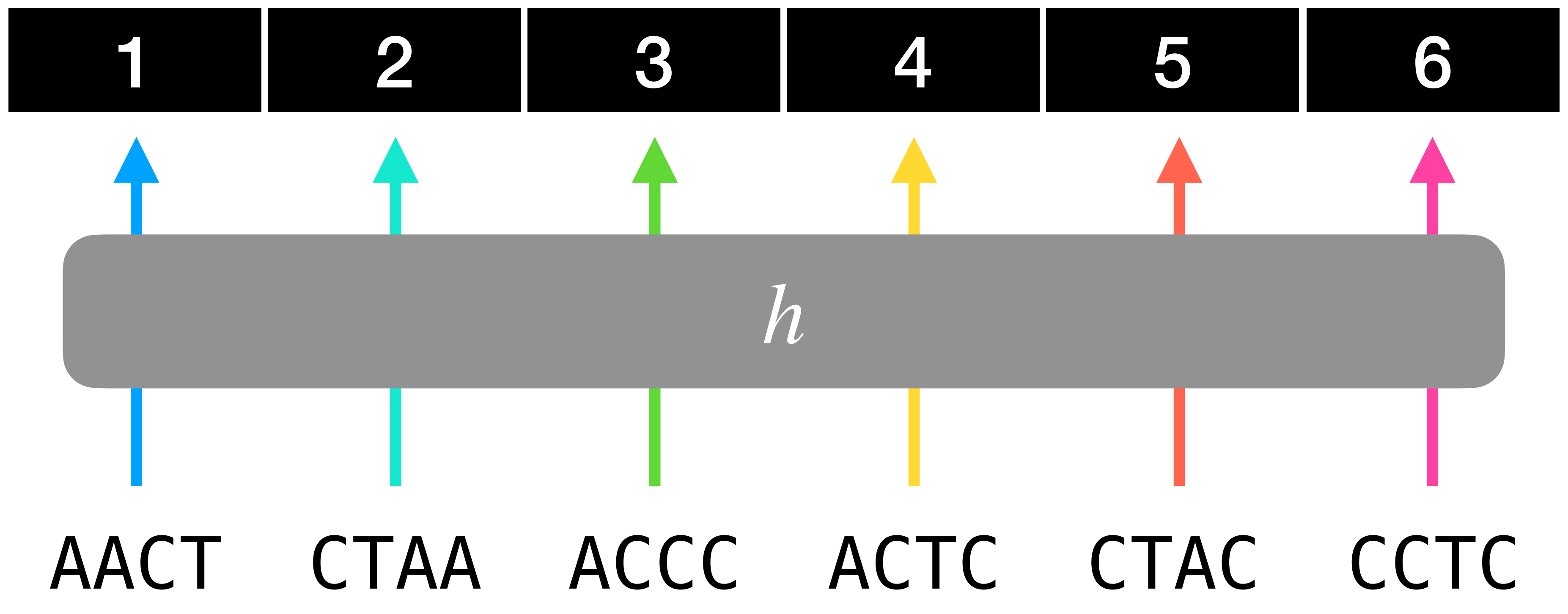
Performance of union-find algorithms

Algorithm	Union	Find
quick-find	n	1
quick-union	tree height	tree height
weighted quick-union	log n	log n
weighted quick-union with path compression	almost 1 (*inverse Ackermann*)	almost 1 (*inverse Ackermann*)

- Performance for n items.
- All algorithms use an array (or two) of size n as their data structure.
- From Chapter 1.5 of “Algorithms”, 4-th Ed., Sedgewick and Wayne.

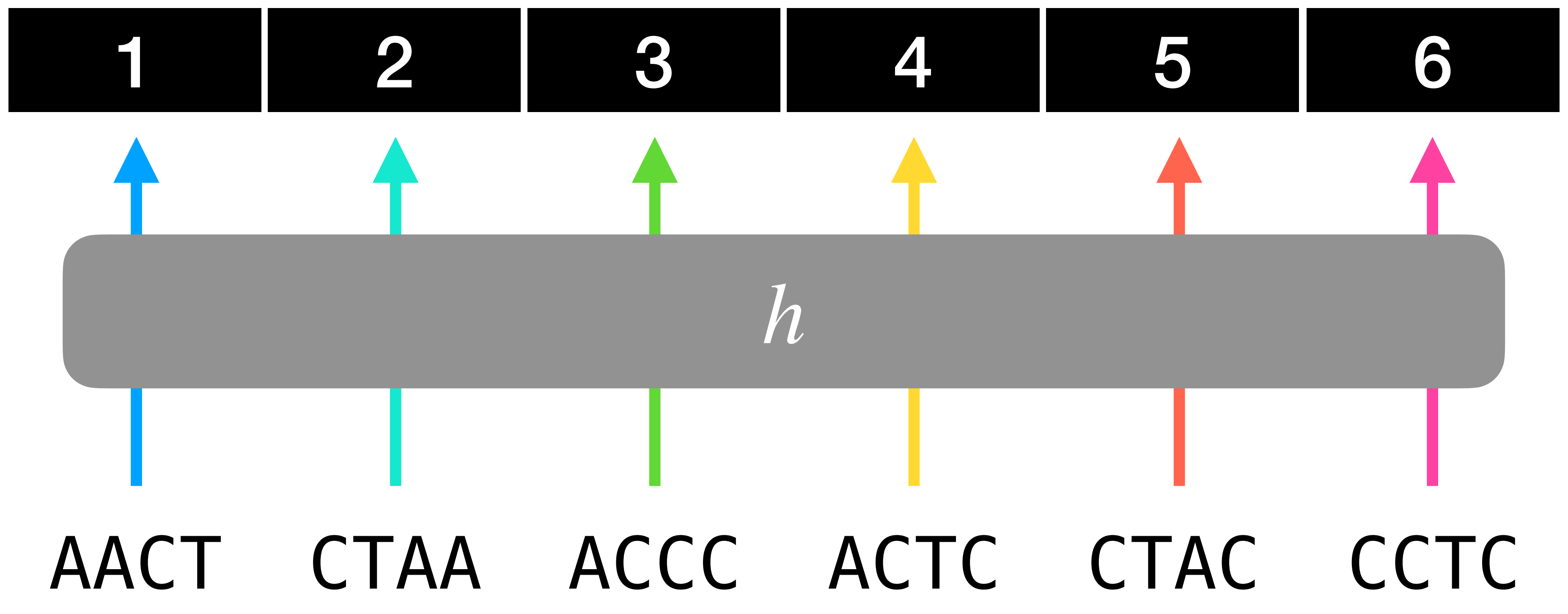
Union-find implementation detail

- A complicating matter for BCALM is that is not trivial to assign integer ids to k-mers, **using little space**, to implement union-find.
- A classic hash table could be used but it also stores the k-mers themselves.



Union-find implementation detail

- A complicating matter for BCALM is that is not trivial to assign integer ids to k-mers, **using little space**, to implement union-find.
- A classic hash table could be used but it also stores the k-mers themselves.
- But **minimal perfect hashing** can be used.
- That is, we build a function h that maps all the n distinct k-mers into the integers $[1..n]$ without collisions and without storing them at all!
- Such functions can be built very efficiently and with space usage between 1.8 — 3.0 bits per key (very close to optimal).



GGCAT

Cracco and Tomescu, 2023

- A refined and finely-engineered version of BCALM.
- **Idea 1.** Merge the k-mer counting step with unitig compaction. A problem in BCALM is that the input set of k-mers must be first materialised on disk.
- This first step takes a non-trivial amount of time (although k-mer counting algorithms are very efficient) and the final input set can take a large amount of disk space.
- **Idea 2.** Avoid union-find data structure but use a randomised approach.
- Many other important practical improvements, e.g., intermediate files are written to disk compressed, better multithreading, better hash tables for SIMPLE, etc.

Super-k-mers

- GGCAT avoids the k-mer counting step of BCALM by splitting the input strings of S into **super-(k-1)-mers** and inserting unique k-mers in the hash table used by SIMPLE.

- **Super-k-mer.** Given a string, a super-k-mer is a **maximal** sequence of consecutive k-mers having the same minimizer.

- Example for $k = 13$ and $\ell = 4$:

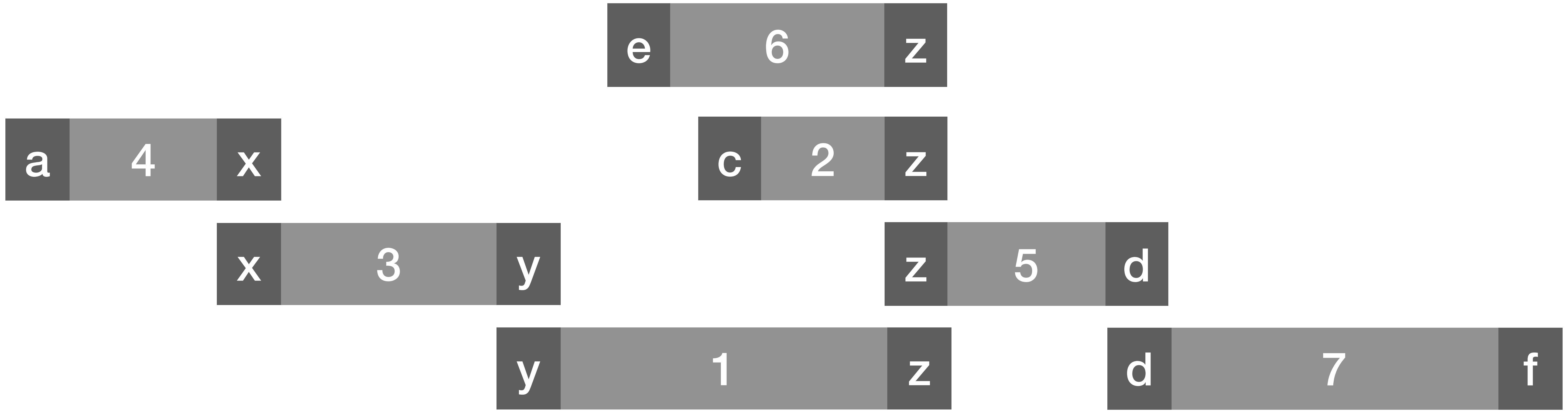
```
ACGGTAGAACCGATTCAAATTCGATCGATTAATTAGAGCGATAAC...
ACGGTAGAACCGA
 CGGTAGAACCGAT
  GGTAGAACCGATT
   GTAGAACCGATTC
    TAGAACCGATTCA
     AGAACCGATTCAA
      GAACCGATTCAAA
       AACCGATTCAAAT
        ...
```

super-k-mer

- A super-k-mer is a space-efficient representation for its set of k-mers.

- Buckets in GGCAT are made of **super-(k-1)-mers** and not of individual k-mers which reduces space on disk (by a lot!).

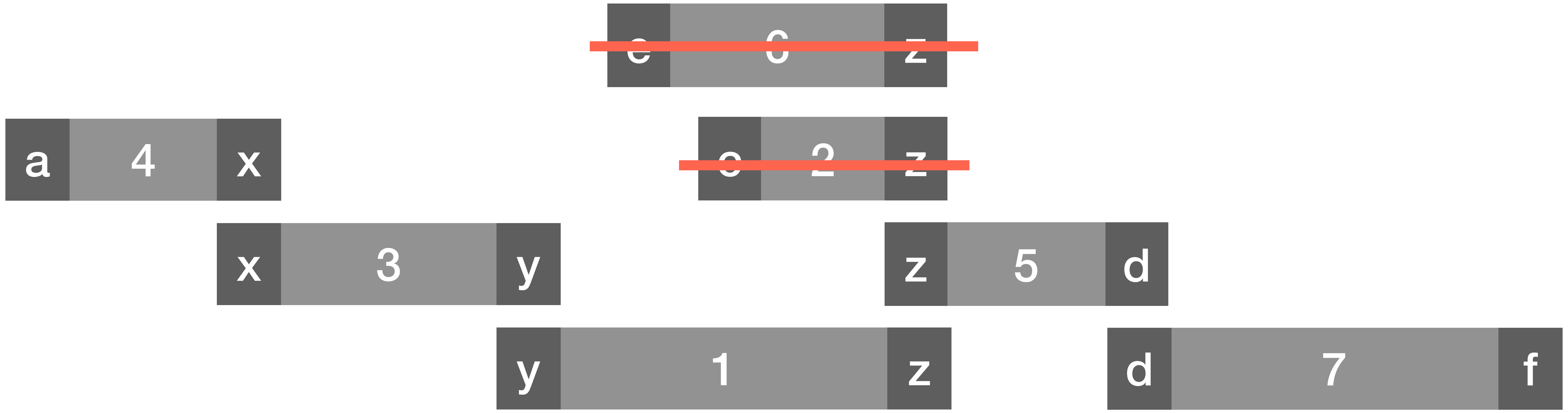
Linking non-maximal unitigs



- We create a list of pairs (first/last k-mer, unitig-id), sorted by k-mer:

  L = [(a,4), (c,2), (d,5), (d,7), (e,6), (f,7) (x,4), (x,3), (y,3), (y,1), (z,1), (z,2), (z,5), (z,6)].

Linking non-maximal unitigs



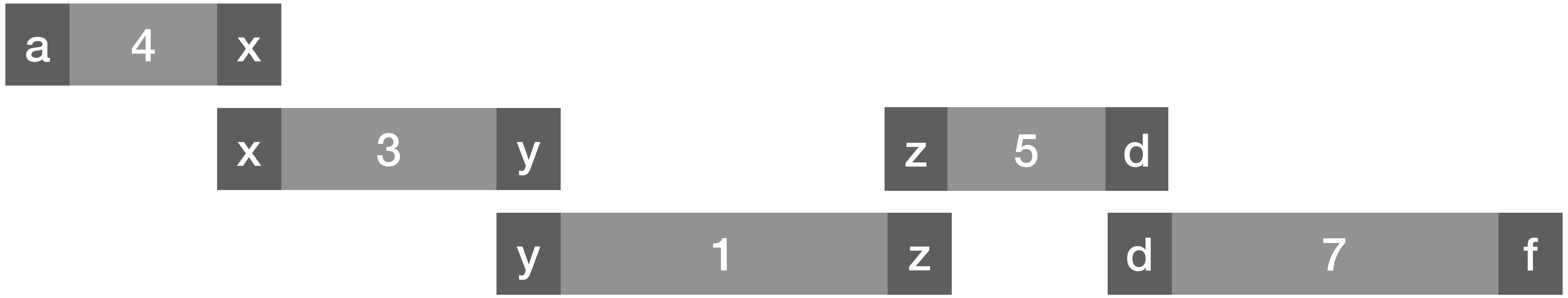
- We create a list of pairs (first/last k-mer, unitig-id), sorted by k-mer:

  L = [(a,4), (c,2), (d,5), (d,7), (e,6), (f,7) (x,4), (x,3), (y,3), (y,1), (z,1), (z,2), (z,5), (z,6)].

- And only unitig-id pairs that should be merged are retained: L = [(5,7), (3,4), (1,3)].

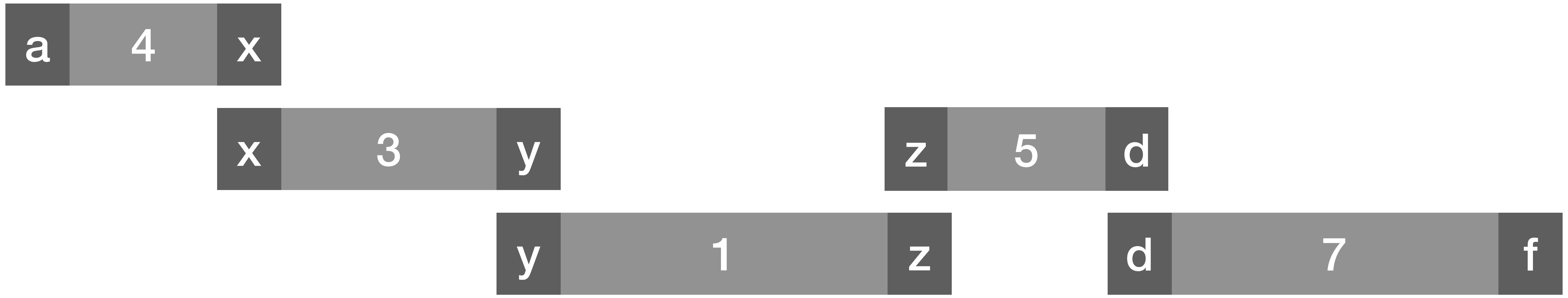
- At the beginning all unitig ids are marked as **unsealed**.

Linking non-maximal unitigs: randomization

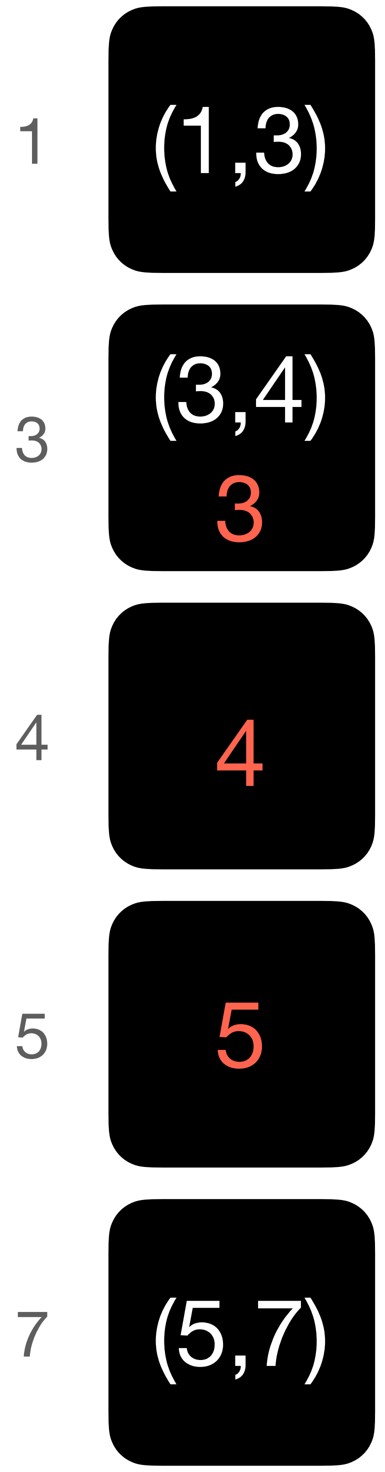


L = [(5,7), (3,4), (1,3)]

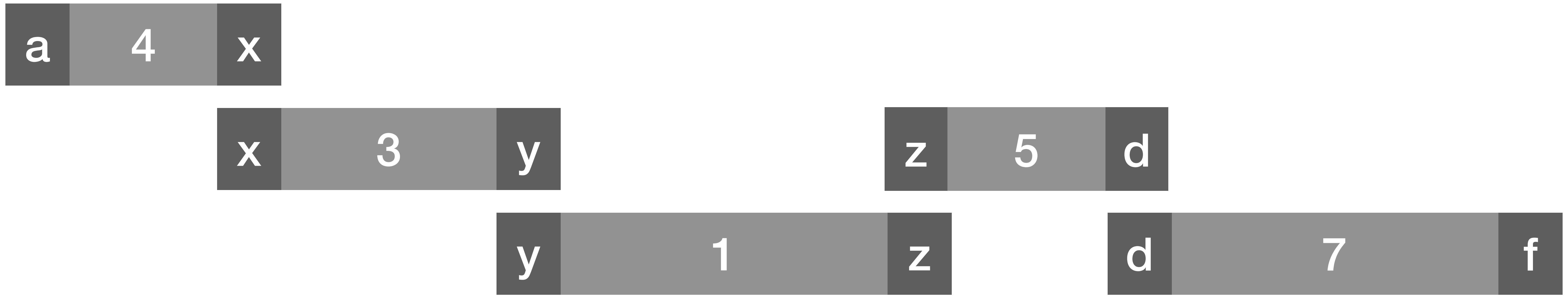
Linking non-maximal unitigs: randomization



L = [(5,7), (3,4), (1,3)]



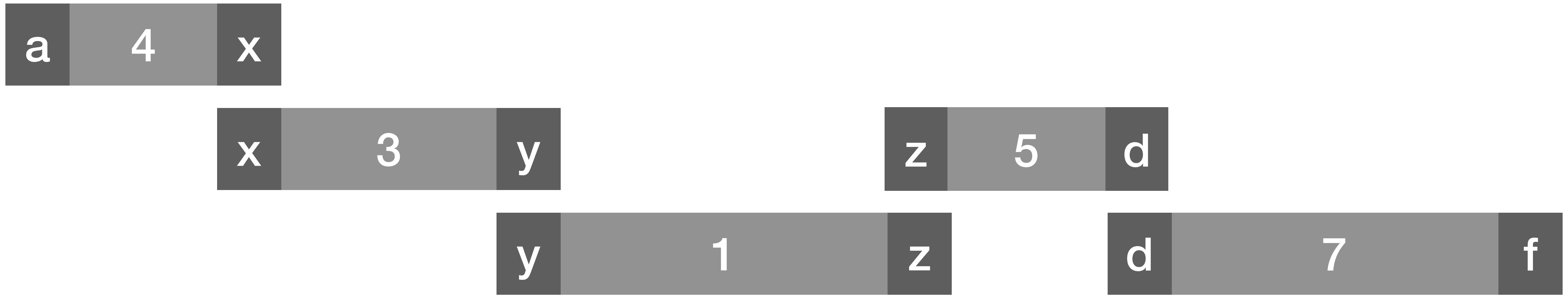
Linking non-maximal unitigs: randomization



L = [(5,7), (3,4), (1,3)]

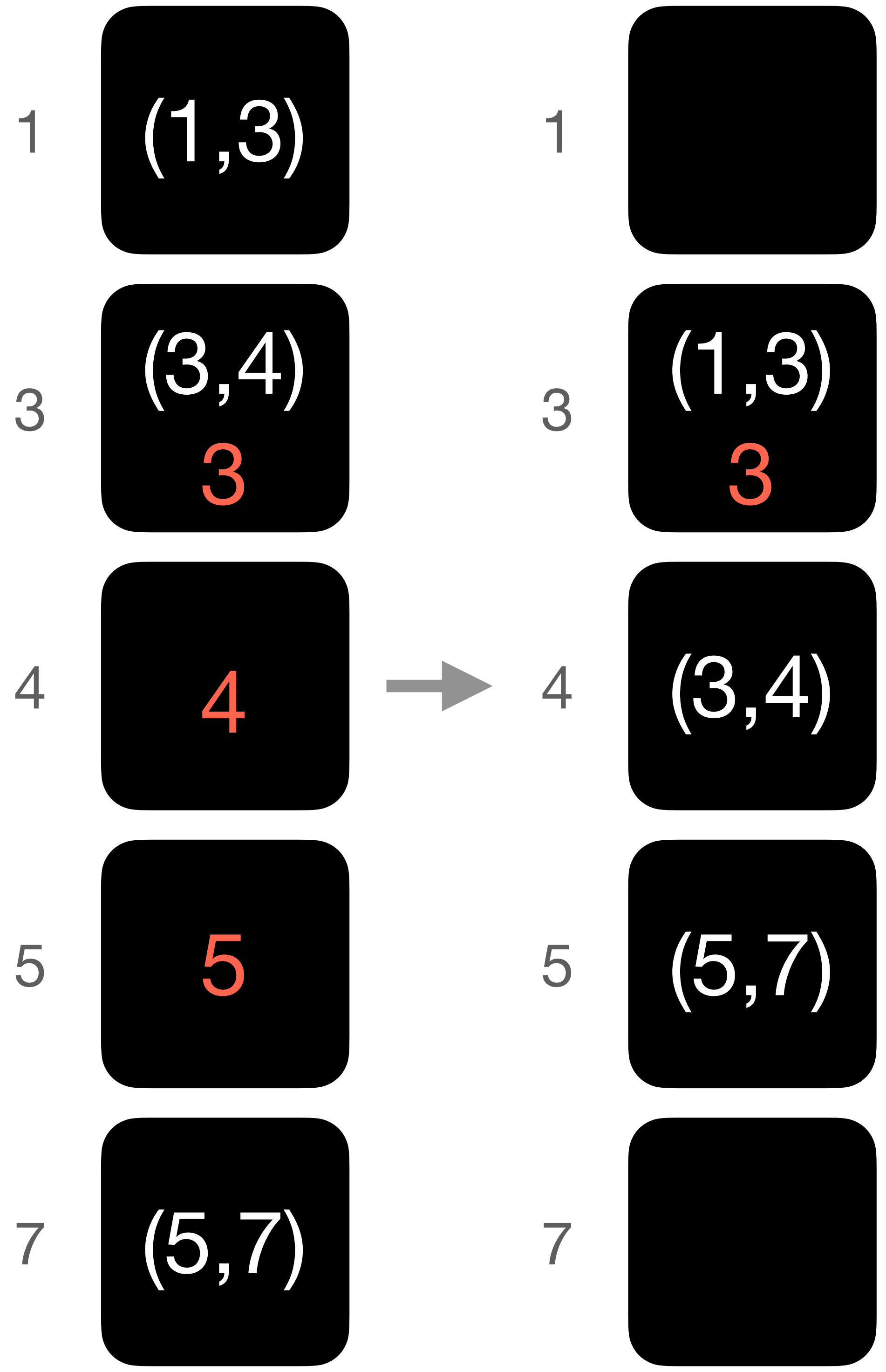
L = [(5,**7**), (3,4), (**1**,3)]

Linking non-maximal unitigs: randomization

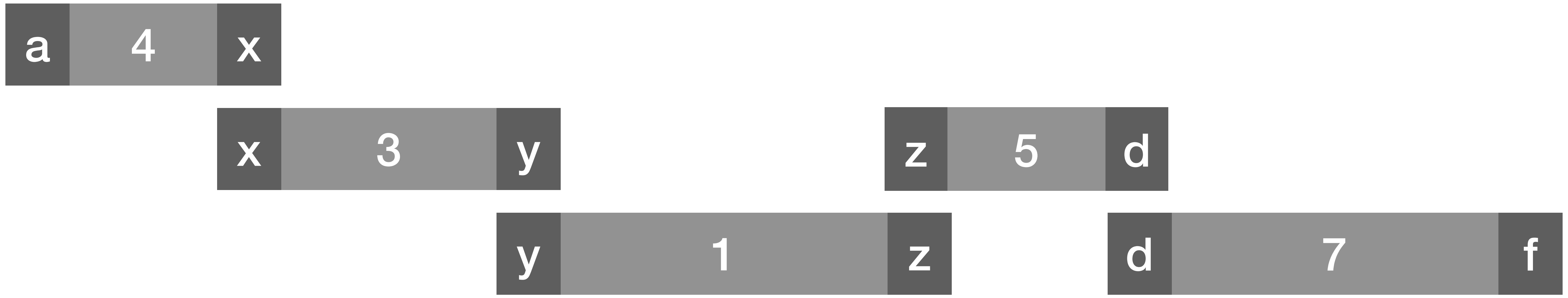


L = [(5,7), (3,4), (1,3)]

L = [(5,**7**), (3,4), (**1**,3)]



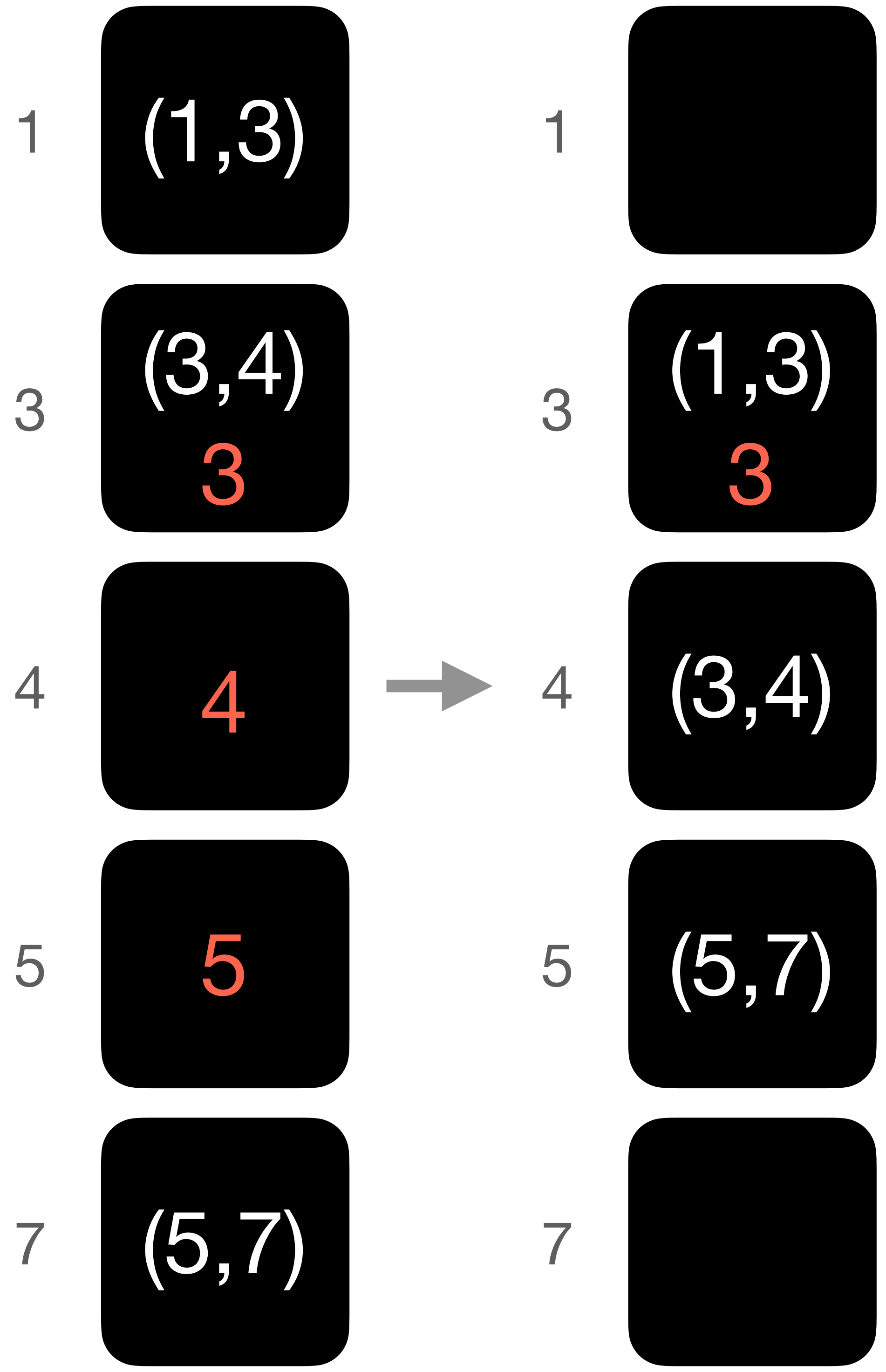
Linking non-maximal unitigs: randomization



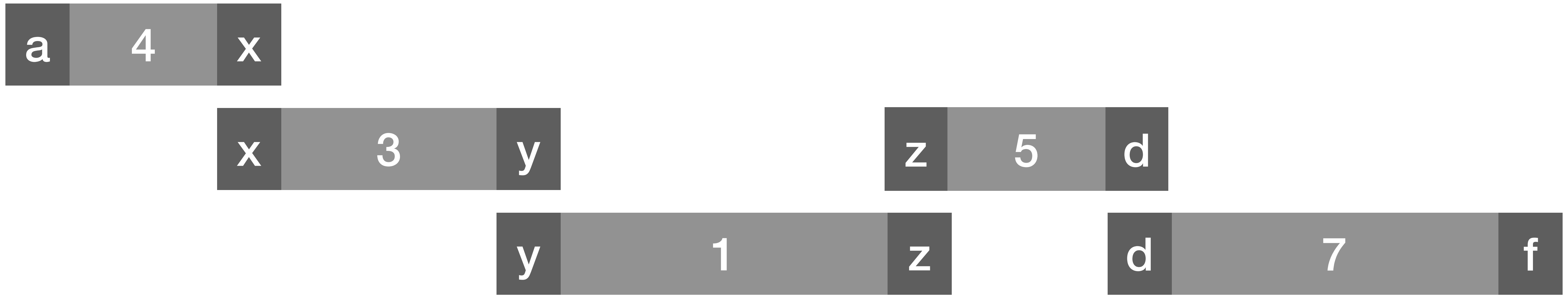
L = [(5,7), (3,4), (1,3)]

L = [(5,**7**), (3,4), (**1**,3)]

L = [(3,**4**)]



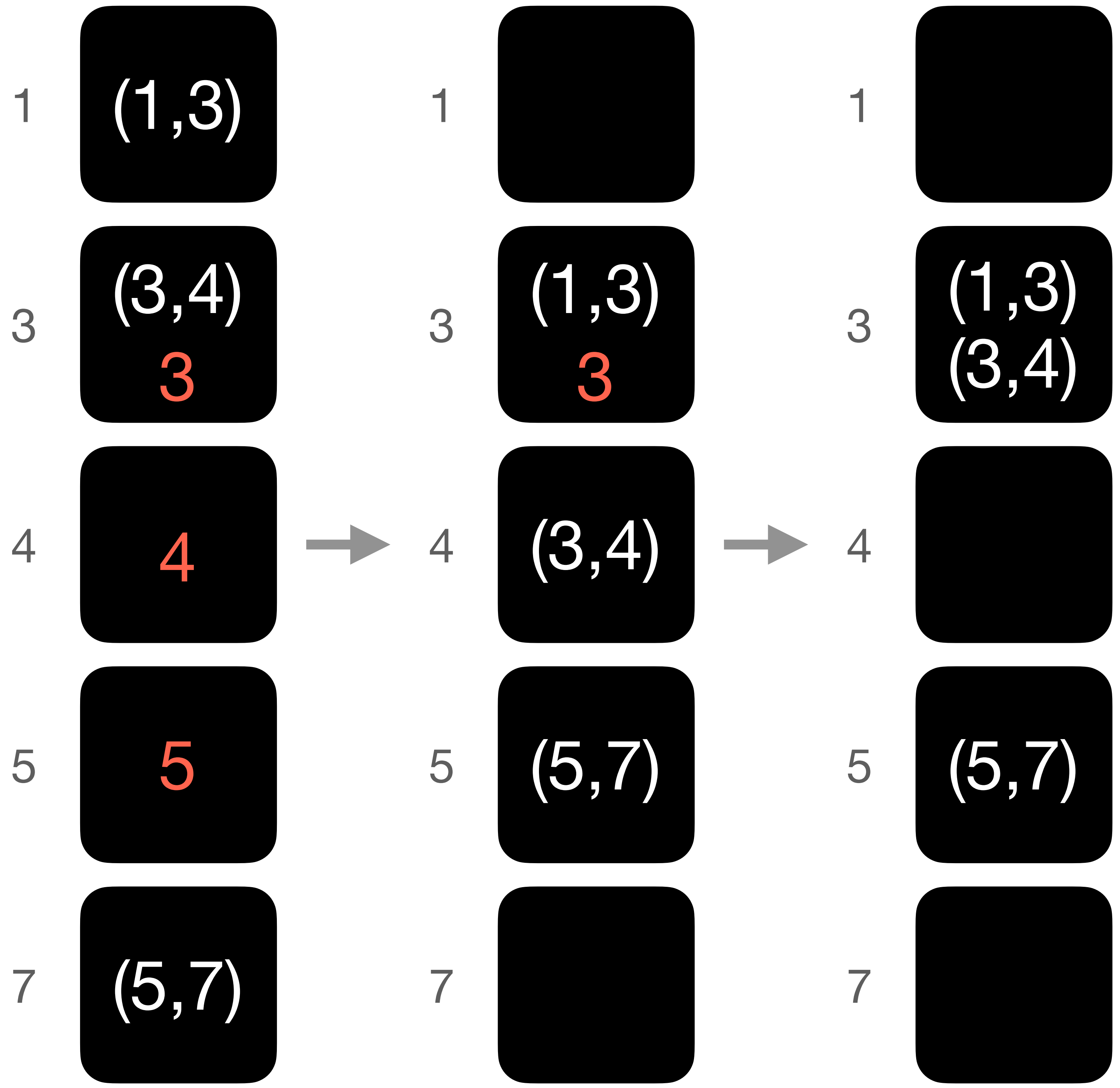
Linking non-maximal unitigs: randomization



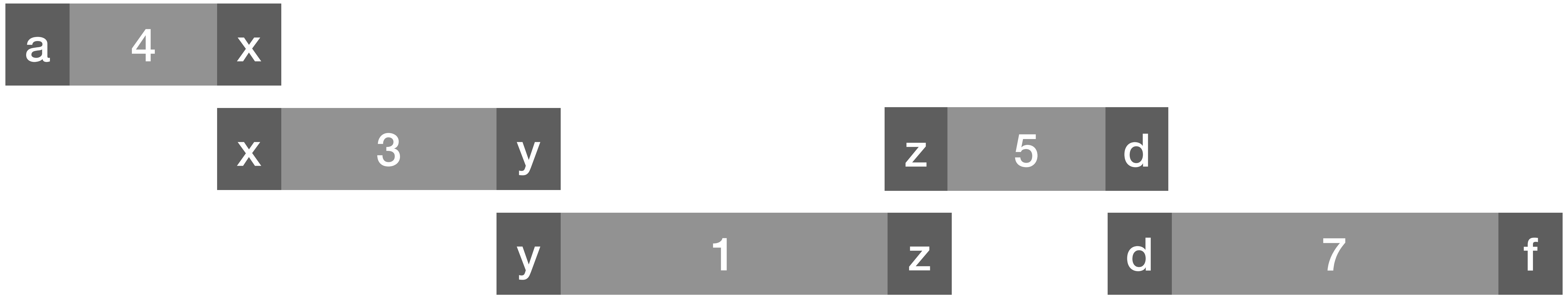
L = [(5,7), (3,4), (1,3)]

L = [(5,**7**), (3,4), (**1**,3)]

L = [(3,**4**)]



Linking non-maximal unitigs: randomization

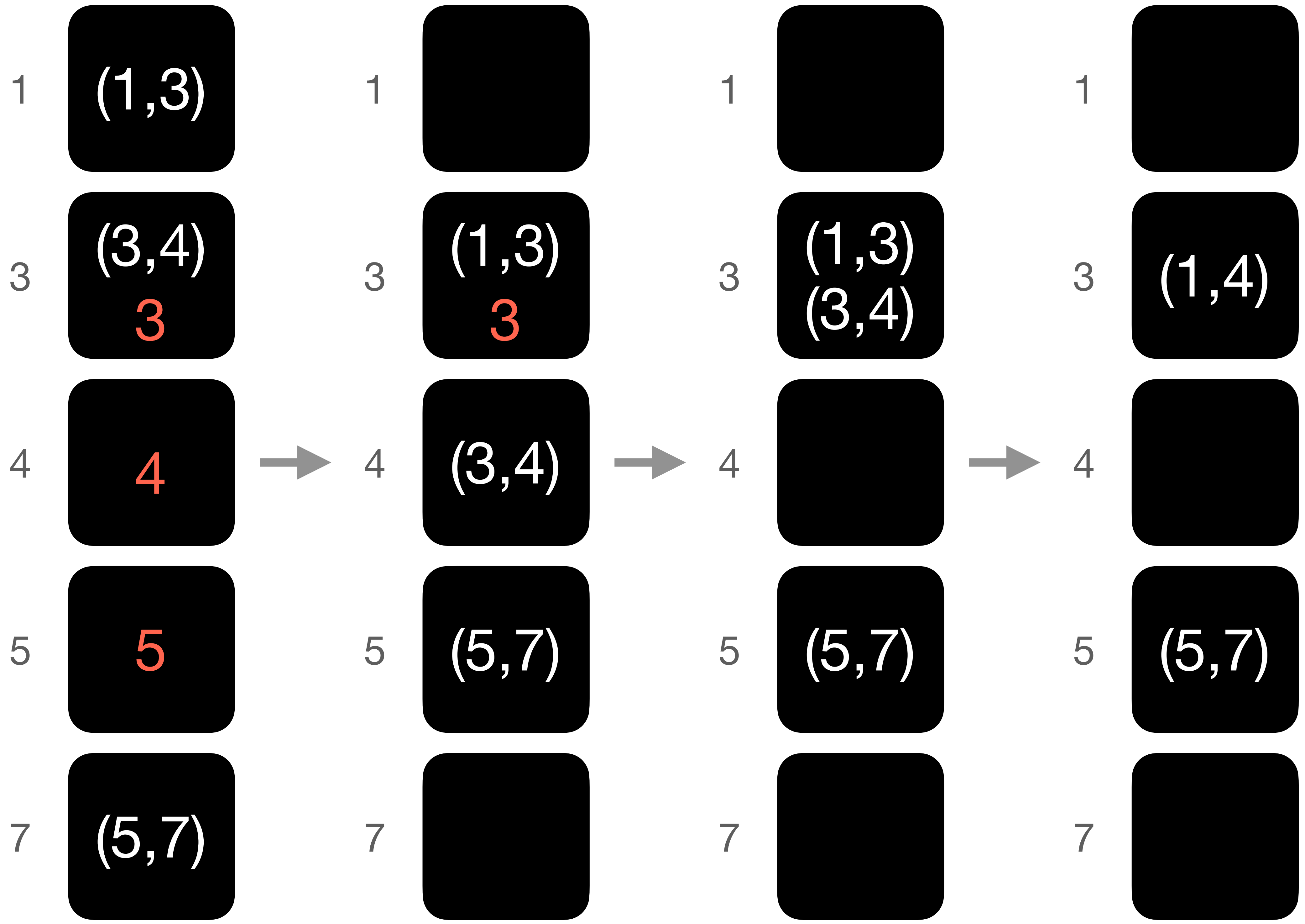


L = [(5,7), (3,4), (1,3)]

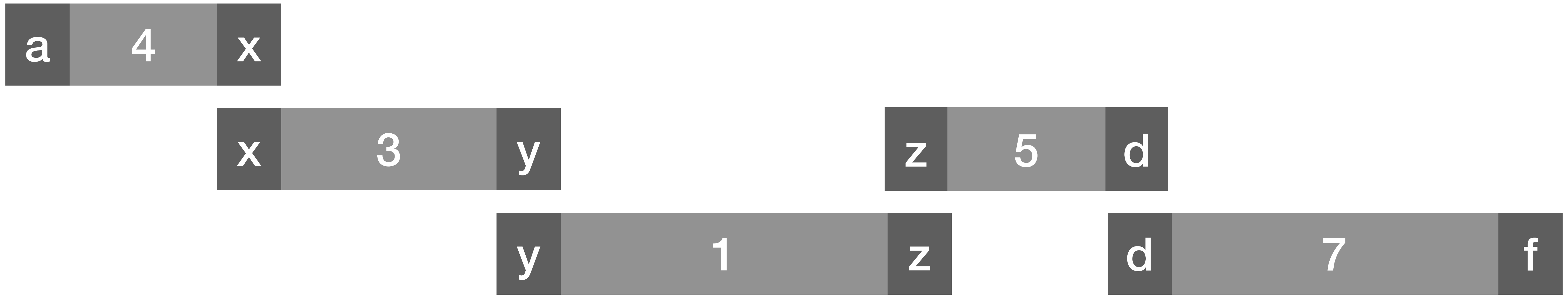
L = [(5,**7**), (3,4), (**1**,3)]

L = [(3,**4**)]

L = [ ]



Linking non-maximal unitigs: randomization

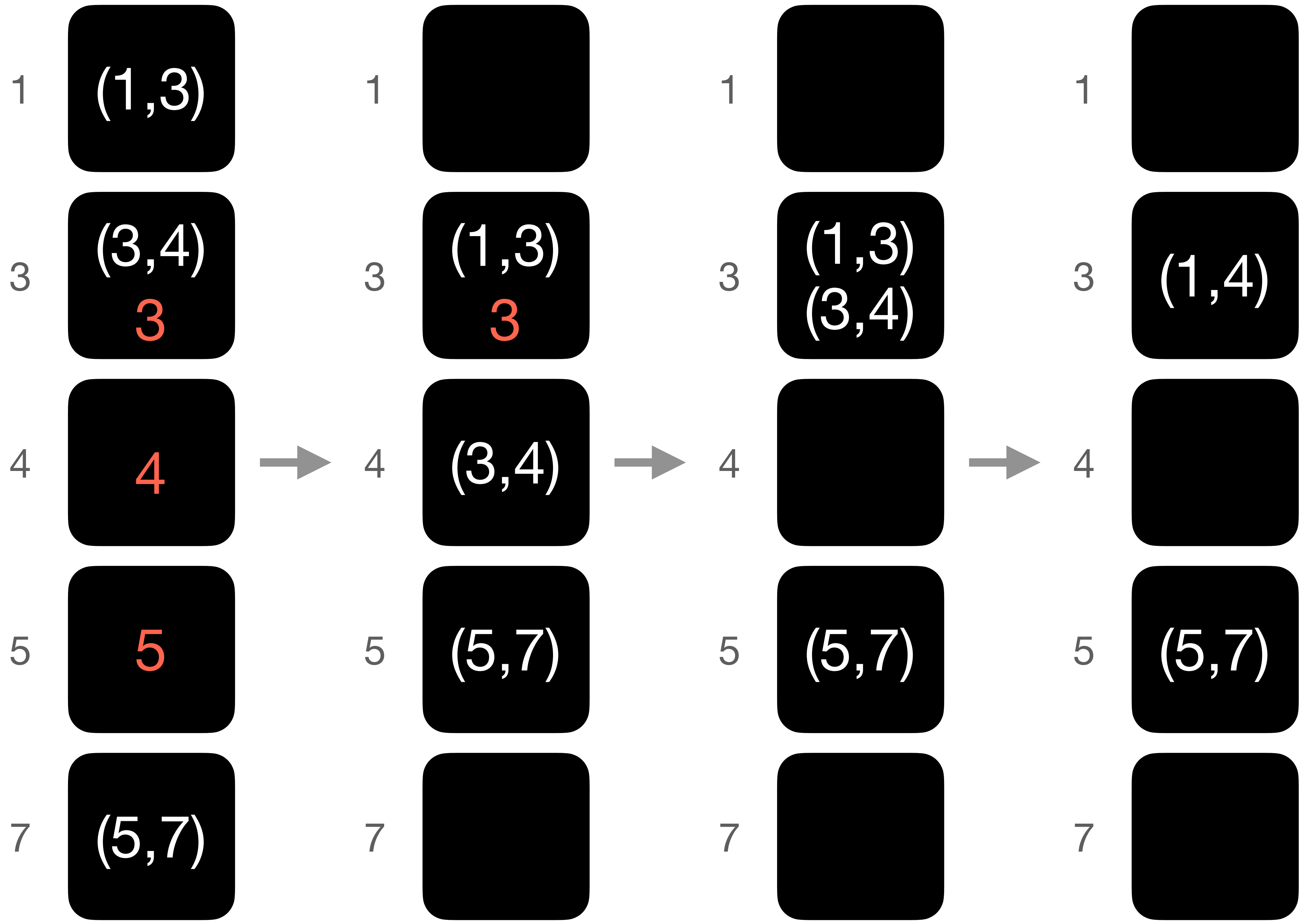


L = [(5,7), (3,4), (1,3)]

L = [(5,**7**), (3,4), (**1**,3)]

L = [(3,**4**)]

L = [ ]



- Since each element from a pair is selected **uniformly at random**, given two pairs that should be merged, they will end up in the same bucket, with probability **at least 1/4**.
- Thus, unitigs that have to be linked will end up in the same bucket after just four steps in expectation.

Performance

A portion of Table 1 from [Khan et al., 2022].

Dataset	k	Thread-count	BCALM 2	CUTTLEFISH 2
Human	27	8	04 h 23 min (6.7)	**01 h 13 min (3.2)**
		16	04 h 58 min (8.9)	**56 min (3.3)**
	55	8	04 h 01 min (7.4)	**02 h 20 min (3.5)**
		16	04 h 26 min (10.5)	**02 h 02 min (3.7)**
Human RNA-seq	27	8	02 h 58 min (3.8)	**30 min (2.9)**
		16	02 h 46 min (3.9)	**20 min (3.0)**
Gut microbiome	27	16	02 h 34 min (7.7)	**26 min (3.5)**
	55		03 h 02 min (12.5)	**44 min (4.0)**

A portion of Table 2
from [Cracco and Tomescu, 2023].

Data set	Server	k	Cuttlefish 2	GGCAT
Human reads	*Small*	27	1 h:15 min (3.95 GB) [209 GB]	1 h:16 min (4.54 GB) [220 GB]
		63	2 h:07 min (4.23 GB) [140 GB]	1 h:03 min (7.11 GB) [156 GB]
Gut microbiome reads	*Small*	27	0 h:30 min (3.35 GB) [78 GB]	0 h:22 min (6.09 GB) [78 GB]
		63	1 h:08 min (3.86 GB) [107 GB]	0 h:19 min (5.42 GB) [51 GB]
		119	1 h:04 min (3.13 GB) [97 GB]	0 h:12 min (5.33 GB) [32 GB]
Salmonella genomes (309 K)	*Small*	27	6 h:59 min (4.38 GB) [1515 GB]	3 h:38 min (3.46 GB) [378 GB]
		63	12 h:02 min (3.88 GB) [1145 GB]	3 h:31 min (3.96 GB) [274 GB]
		119	17 h:07 min (3.95 GB) [1088 GB]	3 h:39 min (4.12 GB) [279 GB]
		255	77 h:58 min (4.82 GB) [1056 GB]	3 h:44 min (4.33 GB) [325 GB]

Performance

A portion of Table 1 from [Khan et al., 2022].

Dataset	k	Thread-count	BCALM 2	CUTTLEFISH 2
Human	27	8	04 h 23 min (6.7)	**01 h 13 min** (**3.2**)
		16	04 h 58 min (8.9)	**56 min** (**3.3**)
	55	8	04 h 01 min (7.4)	**02 h 20 min** (**3.5**)
		16	04 h 26 min (10.5)	**02 h 02 min** (**3.7**)
Human RNA-seq	27	8	02 h 58 min (3.8)	**30 min** (**2.9**)
		16	02 h 46 min (3.9)	**20 min** (**3.0**)
Gut microbiome	27	16	02 h 34 min (7.7)	**26 min** (**3.5**)
	55		03 h 02 min (12.5)	**44 min** (**4.0**)

BCALM is outperformed by CUTTLEFISH which is, in turn, outperformed by GGCAT!

A portion of Table 2
from [Cracco and Tomescu, 2023].

Data set	Server	k	Cuttlefish 2	GGCAT
Human reads	*Small*	27	1 h:15 min (3.95 GB) [209 GB]	1 h:16 min (4.54 GB) [220 GB]
		63	2 h:07 min (4.23 GB) [140 GB]	1 h:03 min (7.11 GB) [156 GB]
Gut microbiome reads	*Small*	27	0 h:30 min (3.35 GB) [78 GB]	0 h:22 min (6.09 GB) [78 GB]
		63	1 h:08 min (3.86 GB) [107 GB]	0 h:19 min (5.42 GB) [51 GB]
		119	1 h:04 min (3.13 GB) [97 GB]	0 h:12 min (5.33 GB) [32 GB]
Salmonella genomes (309 K)	*Small*	27	6 h:59 min (4.38 GB) [1515 GB]	3 h:38 min (3.46 GB) [378 GB]
		63	12 h:02 min (3.88 GB) [1145 GB]	3 h:31 min (3.96 GB) [274 GB]
		119	17 h:07 min (3.95 GB) [1088 GB]	3 h:39 min (4.12 GB) [279 GB]
		255	77 h:58 min (4.82 GB) [1056 GB]	3 h:44 min (4.33 GB) [325 GB]

References

1. Chikhi, Rayan, et al. "On the representation of de Bruijn graphs." *Journal of Computational Biology* 22.5 (2015): 336-352.

2. Chikhi, Rayan, Antoine Limasset, and Paul Medvedev. "Compacting de Bruijn graphs from sequencing data quickly and in low memory." *Bioinformatics* 32.12 (2016): i201-i208. https://github.com/GATB/bcalm

3. Minkin, Ilia, Son Pham, and Paul Medvedev. "TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes." *Bioinformatics* 33.24 (2017): 4024-4032. https://github.com/medvedevgroup/TwoPaCo

4. Khan J, Patro R. 2021. Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. Bioinformatics 37: i177–i186. doi:10.1093/bioinformatics/btab309 https://github.com/COMBINE-lab/cuttlefish

5. Khan J, Kokot M, Deorowicz S, Patro R. 2022. Scalable, ultra-fast, and low- memory construction of compacted de Bruijn graphs with Cuttlefish 2. Genome Biol 23: 190. doi:10.1186/s13059-022-02743-6 https://github.com/COMBINE-lab/cuttlefish

6. Cracco, Andrea, and Alexandru I. Tomescu. "Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT." *Genome Research* 33.7 (2023): 1198-1207. https://github.com/algbio/GGCAT