Compressed indexing for consecutive occurrences

P. Gawrychowski, G. Gourdel, T. Starikovskaya, T.A. Steiner

CPM 2023

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- If the query is a string, multiple space- and time-efficient solutions exist

However, *it is desirable to allow for* ***more general queries!***

- For regular expression patterns, there cannot be a data structure with polynomial-time preprocessing and sublinear query time, conditioned on the online matrix-vector multiplication conjecture [Thankanchan and Gibney, 2021]

- What about simpler models with just two patterns?

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- If we are looking for all texts in a collection that contains two string patterns P_1, P_2, or all texts containing P_1 but not P_2, the asymptotically fastest linear-space solutions use $O(\sqrt{N})$ query time, where N is the total length of the texts [Hon et al. 2010, Hon et al. 2012]
- This was shown to be optimal conditioned on Boolean Matrix Multiplication [Larsen et al. 2014] and on the 3SUM conjecture [Kopelowitz et al. 2016]

Indexing for complex queries

Indexing: preprocess a text or a collection of texts into a data structure that allows locating occurrences of a query pattern in the texts.

- [Kopelowitz and Krauthgamer 2016] considered the problem of retrieving the pair of closest occurrences of two patterns P_1, P_2 in a text T.
- For a text of length N, they showed an index using space $\tilde{O}(N^{1.5})$ with $\tilde{O}(N\sqrt{N})$ preprocessing time and $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$ query time.
- By establishing a connection with Boolean Matrix Multiplication, they highlighted a difficulty in removing the $\sqrt{N}$ factor both from the preprocessing and the query time.

$\tilde{O}$ hides the logarithmic factors.

This work

Gapped indexing for consecutive occurrences: preprocess an text T of length N into a data structure, which allows, given a range $[a, b]$ and two patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

This work

Gapped indexing for consecutive occurrences: preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

- [Navarro and Thankanchan 2016] For the case $P_1 = P_2$, $O(N \log N)$-space index with $O(|P_1| + |P_2| + \mathrm{occ})$ query time.
- [Bille et al. 2021] For the general case $P_1 \neq P_2$, no $\tilde{O}(N)$-space index can achieve $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$ query time conditioned on the Set Disjointness conjecture.

For highly compressible texts, can we design an efficient index for this problem?

$\tilde{O}$ hides the logarithmic factors.

Choice of compression method

The answer, of course, depends on the chosen compression method…

- We assume that the text is represented by a straight-line program (SLP), which is a context-free grammar describing exactly one string.

Example: The SLP $\{A \to BC, B \to ba, C \to DD, D \to na\}$ generates $banana$.

- SLPs are **capable of describing strings of exponential length** (in the size of the representation).

- Capture the popular **Lempel-Ziv compression method** up to a log factor.

- On the other hand, SLPs provide a convenient interface, allowing e.g. for efficient random access [Bille et al. 2015].

Indexing in compressed space

Assuming that a string T of length N is described by an SLP with g productions, there are multiple $\tilde{O}(g)$-space indexes for **classic pattern matching**:

Space	Query time	Reference
$O(g)$	$O(m \log \log N + \mathrm{occ} \log g)$	Claude and Navarro 2012
$O(g \log N)$	$O((m + \mathrm{occ}) \log g)$	Claude et al. 2021
$O(g \log N)$	$O(m + \mathrm{occ} \log^{\varepsilon} N)$	Christiansen et al. 2021
$O(g \log N)$	$O((m \log m + \mathrm{occ}) \log g)$	Díaz-Domínguez et al. 2021

Lower bounds for compressed data

Some problems cannot avoid a high dependency on the size of an uncompressed string:

- Pattern matching with wildcards [Aboud et al. 2017]
- Longest common subsequence [Aboud et al. 2017]
- Median edit distance [Kociumaka et al. 2022]
- Center edit distance [Kociumaka et al. 2022]

What about consecutive pattern matching?

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Cases	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^3 N \log \log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^4 N \log \log N)$

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Achieving $\tilde{O}(g)$ space
and $\tilde{O}(m + \text{occ})$ query time
would contradict the lower
bound of Bille et al. 2021

Cases	Space	me
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$+ \text{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log\log N)$

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated … $a, b]$.

> In the unbounded case, it might be possible to achieve $\tilde{O}(g)$ space and $\tilde{O}(m + \mathrm{occ})$ query time

Cases	Space	y time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \mathrm{occ}) \cdot \log^4 N \log\log N)$

Our results

Gapped indexing for consecutive occurrences:
preprocess an N-length text T into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, of length m to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Cases	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^3 N \log \log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log \log N)$

We obtain those results by using **locally consistent grammars** !

Run-length SLP

Run-length SLP a set of non-terminals, a set of terminals, an initial symbol, and a set of productions, such that:

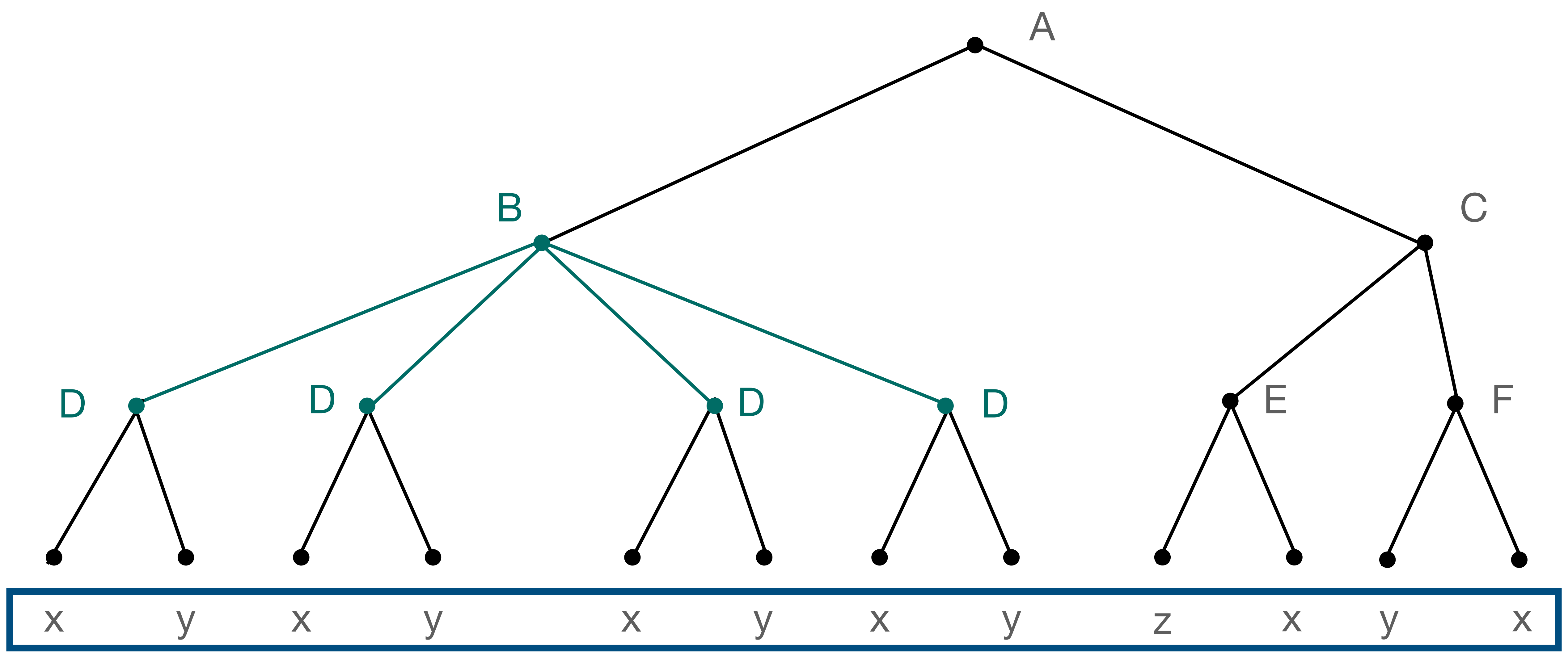
- Each production has form $A \rightarrow BC$ or $A \rightarrow B^k$, where A is a non terminal and B, C can be either terminals or non-terminals.
- Every non-terminal is on the left-hand side of exactly one production (=> it generates exactly one string).

Expansion(S), is the string “generated” by the non-terminal S.

The string obtained by iterative replacement of non-terminals by the right-hand sides of the production rules, until only terminals remain.

A run-length SLP describes the expansion of its initial symbol.

Run-length SLP

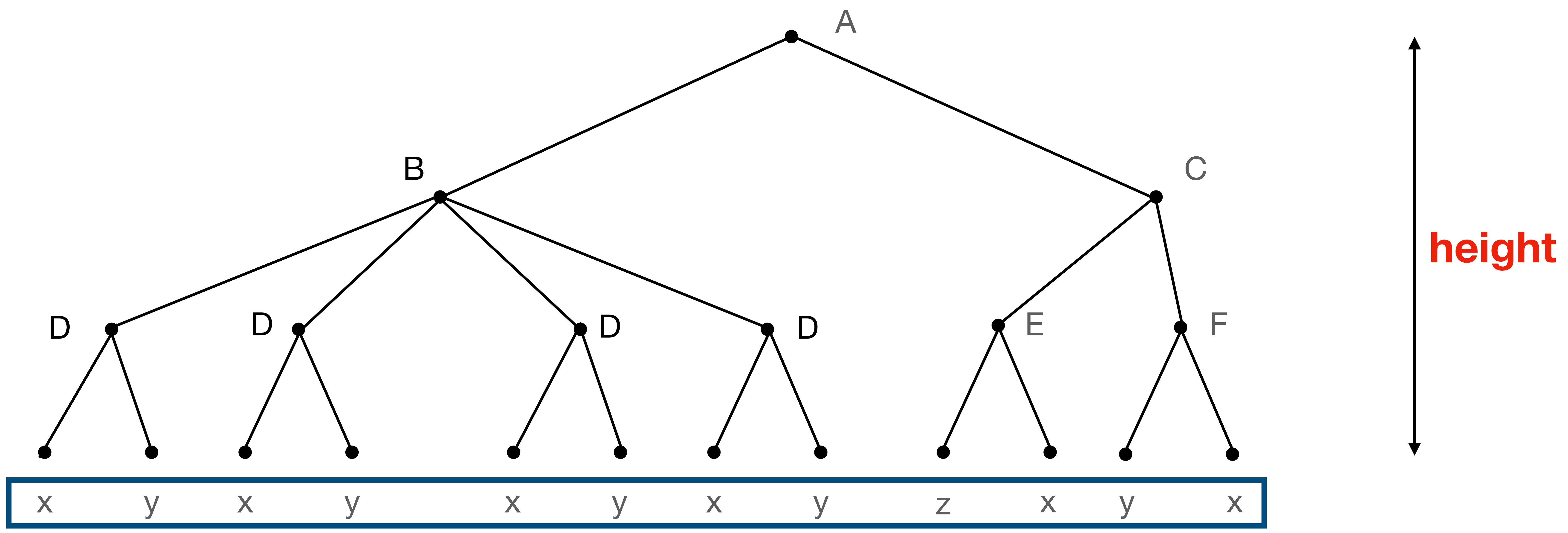


The parse tree of a run-length SLP

non-terminals $= \{A, B, C, D, E, F\}$ and terminals $= \{x, y, z\}$

productions: $A \to BC$, $B \to D^4$, $D \to xy$, $C \to EF$, $E \to zx$, $F \to yx$

Run-length SLP

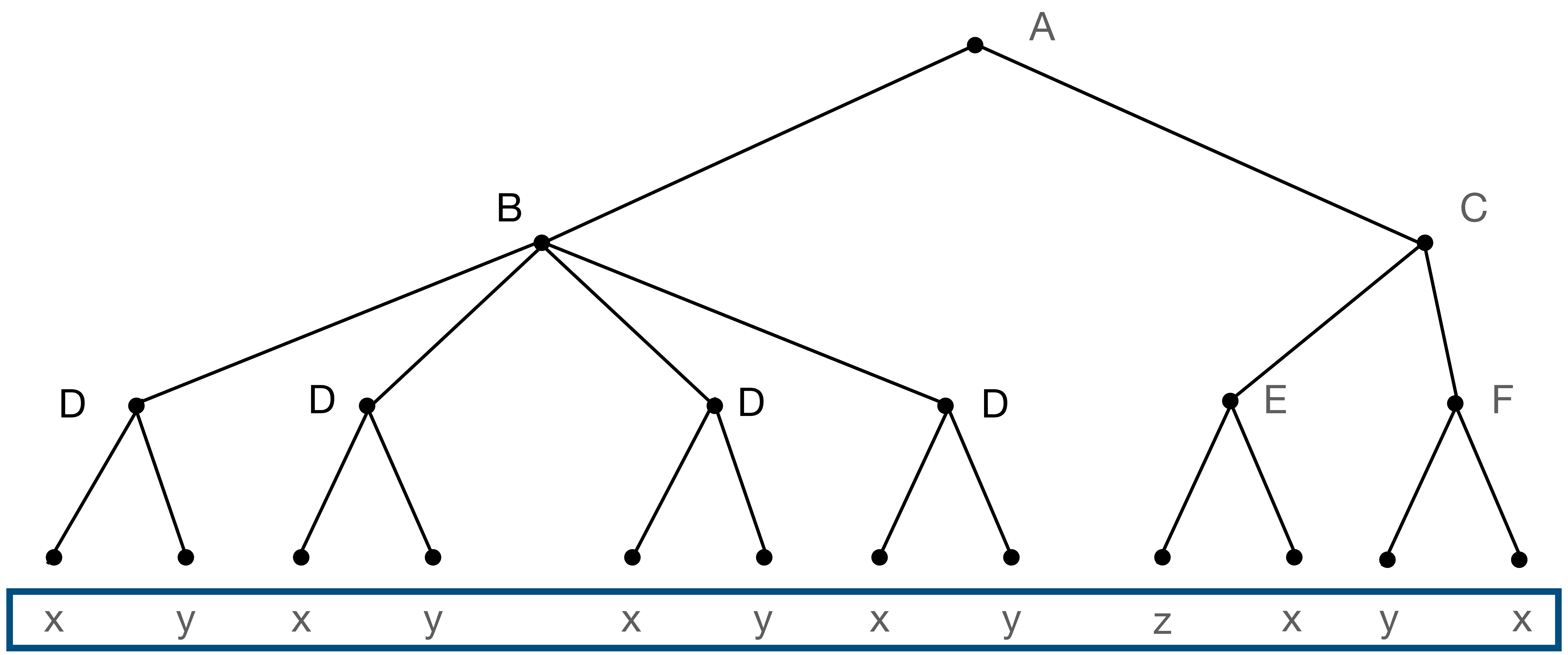


The parse tree of a run-length SLP

non-terminals $= \{A, B, C, D, E, F\}$ and terminals $= \{x, y, z\}$

productions: $\underbrace{A \to BC, B \to D^4, D \to xy, C \to EF, E \to zx, F \to yx}_{\text{size}}$

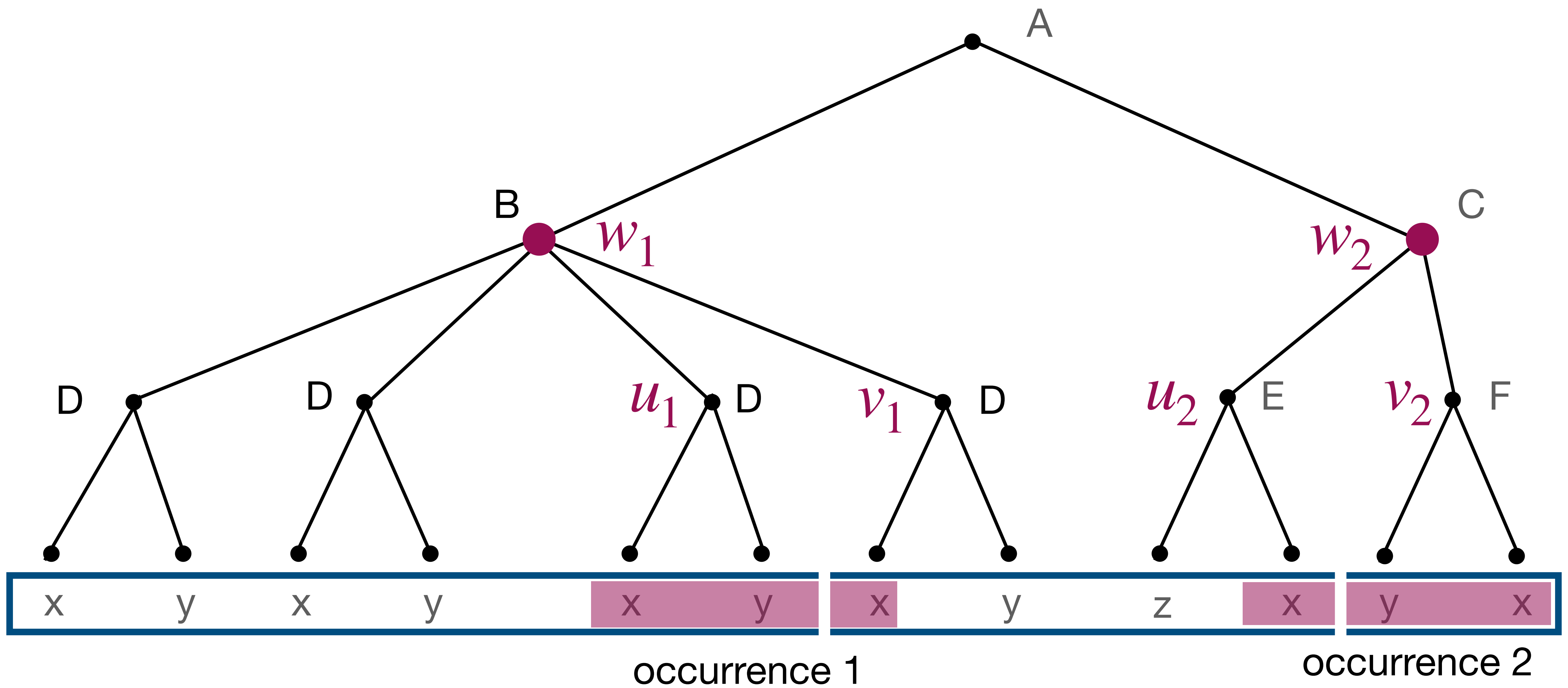
SLP to a locally-consistent run-length SLP



Corollary of [Gawrychowski et al. 2018]

There is a Las-Vegas algorithm that converts an SLP of size g describing a string T of length N into a **locally-consistent** run-length SLP of size $O(g \log N)$ and height $O(\log N)$ describing the same string T in $O(g \log N)$ time.

SLP to a locally-consistent run-length SLP



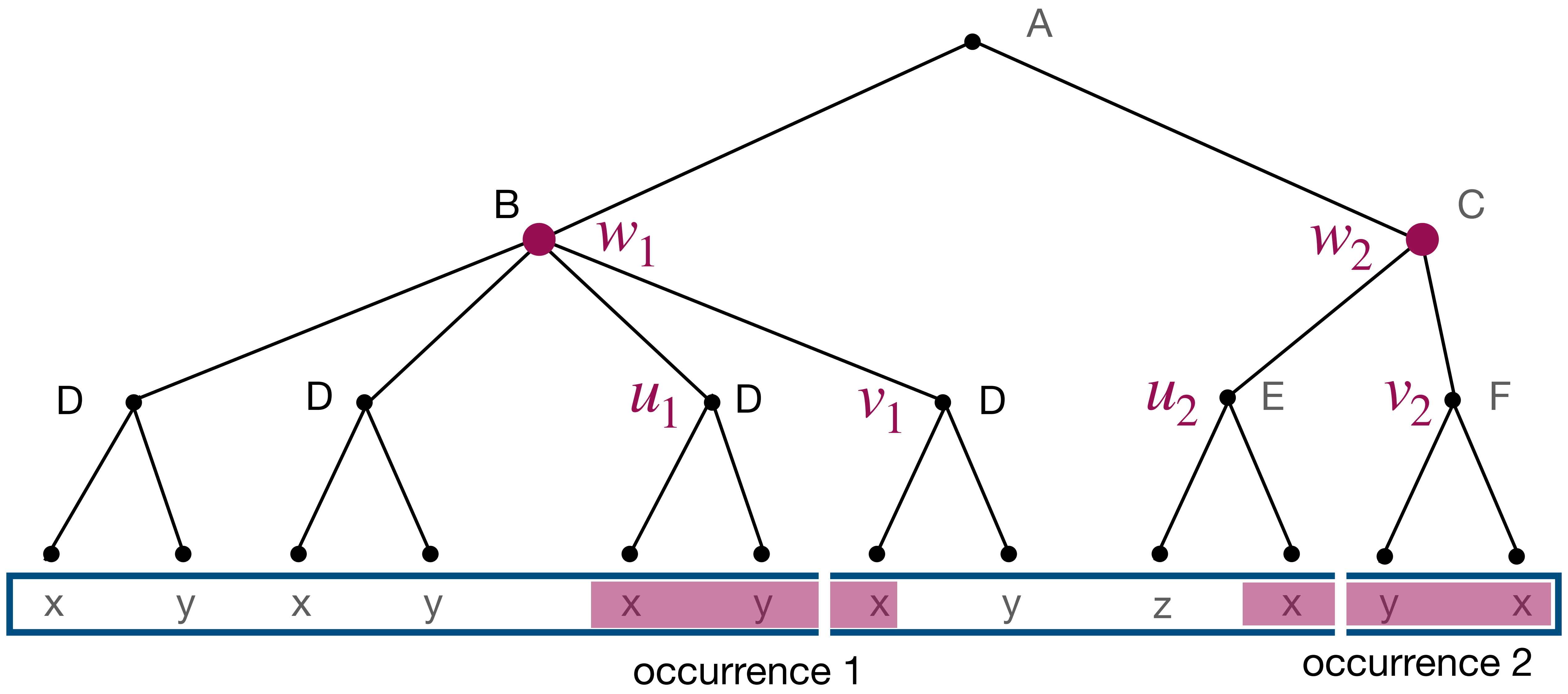
Definition: Split

For an occurrence $T[\ell, r]$ of a pattern P, let w be the lowest node of a parse tree containing it, we say that $T[\ell, r]$ is **relevant** for the label of w (a non-terminal).

$T[\ell, r]$ is **split at position i** if there exist children u, v of w such that $T[\ell, \ell + i]$ is contained in u and $T[\ell + i + 1, r]$ in v.

Example: Occurrence 1 xyx is split at position 2 and relevant for B, occurrence 2 is split at position 1 and relevant for C.

SLP to a locally-consistent run-length SLP



Locally-consistent run-length SLP

There are only $O(\log N)$ possible splits of P, and we can compute them in $O(|P| \log N)$ time

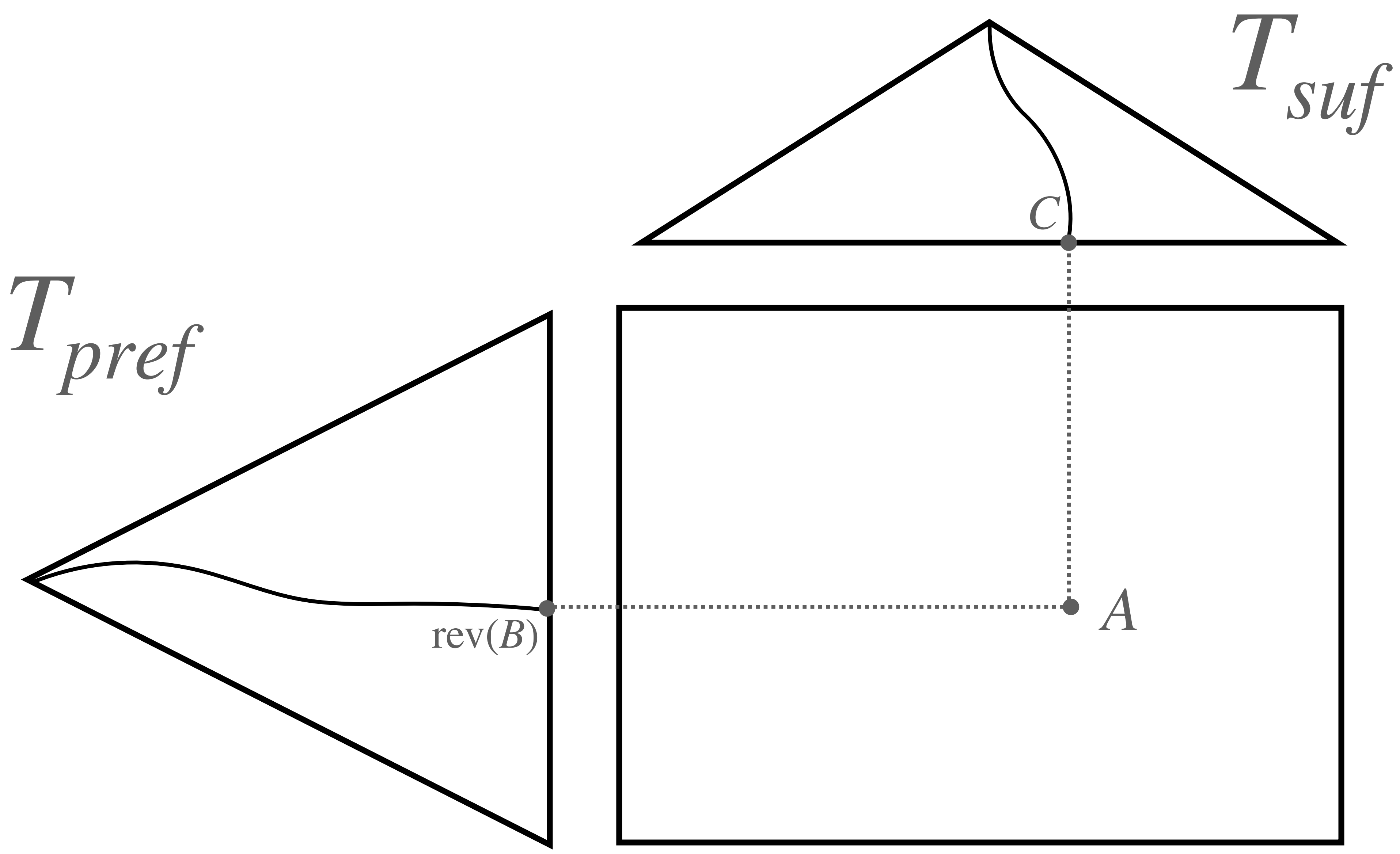
Definition: Split

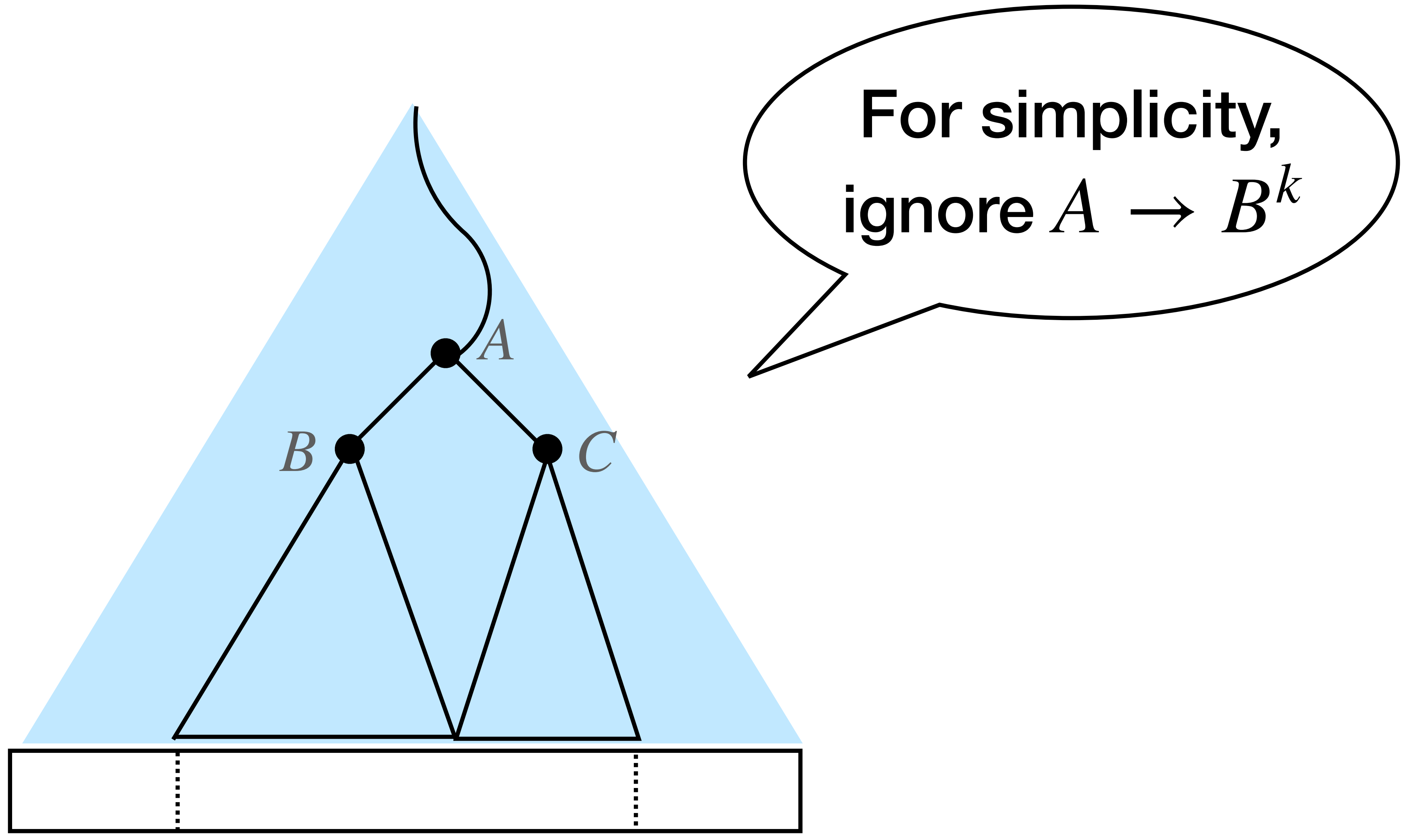
For an occurrence $T[\ell, r]$ of a pattern P, let w be the lowest node of a parse tree containing it, we say that $T[\ell, r]$ is **relevant** for the label of w (a non-terminal).

$T[\ell, r]$ is **split at position i** if there exist children u, v of w such that $T[\ell, \ell + i]$ is contained in u and $T[\ell + i + 1, r]$ in v.

Example: Occurrence 1 xyx is split at position 2 and relevant for B, occurrence 2 is split at position 1 and relevant for C.

Why is local consistency interesting?

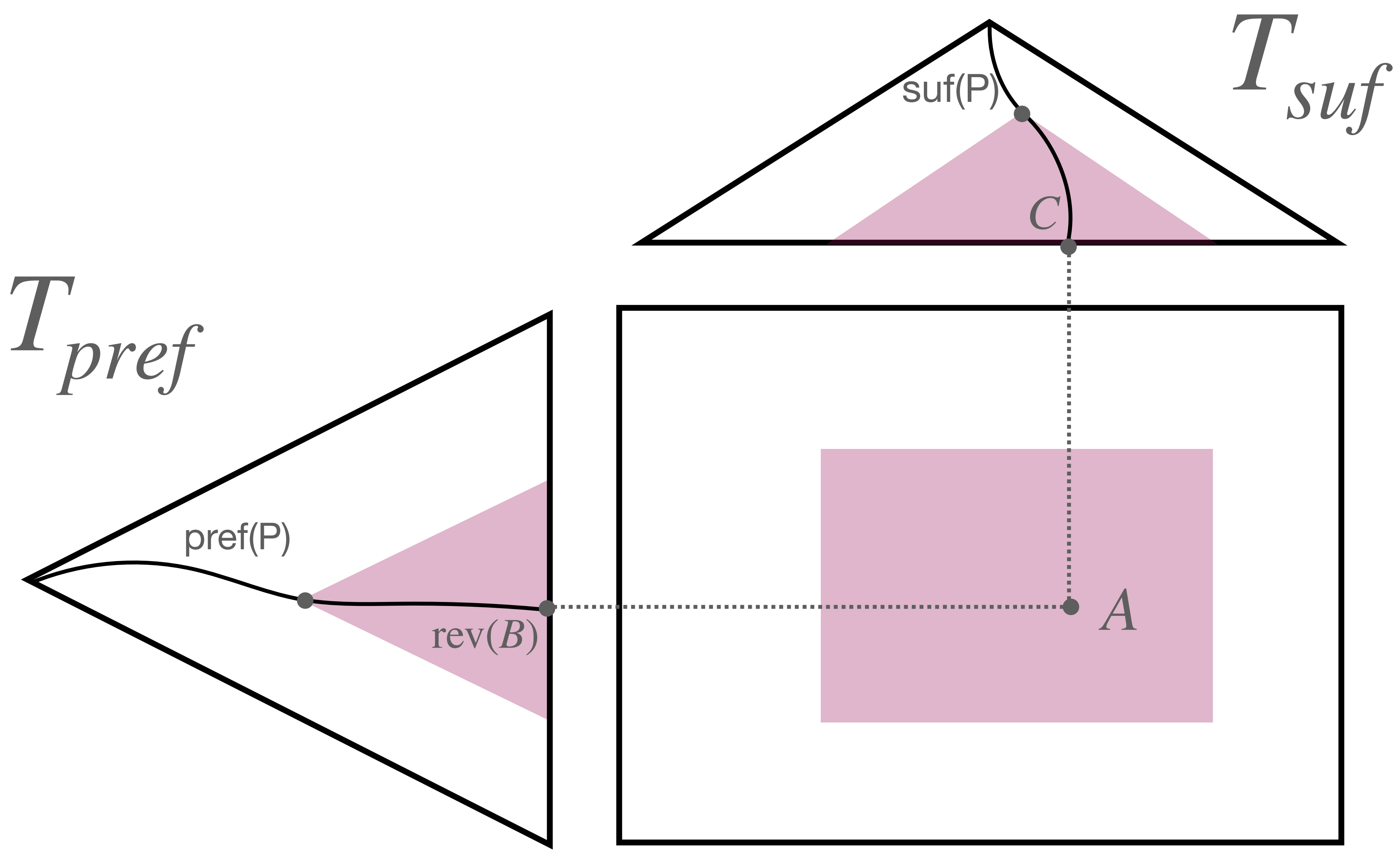


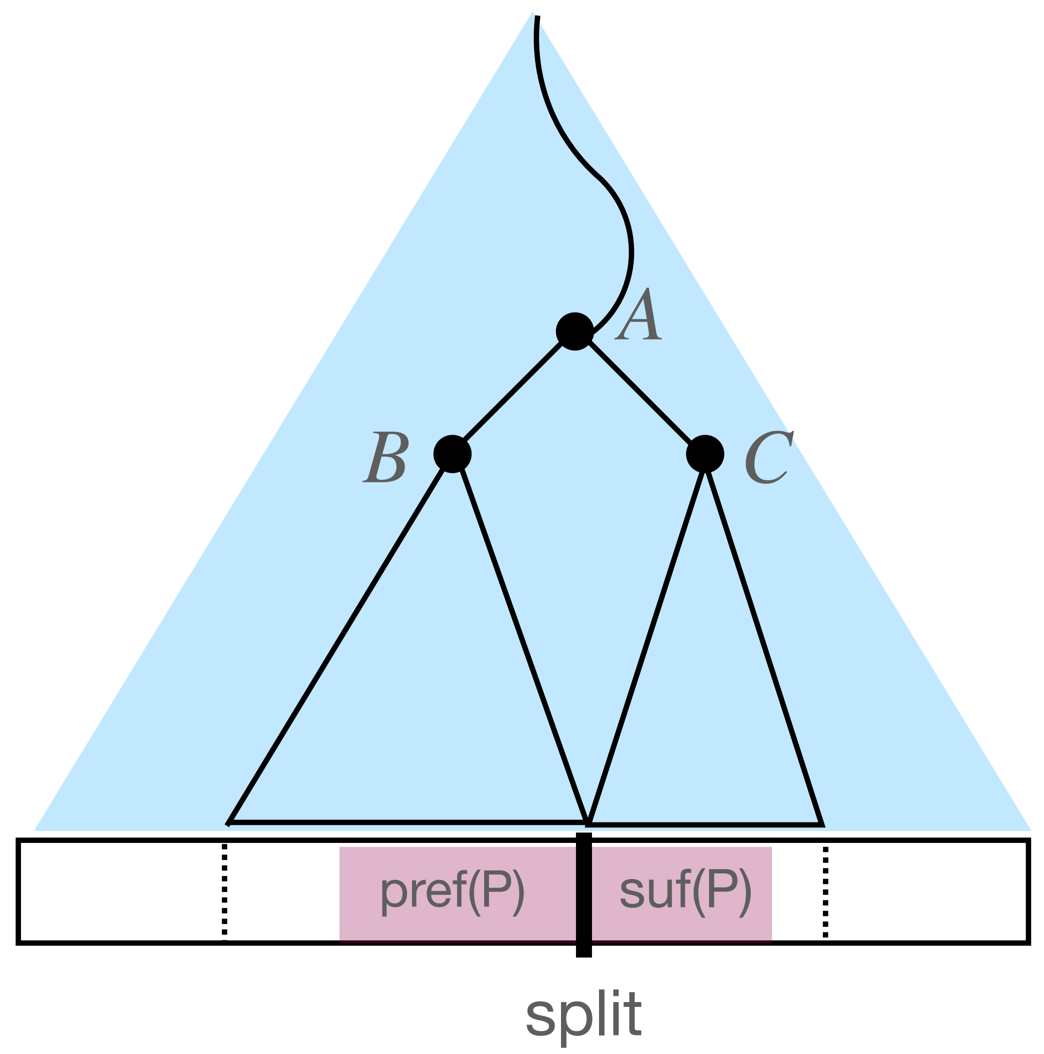


…because we can search for relevant occurrences quickly using the following structure:

- For every $A \to BC$, add $\mathrm{rev}(\mathrm{expansion}(B))$ to T_{pref} and $\mathrm{expansion}(C)$ to T_{suf}
- Create a point (r_B, r_C) (the lexicographic rank of the expensions) for every $A \to BC$
- Build an orthogonal range data structure on the points

Why is local consistency interesting?





To find relevant occurrences of a pattern P in non-terminals:

- For each split s, search for $\mathrm{pref}(P) = \mathrm{rev}(P[1..s])$ in T_{pref} to obtain an interval I_{pref} of leaves starting with it, and for $\mathrm{suf}(P) = P[s+1..]$ in T_{suf} to obtain an interval I_{suf}
- Report all non-terminals in $I_{pref} \times I_{suf}$

We show an even stronger result

For a run-length SLP representing a string T of length N, with size g and height $O(\log n)$,

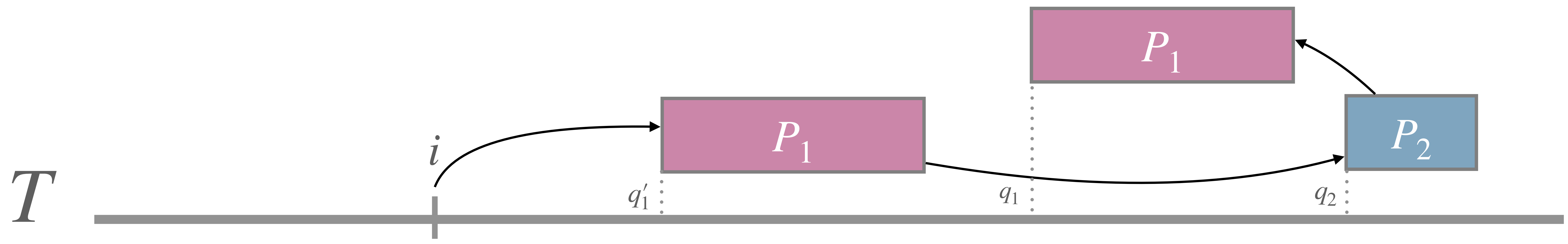
There is **a $O(g^2 \log^2 N)$-space data structure** that preprocesses an m-length pattern P in $O(m \log N + \log^2 N)$ time and can, answer the following queries in $\mathrm{polylog}\, N$ time:

For a given non terminal A, in $\mathrm{expansion}(A)$,

- Report relevant occurrences of P;
- Decide whether there is an occurrence of P;
- Report the leftmost/rightmost occurrence of P ;
- Find a predecessor/successor occurrence of P given a position q.

Corollary: unbounded case

Task: report all consecutive occurrences of P_1, P_2 in a N-length string T described by a run-length SLP of size g and height $\log N$.



1. Find the leftmost occurrence q_1' of P_1 in $T[i\,..\,]$ (successor)
2. Find the leftmost occurrence $q_2 \geq q_1'$ of P_2 (successor)
3. Find the rightmost occurrence $q_1 \leq q_2$ of P_1 (predecessor)
4. Report (q_1, q_2) and set $i = q_2 + 1$

Time $\tilde{O}(m + (\mathrm{occ} + 1)\mathrm{polylog}\, N)$, space $\tilde{O}(g^2)$.

Idea of our index for the case $a = 0$

Task: given an N-length string T described by a run-length SLP of size g and height $\log N$, report all consecutive occurrences of P_1, P_2 in T separated by distance in $[0,b]$.

- For each non-terminal of the grammar, retrieve relevant consecutive occurrences separated by distance in $[0,b]$.
- Generate all consecutive occurrences separated by distance in $[0,b]$ by traversing a pruned parse tree of the grammar (using a standard technique borrowed from classic pattern matching compressed-space indexes).

Conclusion and open questions

Gapped indexing for consecutive occurrences: preprocess a text T of length N into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Case	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^3 N \log \log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log \log N)$

Can better space be achieved?

Is there a compressed-space solution for the general case?

Conclusion and open questions

Gapped indexing for consecutive occurrences: preprocess a text T of length N into a data structure, which allows, given a range $[a, b]$ and two string patterns P_1, P_2, to retrieve all pairs of consecutive occurrences of P_1, P_2 separated by distance $d \in [a, b]$.

Case	Space	Query time
unbounded (a = 0, b = N)	$O(g^2 \log^4 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^3 N \log\log N)$
a = 0	$O(g^5 \log^5 N)$	$O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log\log N)$

Can better space be achieved?

Is there a compressed-space solution for the general case?

Thank you for your attention! Any questions ?