Height Bounded Lempel-Ziv Encodings

Hideo Bannai (TMDU),
Mitsuru Funakoshi (NTT),
Diptarama Hendrian (TMDU) ,
Myuji Matsuda (TMDU),
Simon J. Puglisi (U Helsinki)

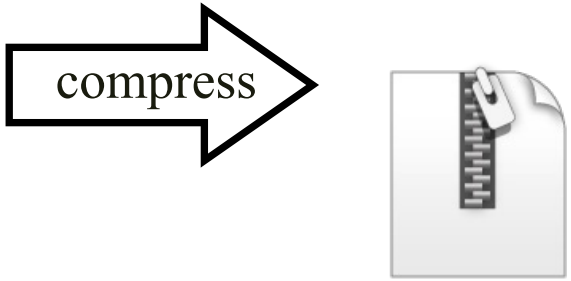
Code: https://github.com/dscalgo/lzhb

Processing Compressed Strings

Processing Compressed Strings



data

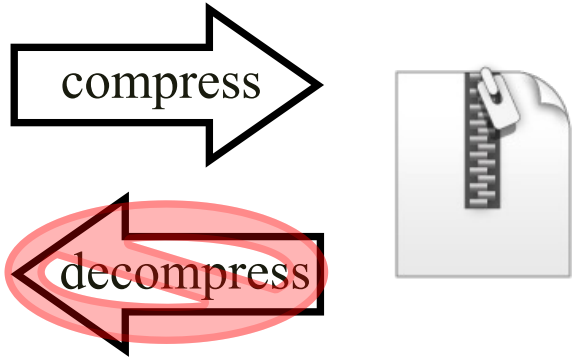


Processing Compressed Strings

BIG

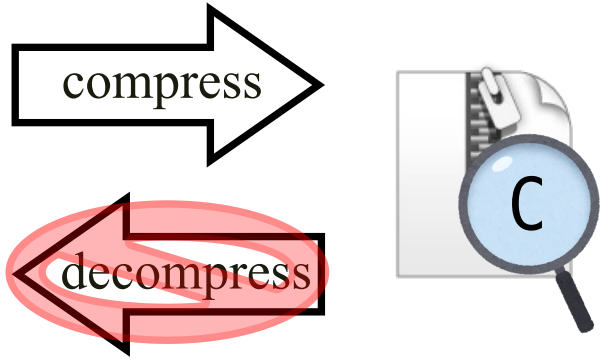


data



Processing Compressed Strings

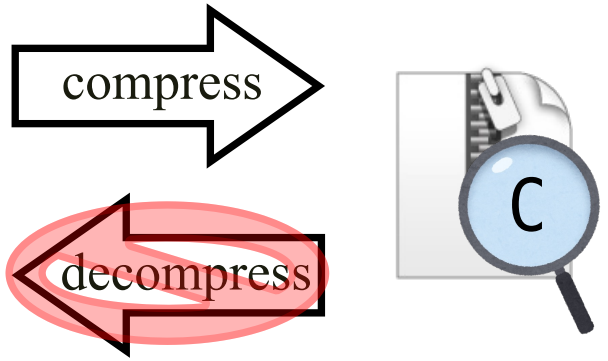




- Radom *access* queries
  - Given compressed representation of string T, answer T[i] for any position i

Processing Compressed Strings

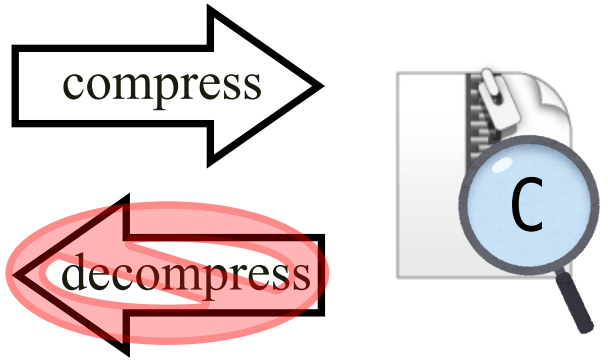




- Radom *access* queries
  - Given compressed representation of string T, answer T[i] for any position i
- Long-standing open problem:
  - Is there a data structure of size O(z) that allows access in O(polylog n) time?
    z: size of LZ77 compressed representation
    Current best worst case size achieving O(polylog n) time is ≒ O(z log n))

Processing Compressed Strings





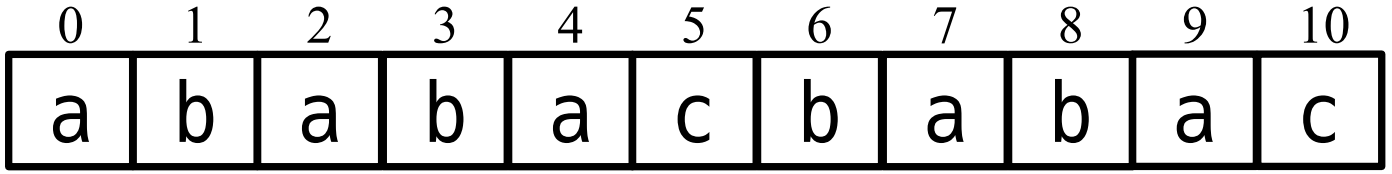
- Radom *access* queries
  - Given compressed representation of string T, answer T[i] for any position i
- Long-standing open problem:
  - Is there a data structure of size O(z) that allows access in O(polylog n) time?
    z: size of LZ77 compressed representation
    Current best worst case size achieving O(polylog n) time is $\fallingdotseq$ O(z log n))
- We propose LZ77 variants that allows tradeoff between access time and space

LZ77 [Ziv&Lempel 1977]

LZ-like encoding

Partitioning of string into ***phrases*** of:

- single symbol ➔ $(1, c)$
- substring having previous occurrence ➔ (len, src)

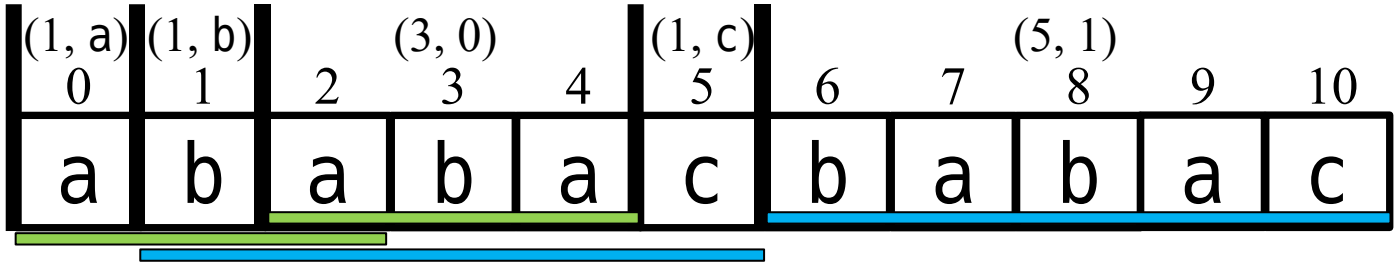


LZ77 [Ziv&Lempel 1977]

LZ-like encoding

Partitioning of string into ***phrases*** of:

- single symbol ➔ $(1, c)$
- substring having previous occurrence ➔ (len, src)

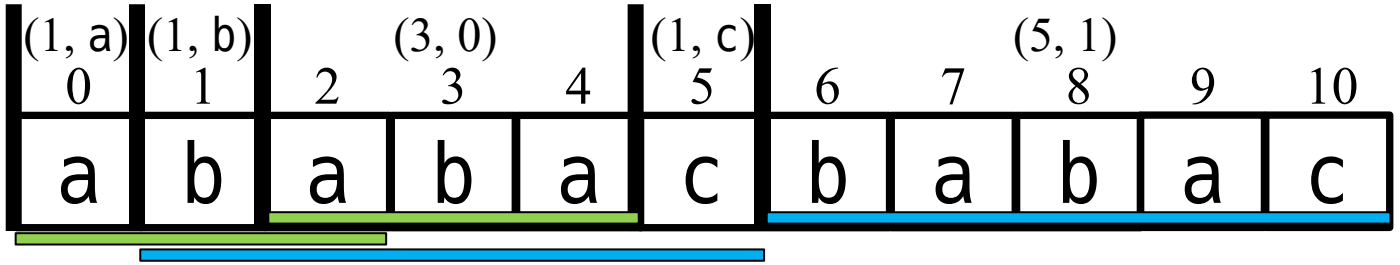


LZ77 [Ziv&Lempel 1977]

LZ-like encoding

Partitioning of string into ***phrases*** of:

- single symbol ➔ $(1, c)$
- substring having previous occurrence ➔ (len, src)



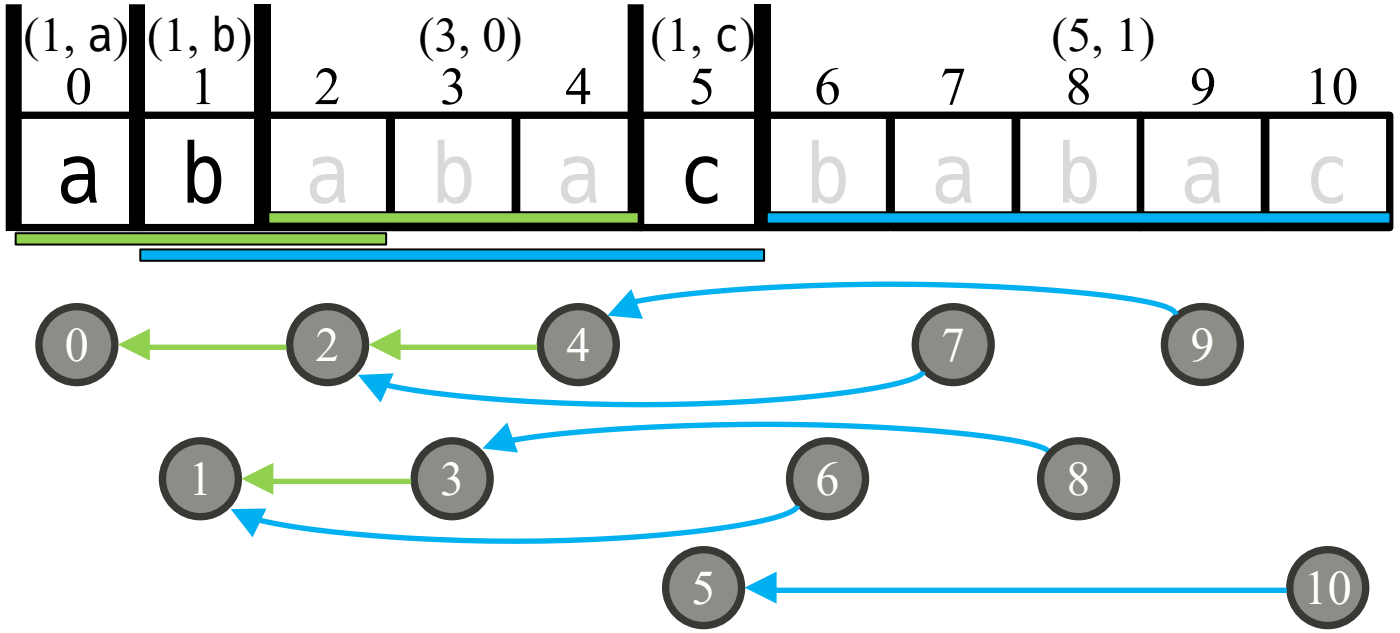
LZ77

Each phrase is computed greedily (longest possible), left-to-right

- # of phrases z is smallest (optimal) among all LZ like encodings
- z is one of the smallest, efficiently computable measures of repetitiveness/compression

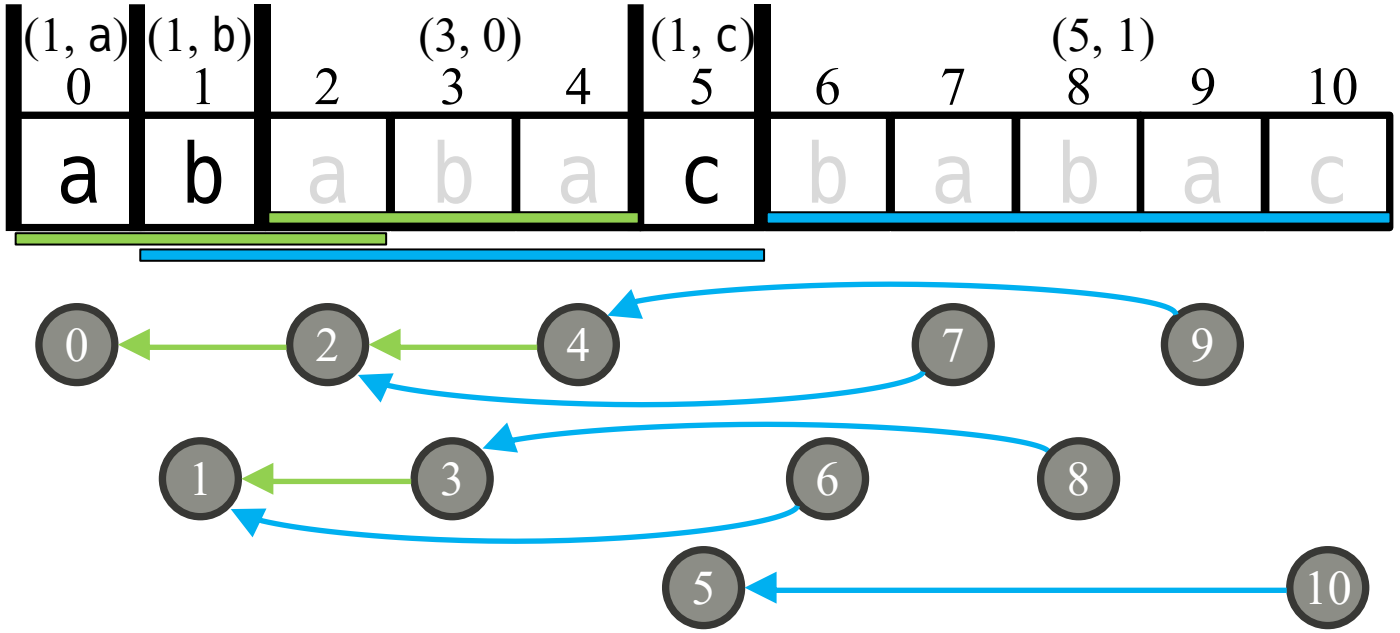
Random Access on LZ-like encoding

- Phrase references induce implicit referencing forest on positions



Random Access on LZ-like encoding

- Phrase references induce implicit referencing forest on positions



- Height of forest = # of steps (predecessor queries) for *access*
- Can be $\Theta(n)$ for LZ77

Height Bounded LZ encoding

- Can we compute LZ-like encoding with height $\leq h$ and having "small" (close to z) # of phrases ?
  h = polylog n gives us O(polylog n) time access

Height Bounded LZ encoding

- Can we compute LZ-like encoding with height $\leq h$ and having "small" (close to z) # of phrases ?
  h = polylog n gives us O(polylog n) time access
- Minimizing # of phrases for given h is NP-hard & APX-hard [Cicalese & Ugazio DLT 2024]

Height Bounded LZ encoding

- Can we compute LZ-like encoding with height $\leq h$ and having "small" (close to z) # of phrases ?
  h = polylog n gives us O(polylog n) time access
- Minimizing # of phrases for given h is NP-hard & APX-hard [Cicalese & Ugazio DLT 2024]
- We propose efficient algorithms for computing *greedy left-to-right* height bounded LZ-like encoding (LZHB), where each phrase is longest possible (i.e., has prev. occ. and satisfies height constraint h)

Height Bounded LZ encoding

- Can we compute LZ-like encoding with height $\leq h$ and having "small" (close to z) # of phrases ?
  h = polylog n gives us O(polylog n) time access
- Minimizing # of phrases for given h is NP-hard & APX-hard [Cicalese & Ugazio DLT 2024]
- We propose efficient algorithms for computing *greedy left-to-right* height bounded LZ-like encoding (LZHB), where each phrase is longest possible (i.e., has prev. occ. and satisfies height constraint h)
  - Related work:
    - LZ-COST [Kreft & Navarro DCC 2010] no efficient algorithm
    - BAT-LZ [Lipták et al. CPM 2024] independent/simultaneous

Our Results

1. First linear time* algorithm for *greedy* LZHB (* assuming constant sized alphabet)
   - $O(n \log \sigma)$ time based on suffix trees
   - $O(n \log n)$ time light-weight implementation based on suffix arrays

   (cf. BAT-LZ [Lipták et al. 2024] : $O(n \log^3 n)$)

Our Results

1. First linear time* algorithm for *greedy* LZHB (* assuming constant sized alphabet)
   - $O(n \log \sigma)$ time based on suffix trees
   - $O(n \log n)$ time light-weight implementation based on suffix arrays

   (cf. BAT-LZ [Lipták et al. 2024] : $O(n \log^3 n)$)
2. Algorithms for *greedier* – choose occ. of phrase with smallest max height
   - $O(n \log \sigma + \#occ)$ time (#*occ*: total # of prev. occ. of all phrases = $O(z'n)$)
   - $O(n \log n + \#occ \log n)$ time

   (cf. BAT-LZ [Lipták et al. 2024] : $O(z'n^2 \log n)$)

Our Results

1. First linear time* algorithm for *greedy* LZHB (* assuming constant sized alphabet)
   - $O(n \log \sigma)$ time based on suffix trees
   - $O(n \log n)$ time light-weight implementation based on suffix arrays

   (cf. BAT-LZ [Lipták et al. 2024] : $O(n \log^3 n)$)
2. Algorithms for *greedier* – choose occ. of phrase with smallest max height
   - $O(n \log \sigma + \#occ)$ time (#*occ*: total # of prev. occ. of all phrases = $O(z'n)$)
   - $O(n \log n + \#occ \log n)$ time

   (cf. BAT-LZ [Lipták et al. 2024] : $O(z'n^2 \log n)$)
3. New LZ-like encoding variant exploiting *periodicity* to further reduce height.

Our Results

1. First linear time* algorithm for *greedy* LZHB (* assuming constant sized alphabet)
   - $O(n \log \sigma)$ time based on suffix trees
   - $O(n \log n)$ time light-weight implementation based on suffix arrays

   (cf. BAT-LZ [Lipták et al. 2024] : $O(n \log^3 n)$)
2. Algorithms for *greedier* – choose occ. of phrase with smallest max height
   - $O(n \log \sigma + \#occ)$ time (#*occ*: total # of prev. occ. of all phrases = $O(z'n)$)
   - $O(n \log n + \#occ \log n)$ time

   (cf. BAT-LZ [Lipták et al. 2024] : $O(z'n^2 \log n)$)
3. New LZ-like encoding variant exploiting *periodicity* to further reduce height.
4. Showed that $\exists c$ s.t. smallest $(c \log n)$-bounded LZ-like encoding is always at most $O(g_{rl})$ and can be $o(g_{rl})$ for some string families.
   (g_{rl}: size of smallest run-length grammar)

$z \preccurlyeq (c \log n)\text{-OPTLZHB} \prec g_{rl}$

Our Results

1. First linear time* algorithm for *greedy* LZHB (* assuming constant sized alphabet)
   - $O(n \log \sigma)$ time (σ: alphabet size) based on suffix trees
   - $O(n \log n)$ time light-weight implementation based on suffix arrays

   (cf. BAT-LZ [Lipták et al. 2024] : $O(n \log^3 n)$)

2. Algorithms for *greedier* – choose occ. of phrase with smallest max height
   - $O(n \log \sigma + \#occ)$ time (#*occ*: total # of prev. occ. of all phrases = $O(z'n)$)
   - $O(n \log n + \#occ \log n)$ time

   (cf. BAT-LZ [Lipták et al. 2024] : $O(z'n^2 \log n)$)

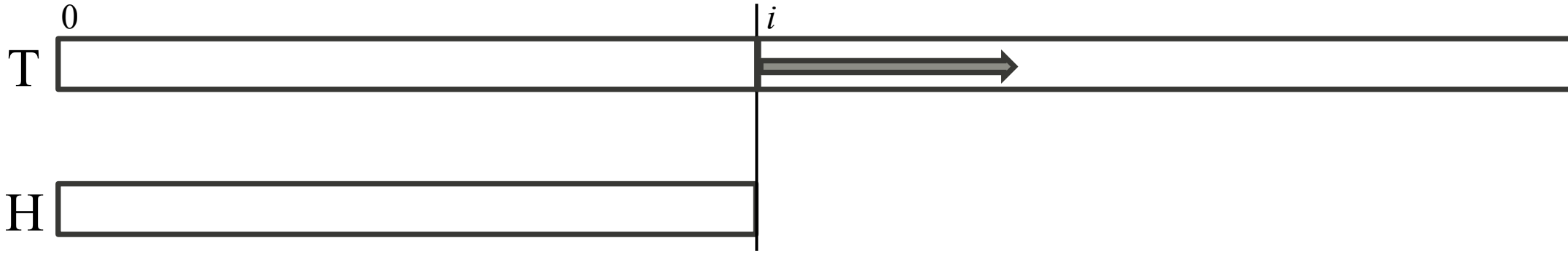
3. New LZ-like encoding variant exploiting *periodicity* to further reduce height.

4. Showed that $\exists c$ s.t. smallest ($c \log n$)-bounded LZ-like encoding is always at most $O(g_{rl})$ and can be $o(g_{rl})$ for some string families.
   (g_{rl}: size of smallest run-length grammar)

$z \preccurlyeq (c \log n)\text{-OPTLZHB} \prec g_{rl}$

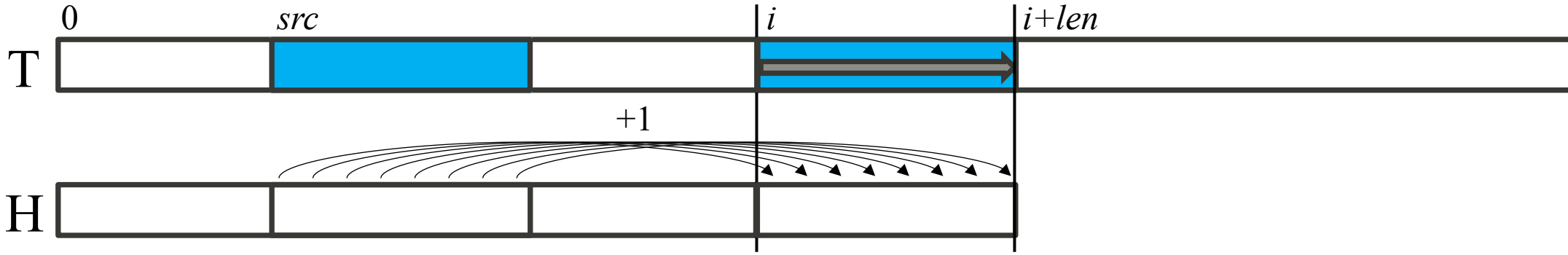
Computing the next phrase

- Given encoding for T[0..*i*), and heights H[0..*i*), compute: next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)



Computing the next phrase

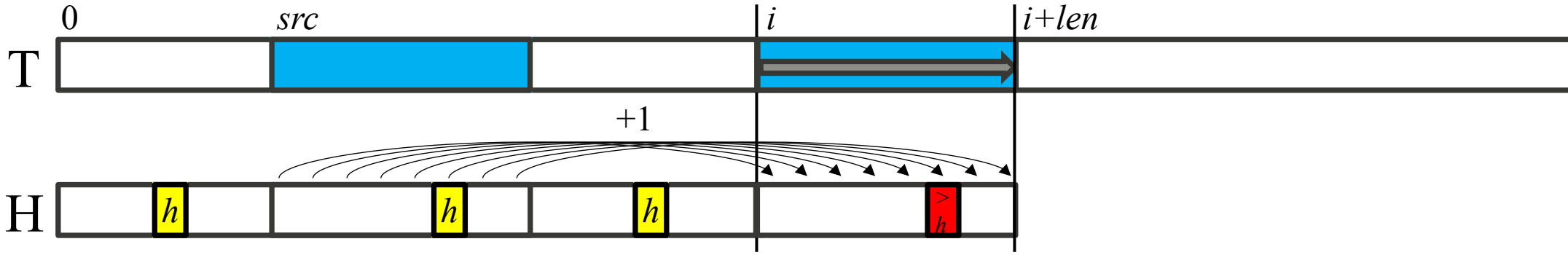
- Given encoding for T[0..*i*), and heights H[0..*i*), compute:
  next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)



- Observation: Referencing adds 1 to height.
  We shouldn't reference positions that have height = *h*.

Computing the next phrase

- Given encoding for T[0..*i*), and heights H[0..*i*), compute:
  next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)

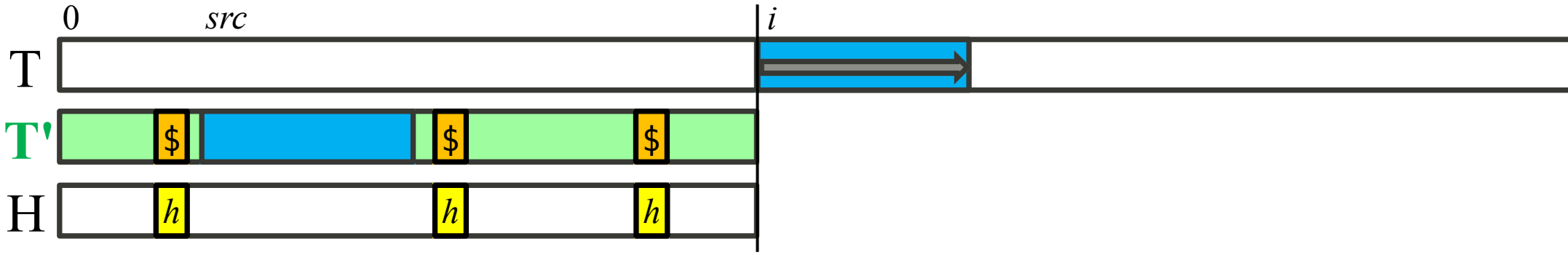


- Observation: Referencing adds 1 to height.
  We shouldn't reference positions that have height = *h*.

How can we avoid referencing these positions?

Computing the next phrase

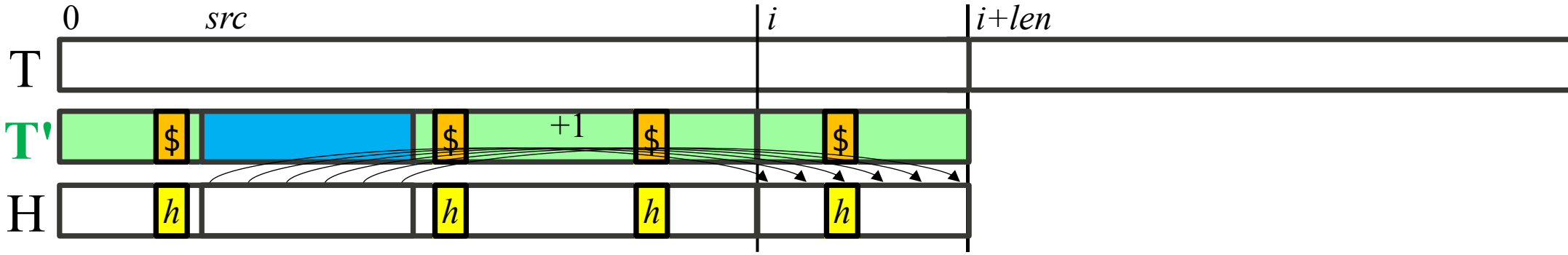
- Given encoding for T[0..*i*), and heights H[0..*i*), compute:
  next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)



- Observation: Referencing adds 1 to height.
  We shouldn't reference positions that have height = *h*.
- Key Idea:
  - **T'** : Replace symbols at positions of T having height = *h* with \$ ($\notin \Sigma$)
  - Find longest prefix of T[*i*..*n*) that occurs in **T'[0..*i*)**

Computing the next phrase

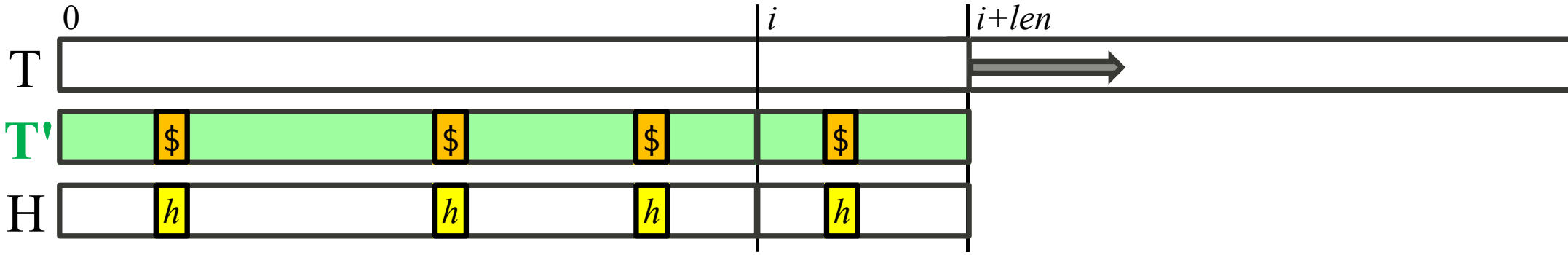
- Given encoding for T[0..*i*), and heights H[0..*i*), compute:
  next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)



- Observation: Referencing adds 1 to height.
  We shouldn't reference positions that have height = *h*.
- Key Idea:
  - **T'** : Replace symbols at positions of T having height = *h* with \$ ($\notin \Sigma$)
  - Find longest prefix of T[*i*..*n*) that occurs in **T'[0..*i*)**

Computing the next phrase

- Given encoding for T[0..*i*), and heights H[0..*i*), compute:
  next phrase T[*i*..*i*+*len*), its *src*, and H[*i*..*i*+*len*)



- Observation: Referencing adds 1 to height.
  We shouldn't reference positions that have height = *h*.
- Key Idea:
  - **T'** : Replace symbols at positions of T having height = *h* with \$ ($\notin \Sigma$)
  - Find longest prefix of T[*i*..*n*) that occurs in **T'[0..*i*)**

Finding longest prefix of T[i..n) occurring in T'[0..i)

- Solution 1: suffix tree ST (of T'[0..i))

- Solution 2: suffix arrays SA

Finding longest prefix of T[$i..n$) occurring in T'[0..i)

- Solution 1: suffix tree ST (of T'[0..i))
  - Can find longest prefix in O($len \cdot \log \sigma$) time, total O($n \log \sigma$) time
  - ST of T' can be built *on-line* in total O($n \log \sigma$) time

  (some technicalities with occurrences/self-referencing phrases, but works out – see paper for details)
- Solution 2: suffix arrays SA

Finding longest prefix of T[$i..n$) occurring in T'[0..i)

- Solution 1: suffix tree ST (of T'[0..i))
  - Can find longest prefix in O($len \cdot \log \sigma$) time, total O($n \log \sigma$) time
  - ST of T' can be built *on-line* in total O($n \log \sigma$) time

  (some technicalities with occurrences/self-referencing phrases, but works out – see paper for details)
- Solution 2: suffix arrays SA
  - SA ≒ memory efficient ST, but not efficient to build on-line.

  ➔ We simulate ST of T' built on-line, using *static* suffix array of T.

Finding longest prefix of T[*i*..*n*) occurring in T'[0..*i*)

- Solution 1: suffix tree ST (of T'[0..*i*))
  - Can find longest prefix in O(*len* · log σ) time, total O(*n* log σ) time
  - ST of T' can be built *on-line* in total O(*n* log σ) time

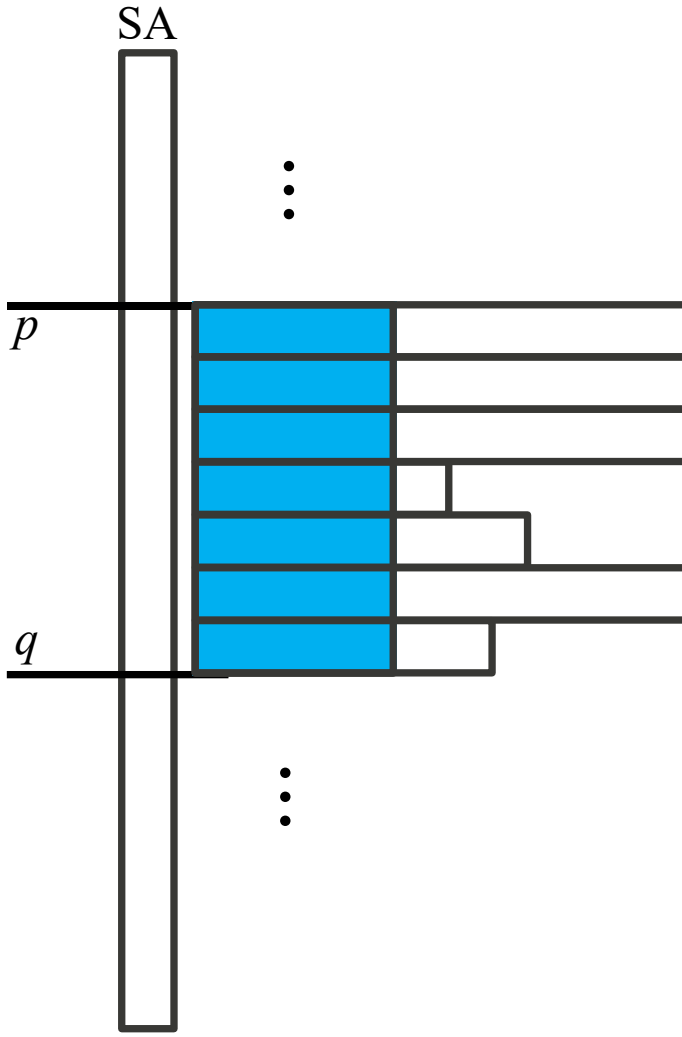
  (some technicalities with occurrences/self-referencing phrases, but works out – see paper for details)
- Solution 2: suffix arrays SA
  - SA ≒ memory efficient ST, but not efficient to build *on-line*.

  ➔ We simulate ST of T' built on-line, using *static* suffix array of T.

BAT-LZ uses dynamic 3D range existence queries,
we use dynamic 1D range maximum queries

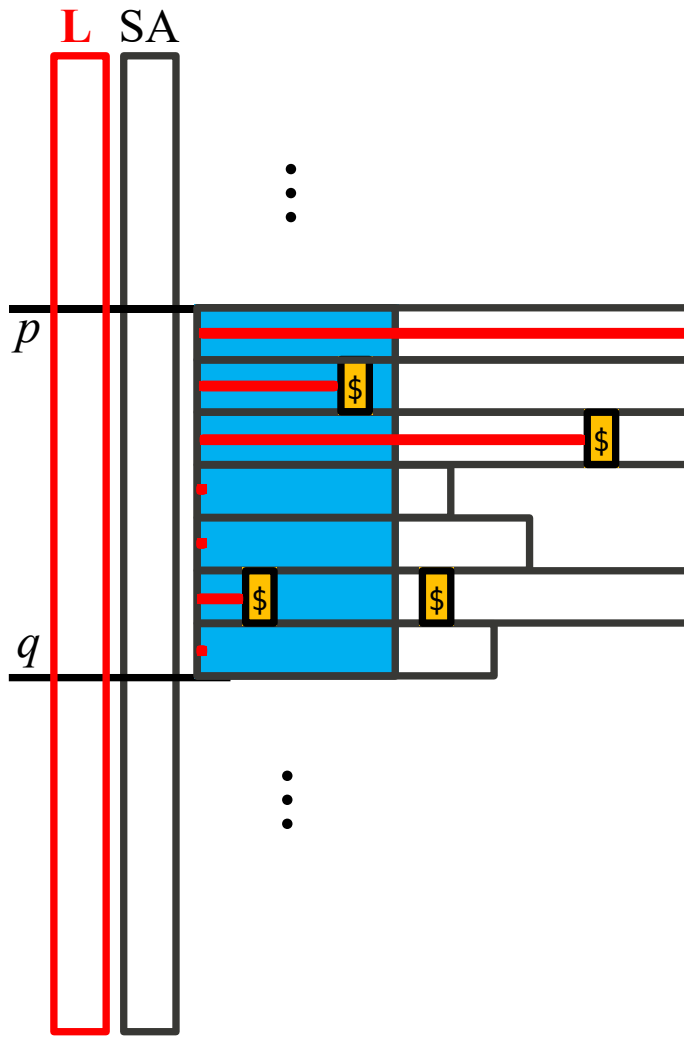
Simulating *on-line ST* of T' using *static SA* of T

- Suffix Array SA of T
  - array of (starting positions of) lex-sorted suffixes
  - occurrences of substring in T
    ⇔ range in SA (can be found in O(log n) time /symbol)



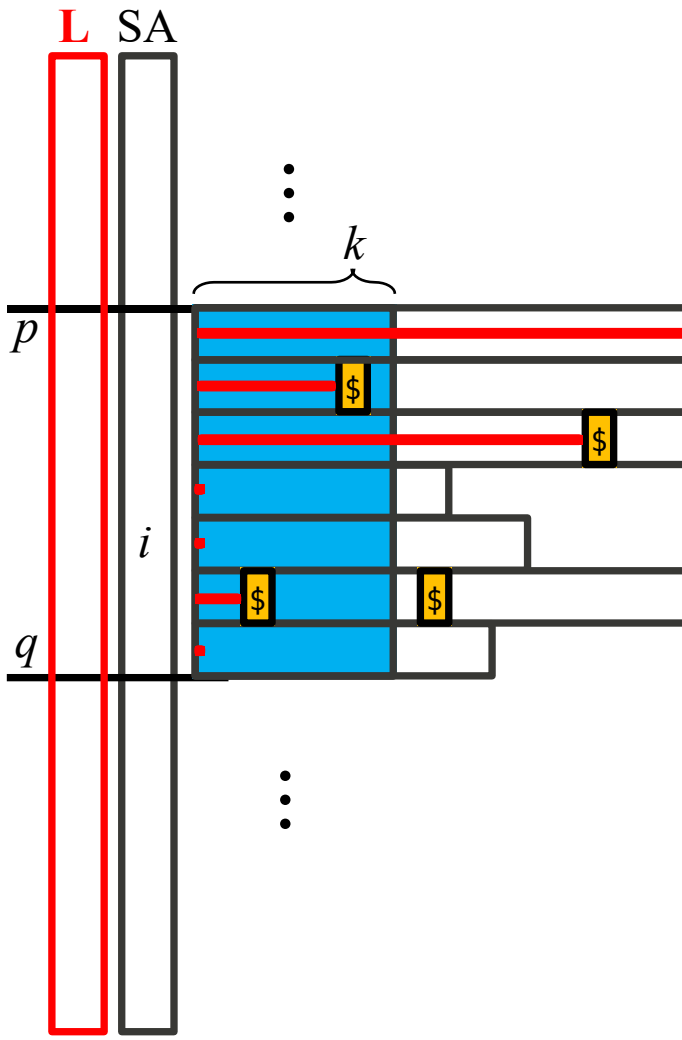
Simulating *on-line ST* of T' using *static SA* of T

- Suffix Array SA of T
  - array of (starting positions of) lex-sorted suffixes
  - occurrences of substring in T
    ⇔ range in SA (can be found in O(log n) time /symbol)
- Maintain dynamic array **L**
  - **L[j]** = valid length of suffix T'[SA[j]..n)
    - initially 0
    - set to n – SA[j] when T[SA[j]..n) added to T'
    - updated after first $ introduced in T'[SA[j]..n)



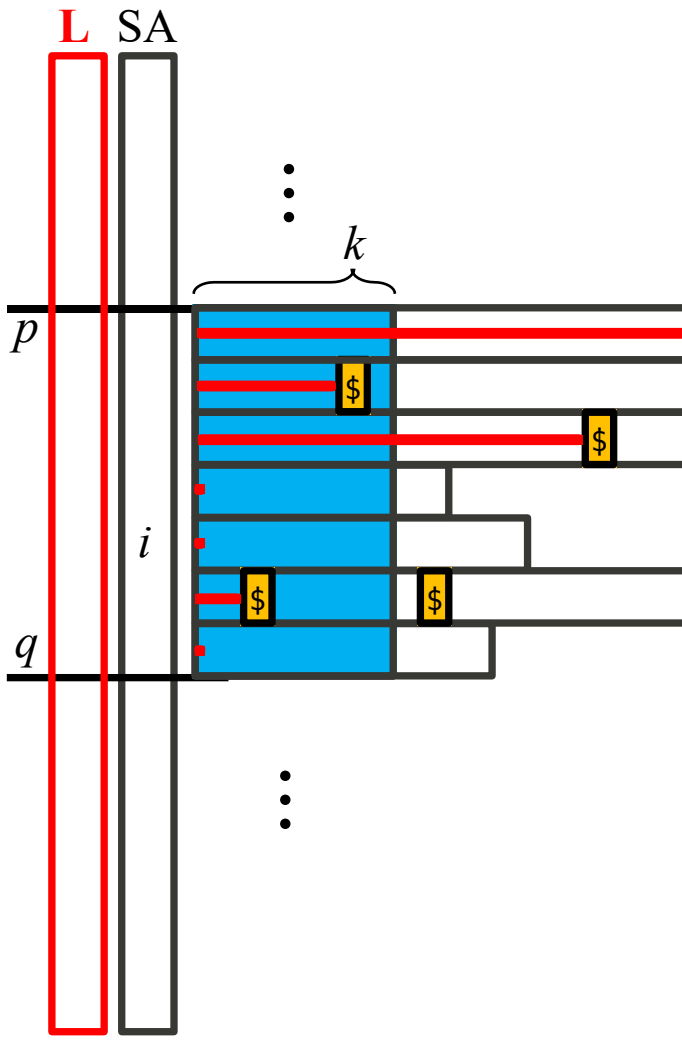
Simulating *on-line ST* of T' using *static SA* of T

- Suffix Array SA of T
  - array of (starting positions of) lex-sorted suffixes
  - occurrences of substring in T
    ⇔ range in SA (can be found in O(log n) time /symbol)
- Maintain dynamic array **L**
  - **L[j]** = valid length of suffix T'[SA[j]..n)
    - initially 0
    - set to n – SA[j] when T[SA[j]..n) added to T'
    - updated after first \$ introduced in T'[SA[j]..n)
- T[i..i+k) with range SA[p..q] contains valid occurrence in T'[0..i) ⇔ max **L[p..q]** ≥ k



Simulating *on-line ST* of T' using *static SA* of T

- Suffix Array SA of T
  - array of (starting positions of) lex-sorted suffixes
  - occurrences of substring in T
    ⇔ range in SA (can be found in O(log n) time /symbol)
- Maintain dynamic array **L**
  - **L[j]** = valid length of suffix T'[SA[j]..n)
    - initially 0
    - set to n – SA[j] when T[SA[j]..n) added to T'
    - updated after first $ introduced in T'[SA[j]..n)
- T[i..i+k) with range SA[p..q] contains valid occurrence in T'[0..i) ⇔ max **L[p..q]** ≥ k



O(n) total updates and dynamic range maximum queries ➔ O(n log n) time

Computational Experiments (throughput)

O(n log σ) time suffix tree solution
O(n log n) time suffix array solution
BAT-LZ

greedy
greedier

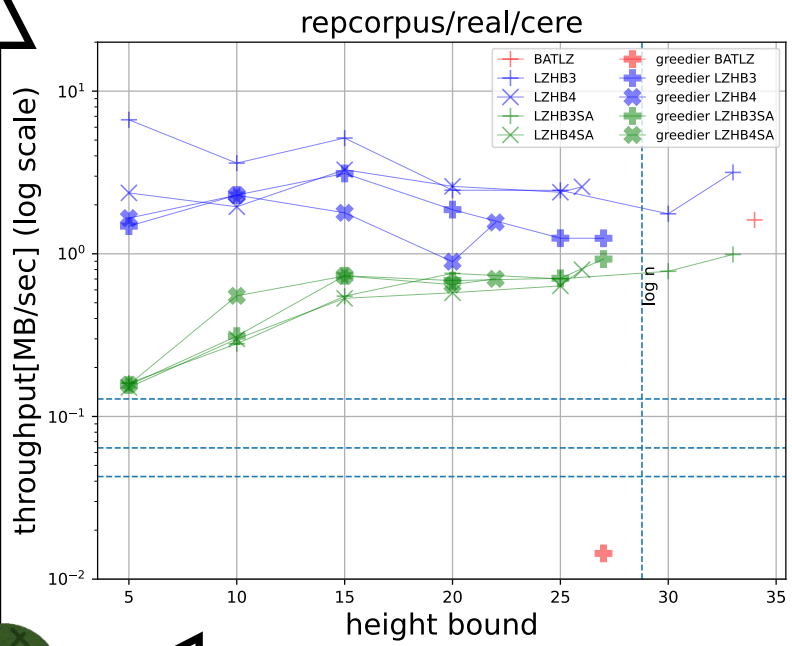
\+ traditional encoding
× new encoding

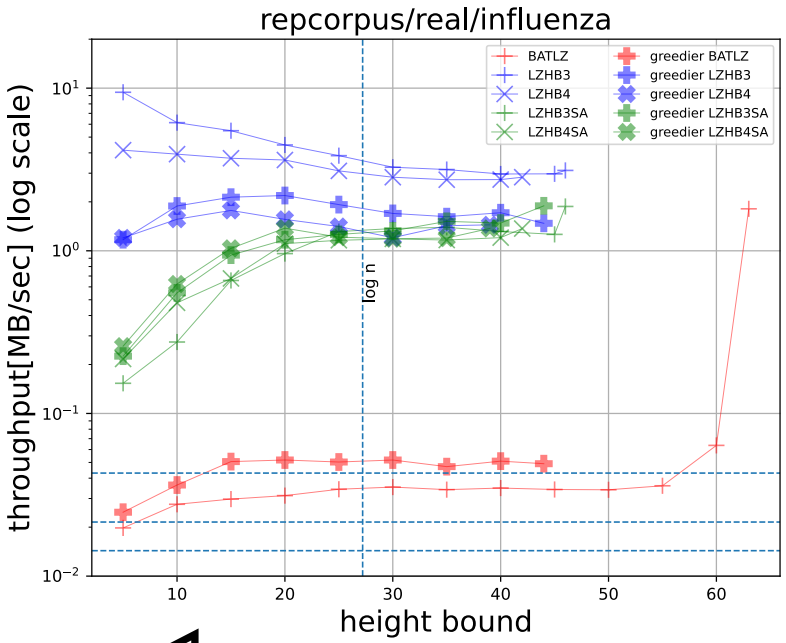


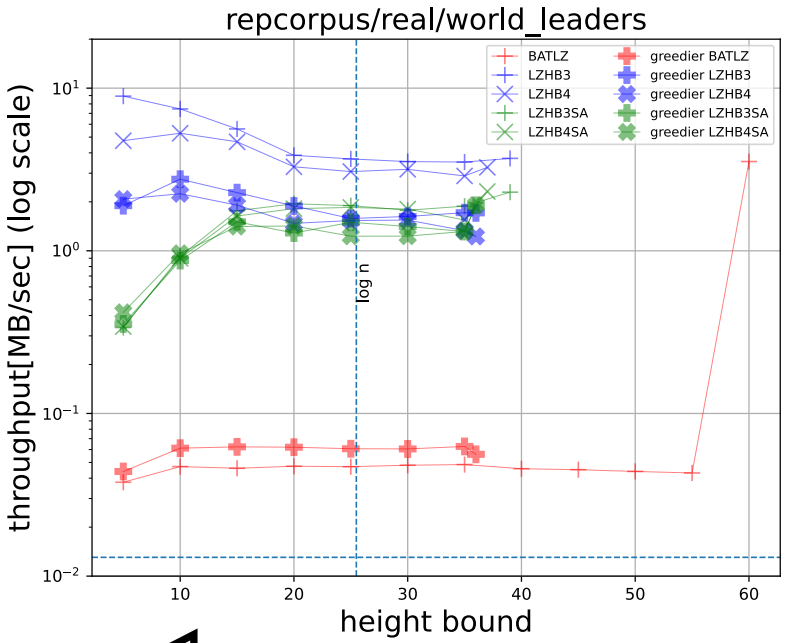
FAST

slow









short TALL

short TALL

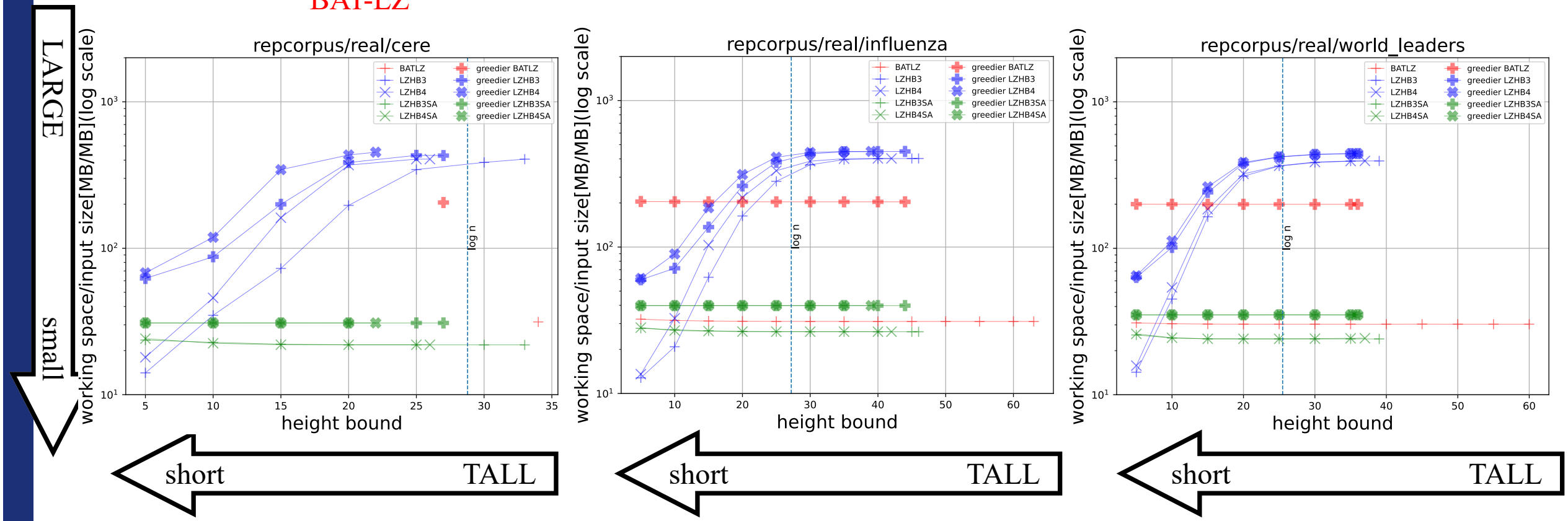
short TALL

Computational Experiments (memory usage)

O(n log σ) time suffix tree solution
O(n log n) time suffix array solution
BAT-LZ

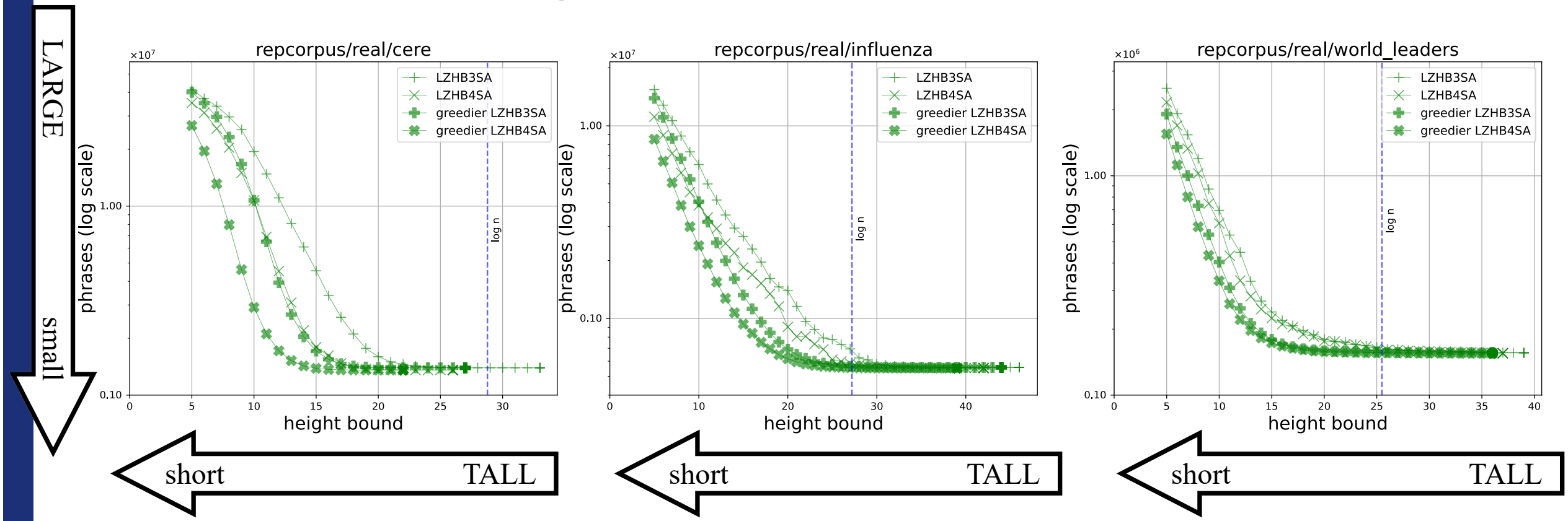
greedy
greedier

\+ traditional encoding
× new encoding



Computational Experiments (# phrases)

greedy + traditional encoding
greedier × new encoding



Conclusion

Summary

- Efficient algorithms LZHB for greedy height bounded LZ-encodings
  - First linear time algorithm (assuming constant sized alphabet)
  - O(n log n) lightweight implementation using suffix arrays
  - Extension to *greedier* heuristic
  - Theoretically and practically faster than BAT-LZ (greedy & greedier)
  - Code: https://github.com/dscalgo/lzhb
- New LZ-encoding variants
- Relations with other repetitiveness measures

Conclusion

Summary

- Efficient algorithms LZHB for greedy height bounded LZ-encodings
  - First linear time algorithm (assuming constant sized alphabet)
  - O(n log n) lightweight implementation using suffix arrays
  - Extension to *greedier* heuristic
  - Theoretically and practically faster than BAT-LZ (greedy & greedier)
  - Code: https://github.com/dscalgo/lzhb
- New LZ-encoding variants
- Relations with other repetitiveness measures

Future work & Open problems

- Engineer random-access data structures based on height-bounded LZ

Conclusion

Summary

- Efficient algorithms LZHB for greedy height bounded LZ-encodings
  - First linear time algorithm (assuming constant sized alphabet)
  - O(n log n) lightweight implementation using suffix arrays
  - Extension to *greedier* heuristic
  - Theoretically and practically faster than BAT-LZ (greedy & greedier)
  - Code: https://github.com/dscalgo/lzhb
- New LZ-encoding variants
- Relations with other repetitiveness measures

Future work & Open problems

- Engineer random-access data structures based on height-bounded LZ
- Can we "balance" LZ77?
  (achieve polylog height with O(z) phrases)