Efficient Parallel Output-Sensitive Edit Distance

Xiangyun Ding, Xiaojun Dong, Yan Gu, Youzhe Liu, Yihan Sun
University of California, Riverside

Full version: https://arxiv.org/pdf/2306.17461.pdf

Code: https://github.com/ucrparlay/Edit-Distance

Problem Definitions: Edit Distance

- **Input:** $A[1..n]$ and $B[1..m]$ over a fixed alphabet Σ (varieties of characters)
- **Output: Levenshtein distance**
  - unit-cost, single-character
  - insertions, deletions, and substitutions (replacement).

Input:

a	m	s	t	e	r	d	a	m

a	s	t	e	r	i	s	m

Problem Definitions: Edit Distance

- **Input:** $A[1..n]$ and $B[1..m]$ over a fixed alphabet Σ (varieties of characters)
- **Output: Levenshtein distance**
  - unit-cost, single-character
  - insertions, deletions, and substitutions (replacement).

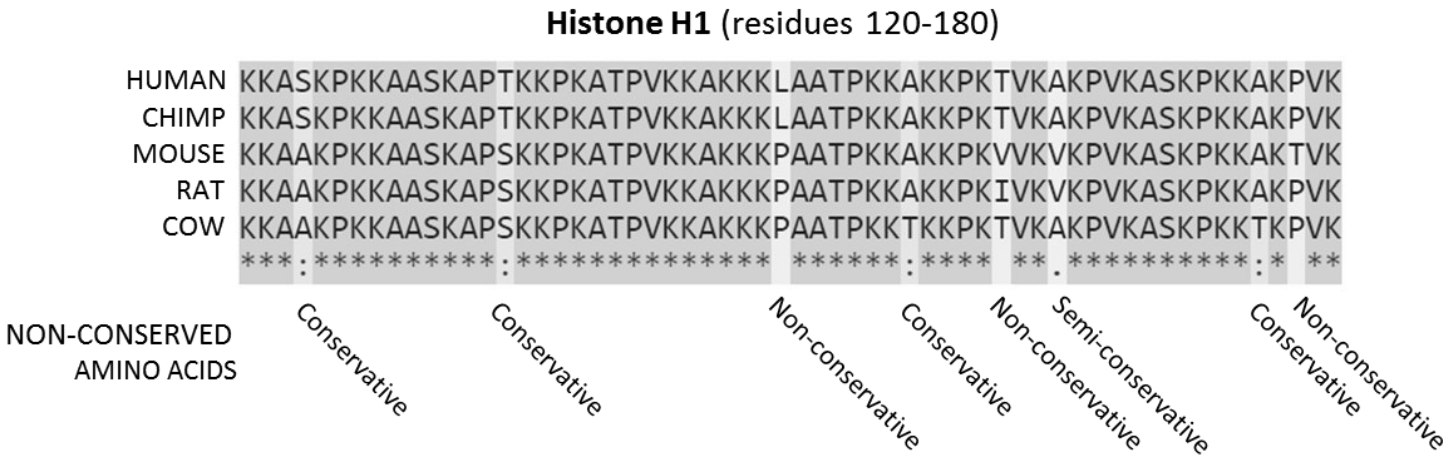
Input:

a	m	s	t	e	r	d	a	m

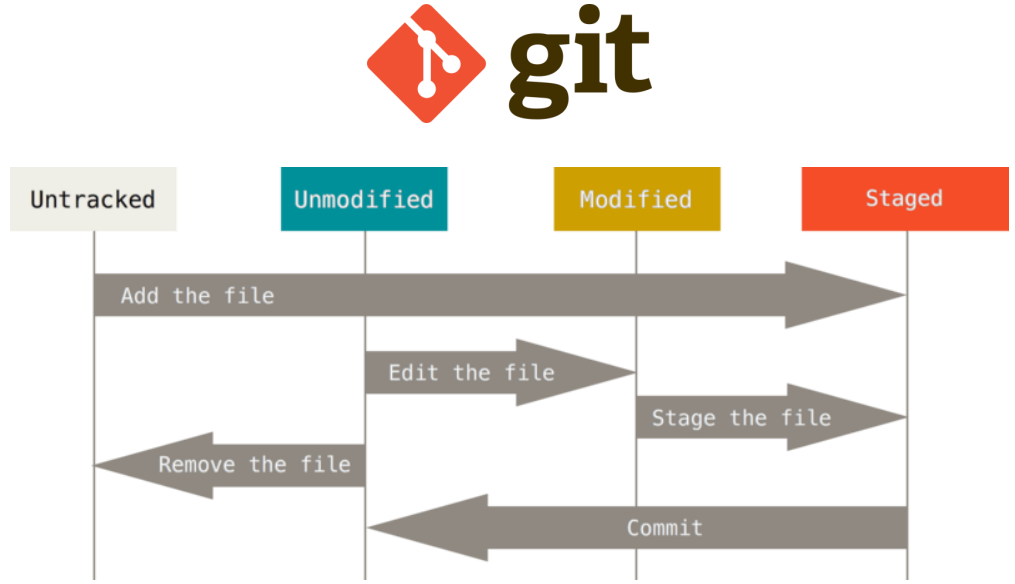
a	s	t	e	r	i	s	m

Output: The edit distance $\boldsymbol{k = 3}$

Edit distance is widely used



Bioinformatics: Sequence alignment [1]

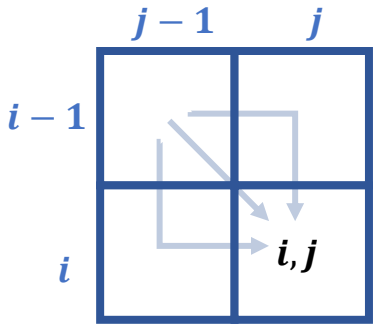


Version control tools: Git[2]

[1] Image from Wikipedia
[2] Image from Git official website

The classic algorithm

Dynamic programming



$$T[i,j] = min \begin{cases} T[i-1,j]+1 \\ T[i,j-1]+1 \\ T[i-1,j-1]+\delta(x[i-1],y[j-1]) \end{cases}$$

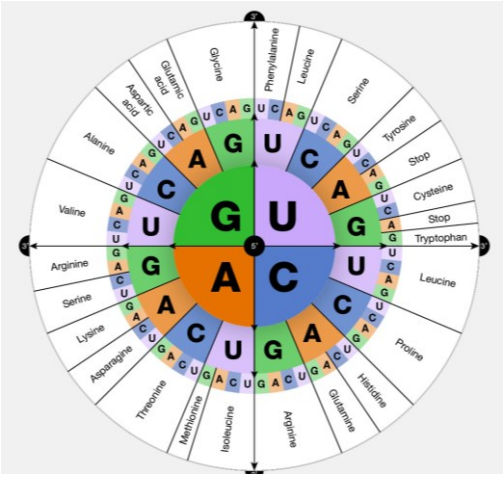
$$= 0 \quad \text{If } x[i-1] = y[j-1]$$
$$= 1 \quad \text{If } x[i-1] \neq y[j-1]$$

size of the two sequences

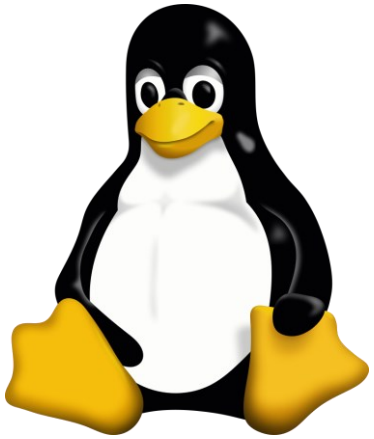
Work (time complexity): $O(mn)$

		A	s	t	e	r	i	s	m
	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
m	2	1	1	2	3	4	5	6	6
s	3	2	1	2	3	4	5	5	6
t	4	3	2	1	2	3	4	5	6
e	5	4	3	2	1	2	3	4	5
r	6	5	4	3	2	1	2	3	4
d	7	6	5	4	3	2	2	3	4
a	8	7	6	5	4	3	3	3	4
m	9	8	7	6	5	4	4	4	3

The sizes of m and n are quite large!



Bacteria*: about **4.6 million** base pairs



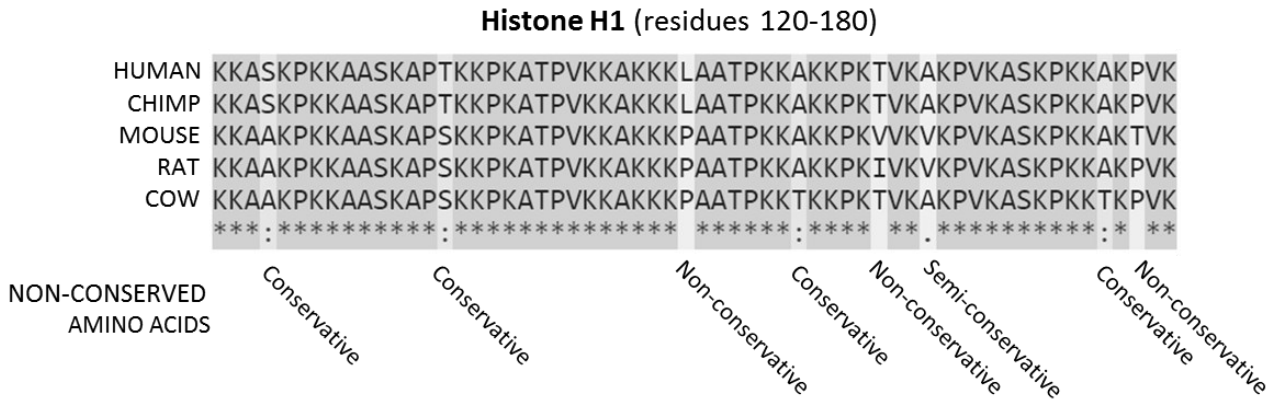
Linux kernel*: around **27 million** lines of code

Input: millions-scale or even larger

Better solutions for large-scale applications in practice?

* Figure from Wikipedia

The two sequences are very similar!



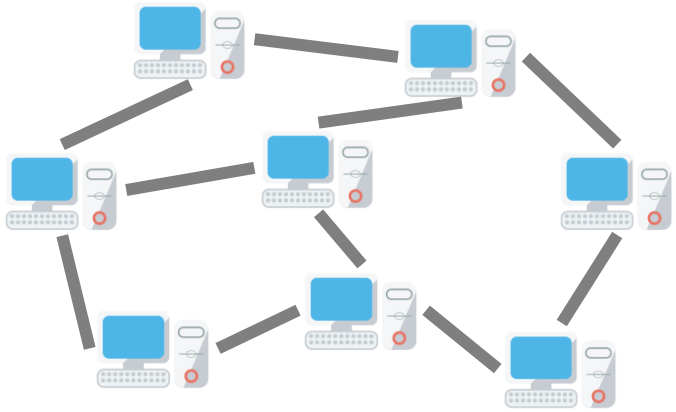
Alignment: edits are **small**
(we care alignments on similar sequence pairs)

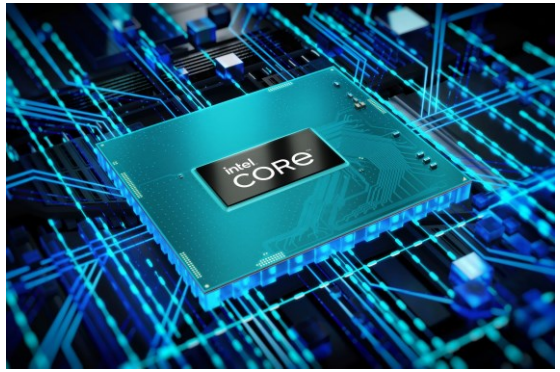
Treat different sizes of changes in different ways

The two sequences are generally very similar! $\longrightarrow$ #edits: $k = o(n)$

Output-sensitive algorithms!
$\Rightarrow$ When $k = o(n)$, the cost is $o(nm)$

Another resource: parallelism





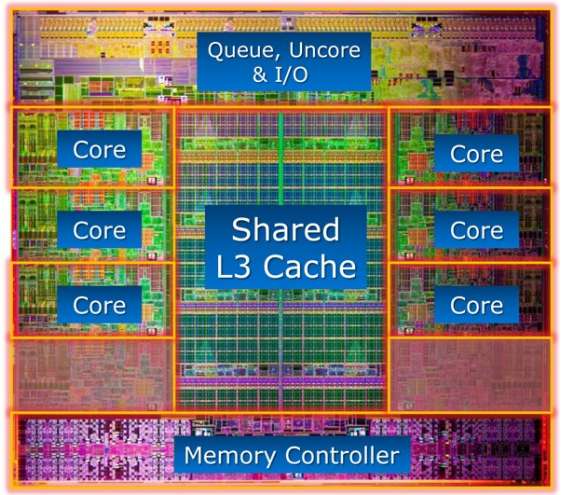
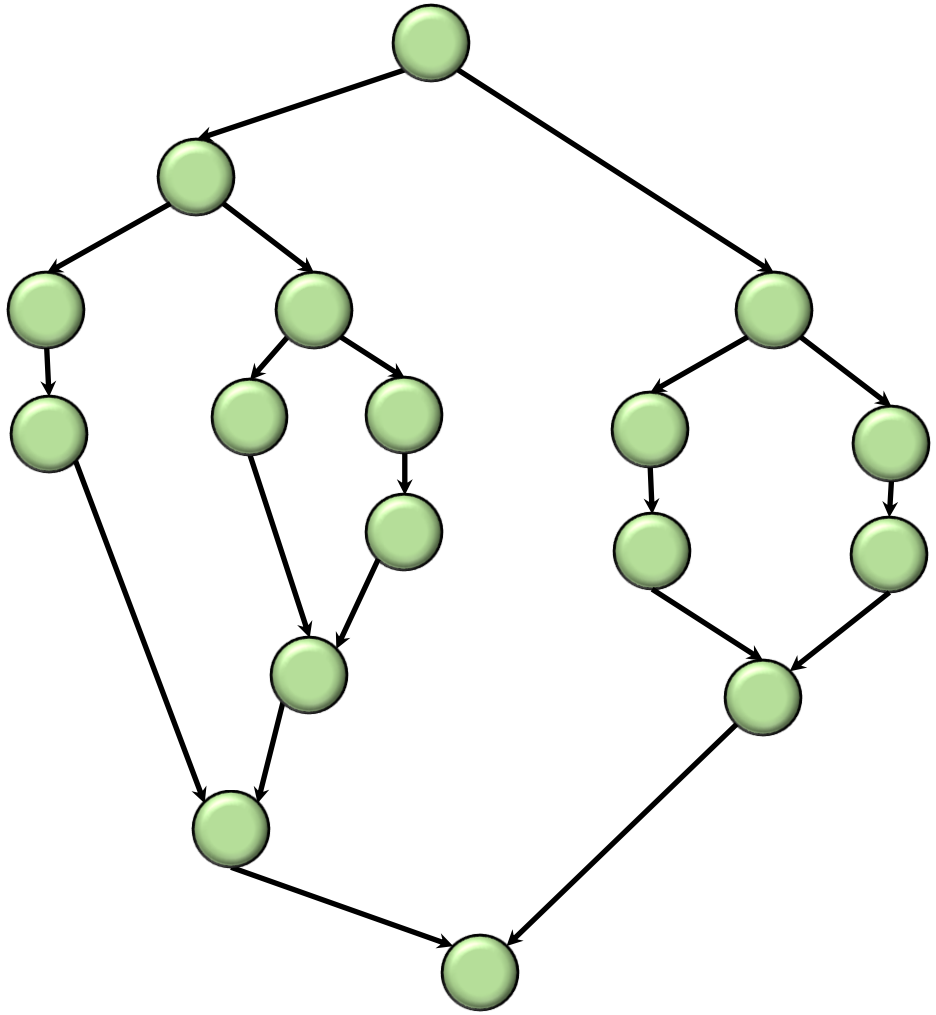


Figure credit: Wikipedia

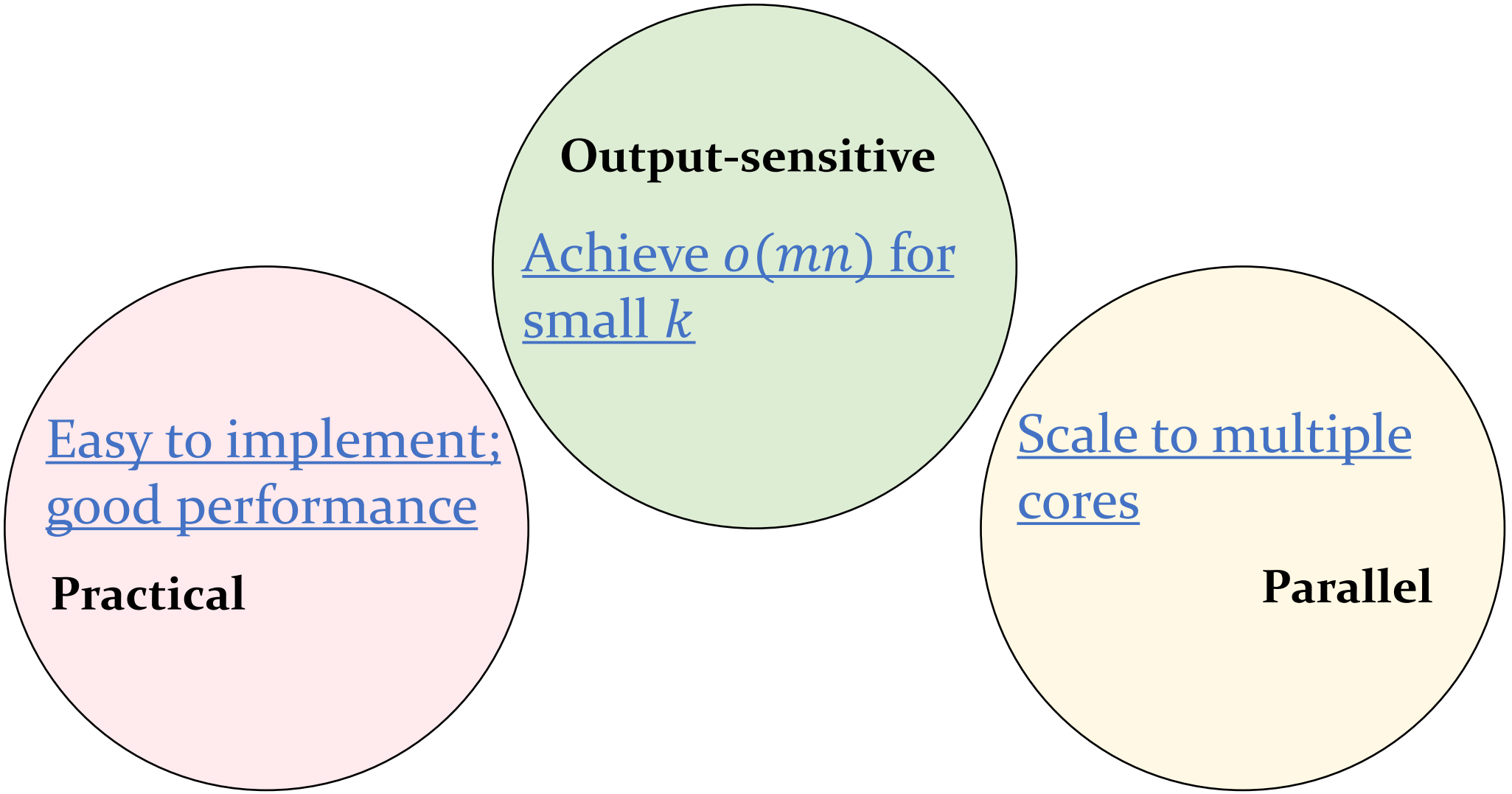
Parallelism is ubiquitous and can be used to accelerate algorithms

Computational Model

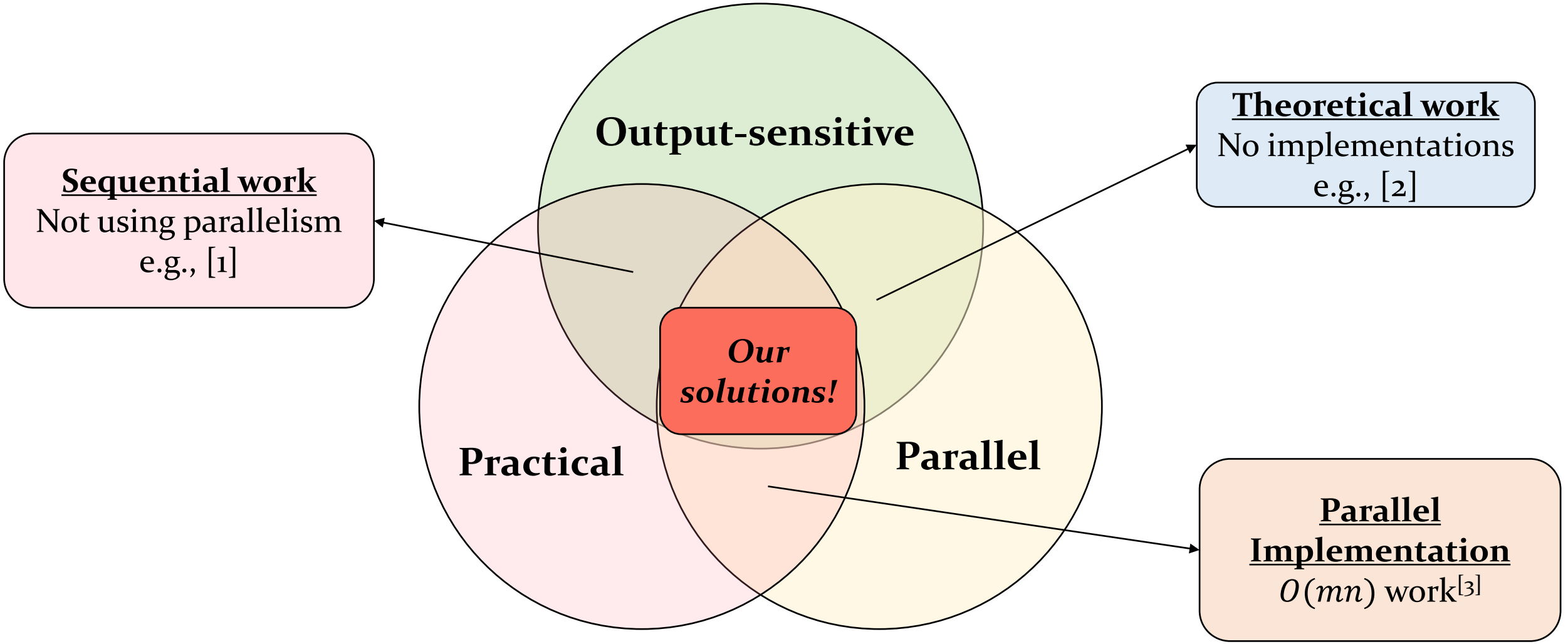
- Shared-memory multi-core setting
- **Work**: the total number of operations
  - = running time on one core
- **Span** (depth): the length of the longest dependency chain
  - = running time on an infinite number of cores



Our goals

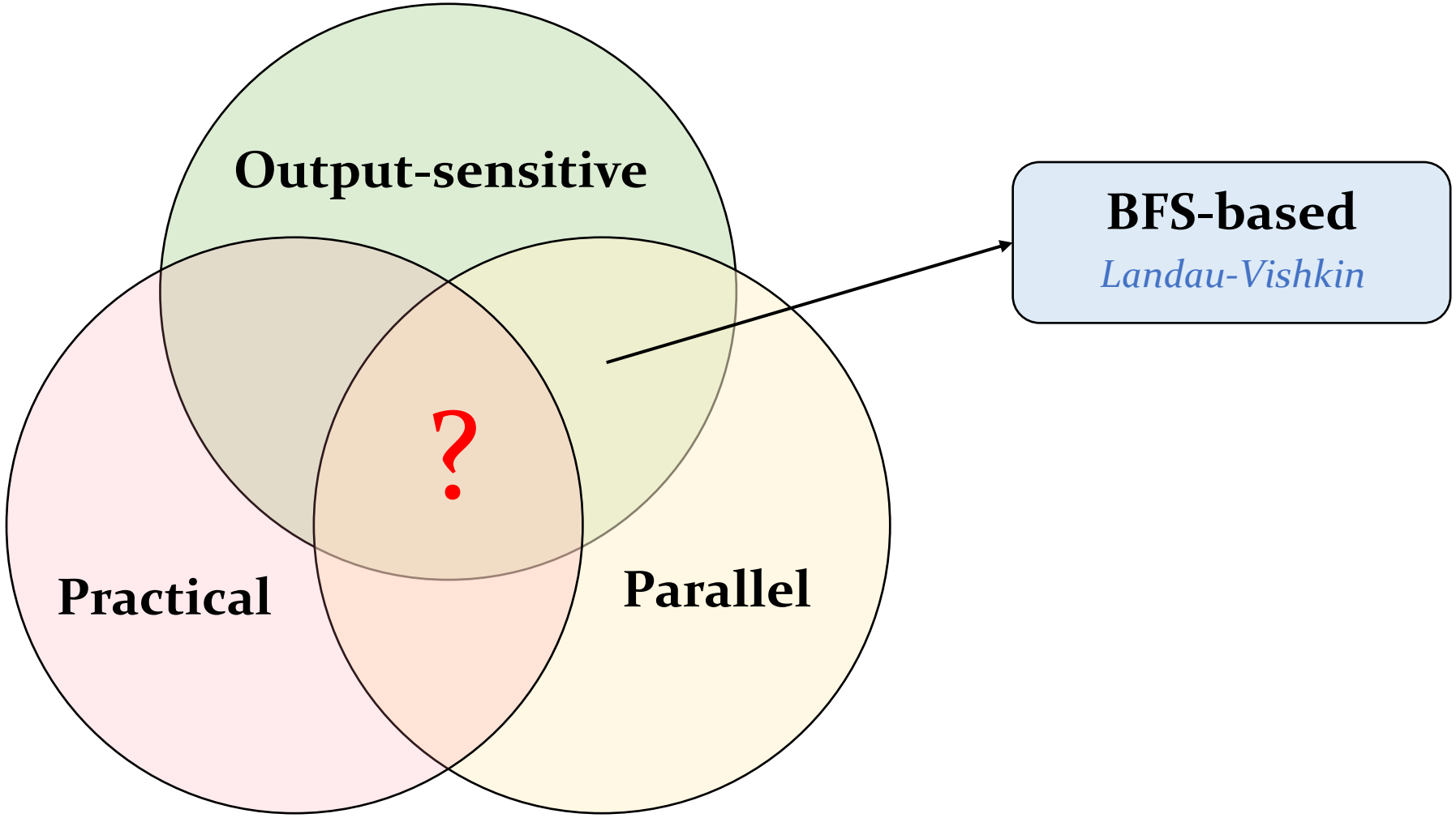

Output-sensitive
Achieve $o(mn)$ for small k
Easy to implement; good performance
Practical
Scale to multiple cores
Parallel

Our goals

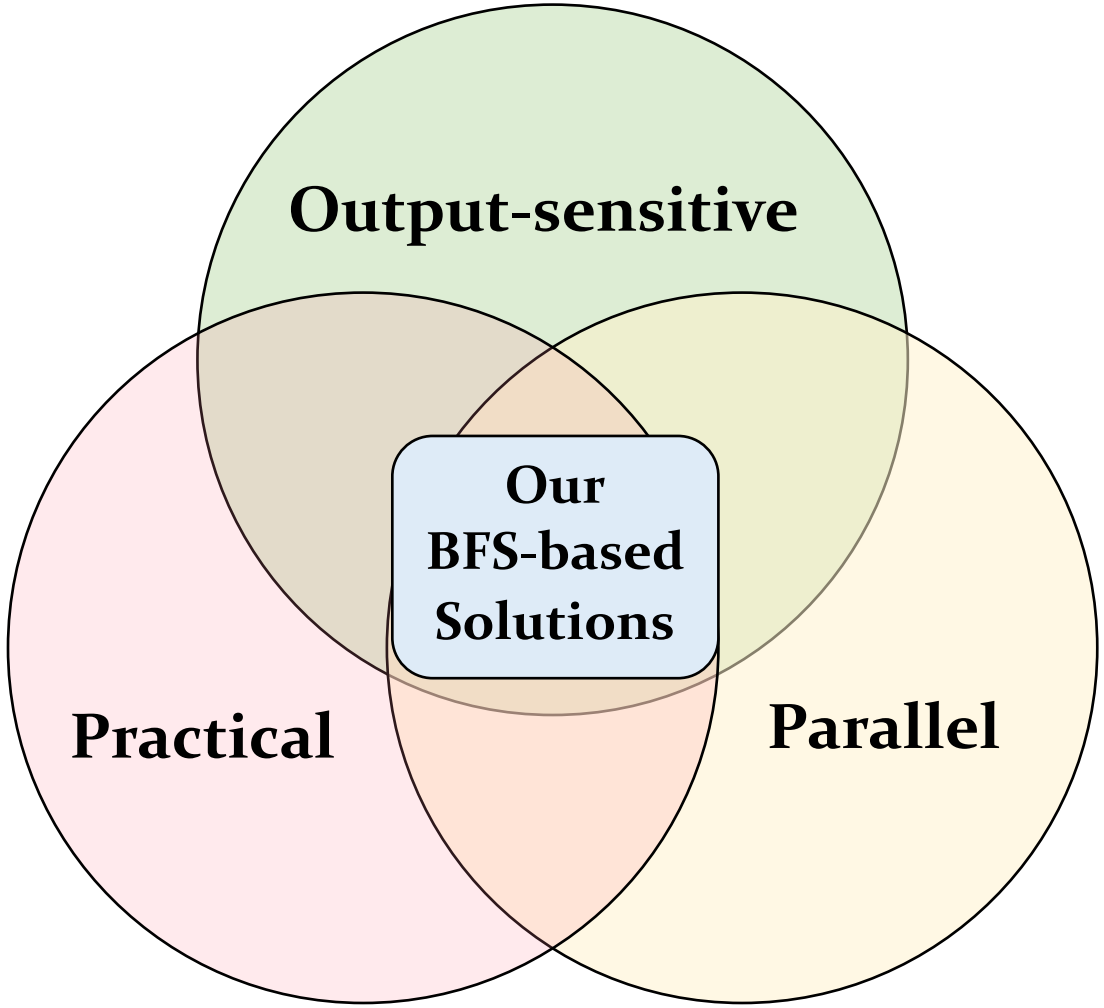


[1] Marçais, DeBlasio, Pandey, and Kingsford. Locality-sensitive hashing for the edit distance. Bioinformatics, 2019.
[2] Landau and Vishkin. Fast parallel and serial approximate string matching. J. Algorithms, 1989.
[3] Blelloch, Anderson, and Dhulipala. "ParlayLib-a toolkit for parallel algorithms on shared-memory multicore machines

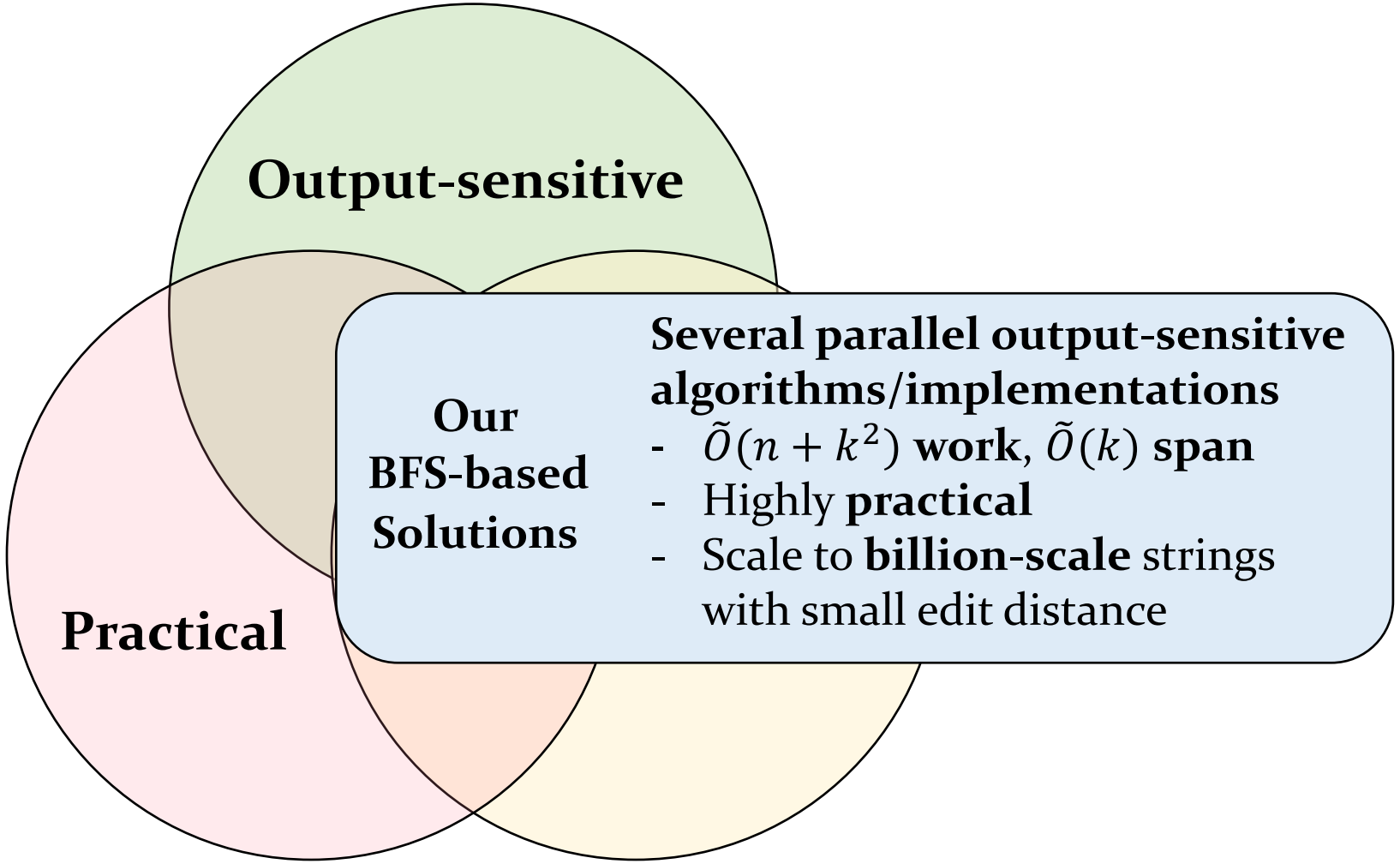
Our contributions: in practice



Our contributions: in practice



Our contributions: in practice

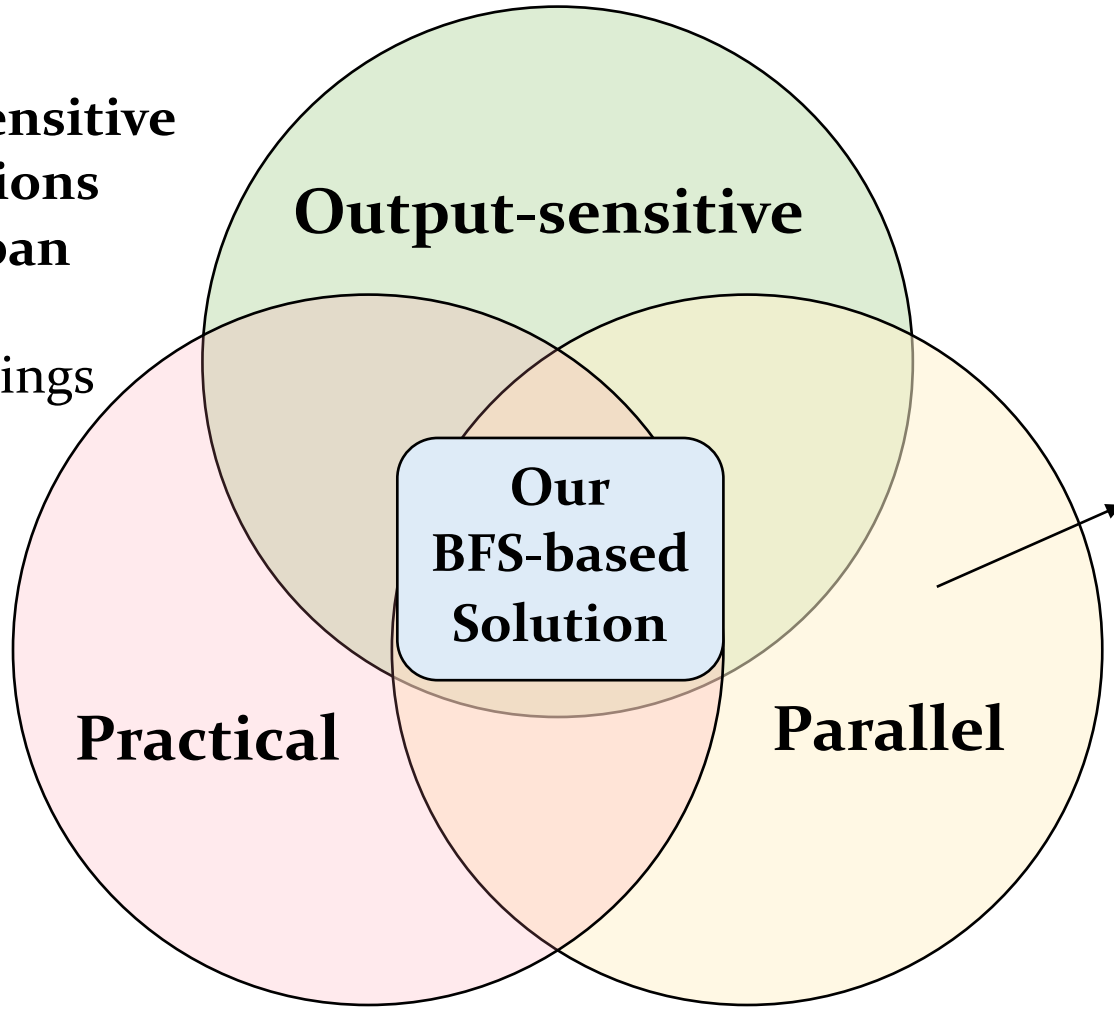


Our contributions: in theory

In practice

Several parallel output-sensitive algorithms/implementations

- $\tilde{O}(n + k^2)$ **work**, $\tilde{O}(k)$ **span**
- Highly **practical**
- Scale to **billion-scale** strings with small distance



DaC (Divide-and-conquer)

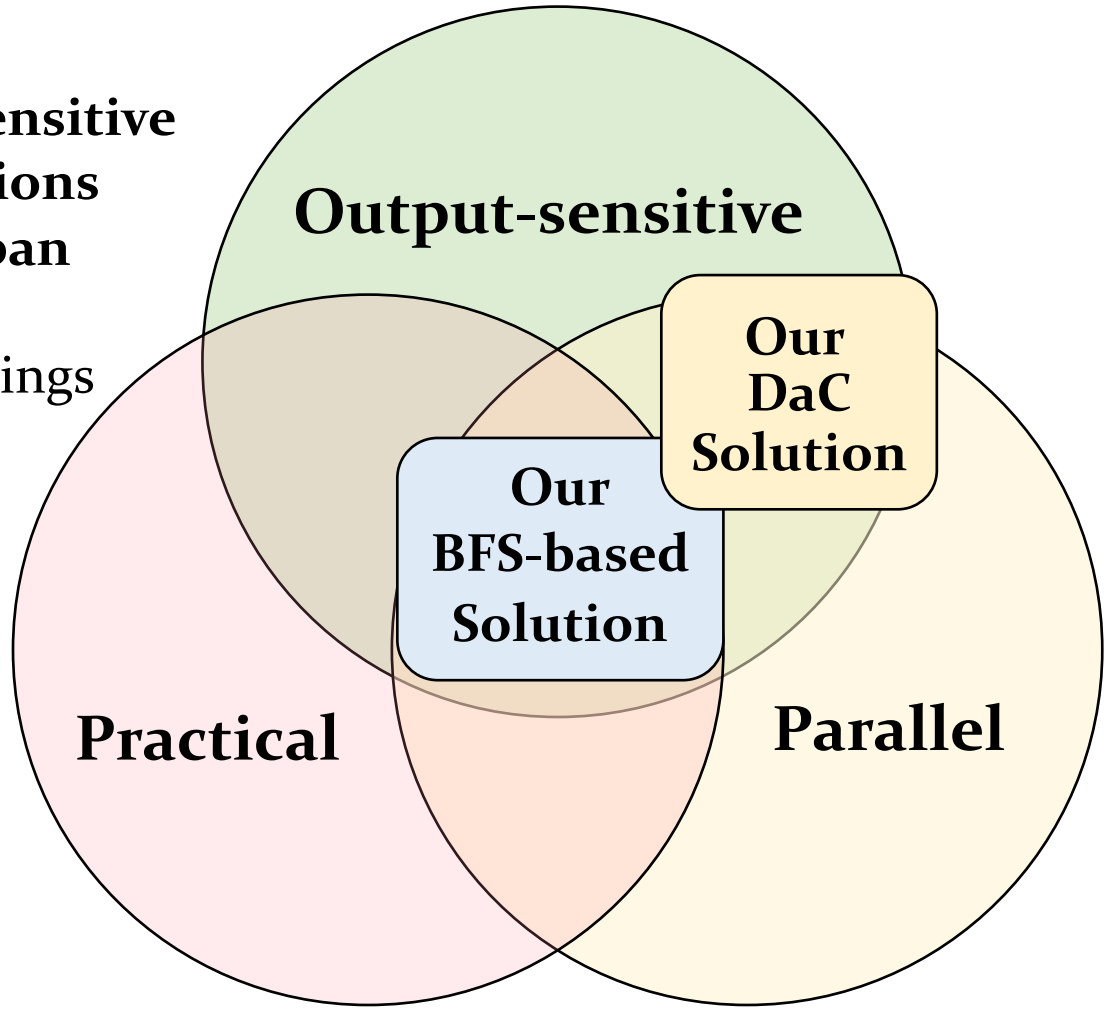
The AALM algorithm[1]

Our contributions: in theory

In practice

Several parallel output-sensitive algorithms/implementations

- $\tilde{O}(n + k^2)$ **work**, $\tilde{O}(k)$ **span**
- Highly **practical**
- Scale to **billion-scale** strings with small distance

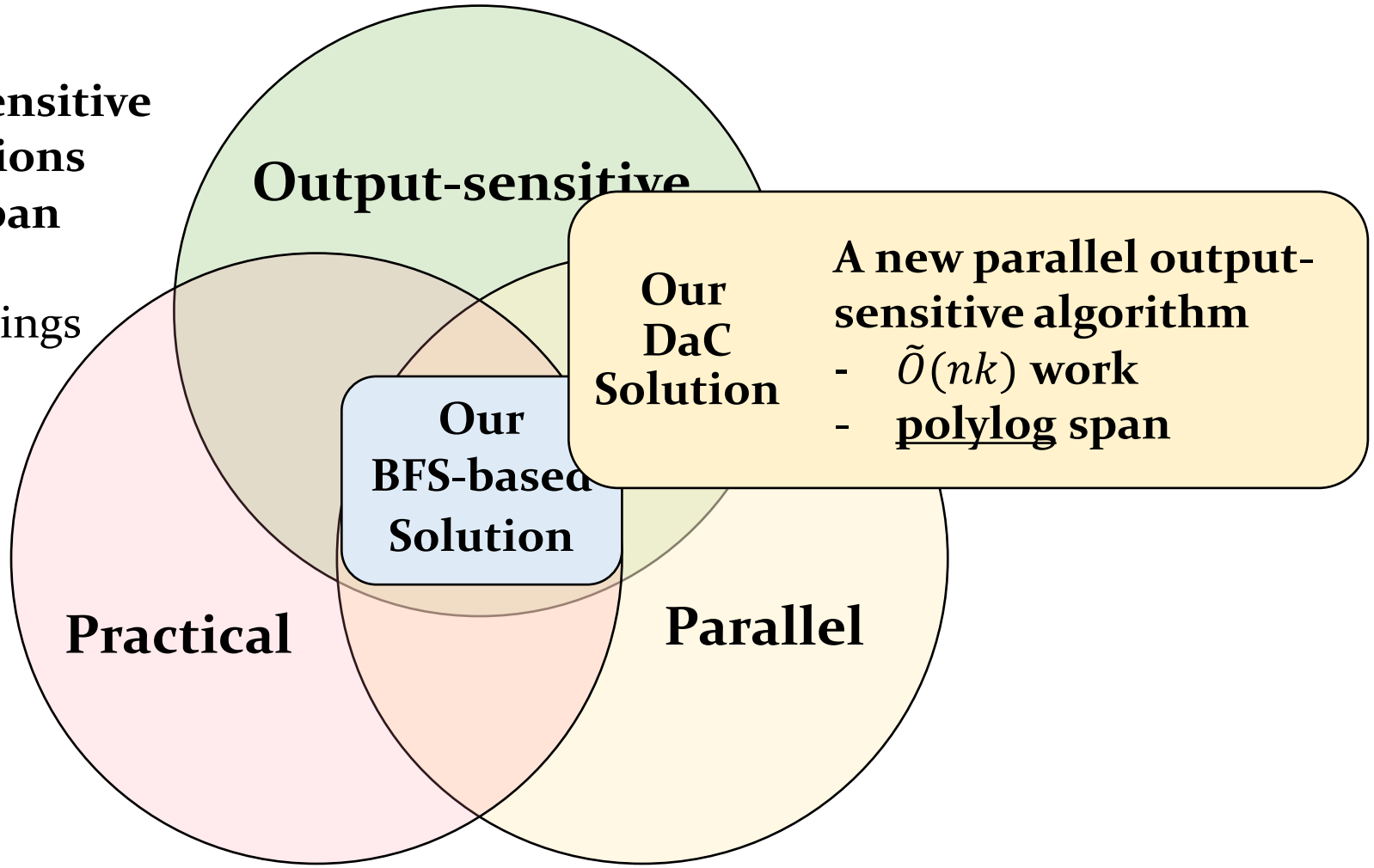


Our contributions: in theory

In practice

Several parallel output-sensitive algorithms/implementations

- $\tilde{O}(n + k^2)$ **work**, $\tilde{O}(k)$ **span**
- Highly **practical**
- Scale to **billion-scale** strings with small distance

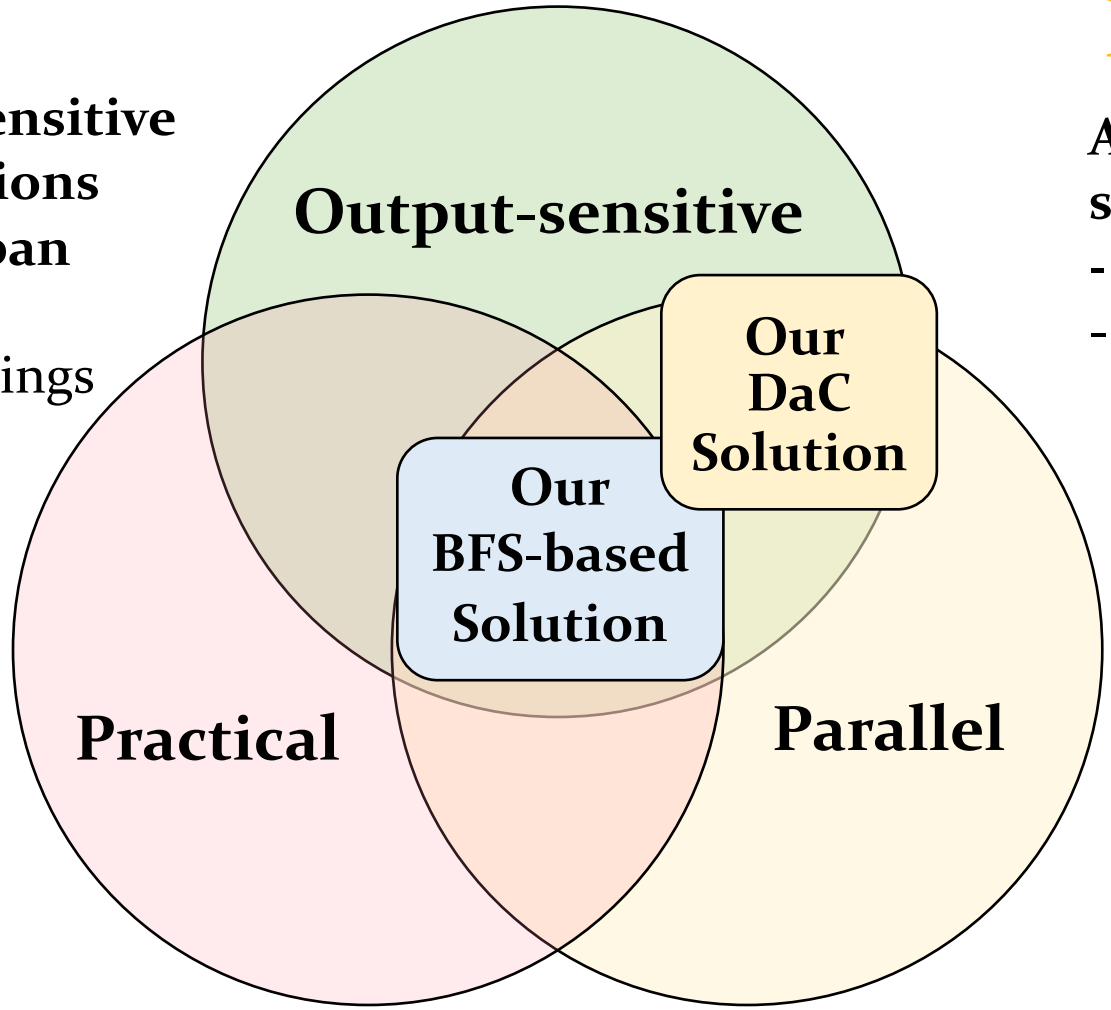


Our contributions

In practice

Several parallel output-sensitive algorithms/implementations

- $\tilde{O}(n + k^2)$ **work**, $\tilde{O}(k)$ **span**
- Highly **practical**
- Scale to **billion-scale** strings with small distance



In theory

A new parallel output-sensitive algorithm

- $\tilde{O}(nk)$ **work**
- **polylog span**

In practice

Parallel BFS-based algorithms

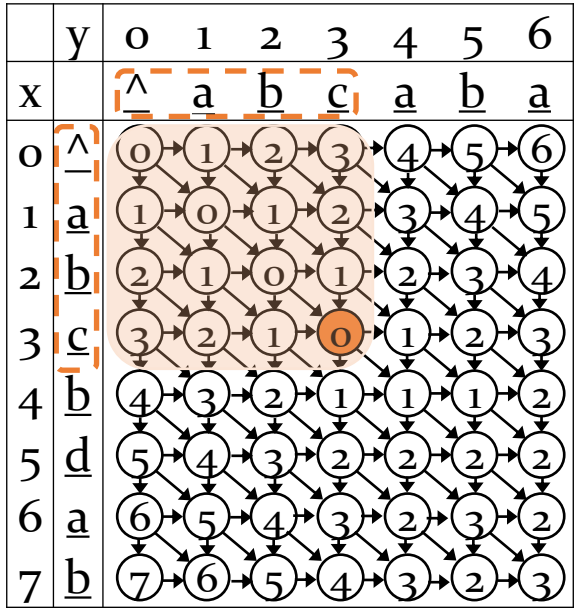
BFS on the DAG

	y	0	1	2	3	4	5	6
x		^	a	b	c	a	b	a
0	^	0	1	2	3	4	5	6
1	a	1	0	1	2	3	4	5
2	b	2	1	0	1	2	3	4
3	c	3	2	1	0	1	2	3
4	b	4	3	2	1	1	1	2
5	d	5	4	3	2	2	2	2
6	a	6	5	4	3	2	3	2
7	b	7	6	5	4	3	2	3

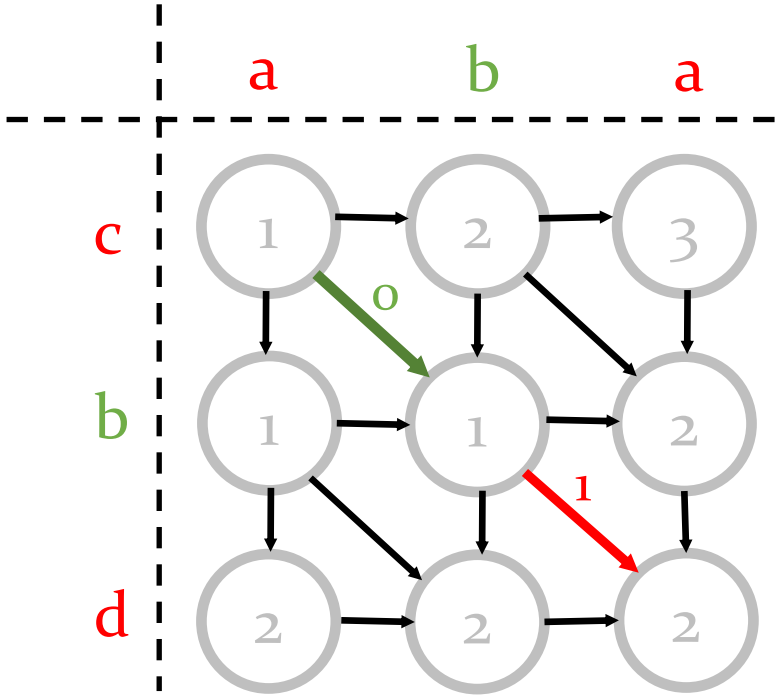
BFS on the DAG

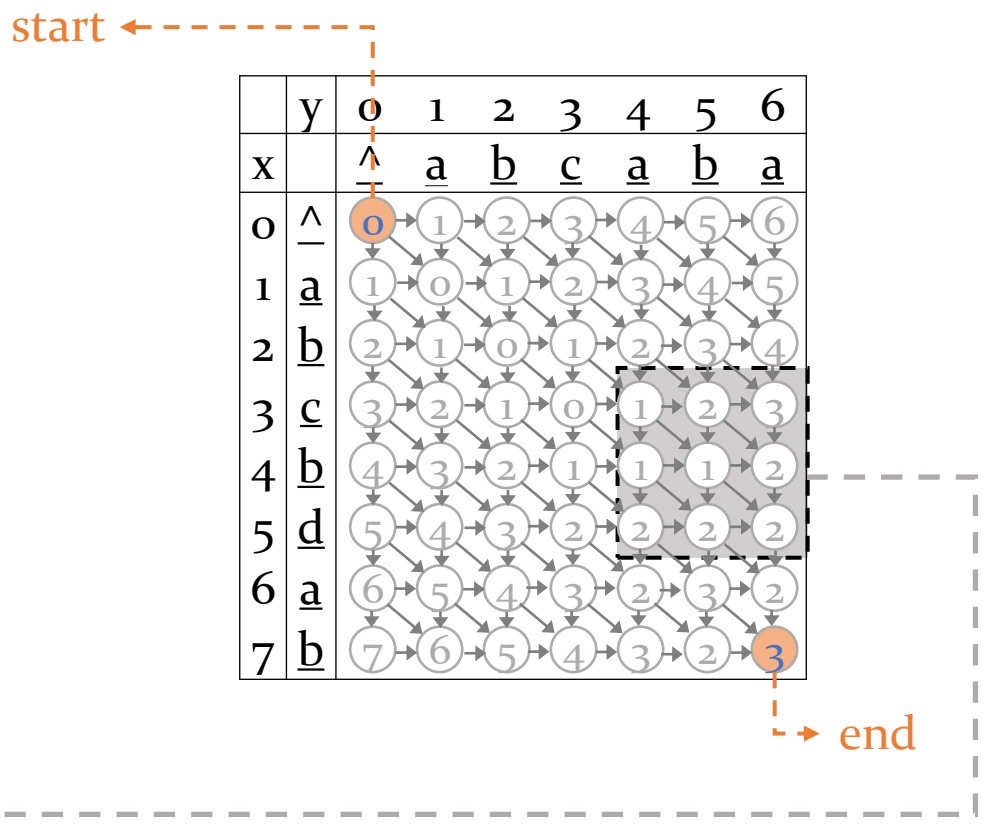
	y	0	1	2	3	4	5	6
x		^	a	b	c	a	b	a
0	^	0	1	2	3	4	5	6
1	a	1	0	1	2	3	4	5
2	b	2	1	0	1	2	3	4
3	c	3	2	1	0	1	2	3
4	b	4	3	2	1	1	1	2
5	d	5	4	3	2	2	2	2
6	a	6	5	4	3	2	3	2
7	b	7	6	5	4	3	2	3

BFS on the DAG



BFS on the DAG

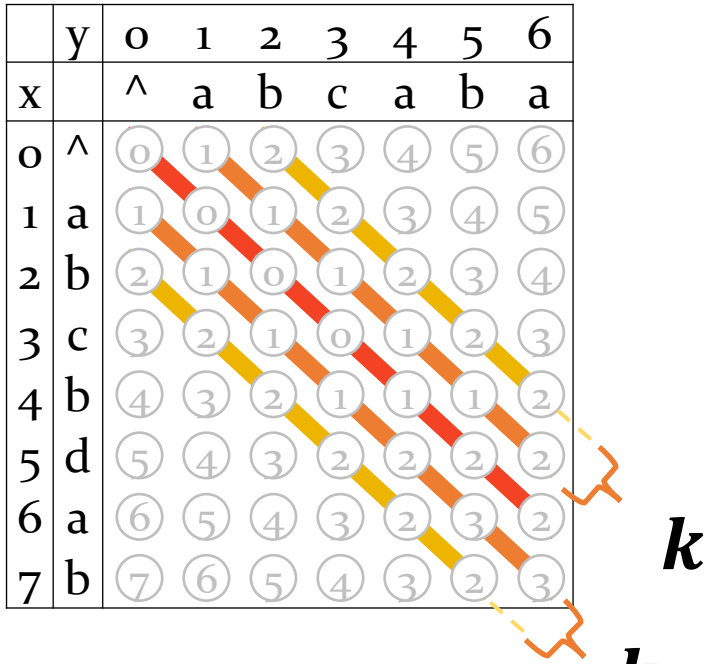




BFS on the DAG: two key observations

	y	0	1	2	3	4	5	6
x		^	a	b	c	a	b	a
0	^	0	1	2	3	4	5	6
1	a	1	0	1	2	3	4	5
2	b	2	1	0	1	2	3	4
3	c	3	2	1	0	1	2	3
4	b	4	3	2	1	1	1	2
5	d	5	4	3	2	2	2	2
6	a	6	5	4	3	2	3	2
7	b	7	6	5	4	3	2	3

BFS on the DAG: two key observations



The edit distance is 3

BFS on the DAG: two key observations

Vertices (states) with $|x - y| > k$ will **NOT** be touched

Work: $O(kn)$

↓

Output-sensitive !

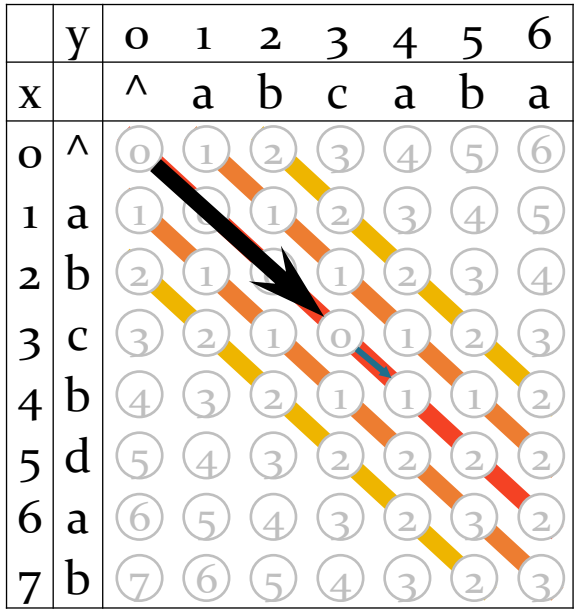
	y	0	1	2	3	4	5	6
x		^	a	b	c	a	b	a
0	^	0	1	2	3	4	5	6
1	a	1	0	1	2	3	4	5
2	b	2	1	0	1	2	3	4
3	c	3	2	1	0	1	2	3
4	b	4	3	2	1	1	1	2
5	d	5	4	3	2	2	2	2
6	a	6	5	4	3	2	3	2
7	b	7	6	5	4	3	2	3

The edit distance is **3**

BFS on the DAG: two key observations

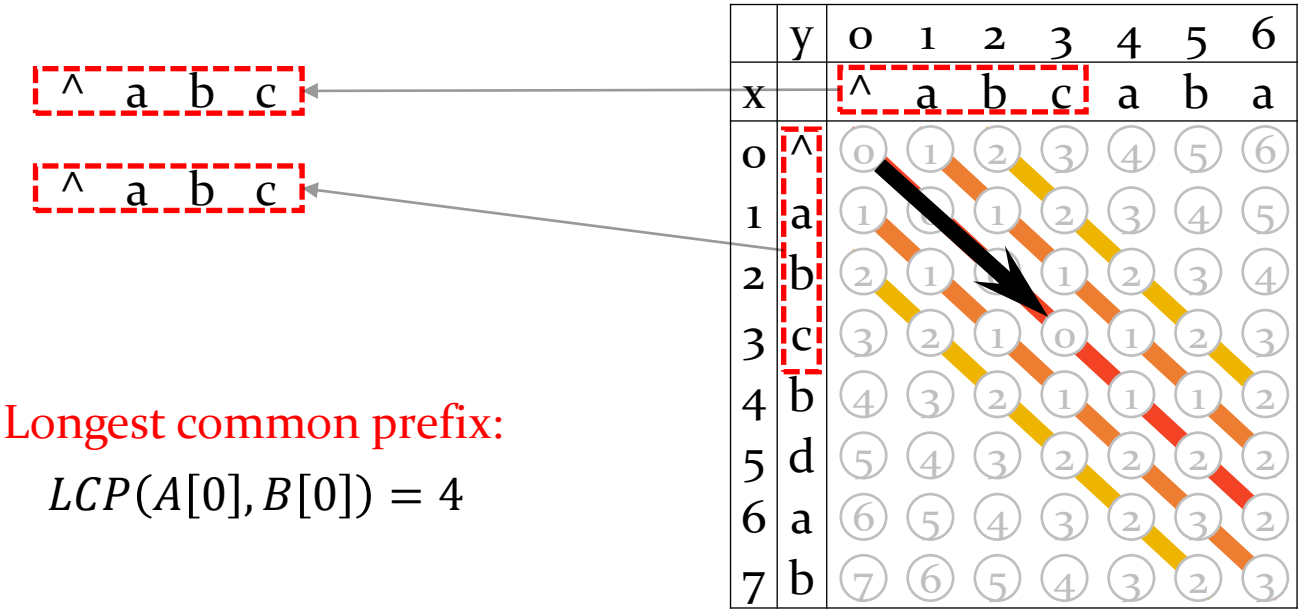
Shown by **Ukkonen** in his paper:

Algorithms for approximate string matching. Information and Control, 1985.



The edit distance is **3**

BFS on the DAG: two key observations

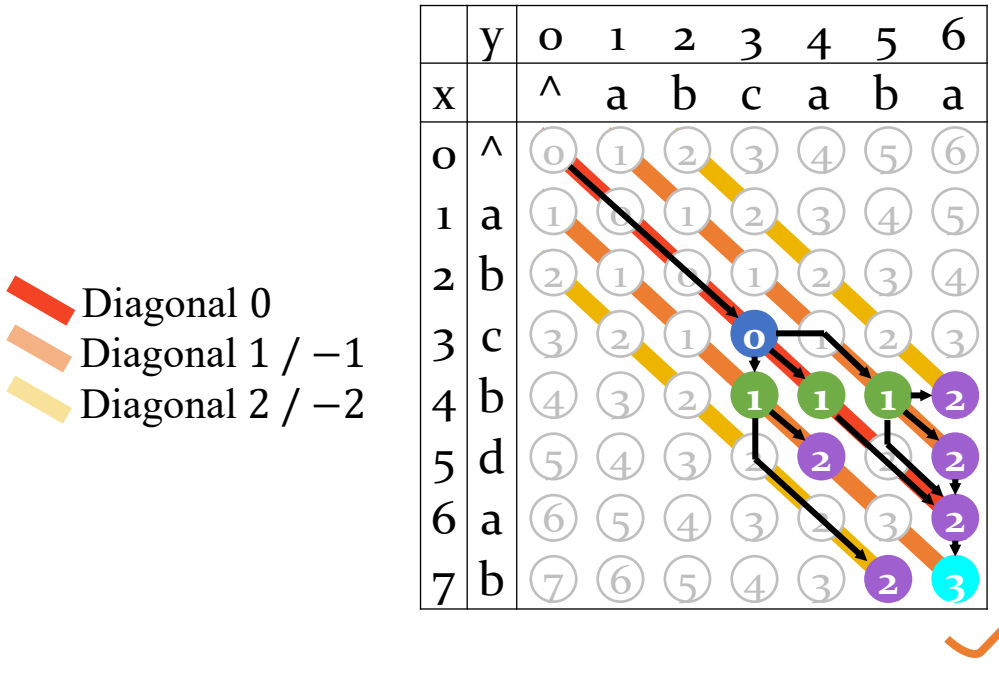


Longest common prefix:

$LCP(A[0], B[0]) = 4$

The edit distance is **3**

BFS on the DAG: two key observations



1. Restricted computing area → k rounds
2. LCP computing instead → $O(k)$ vertices per round

→ $O(k^2)$ states

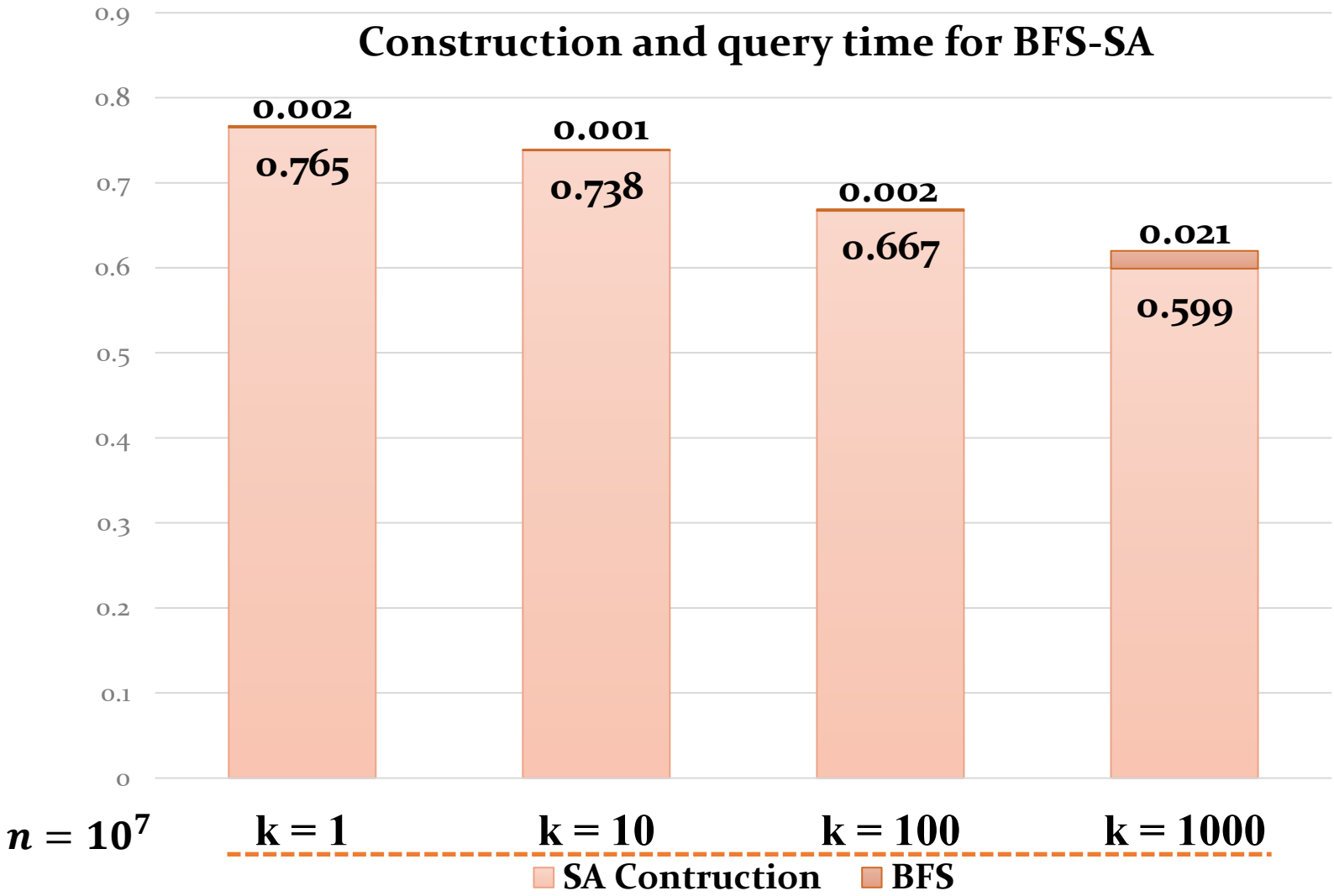
How to compute the LCP

- Suffix tree by Landau and Vishkin[1]
  - Hard to implement, especially in parallel settings
  - the large constant of its work leads to a poor performance in practice

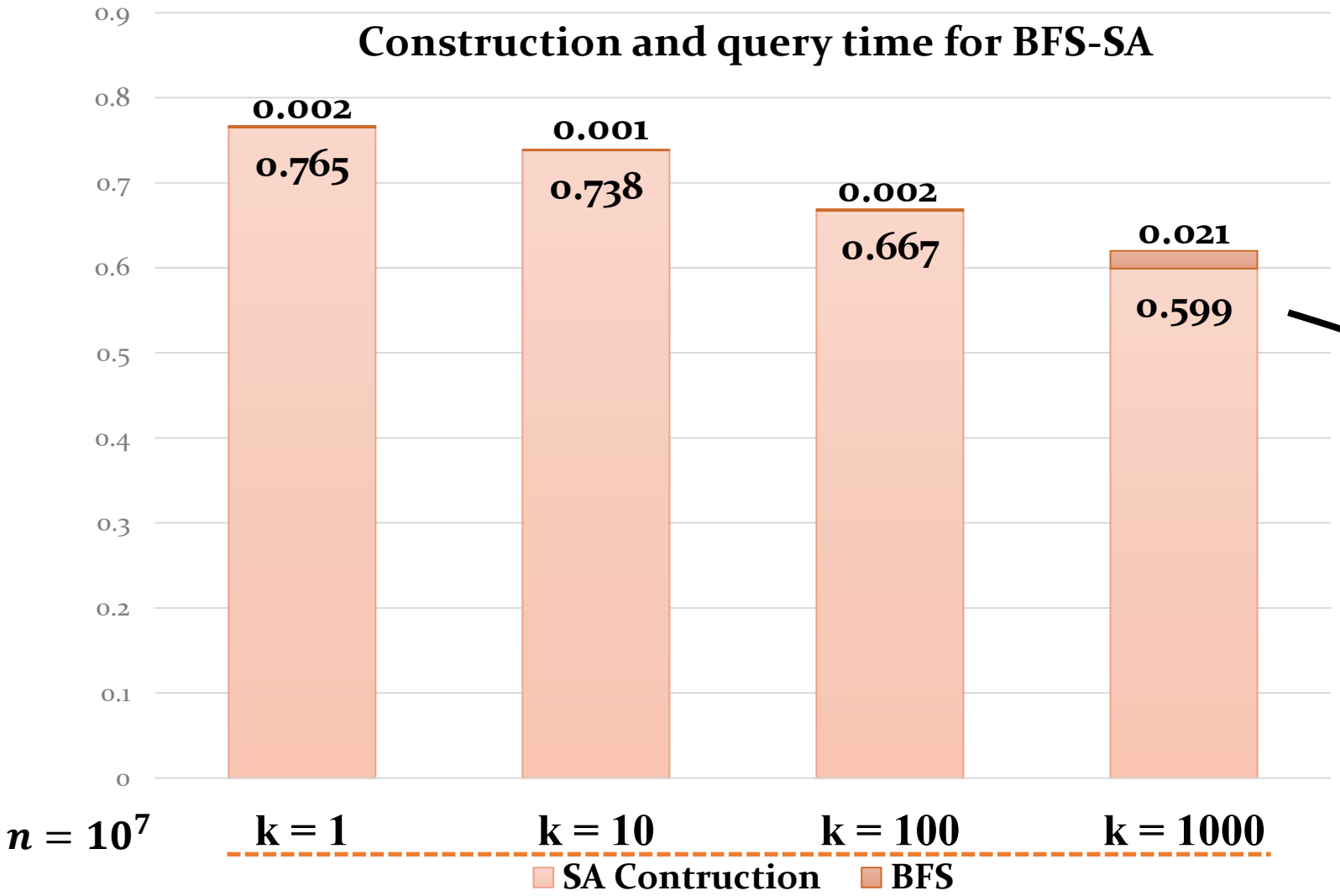
- **Our improvement & implementation: BFS-SA**
  - Alternative data structure: Suffix array[2]
  - Easy to implement in parallel
  - Achieve better performance

[1] Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. J. Algorithms, 1989.
[2] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction, ICALP, 2003.

BFS-SA performance at a glance



BFS-SA performance at a glance

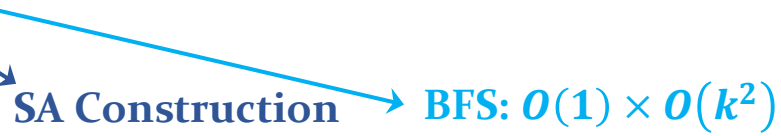


$n = 10^7$
$k = 1 \sim 1000$
Output-sensitive?

The construction time is too long!

Solutions with more practical design?

- Look at the cost of BFS-SA
  - **Work**: $O(n + k^2)$
  - **Span**: $\tilde{O}(k)$



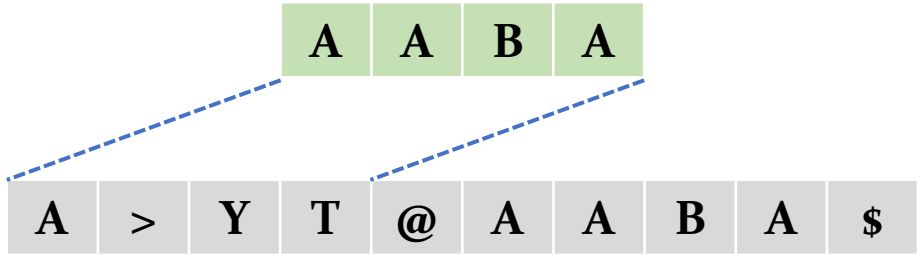
- Expecting trade-offs for better solutions
  - **Accelerate** the **construction** speed as much as possible
  - Acceptable **sacrifice** on **BFS** query work

Our contribution in practice: faster algorithm using hashing

Parallel BFS-based algorithms

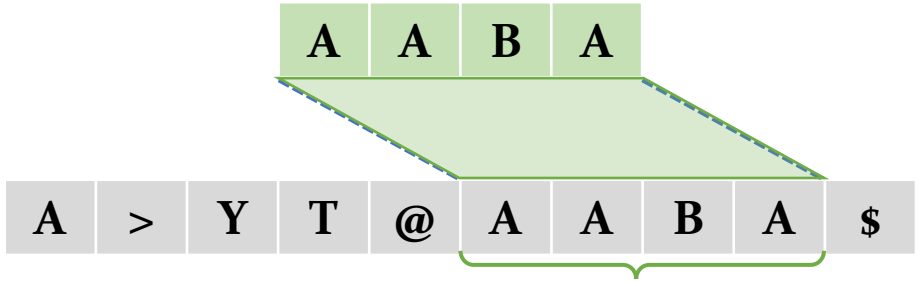
Prefix table for edit distance

- **String hashing** for string matching
  - Compare the (sub)string based on hashing values



Prefix table for edit distance

- **String hashing** for string matching
  - Compare the (sub)string based on hashing values



Prefix table for edit distance

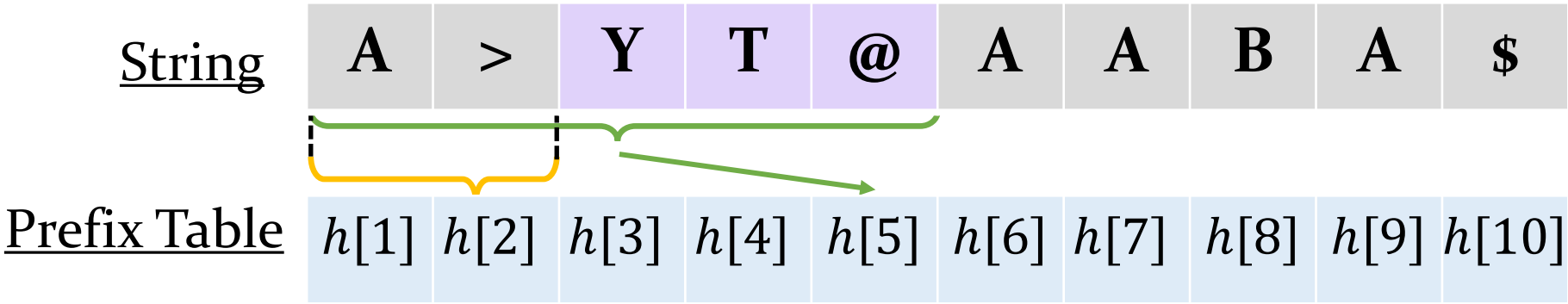
- **The "string hashing arithmetic"**
  - Preprocessing a "prefix table"

String	A	>	Y	T	@	A	A	B	A	$
Prefix Table	$h[1]$	$h[2]$	$h[3]$	$h[4]$	$h[5]$	$h[6]$	$h[7]$	$h[8]$	$h[9]$	$h[10]$

$$h[4] = h(A[1..4]) = h[3] \oplus h(A[4])$$

Prefix table for edit distance

- **The "string hashing arithmetic"**
  - Preprocessing a "prefix table"
  - Extracting the hashing value of a substring in $O(1)$ work



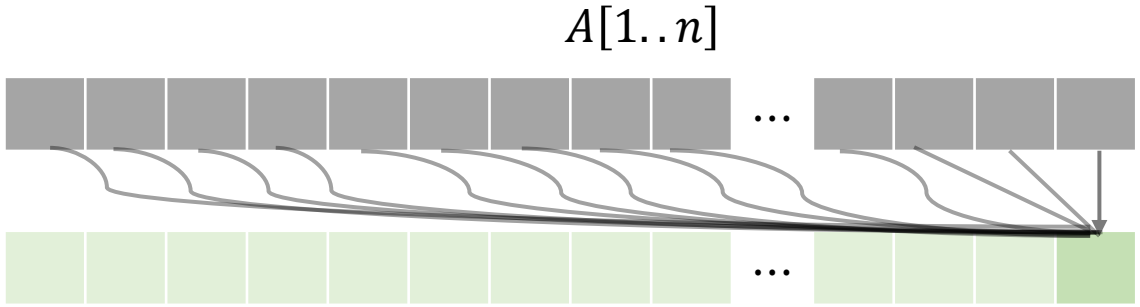
$$h(A[2..5]) = h[5] \ominus h[2]$$

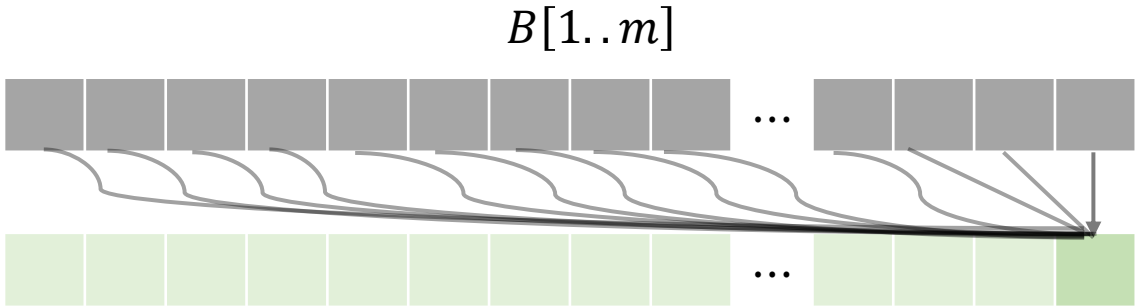
Prefix table construction

- Prefix tables construction for fast LCP query

Prefix table construction

- Prefix tables construction for fast LCP query

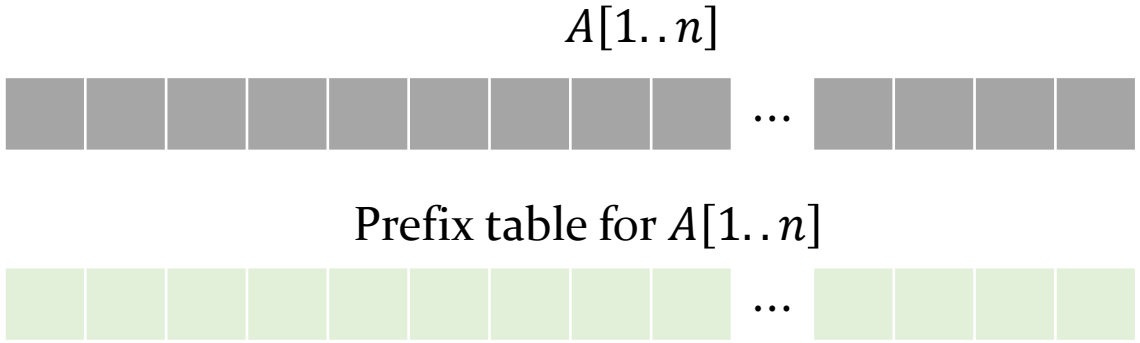


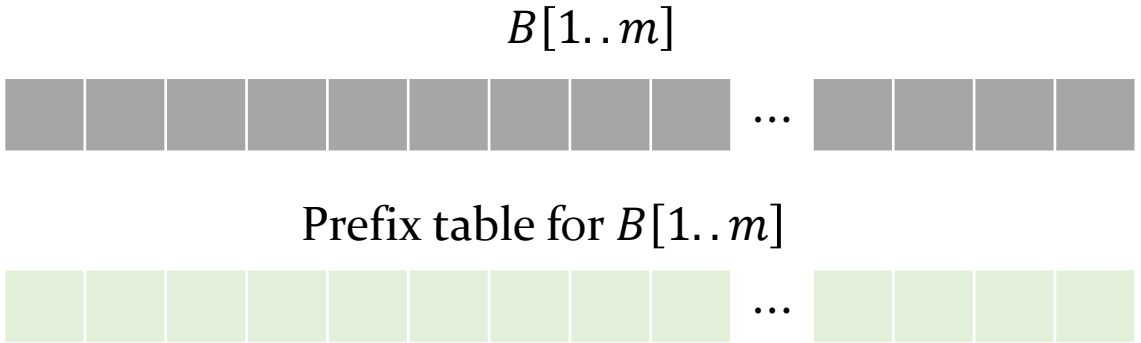


Prefix sum in parallel

Query based on the prefix tables

- Prefix tables construction for fast LCP query

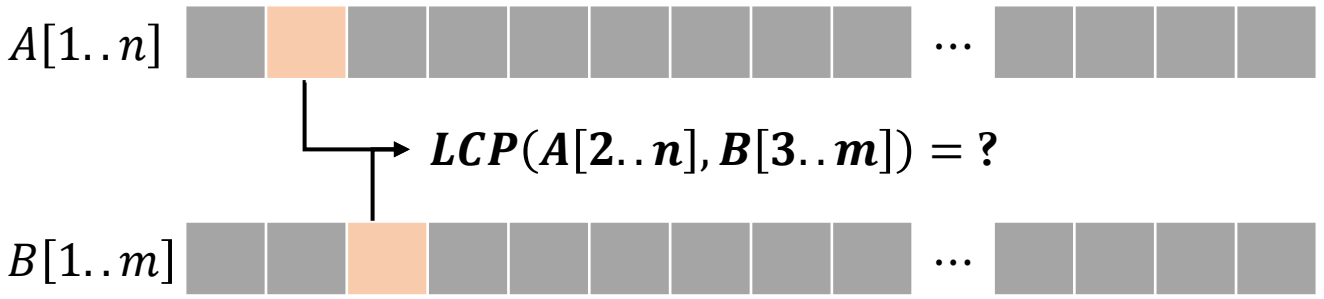




For construction:
Work : $O(n)$
Span: $O(\log n)$

Query based on the prefix tables

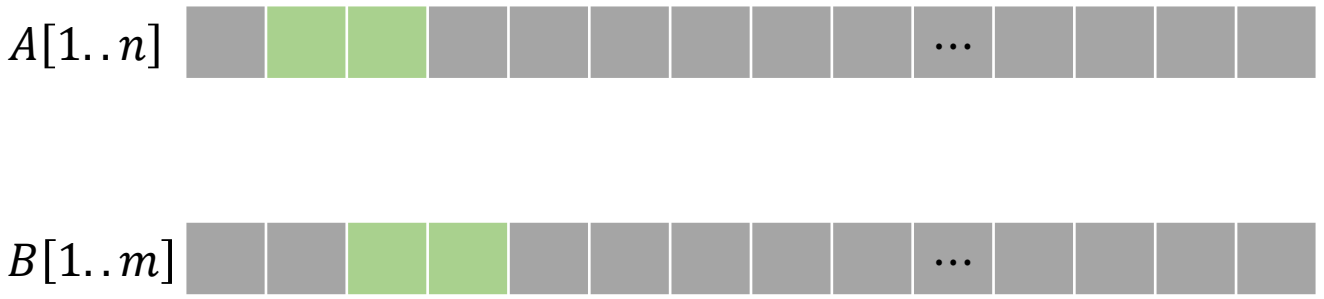
- LCP query based on the prefix tables



Dual binary search: to find the longest common prefix

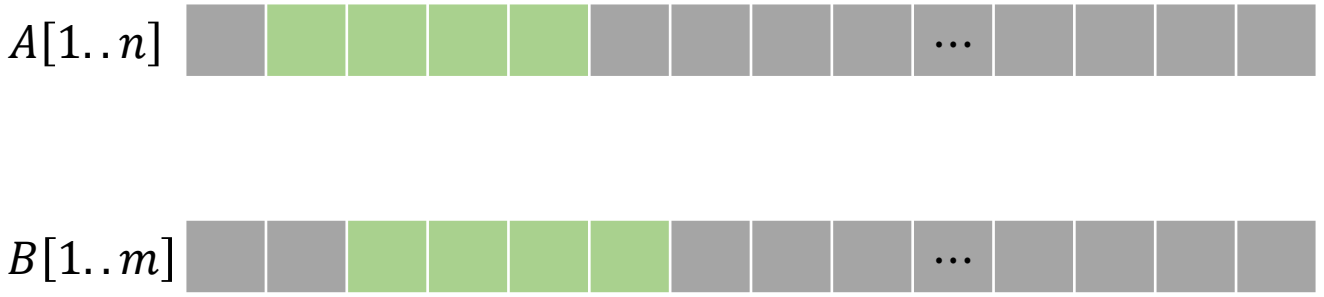
Query based on the prefix tables

- LCP query based on the prefix tables



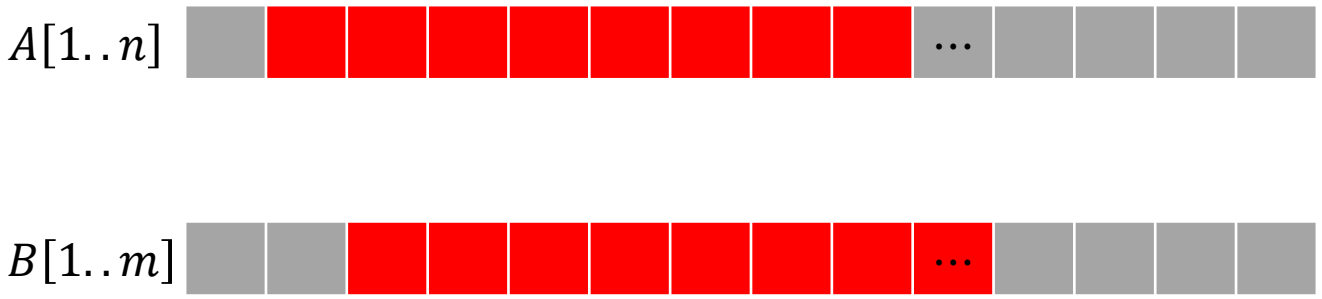
Query based on the prefix tables

- LCP query based on the prefix tables



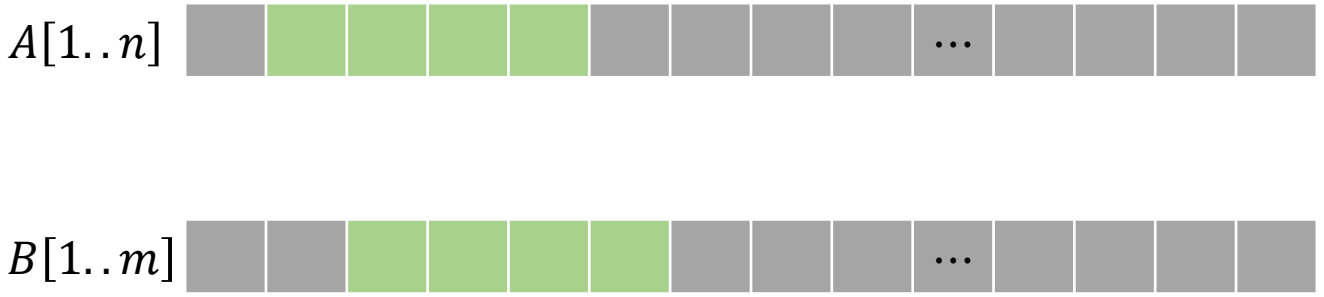
Query based on the prefix tables

- LCP query based on the prefix tables



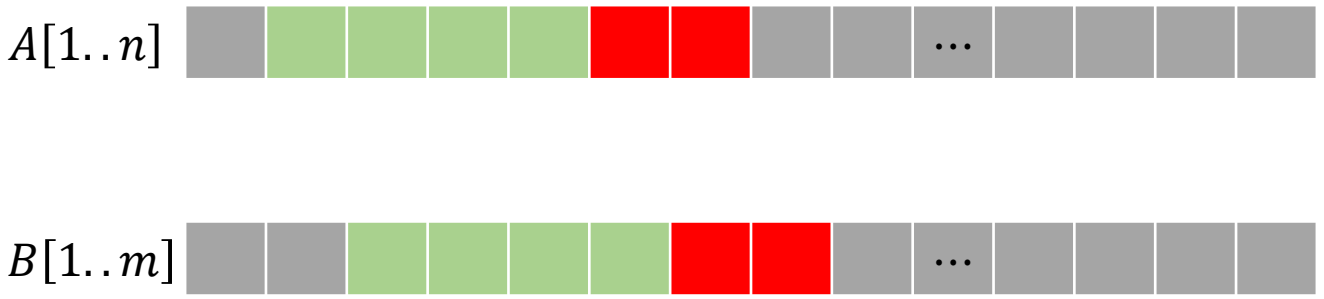
Query based on the prefix tables

- LCP query based on the prefix tables



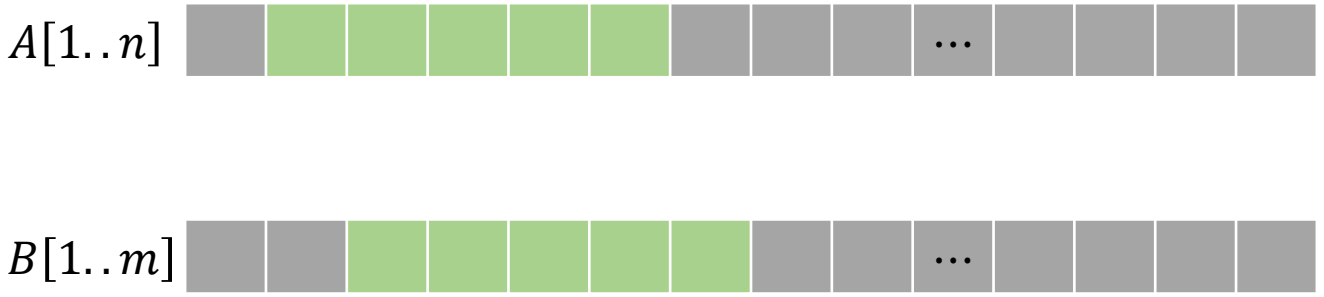
Query based on the prefix tables

- LCP query based on the prefix tables



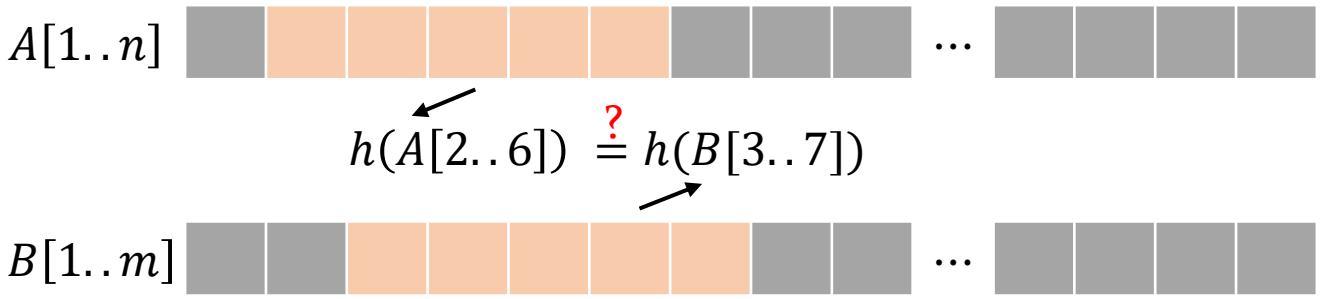
Query based on the prefix tables

- LCP query based on the prefix tables



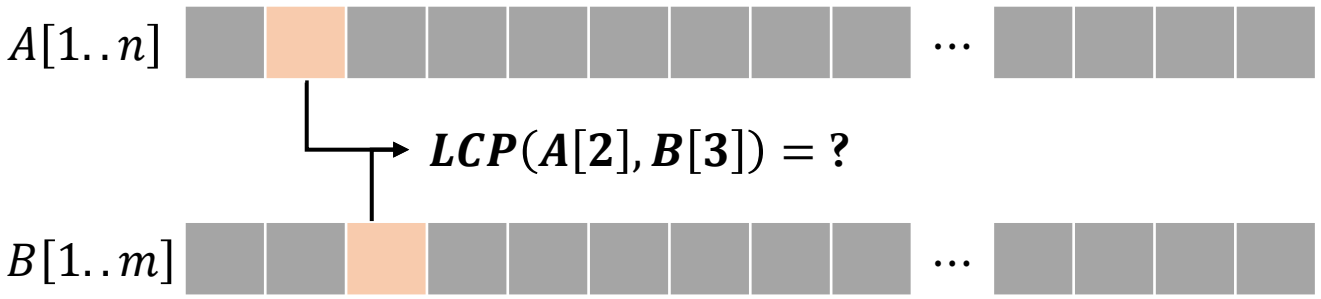
Query based on the prefix tables

- LCP query based on the prefix tables



Query based on the prefix tables

- LCP query based on the prefix tables



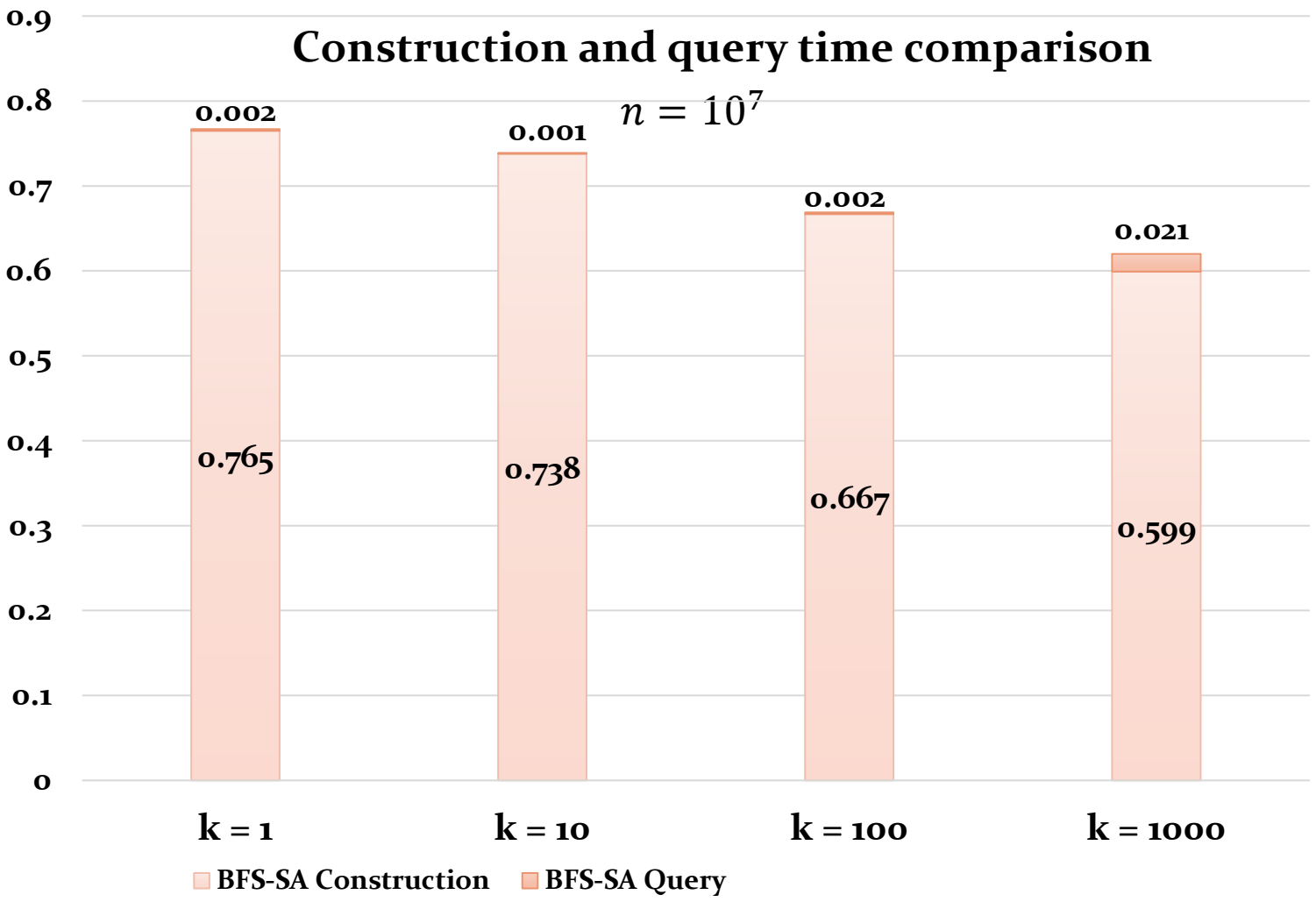
BFS-Hash

Work for construction: $O(n)$

Work for each query: $O(\log n)$

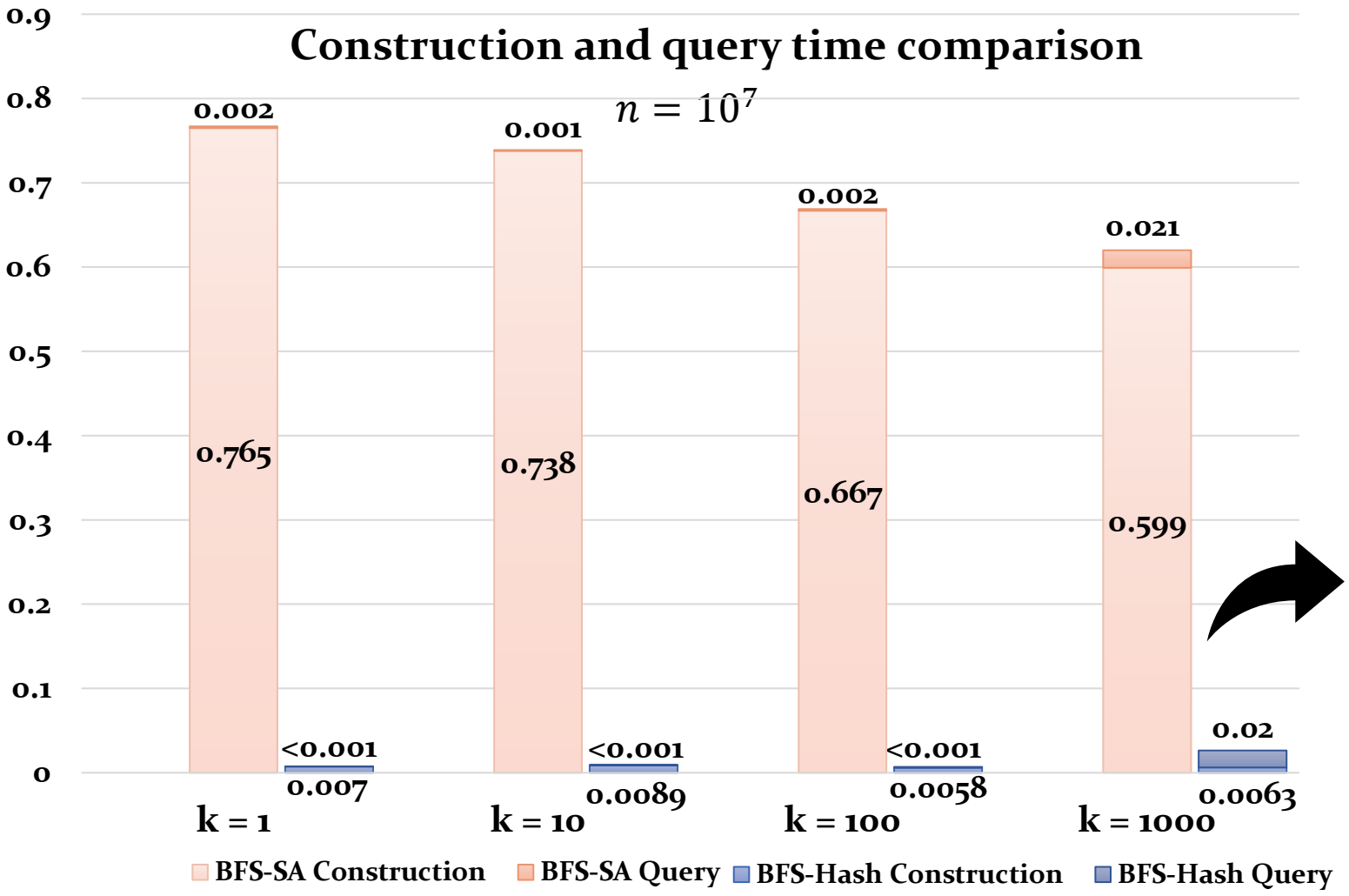
Total work: $O(n + k^2 \log n)$

Did BFS-Hash actually improve the performance?



Lower is better

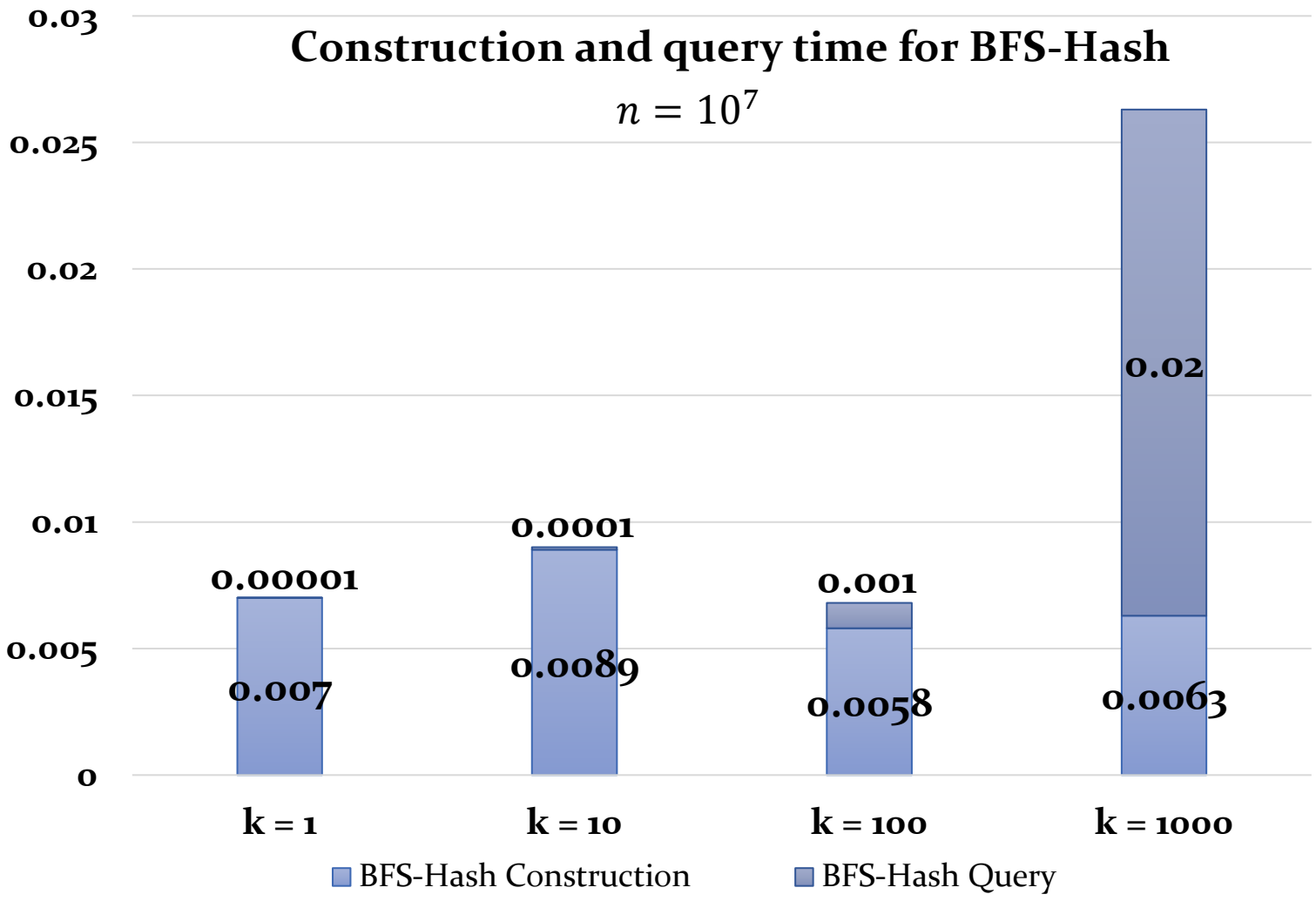
BFS-Hash is significantly better than BFS-SA



Lower is better

BFS-Hash is 24 × better than **BFS-SA**

BFS-Hash is significantly better than BFS-SA



What about the space of prefix tables?

We use the **same** size for hash coding preprocessing $\longrightarrow$ $O(n)$

Do we have ideas to **reduce the space usage**?

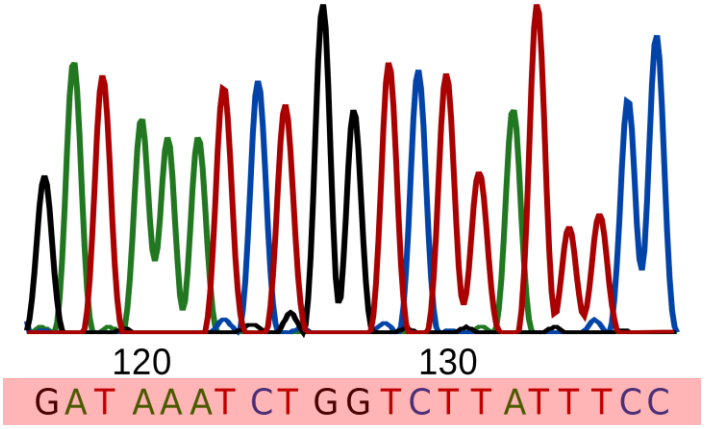
What about the space of prefix tables?

$A[1..n]$

Prefix table for $A[1..n]$

$B[1..m]$

Prefix table for $B[1..m]$



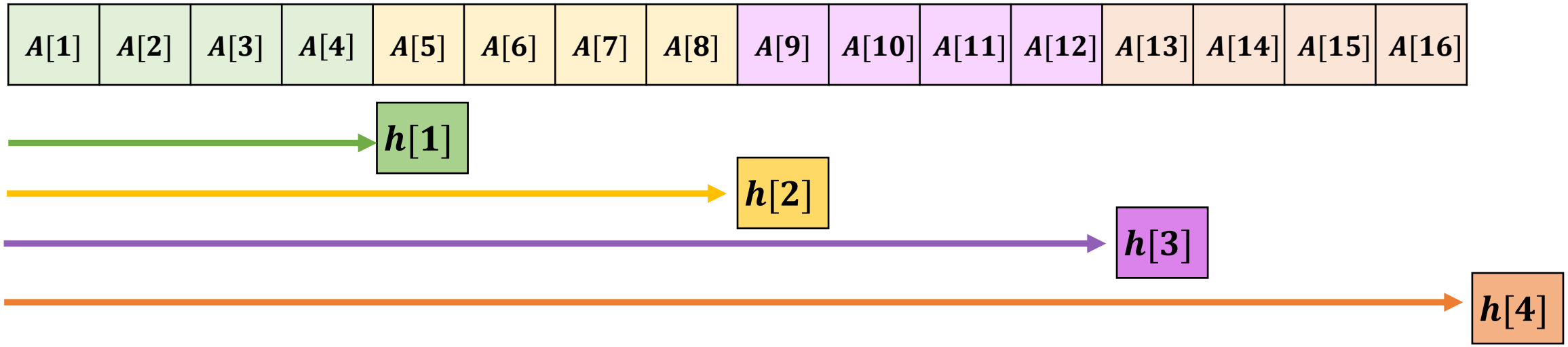
BFS-B-Hash: block-version prefix table

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$	$A[12]$	$A[13]$	$A[14]$	$A[15]$	$A[16]$

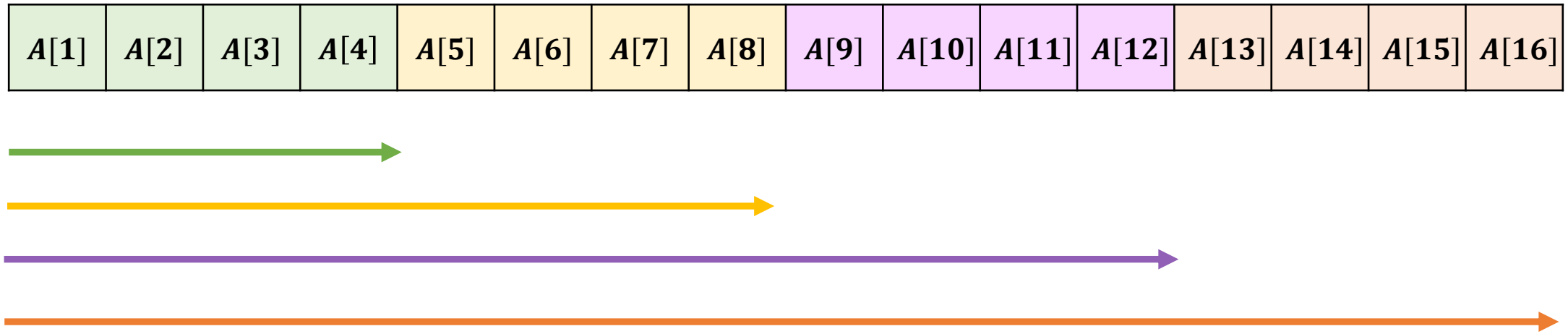
BFS-B-Hash: block-version prefix table

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$	$A[12]$	$A[13]$	$A[14]$	$A[15]$	$A[16]$

BFS-B-Hash: block-version prefix table



BFS-B-Hash: block-version prefix table



The B-Hash Prefix Table

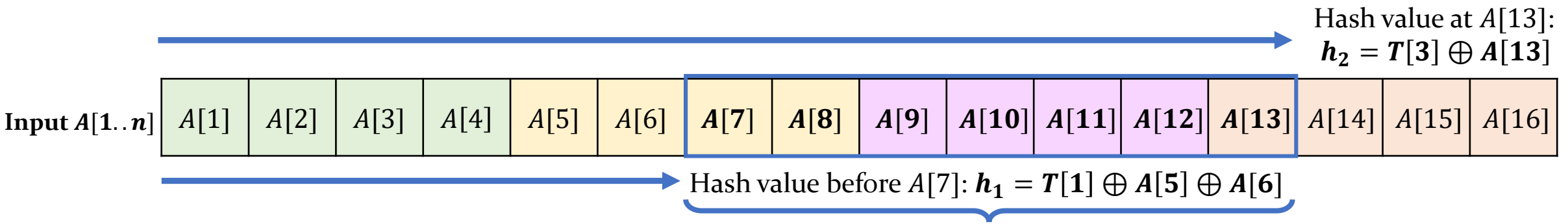
$h[1]$	$h[2]$	$h[3]$	$h[4]$

Block size: 4

Sequence size: 16

B-Hash Prefix table size: 16 / 4 = **4**

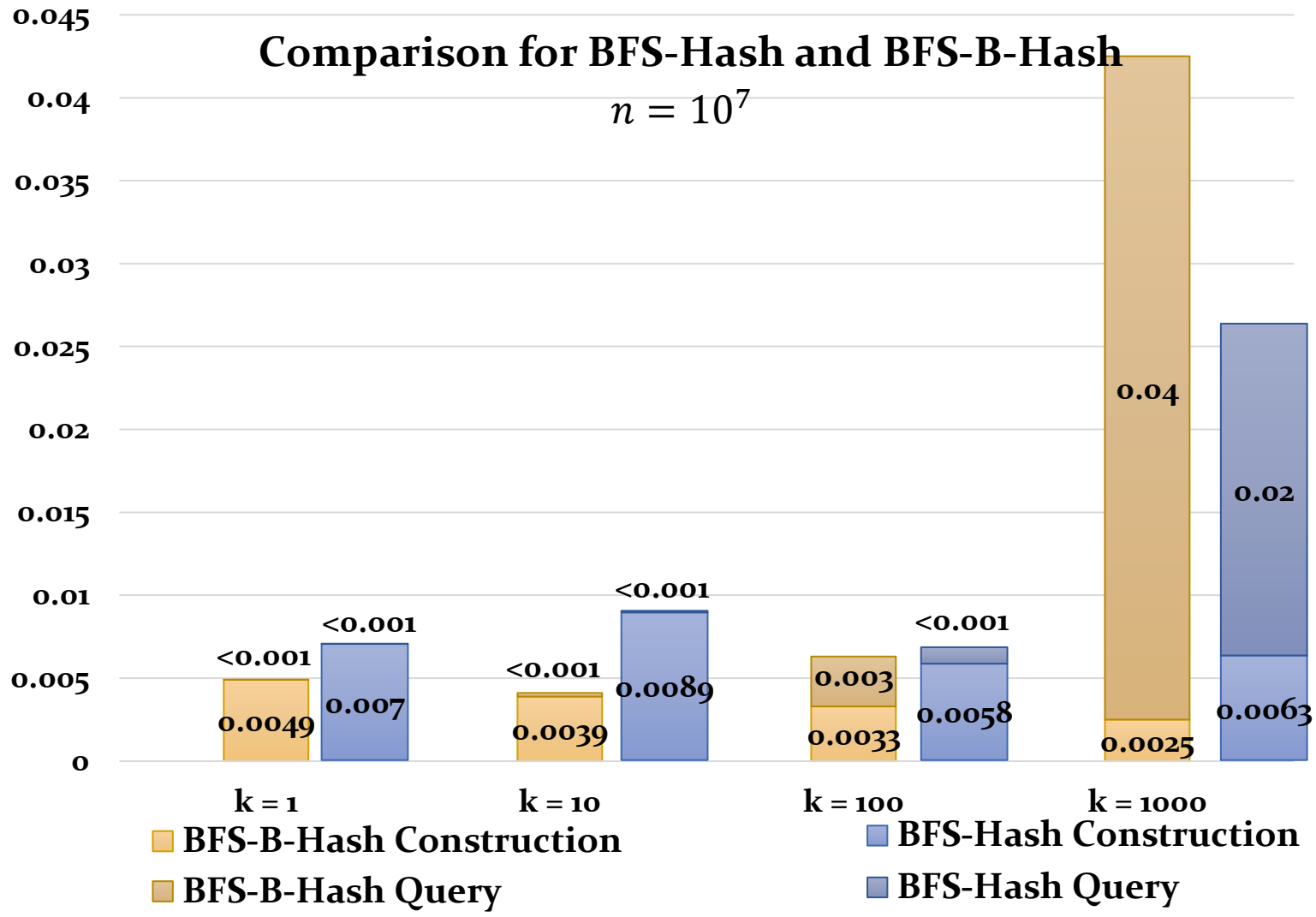
BFS-B-Hash: Query



E.g., Finding hash value of $A[7..13]$

Work for one query: $O(b \log n)$

BFS-B-Hash: block-version prefix table



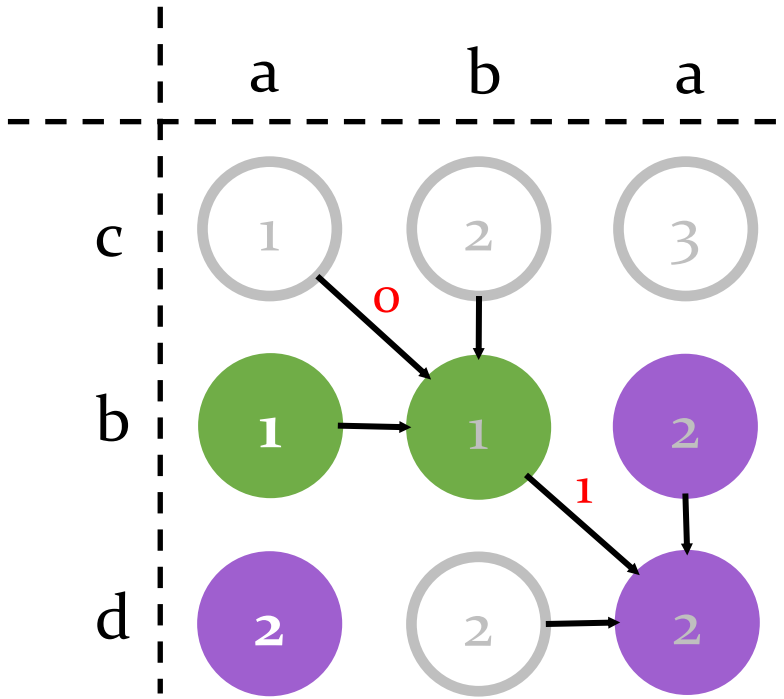
Lower is better

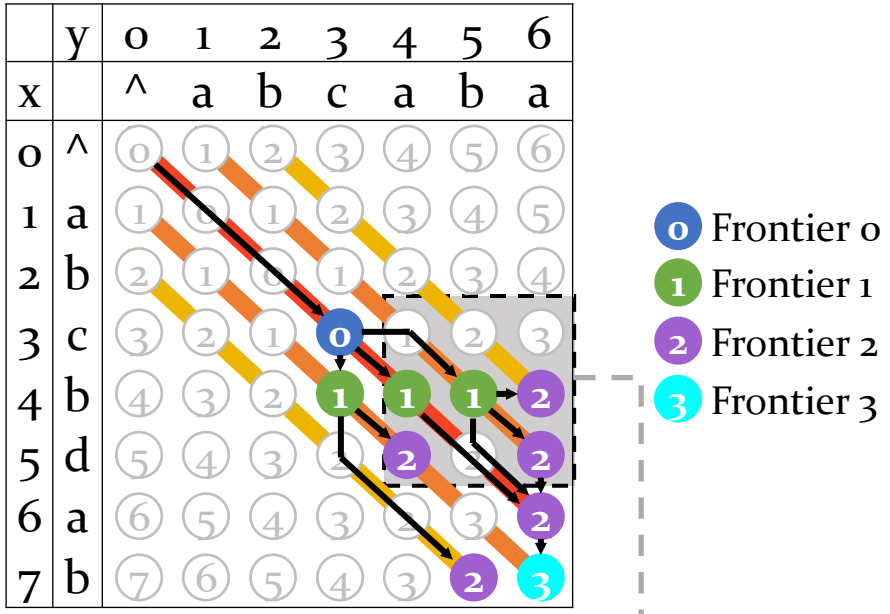
Our contribution in theory

Parallel Divide-and-conquer-based algorithms

Revisit the DP matrix

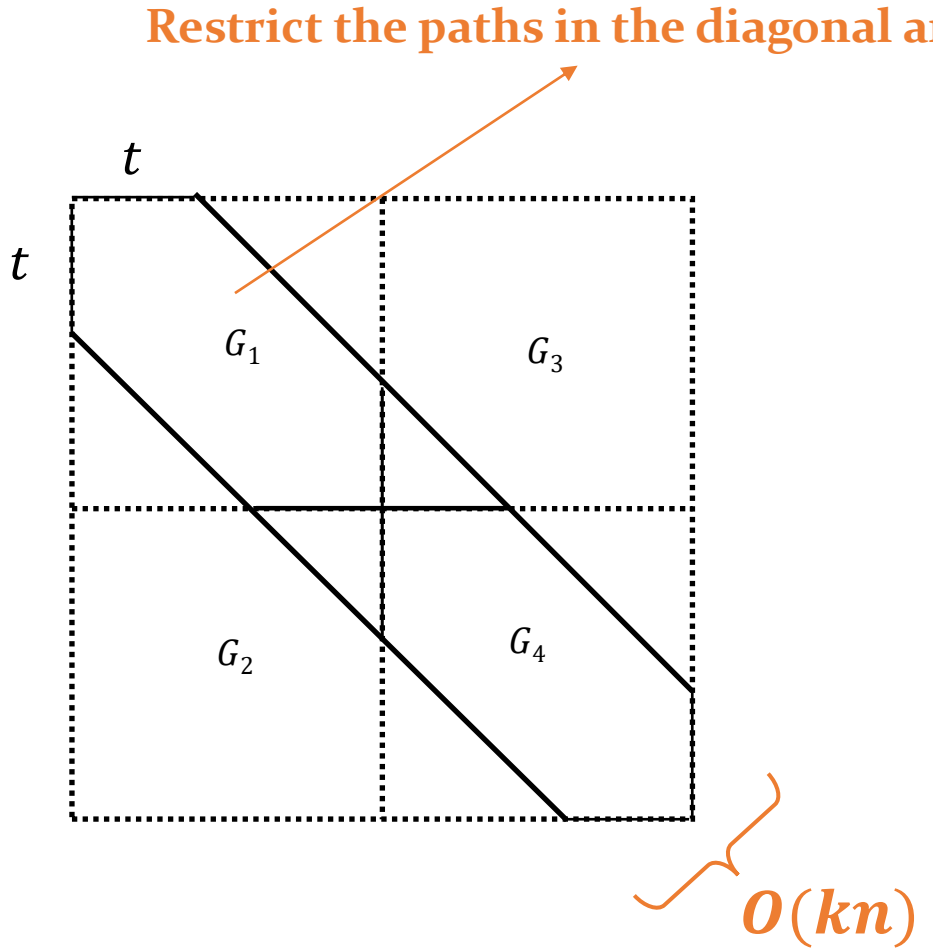
- DP matrix -> Directed acyclic graph (DAG)
- The solution of edit distance is the shortest distance from (0, 0) to (n, m)





Our exploration of the DaC-based approach

- The AALM algorithm[1]
  - The **monotonicity** of the paths
  - Theoretically, the AALM can achieve
    - Work: $O(n^2 \log n)$
    - Span: $O(\log^3 n)$
- Our algorithm **DAC-SD**
  - **Work**: $O(nk \log k)$
  - **Span**: $O(\log n \log^3 k)$



best so far under output-sensitive settings

Our exploration of the DaC-based approach

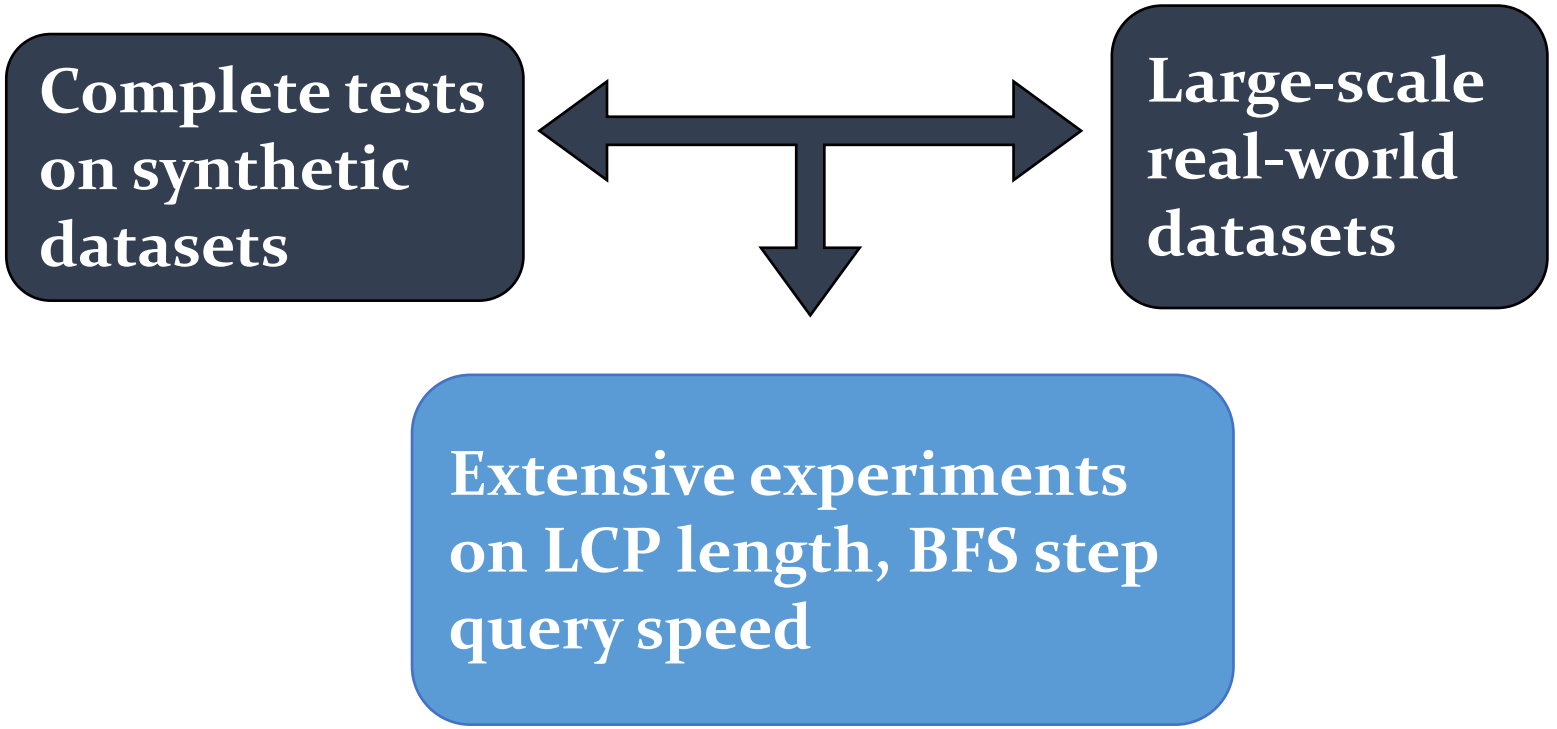
- Our algorithm **DAC-SD**
  - **Work**: $O(nk \log k)$
  - **Span**: $O(\log n \log^3 k)$

- Carefully implemented **DAC-SD** with careful engineering details
  - Relatively worse performance than BFS-based algorithms in practice

Overview of the experiments

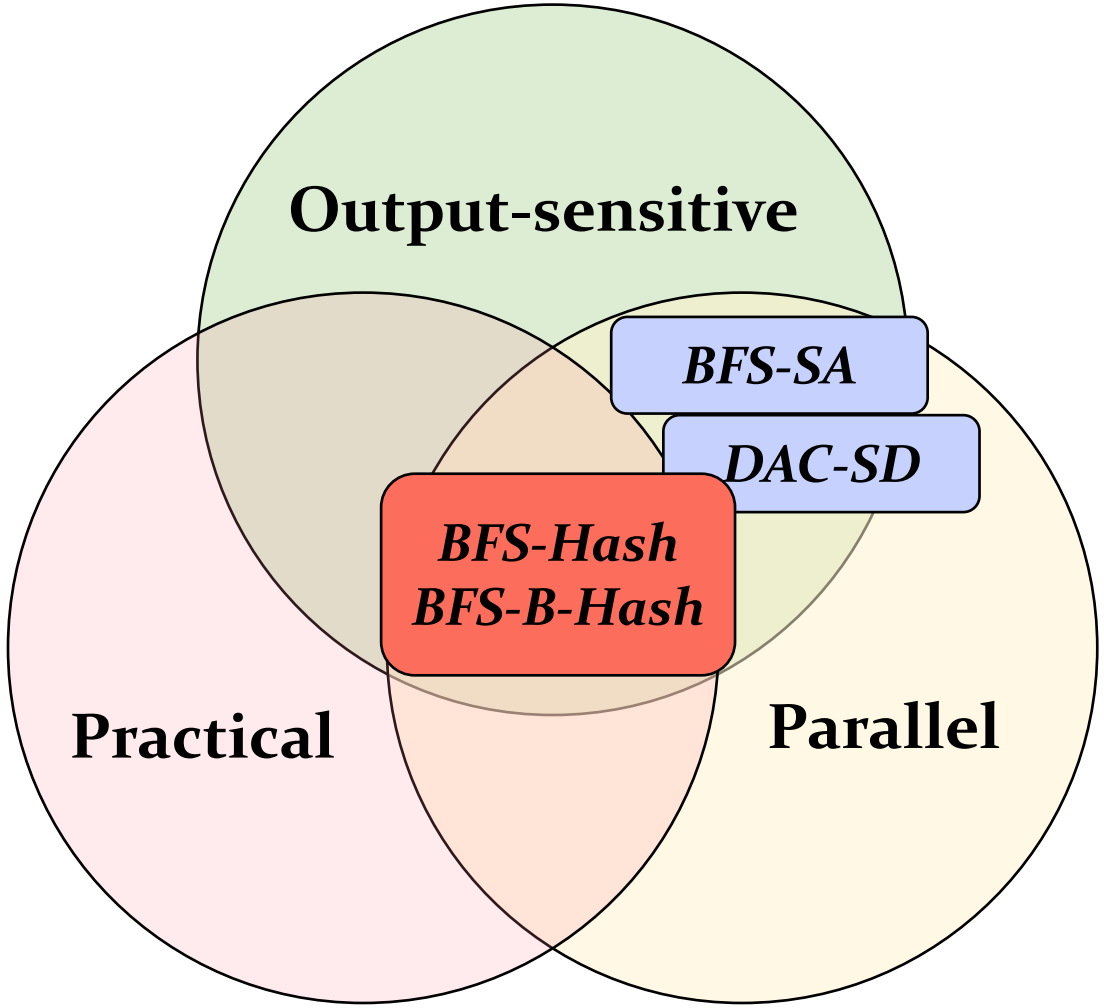
Experiments setup

- ParlayLib[1] for **fork-join parallelism** and **primitives**
- **96-core** (**192 hyperthreads**) machine with four Intel Xeon Gold 6252 CPUs



Conclusions

- Summary



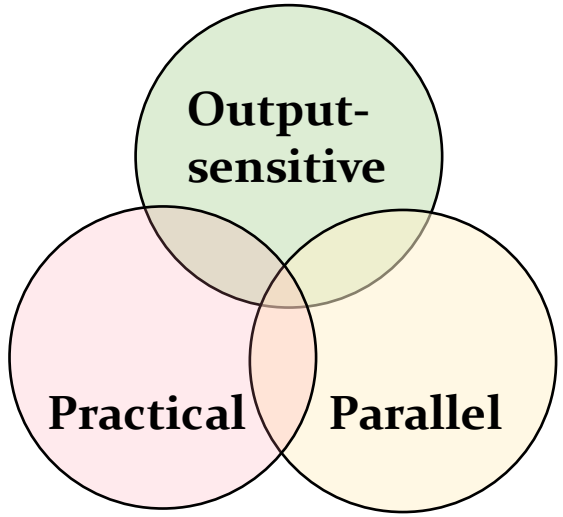
Conclusions

	Work	Span	Space
BFS-SA	$O(n + k^2)$	$\tilde{O}(k)$	$O(n)$
DAC-SD	$O(nk \log k)$	$\tilde{O}(1)$	$O(nk)$
BFS-Hash	$O(n + k^2 \log n)$	$\tilde{O}(k)$	$O(n)$
BFS-B-Hash	$O(n + k^2 b \log n)$	$\tilde{O}\left(\frac{n}{b} + k\right)$	$O(n/b + k)$

Our efficient implementations

Our theoretical improvement: Improved theoretical bound

Our improved algorithms: Best performance in practice



- Code on GitHub:
  - https://github.com/ucrparlay/Edit-Distance

- Contact
  - Youzhe Liu (*yliu908@ucr.edu*)