# Relative Data Structures

Jouni Sirén

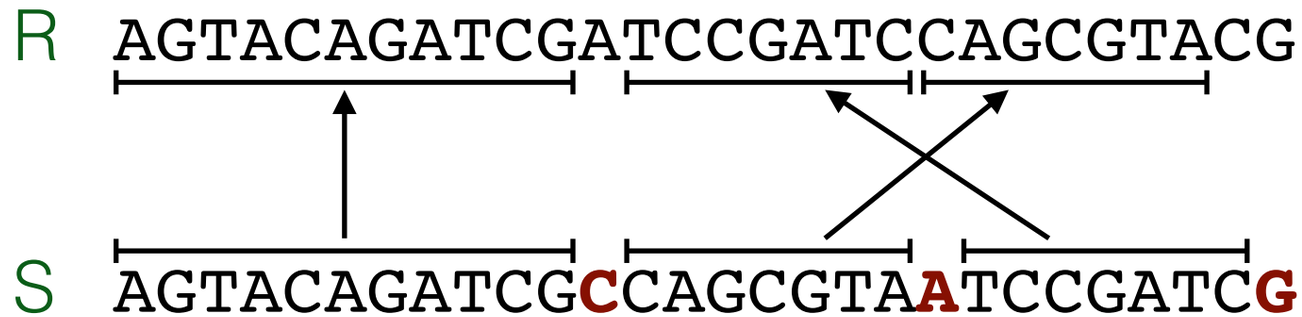
Joint work with Djamal Belazzougui, Christina Boucher, Alexander Bowe, Anthony J. Cox, Andrea Farruggia, Travis Gagie, Simon Gog, Giovanni Manzini, Gonzalo Navarro, and Simon J. Puglisi

# Relative Lempel-Ziv (RLZ)

Relative or reference-based data compression is an old idea.

Given a reference dataset R, we compress a target dataset S as a sequence of operations S | R that generate S from R.

The reference dataset can be a synthetic reference designed to improve compression performance.




S | R: (0,11,**C**), (19,7,**A**), (12,7,**G**)

Go to offset 19 in the reference, take 7 characters, and add the mismatch **A**.

- We can do more with data structures.

- Given data structures D(R) for the reference and D(S) for the target, we compress D(S) relative to D(R) as D(S | R).

- Given D(S | R) and D(R), we can simulate D(S).

- This is similar to persistent data structures, but the focus is on space-efficiency instead of maintaining past states.

- We can also view relative data structures as version control systems with extended functionality.

# Why relative data structures?

- Assume a large collection of similar datasets (e.g. assembled individual genomes).

- We can take advantage of the similarities to compress data structures for the entire collection.

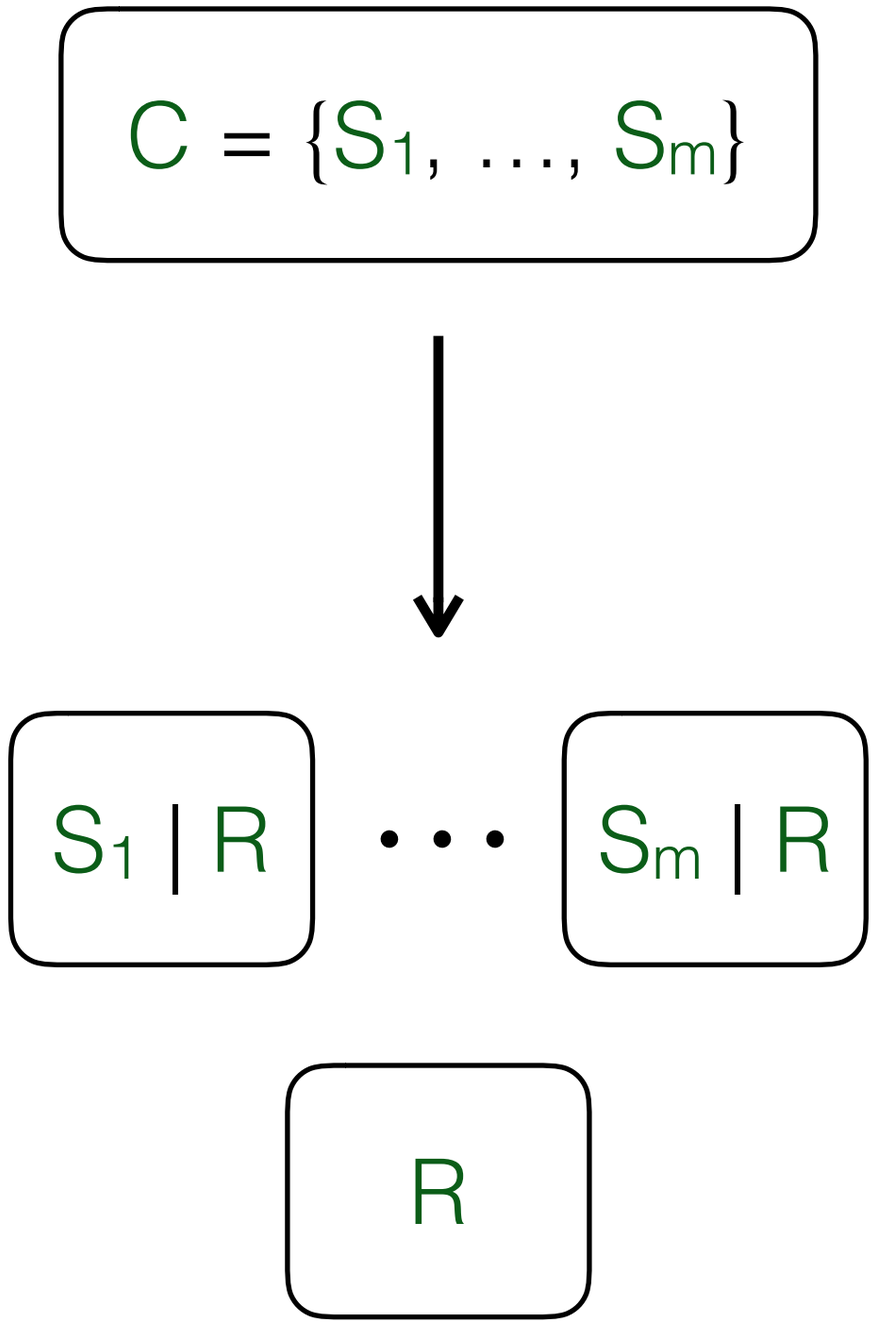
- Such data structures are usually immutable for performance reasons.

- We can replace or augment the compressed data structures with relative data structures to make adding or removing individual datasets faster.

- Somewhat worse compression, as we cannot take advantage of the similarities between $S_i$ and $S_j$ not existing in the reference.

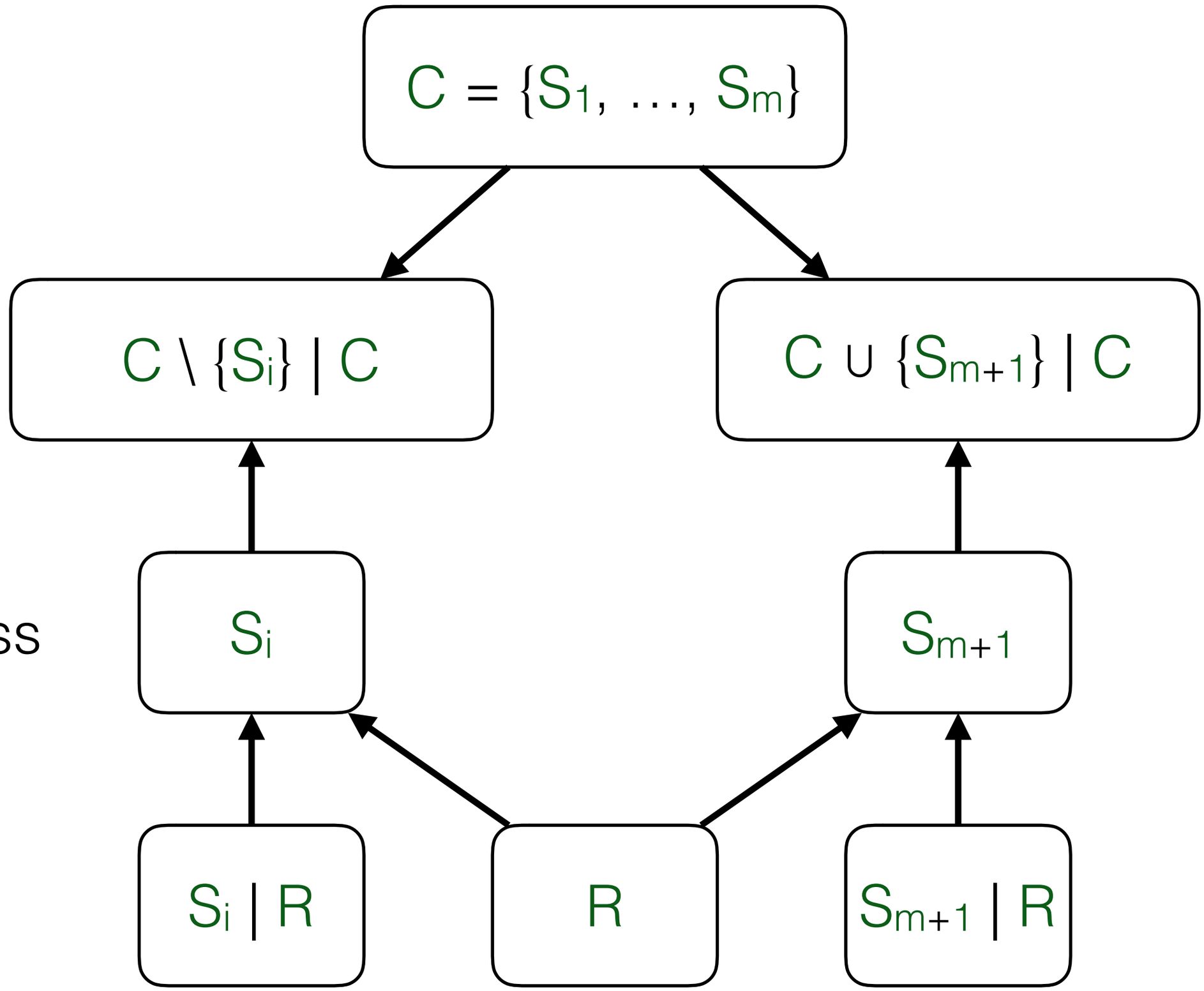
- Queries must be repeated for each dataset.

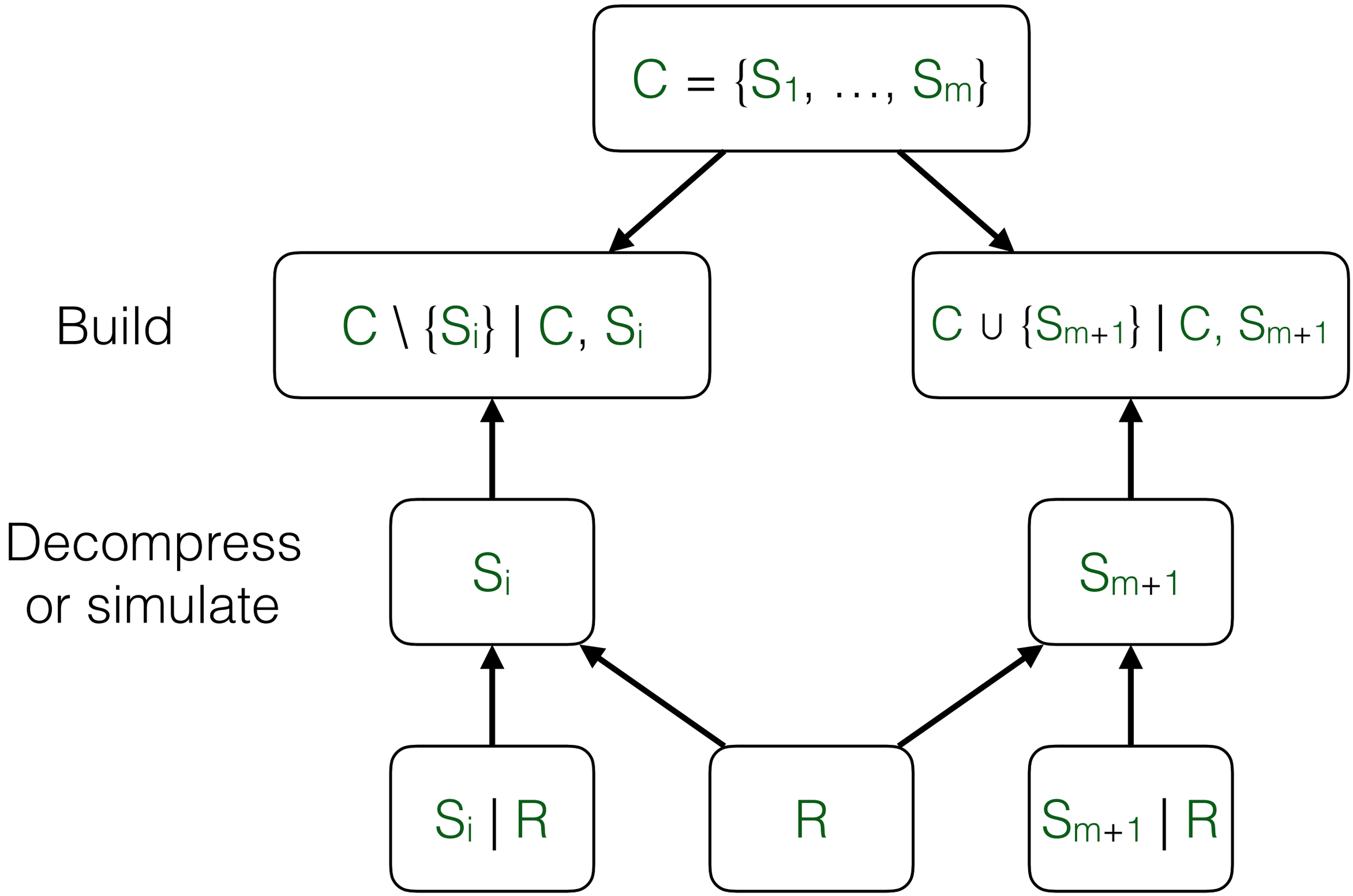
- Queries can be restricted to any subcollection.



Build

Decompress




Build
Decompress
or simulate
C = {S1, …, Sm}
C \ {Si} | C, Si
C ∪ {Sm+1} | C, Sm+1
Si
Sm+1
Si | R
R
Sm+1 | R

# Relative FM-Index

# Burrows-Wheeler transform

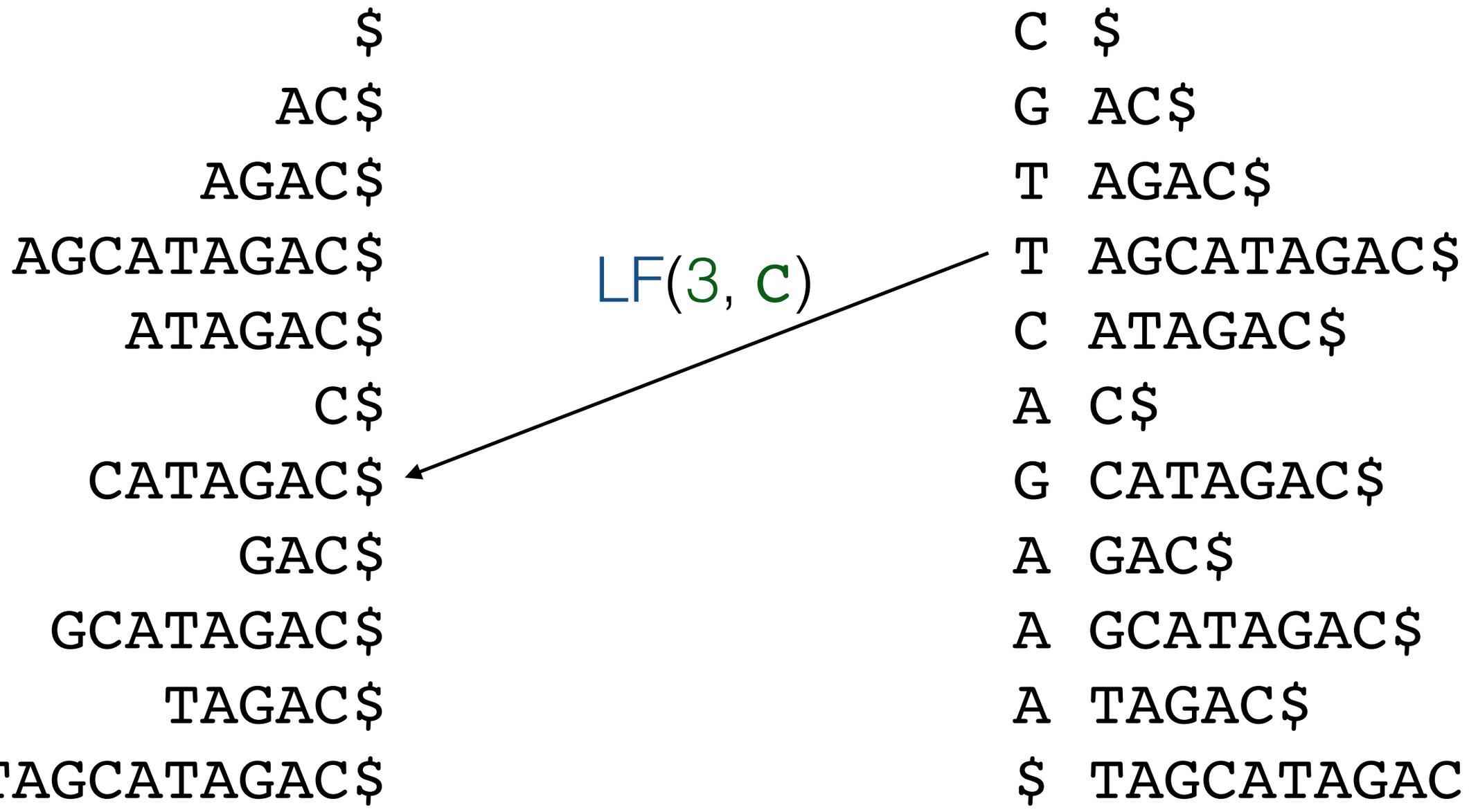
- Add a unique terminator ($) to the end of the text, sort the suffixes in lexicographic order, and output the preceding character for each suffix.

- The permutation is easily reversible and makes the text easier to compress (Burrows & Wheeler, 1994).

- The combinatorial structure is similar to the suffix array, which makes the BWT useful as a space-efficient text index (Ferragina & Manzini, 2000, 2005).

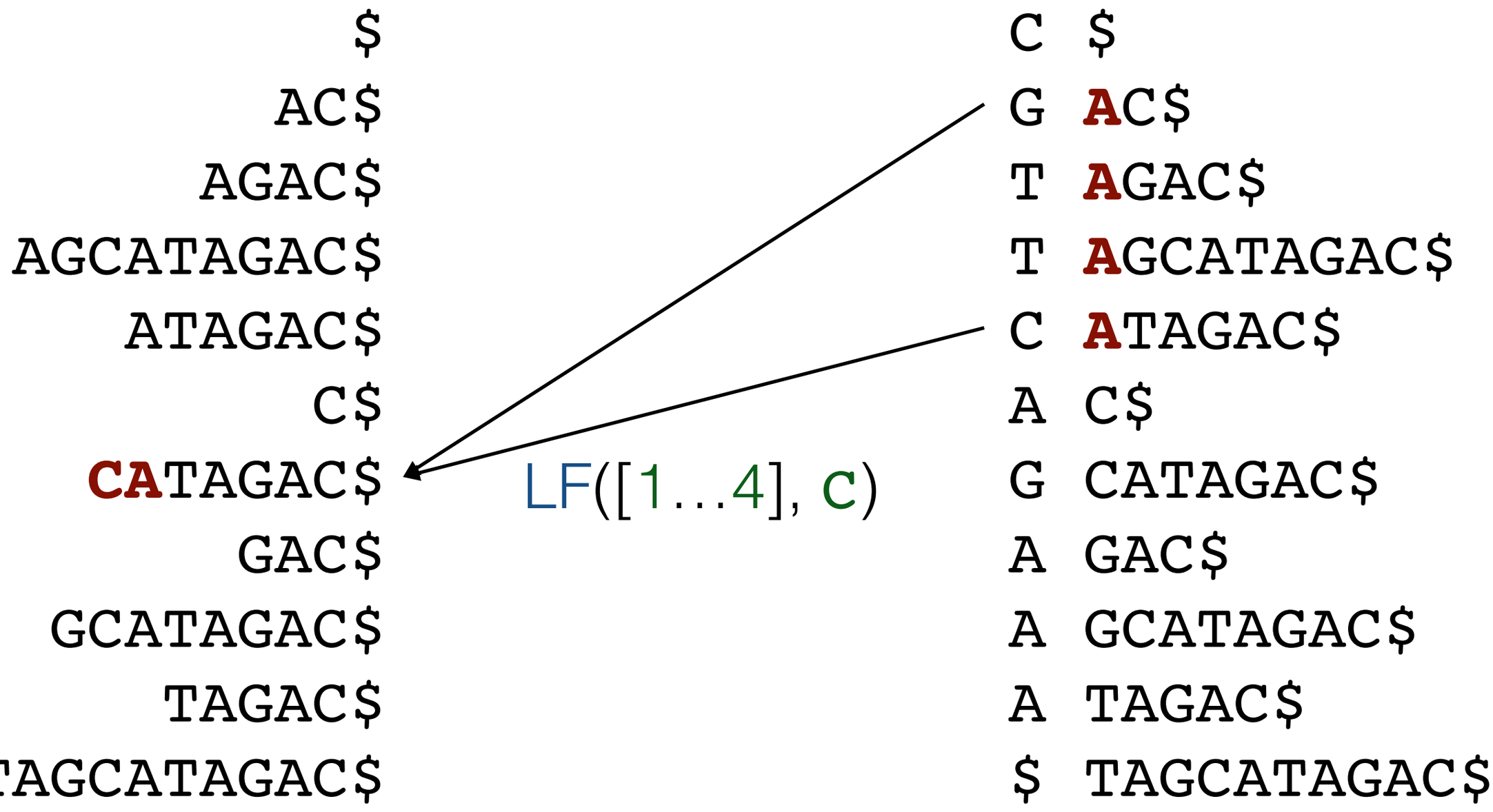
```
                TAGCATAGAC$

C  $
G  AC$
T  AGAC$
T  AGCATAGAC$
C  ATAGAC$
A  C$
G  CATAGAC$
A  GAC$
A  GCATAGAC$
A  TAGAC$
$  TAGCATAGAC$
```

# LF-mapping



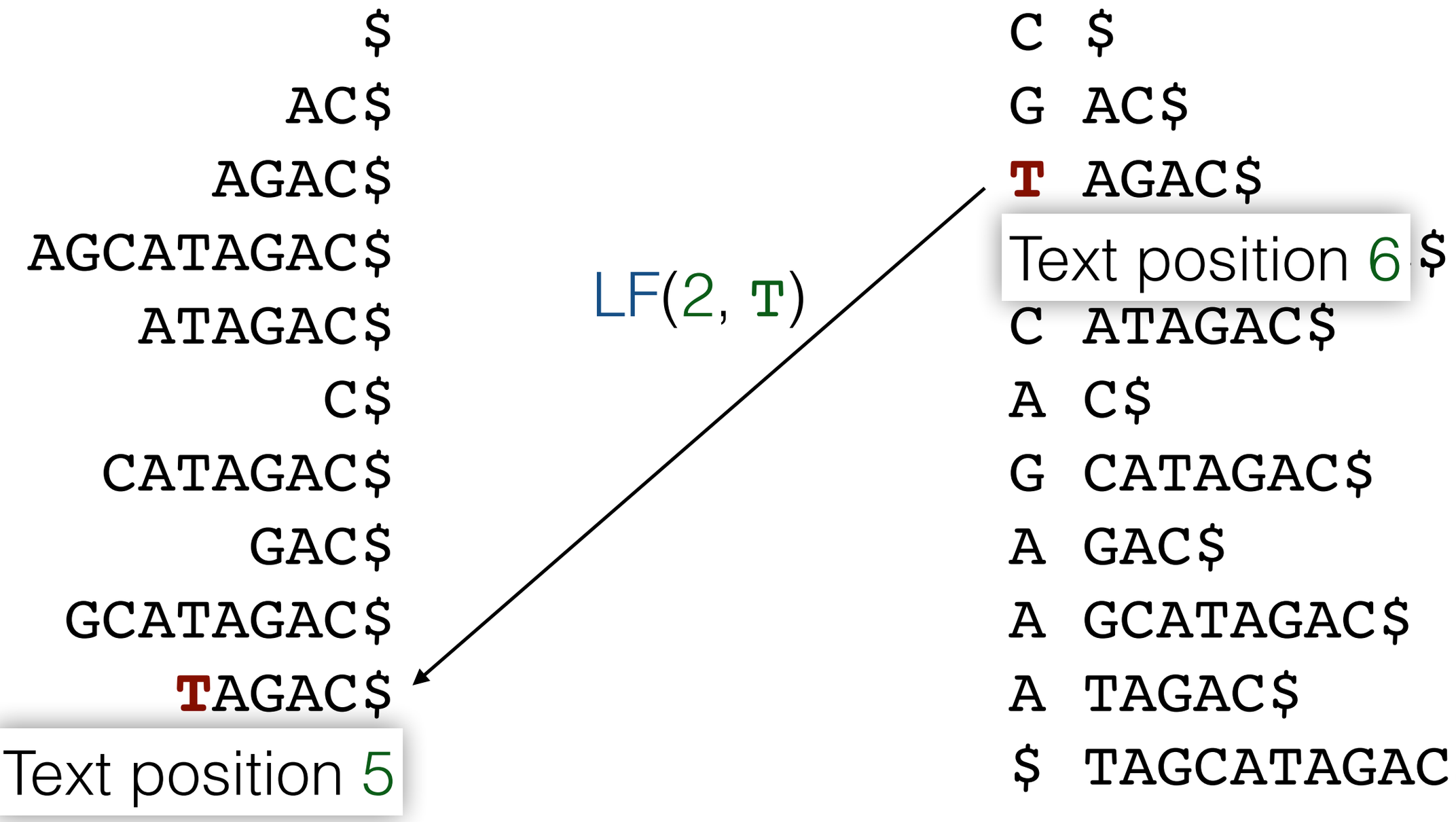

Interpretation: LF(i, c) = C[c] + BWT.rank(i, c) suffixes are **strictly before** the hypothetical suffix.

# Backward searching

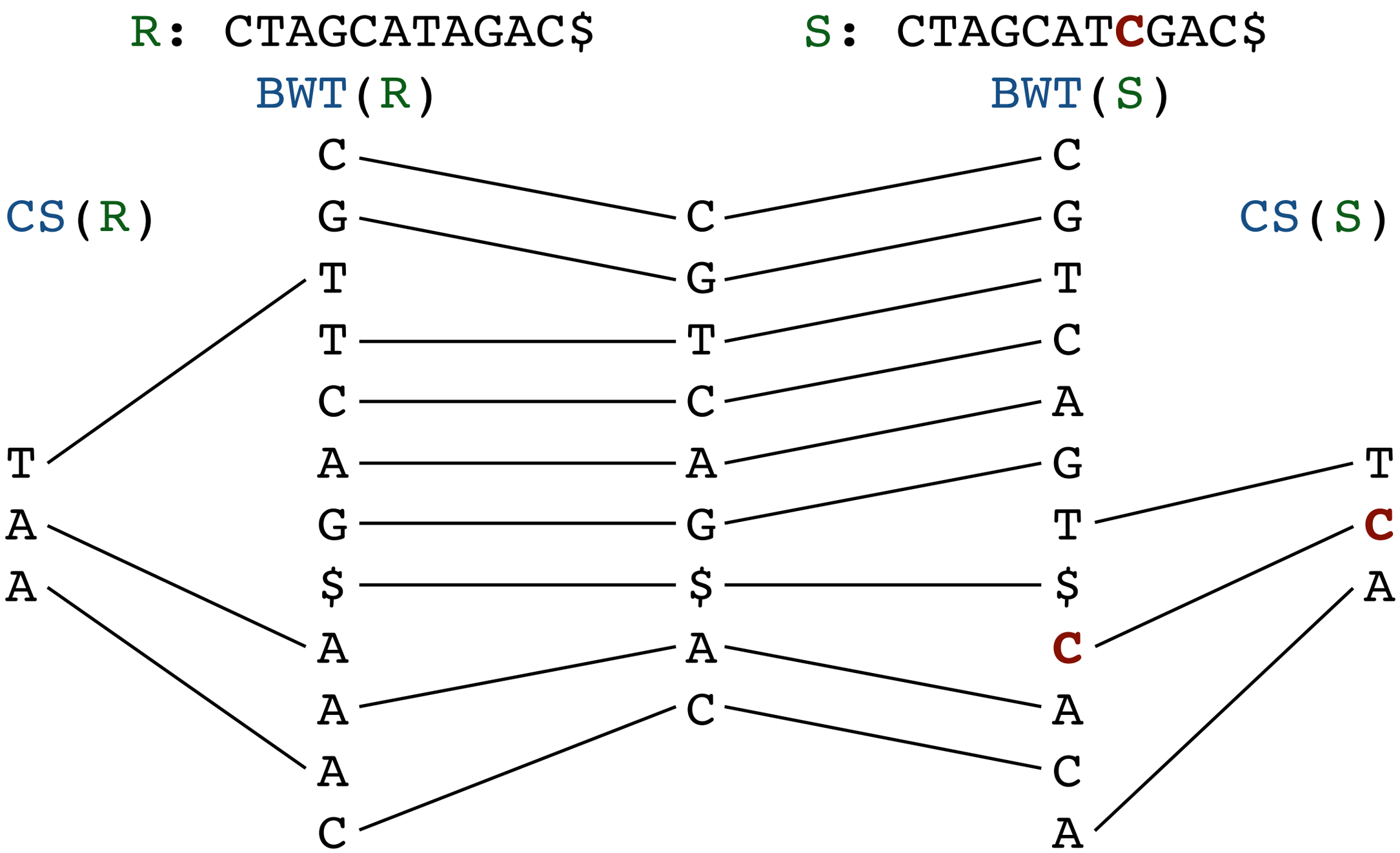



LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) −1]

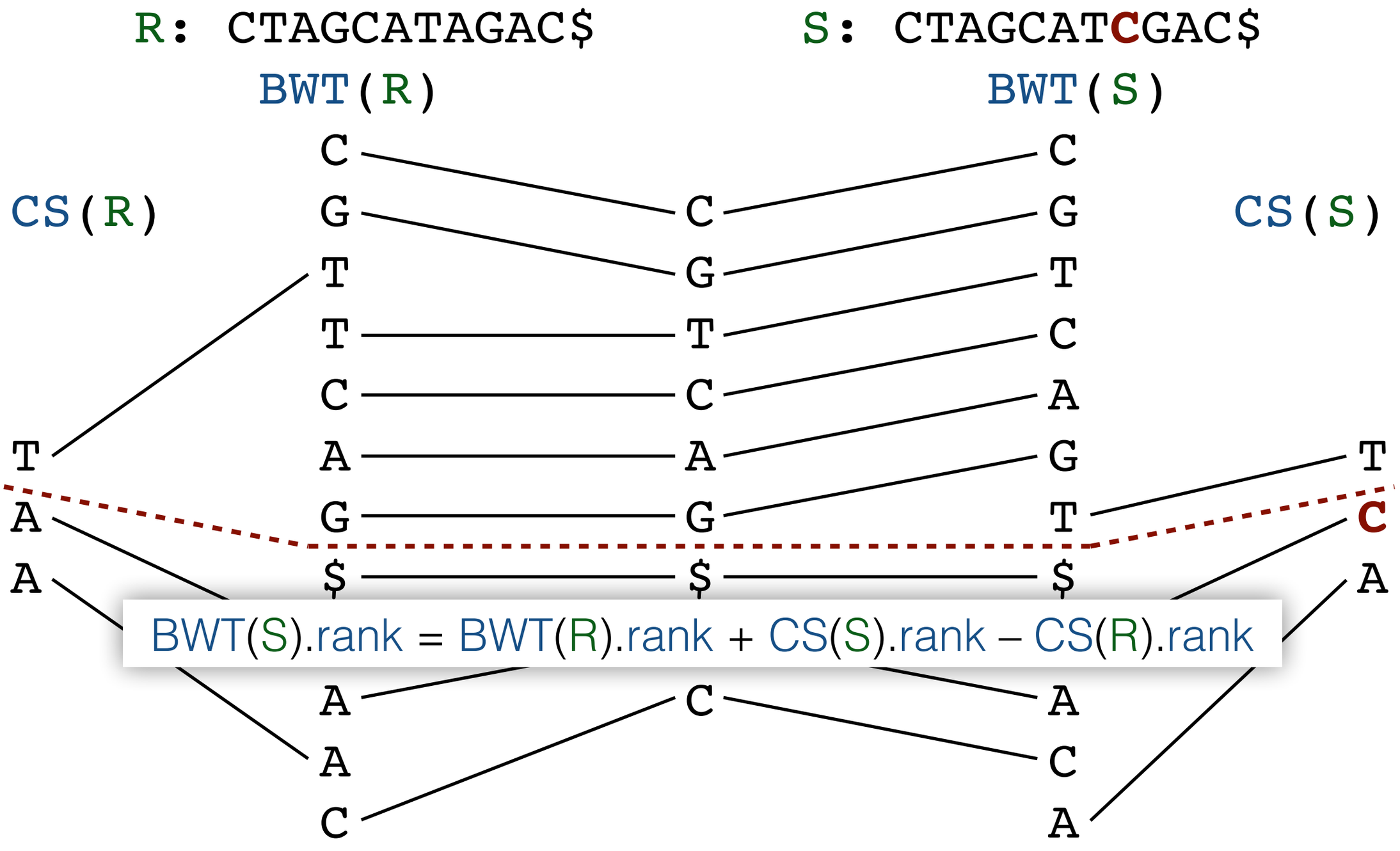
# Locating the occurrences



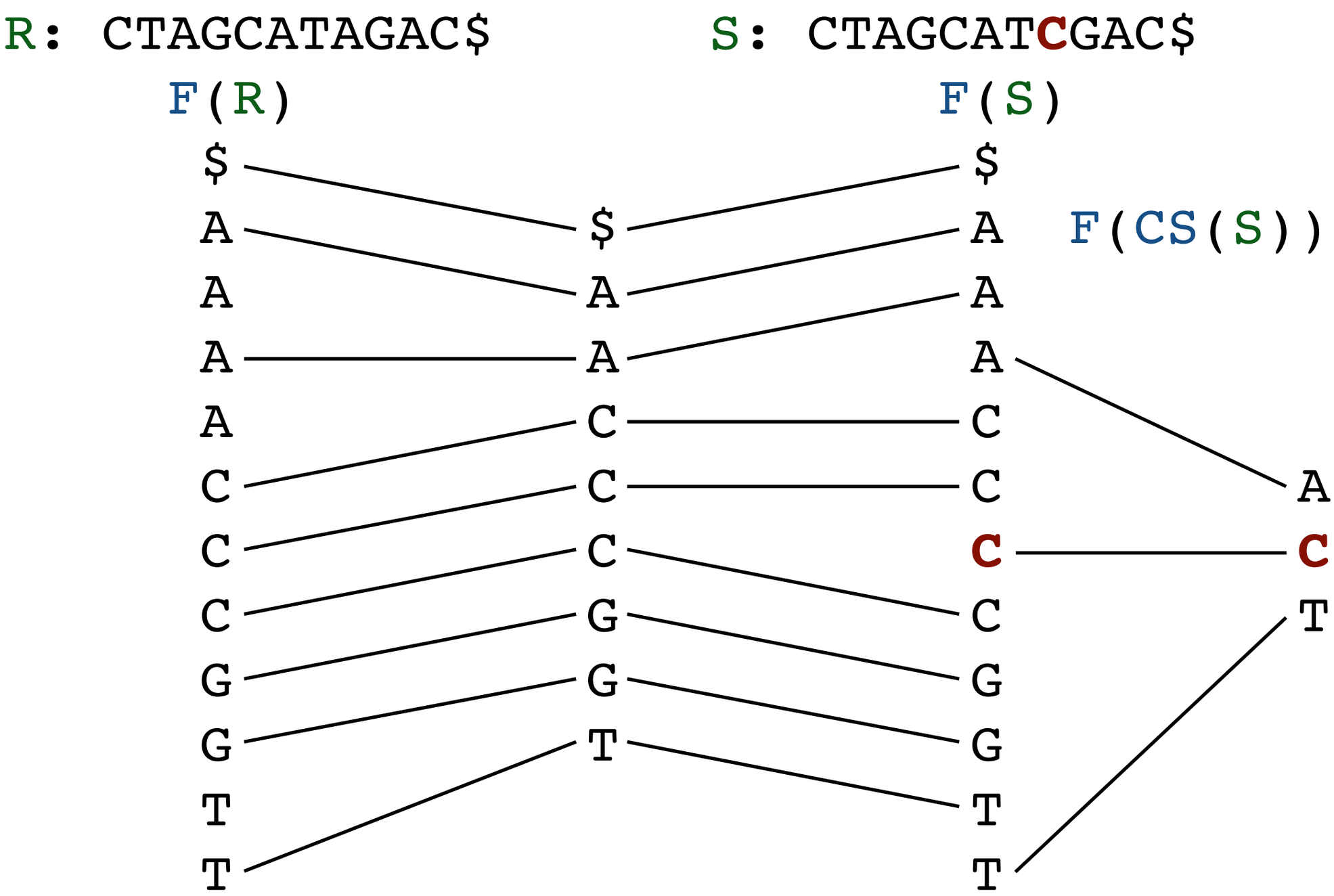

We sample some suffix array pointers and iterate LF-mapping to derive the rest from the samples.



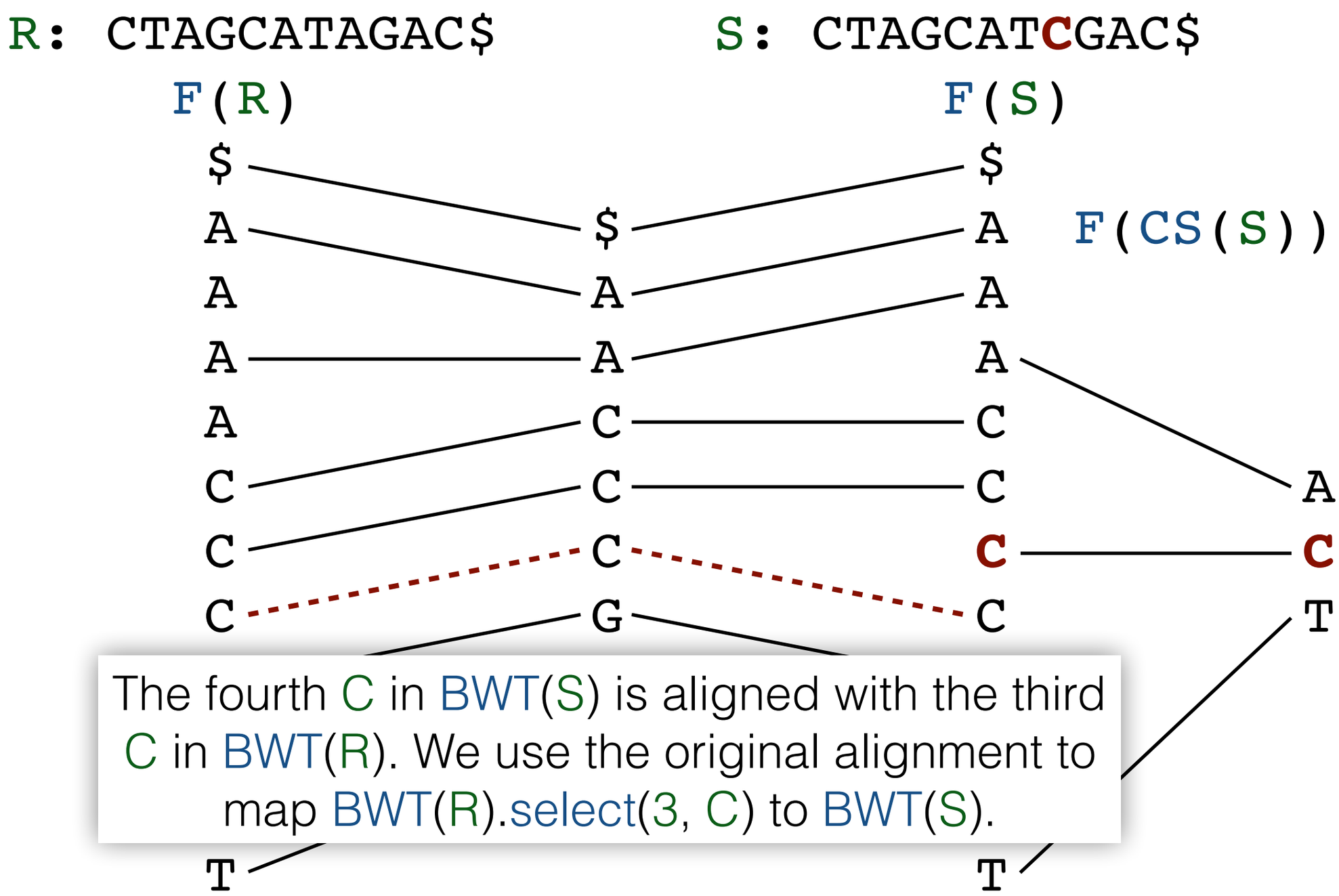

The relative FM-index (Belazzougui, Gagie, Gog, Manzini, and Sirén. SPIRE 2014) simulates the FM-index of S by aligning the BWTs and storing the alignment and the symmetric difference.




The relative FM-index (Belazzougui, Gagie, Gog, Manzini, and Sirén. SPIRE 2014) simulates the FM-index of S by aligning the BWTs and storing the alignment and the symmetric difference.



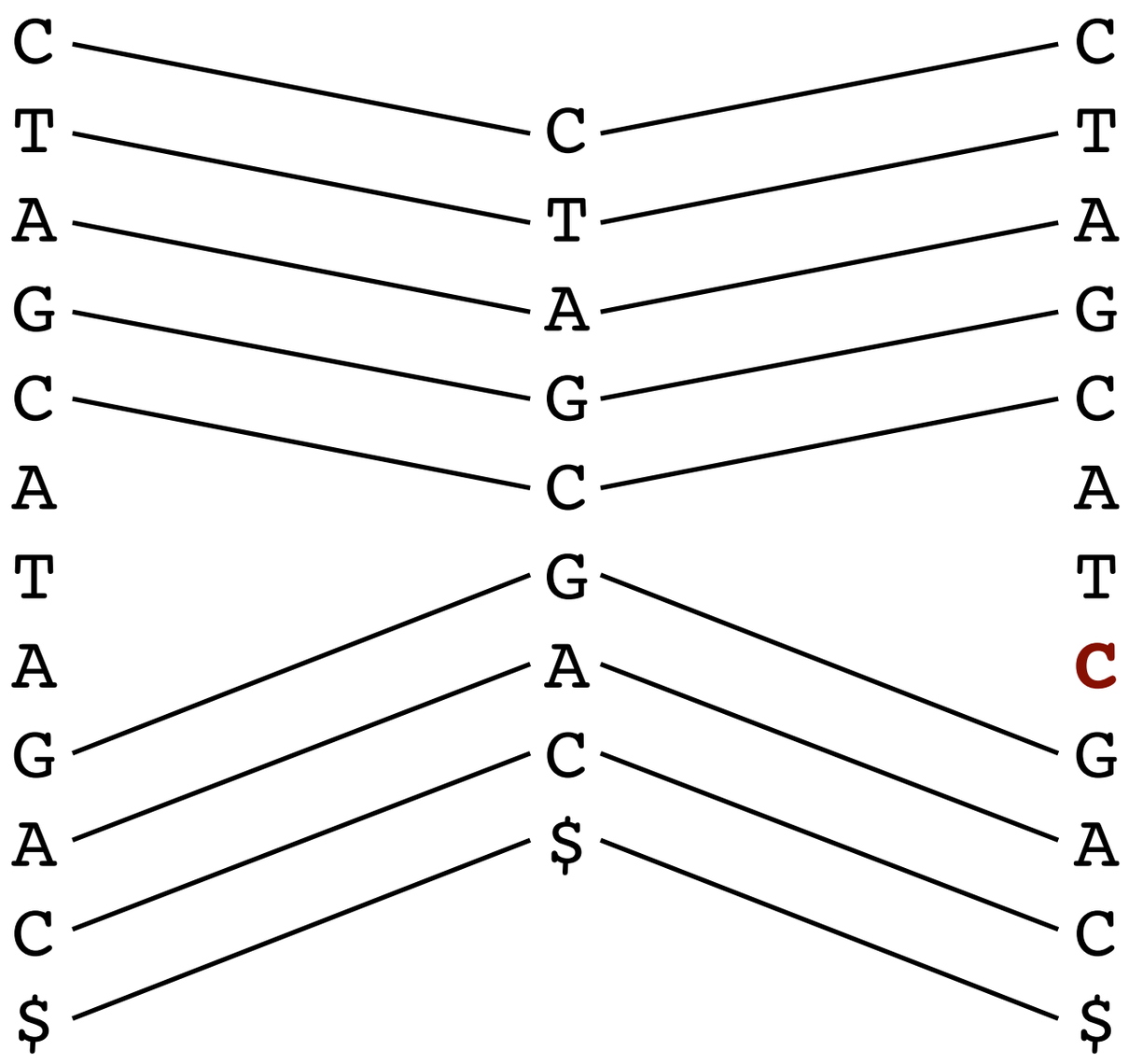

For faster select queries, we store the same alignment after stable sorting (Boucher, Bowe, Gagie, Manzini, and Sirén. SPIRE 2015).




For faster select queries, we store the same alignment after stable sorting (Boucher, Bowe, Gagie, Manzini, and Sirén. SPIRE 2015).

R: CTAGCATAGAC$

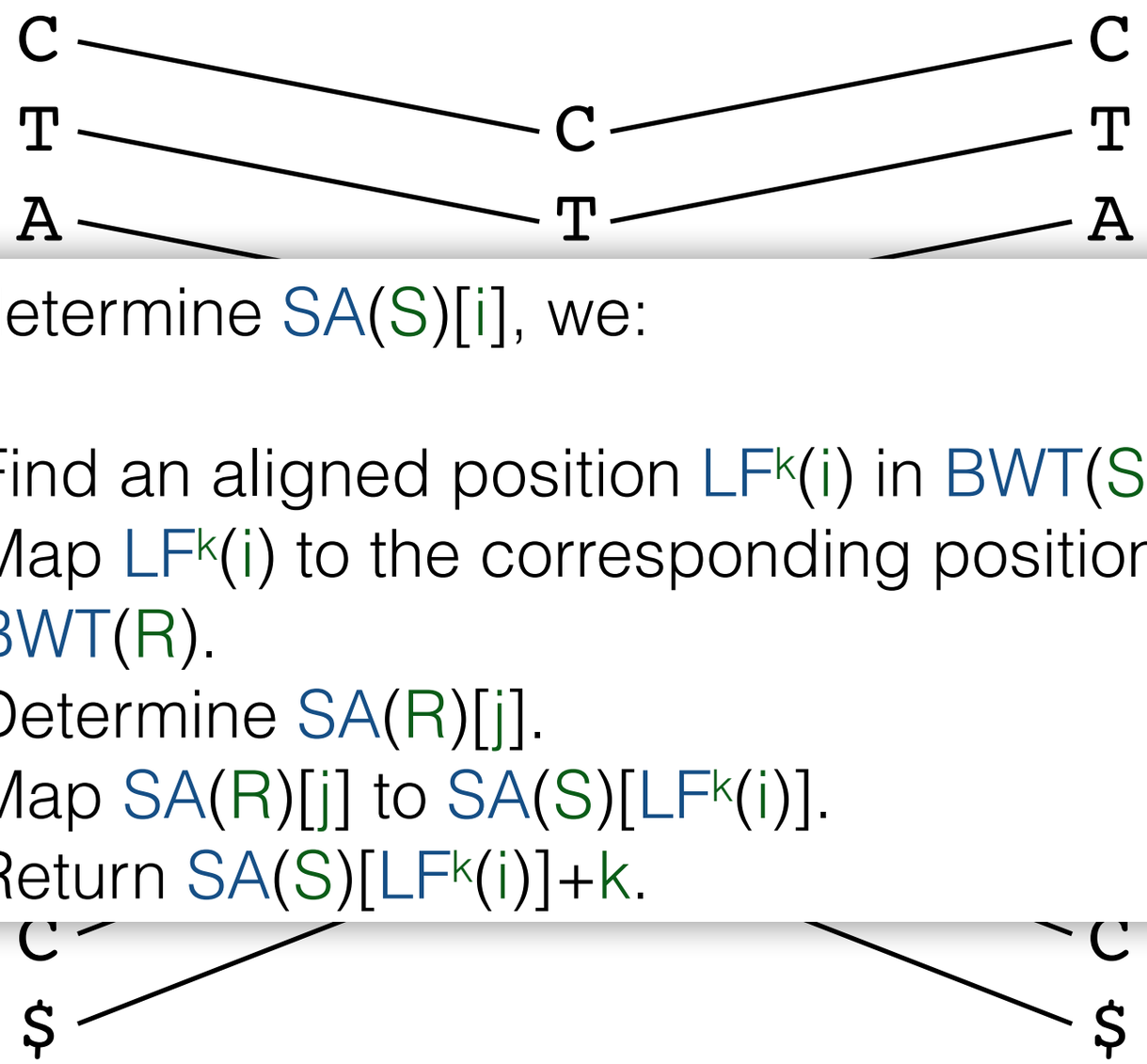
S: CTAGCAT**C**GAC$




The alignment is BWT-invariant, if it is also an alignment of the original texts. We store the alignment in text order to support locate.

R: CTAGCATAGAC$

S: CTAGCAT**C**GAC$




To determine SA(S)[i], we:

1. Find an aligned position $LF^k(i)$ in BWT(S).
2. Map $LF^k(i)$ to the corresponding position j in BWT(R).
3. Determine SA(R)[j].
4. Map SA(R)[j] to $SA(S)[LF^k(i)]$.
5. Return $SA(S)[LF^k(i)]+k$.

An alignment of BWTs is **BWT-invariant**, if it is also an alignment of the original texts. We store the alignment in **text order** to support locate.

```
BWT(R)    C G TT C A G  $ A A AC
BWT(RS)   CCGGTTTCCAAGGT$$ACAAACCA
BWT(S)      C G  T C A GT $ C A  CA
```

- The BWT merging algorithm tells how to interleave BWT(R) and BWT(S) into BWT(RS).

- The interleaving contains essentially the same information as a BWT-invariant alignment of BWT(R) and BWT(S).

- We can use the interleaving as BWT(RS | R, S).

| | Size | find() | find() + locate() |
|---|---|---|---|
| **FMI bit_vector** | 5.83 bpc | 32.9 s | 662 s |
| **FMI rrr_vector** | 4.14 bpc | 142 s | 4490 s |
| **RFM (full reference)** | 1.26 bpc | 254 s | 2092 s |
| **RFM (female reference)** | 1.11 bpc | 292 s | 2074 s |

WT-based FM-index (samples: SA 17, ISA 64) vs. relative FM-index.
Maternal haplotypes of NA12878 vs. human reference genome.
2 million patterns of length 32 with 255 million occurrences.

# Relative Suffix Tree

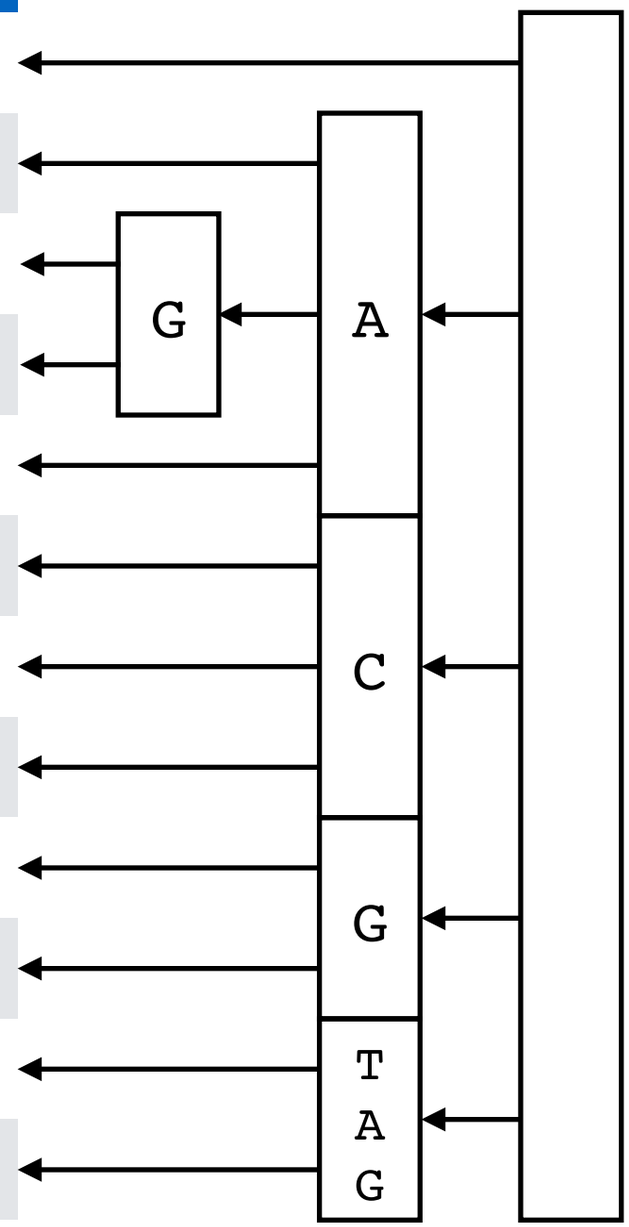
We can simulate the suffix tree with an FM-index and NPR queries (next/previous smaller value, range minimum) in the LCP array. (Fischer et al. TCS, 2009)

LCP[sp…ep] is an internal node of the suffix tree at depth d, if

- LCP[sp] < d;
- LCP[ep+1] < d;
- d ∈ LCP[sp+1…ep]; and
- LCP[sp+1…ep] ≥ d.

(Abouelhoda et al. JDA, 2004)

| Suffixes | LCP |
|---|---|
| $ | 0 |
| AC$ | 0 |
| AGAC$ | 1 |
| AGCATAGAC$ | 2 |
| ATAGAC$ | 1 |
| C$ | 0 |
| CATAGAC$ | 1 |
| CTAGCATAGAC$ | 1 |
| GAC$ | 0 |
| GCATAGAC$ | 1 |
| TAGAC$ | 0 |
| TAGCATAGAC$ | 3 |

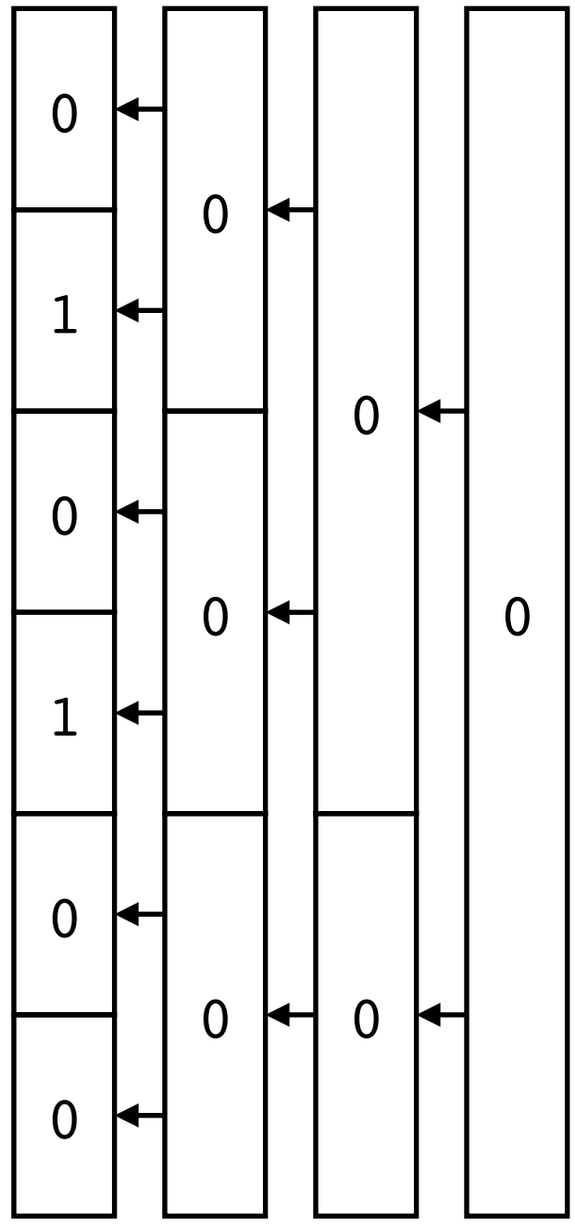


If we have fast sequential access to the LCP array, we can support NPR queries with a range minima tree.

We partition the LCP array into blocks, which become the leaves of the tree. Each node contains the minimal LCP value in the corresponding range.

In practice, we store the minimal values in level order. This allows us to visit all children of a node with a single memory access.

| Suffixes | LCP |
|---|---|
| `$` | 0 |
| `AC$` | 0 |
| `AGAC$` | 1 |
| `AGCATAGAC$` | 2 |
| `ATAGAC$` | 1 |
| `C$` | 0 |
| `CATAGAC$` | 1 |
| `CTAGCATAGAC$` | 1 |
| `GAC$` | 0 |
| `GCATAGAC$` | 1 |
| `TAGAC$` | 0 |
| `TAGCATAGAC$` | 3 |



- The relative suffix tree (Gagie, Navarro, Puglisi, and Sirén. arXiv, 2015) uses RLZ to compress the LCP array. Each phrase becomes a leaf in the range minima tree.

- The parsing uses differential values LCP[i] – LCP[i–1], but the mismatches are stored as absolute LCP values.

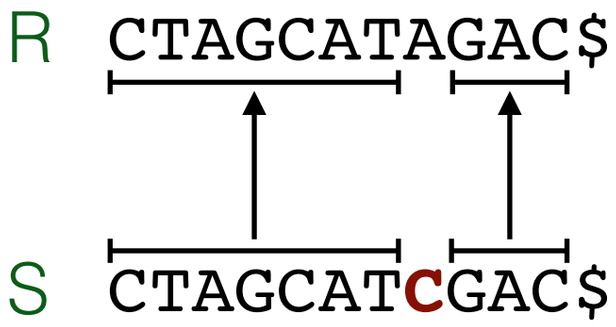
- LCP(S)[i+k] = LCP(S)[i–1] + LCP(R)[j+k] – LCP(R)[j–1], where the phrase starts at LCP(S)[i] and LCP(R)[j].

- The result is surprisingly fast, but the RLZ parsing requires too much space.

- We decided not to publish the paper until we can figure out something better than RLZ.

| | Size | Traversal | Matching statistics |
|---|---|---|---|
| **cst_sct3 lcp_dac** | 18.1 bpc | 18 min | 114 min |
| **cst_sct3 PLCP** | 10.8 bpc | 18 min | 166 min |
| **cst_sada** | 12.3 bpc | 5 min | 308 min |
| **cst_fully** | 4.98 bpc | >2160 min | – |
| **Relative ST slow select()** | 3.18 bpc (RFM 1.11 bpc) | 41 min | 855 min |
| **Relative ST fast select()** | 3.63 bpc | 41 min | 340 min |

Compressed suffix trees from SDSL vs. relative suffix tree.
Maternal haplotypes of NA12878 vs. human (female) reference.
ST traversal and matching statistics vs. another chr1 assembly.



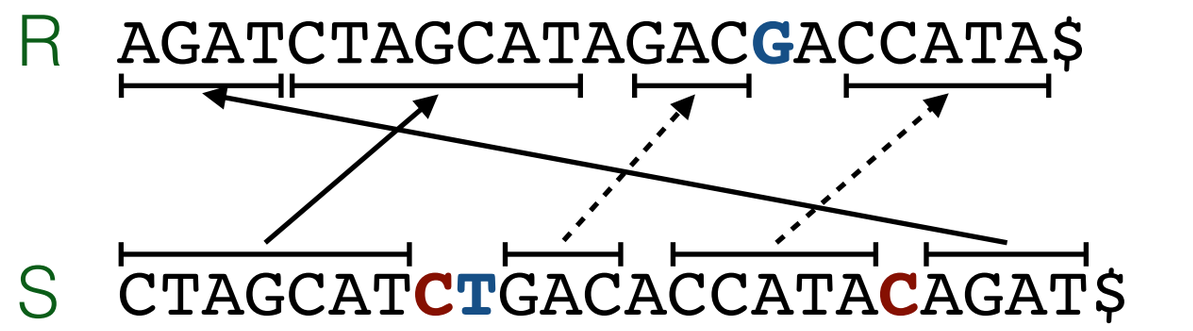

S | R: (0,7,**C**), (8,3,**$**)

RLZ works well with **substitutions**. We can assume that the next phrase starts length+1 characters after the previous one.

Lexicographic sorting transforms substitutions into **indels**.

The substitutions also **change** LCP values.

| R: CTAGCATAGAC$ | | | S: CTAGCATCGAC$ | | |
|---|---|---|---|---|---|
| $ | C | 0 | $ | C | 0 |
| AC$ | G | 0 | AC$ | G | 0 |
| AGAC$ | **T** | **1** | | | |
| AGCATAGAC$ | T | 2 | AGCATCGAC$ | T | **1** |
| ATAGAC$ | C | 1 | ATCGAC$ | C | 1 |
| C$ | A | 0 | C$ | A | 0 |
| CATAGAC$ | G | 1 | CATCGAC$ | G | 1 |
| | | | CGAC$ | **T** | **1** |
| CTAGCATAGAC$ | $ | 1 | CTAGCATCGAC$ | $ | 1 |
| GAC$ | A | 0 | GAC$ | **C** | **0** |
| GCATAGAC$ | A | 1 | GCATCGAC$ | A | 1 |
| TAGAC$ | **A** | **0** | | | |
| TAGCATAGAC$ | C | 3 | TAGCATCGAC$ | C | **0** |
| | | | TCGAC$ | **A** | **1** |




S | R: (4,7,**CT**), (–1,3,**A**), (+1,5,**C**), (0,4,**$**)

RLZAP uses two kinds of phrases:

- Explicit phrases contain an absolute pointer to the reference.
- Adaptive phrases encode the pointer relative to what can be expected based on the previous phrase.

Each phrase can contain multiple mismatch characters at the end. We essentially represent the target sequence as a concatenation of local alignments with the reference.

Cox, Farruggia, Gagie, Puglisi, and Sirén. SPIRE 2016.

# Conclusions

- Relative data structures simulate data structures for a target dataset, given the same data structure for a similar reference dataset.

- The relative FM-index has similar performance to compressed FM-indexes, while using much less space.

- The relative suffix tree also achieves good time/space trade-offs, but there is still much room for improvement.