Distribution-Aware Compressed Full-Text Indexes

Jouni Sirén
University of Helsinki, Finland
with
Paolo Ferragina and Rossano Venturini
University of Pisa, Italy

Jouni Sirén

- PhD (computer science), University of Helsinki, 2012
- Supervisor: Veli Mäkinen
- Thesis: Compressed Full-Text Indexes for Highly Repetitive Collections
- http://iki.fi/jouni.siren/

Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, Niko Välimäki: **Storage and Retrieval of Highly Repetitive Sequence Collections**. Journal of Computational Biology, 2010. Earlier in SPIRE 2008, RECOMB 2009.

Collections of individual genomes or different versions of documents compress extremely well. With them, o(n) bits of overhead information in a CSA can be too much.

CSAs where overhead scales with compressed size.

Jouni Sirén: **Compressed Suffix Arrays for Massive Data**. SPIRE 2009.

Compressed data structures allow handling larger data sets in main memory than with uncompressed ones. Yet how do we build the compressed structures in the first place?

A space-efficient parallel construction algorithm for CSAs. Practical for data sets of tens of gigabytes in size.

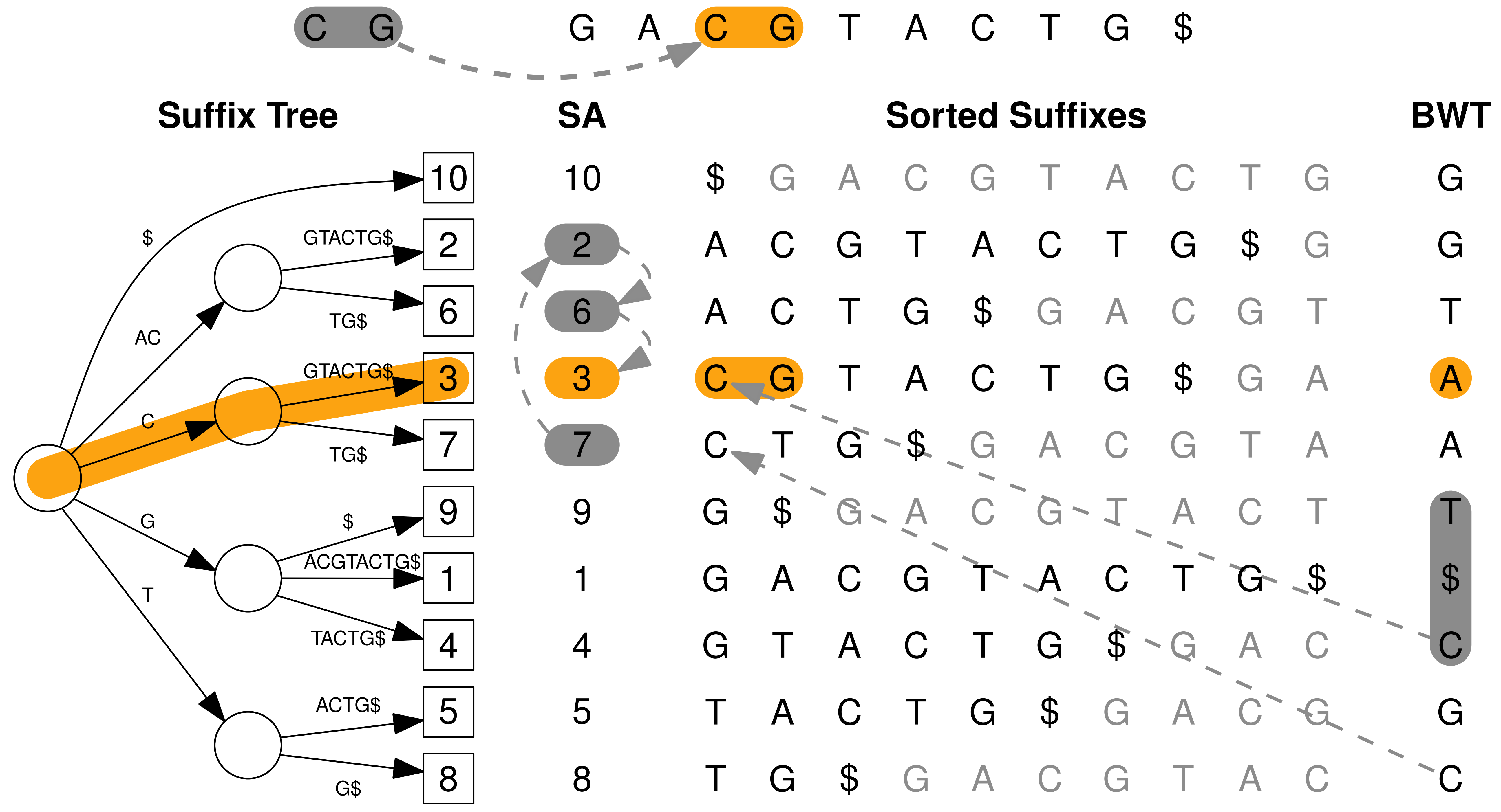
Jouni Sirén, Niko Välimäki, Veli Mäkinen: **Indexing Finite Language Representation of Population Genotypes**. WABI 2011.

Compressed suffix arrays can index sets of sequences. What if we want to index plausible recombinations of those sequences, or a reference sequence and a set of known genetic variation?

CSAs for indexing generalizations of (subgraphs of) De Bruijn graphs.

Contents

Full-text indexes



SA	Sorted Suffixes	BWT
10	$ G A C G T A C T G	G
2	A C G T A C T G $ G	G
6	A C T G $ G A C G T	T
3	C G T A C T G $ G A	A
7	C T G $ G A C G T A	A
9	G $ G A C G T A C T	T
1	G A C G T A C T G $	$
4	G T A C T G $ G A C	C
5	T A C T G $ G A C G	G
8	T G $ G A C G T A C	C

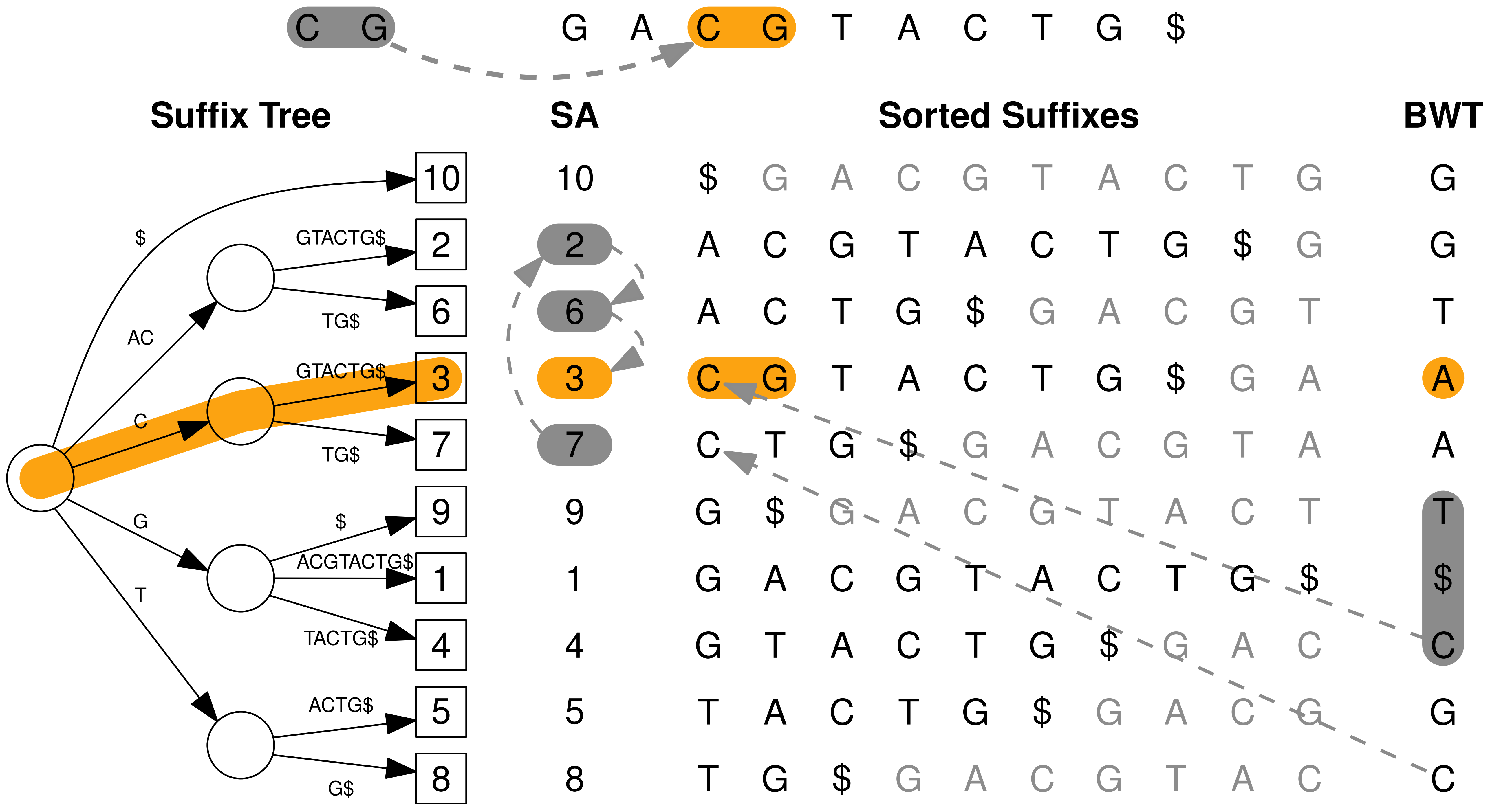
Suffix array

- Pointers to the suffixes of the text in lexicographic order.
- Index size (usually 5 or 9 Bpc) limits the size of the data that can be indexed.
- find(P) returns the lexicographic range of suffixes prefixed by pattern P.
- locate(i) returns the suffix of rank i.

Burrows-Wheeler transform

- Permutation of the text related to SA.
- Instead of a pointer to the suffix, we store the previous character.
- Easier to compress than the original text.
- An index based on the BWT does not require the original text.

Full-text indexes



SA	Sorted Suffixes	BWT
10	$ G A C G T A C T G	G
2	A C G T A C T G $ G	G
6	A C T G $ G A C G T	T
3	C G T A C T G $ G A	A
7	C T G $ G A C G T A	A
9	G $ G A C G T A C T	T
1	G A C G T A C T G $	$
4	G T A C T G $ G A C	C
5	T A C T G $ G A C G	G
8	T G $ G A C G T A C	C

locate(i) with BWT

- $SA[LF(i)] = SA[i] - 1$
- $LF(i) = C[BWT[i]] + \text{rank}_{BWT[i]}(BWT, i)$
- We sample some text positions (i, SA[i]).
- If SA[i] has not been sampled, we compute it as $SA[i] = SA[LF^j(i)] + j$, where $SA[LF^j(i)]$ has been sampled.

Contents

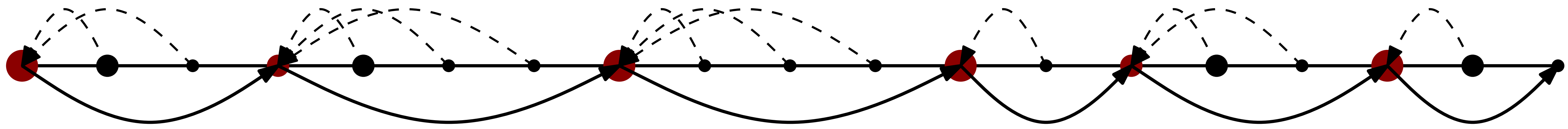
Paolo Ferragina, Jouni Sirén, Rossano Venturini: **Distribution-aware compressed full-text indexes**. ESA 2011.

Locating the occurrences of a pattern with a compressed suffix array is relatively slow. If the query distribution is skewed, we should be able to use that information to solve locate queries faster.

Efficient algorithm for optimizing a CSA for a known query distribution.

Modeling the problem

- Cost of locate() is proportional to the distance $d(i, S)$ to the nearest sampled text position $s \in S$.
- We can only move backward.
- Position i is located with probability $P(i)$.
- Find S of size k that minimizes $\sum_i P(i)\ d(i, S)$.



- Define a DAG with nodes 1 to n+1 and all possible forward edges.
- Edge (s, s') encodes the cost of locating text positions s to s'–1 by using sample s.
- Its weight is $w(s, s') = \sum_{s \leq i < s'} P(i)\,(i - s)$.
- Optimal solution is the set of nodes in a minimum-weight k-link path from 1 to n+1.

Finding the solution

- We want a solution using O(n polylog(n)) time and O(n log n) bits of space.
- Yet there are $\theta(n^2)$ edges in the DAG!
- We need a better algorithm and a way to compute edge weights quickly.

Our DAG satisfies the concave Monge property:

$$w(i+1, j+1) - w(i+1, j) = P(j)\,(j - i - 1) \leq P(j)\,(j - i) = w(i, j+1) - w(i, j)$$
$$\Longrightarrow$$
$$w(i, j) + w(i+1, j+1) \leq w(i, j+1) + w(i+1, j).$$

If the weights are non-negative integers, we can solve the problem in $O(n \log U)$ time, where U is the largest edge weight.

The condition holds, if we replace probabilities $P(i)$ with frequencies $f(i)$ in a query log.

A. Aggarwal, B. Schieber, T. Tokuyama: **Finding a Minimum-Weight *k*-Link Path in Graphs with the Concave Monge Property and Applications**. Discrete & Computational Geometry, 1994.

If there are minimum-weight paths of length a and b, we can combine them to get a minimum-weight path of any length between a and b.

Use binary search to find a weight adjustment q, such that when q is added to all edge weights, there is a minimum-weight path of length k.

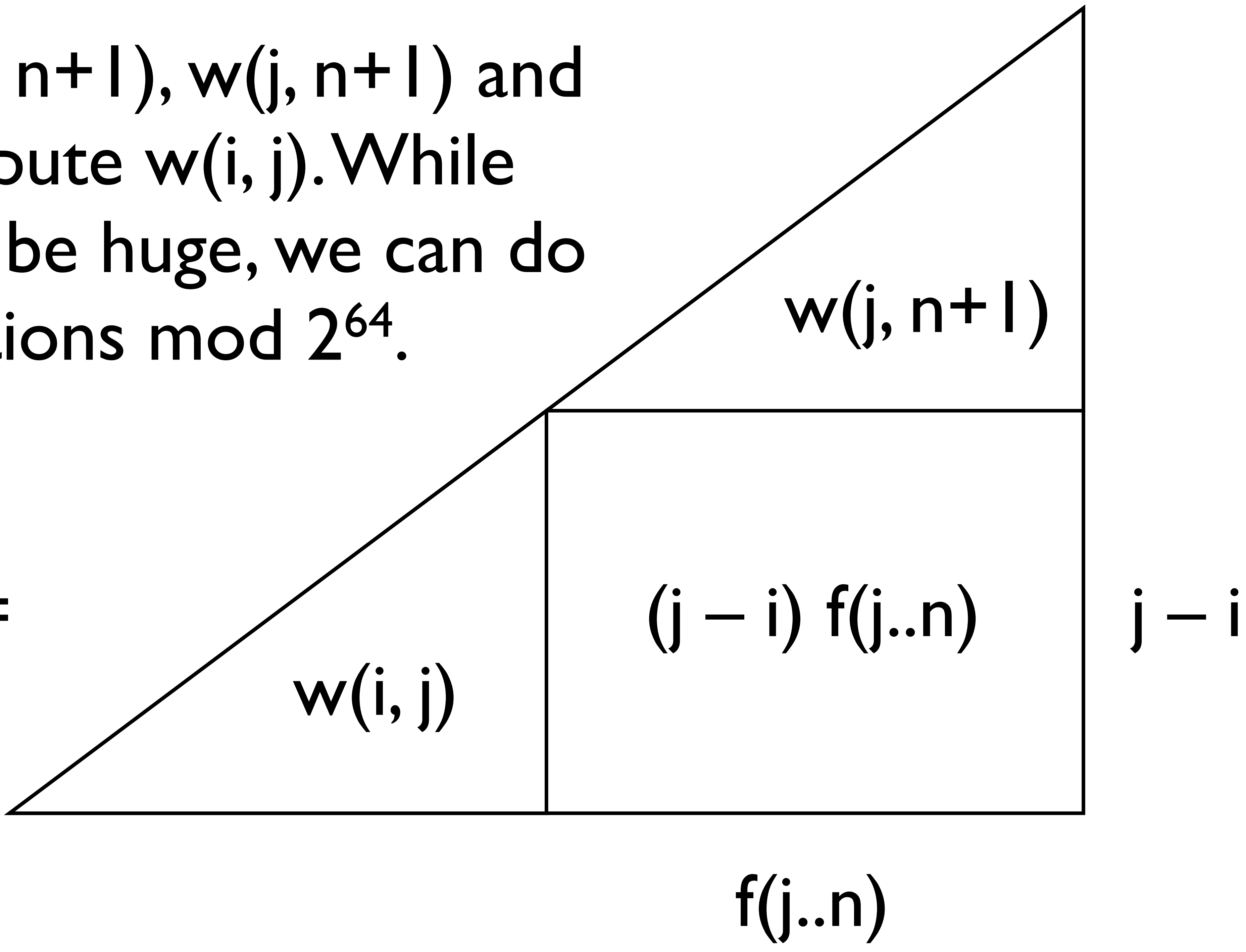
Finding the path

- Finding a minimum-weight path usually involves dynamic programming.
- Concave Monge property allows us to restrict our attention to short edges.
- Several theoretical O(n) time algorithms and a practical O(n log n) time algorithm are based on that restriction.

Edge weights?

We need w(i, n+1), w(j, n+1) and f(j..n) to compute w(i, j). While w(i, n+1) can be huge, we can do the computations mod 2^{64}.

w(i, n+1) =



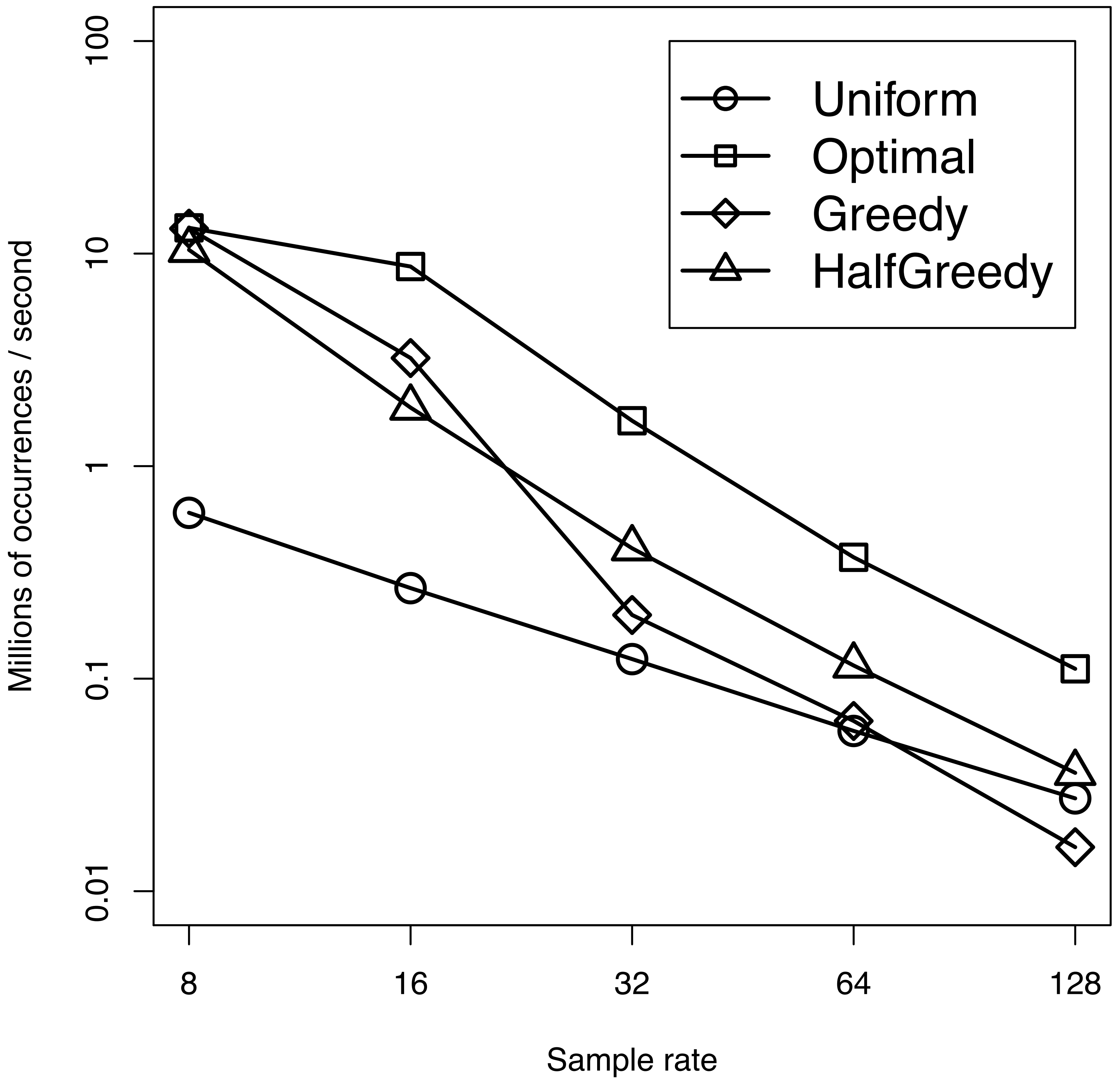
Experimental results

- Text: 1.24 gigabytes of HTML pages.
- Patterns: Search terms from a query log, weighted by term frequencies.
- Queries: 10000 randomly chosen patterns with 122 million occurrences.
- System: 2x 2.53 GHz Xeon E5540 (used only one core), 32 GB memory.

Uniform is the standard sampling strategy.

Greedy samples most frequent text positions.

HalfGreedy is 50% Uniform, 50% Greedy.

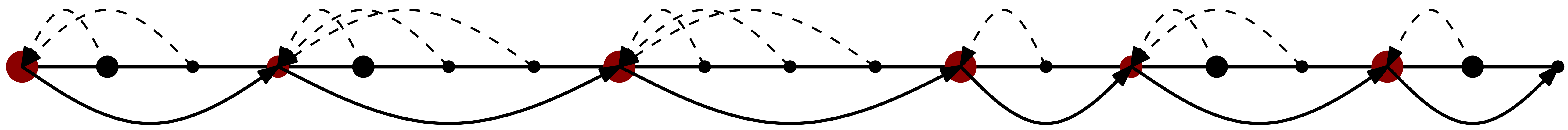


Contents

1. Compressed full-text indexes
2. Optimizing for known query distributions (ESA 2011)
3. **Optimizing for unknown query distributions (work in progress)**

Objectives

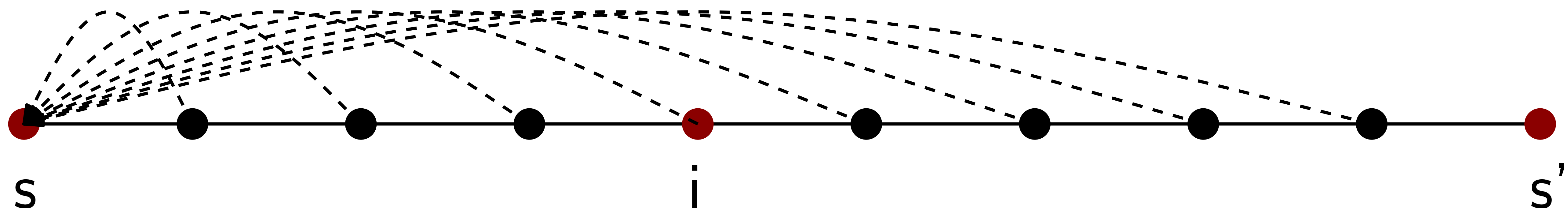
- Learn the distribution online space-efficiently.
- Uniform samples take 6–10 bytes each. Doubling their number doubles the speed.
- Ideal solution would take O(1) time, but O(log n) time with O(1) random memory accesses is acceptable.



Find set S of k points that minimizes the sum
$\sum_i f(i) \min \{ i - s \mid s \in S, s \leq i \}$.

OR

Find a minimum-weight k-link path from 1 to n+1,
where $w(s, s') = w(s, s') = \sum_{s \leq i < s'} f(i)\ (i - s)$.



We have used sample s to retrieve text position i. This can imply that

a) s is a good sample; or
b) i should be sampled, making s less useful as a sample.

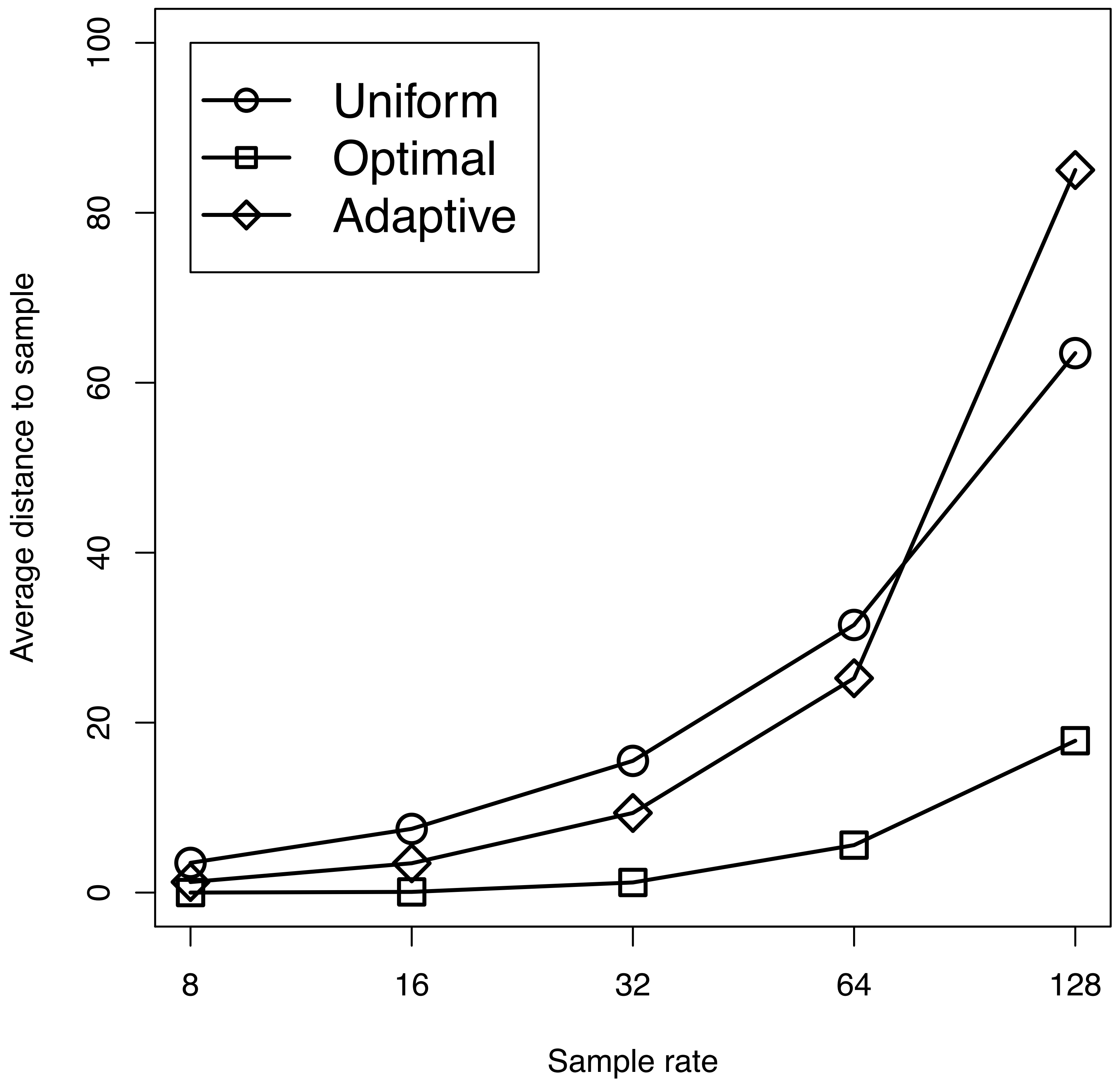
What do we choose?

Basic solution

- Store the sampled positions in a hash table.
- Sample all located positions.
- If there is a collision, drop the old sample.
- Approximates something between Greedy and HalfGreedy.

Same data and patterns as in the static case.

100000 queries with 1.3 billion occurrences.



Additional heuristics

- Use some static samples, as in HalfGreedy.
- Guarantees reasonable worst-case performance.
- Two other heuristics can be used to improve the performance further.

Two hash tables

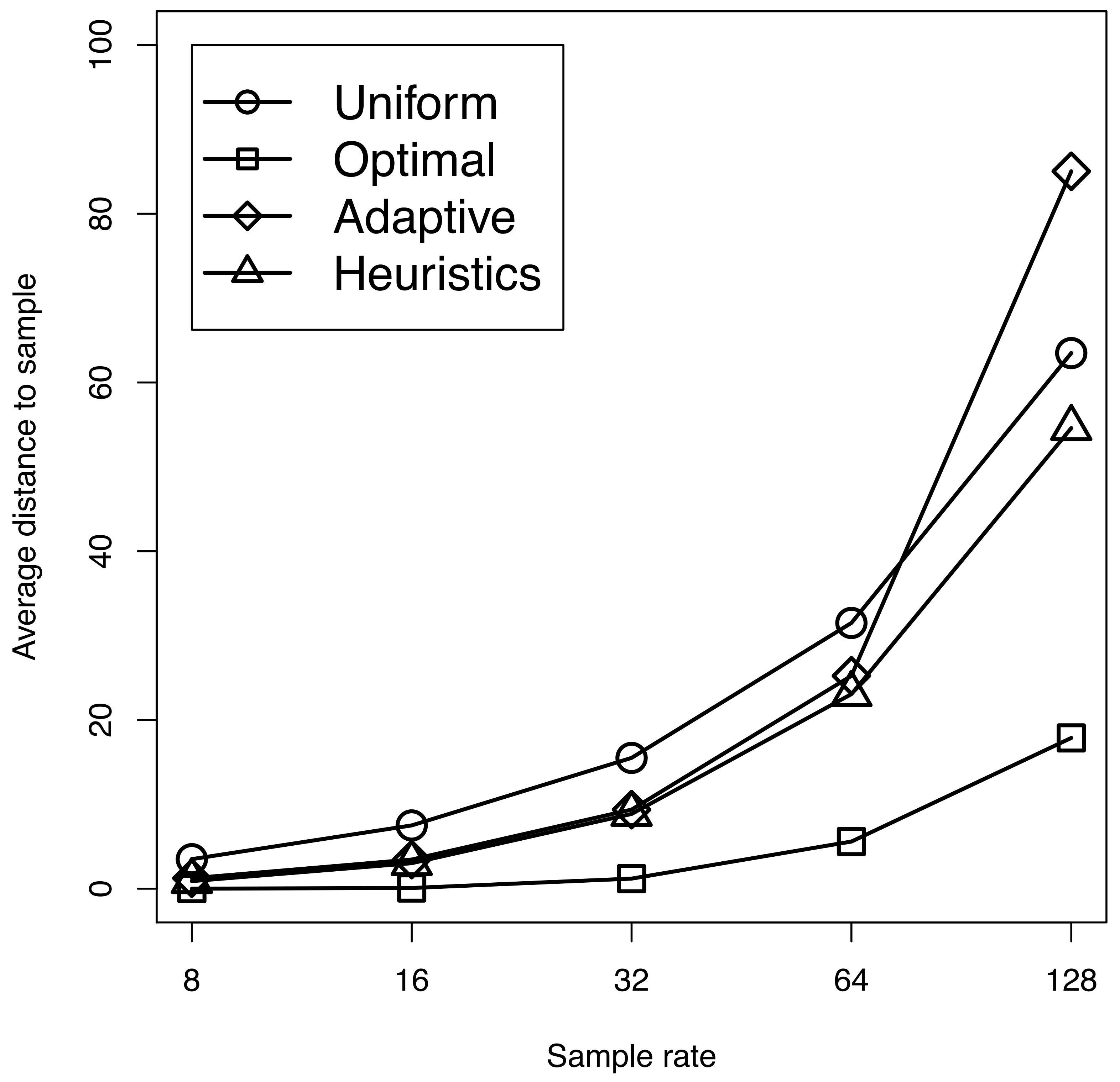
- New samples are added to hash table B.
- If a sample from hash table B is used, it is promoted to hash table A.
- If a collision happens in hash table A, the old sample is demoted to hash table B.
- Good samples are more likely to remain in the hash tables.

Random sampling

- Cost function: $\sum_i f(i) \min \{ i - s \mid s \in S, s \leq i \}$
- If text position i is retrieved by using sample s, we sample it with a probability proportional to i – s.
- Text positions that contribute more to the cost function are more likely to be sampled.

Same data and patterns as in the static case.

100000 queries with 1.3 billion occurrences.



Caching

- In case of a collision, we drop the old sample from the hash table.
- This is similar to random caching policy.
- Would some other policy such as LFU or LRU perform better?
- Can they be implemented in the desired time and space constraints?

Streaming algorithms

- We are looking for an efficient streaming algorithm for approximating the cost function.
- Streaming algorithms already exist for similar problems, such as finding heavy hitters.
- Can they be adapted to our problem?

Thank you!