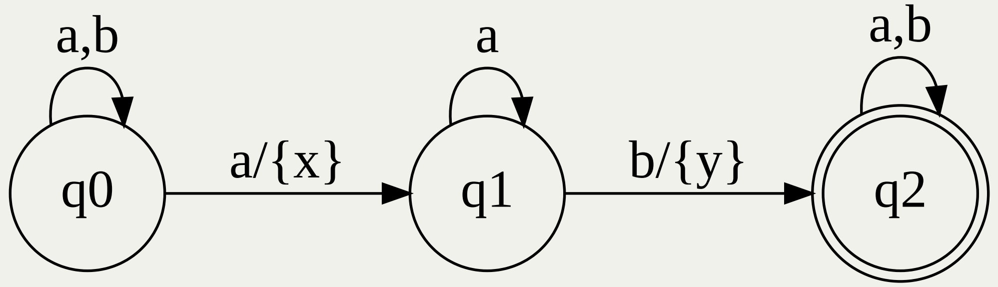
Dynamic direct access of MSO query evaluation over strings

Pierre Bourhis, *Florent Capelli*, Stefan Mengel and Cristian Riveros
CRIL, Université d’Artois
ICDT 2025
25 March 2025

Variable Set Automata

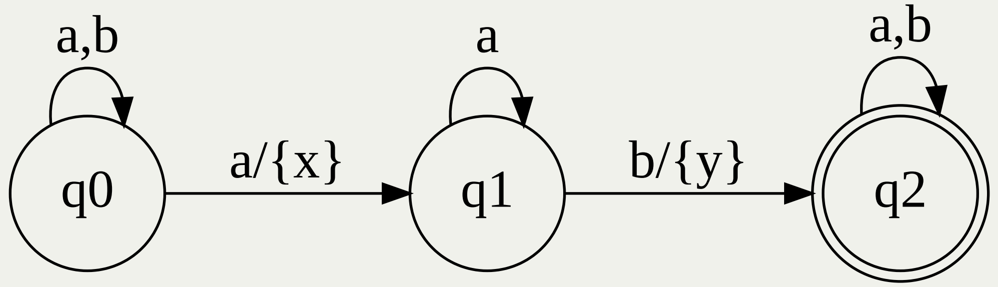
Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12

$[\![A]\!](w)$	x	y

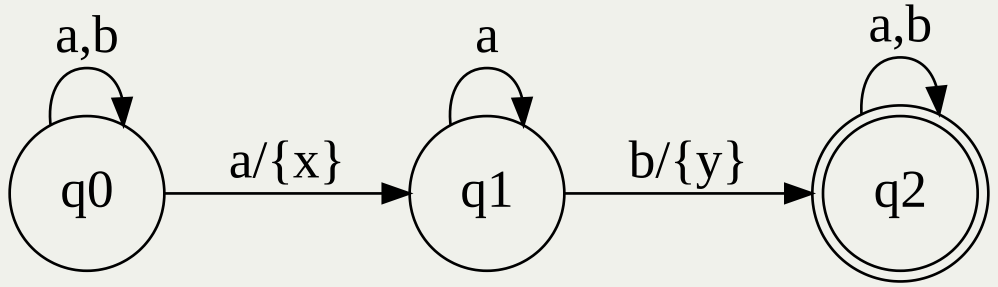
Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12
Run 1		x		y								

$[\![A]\!](w)$	x	y
(run 1)	2	4

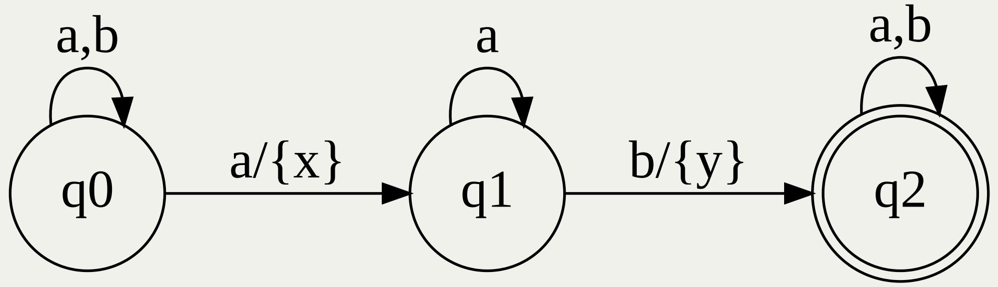
Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12
Run 1		x		y								
Run 2					x			y				

$[\![A]\!](w)$	x	y
(run 1)	2	4
(run 2)	5	8

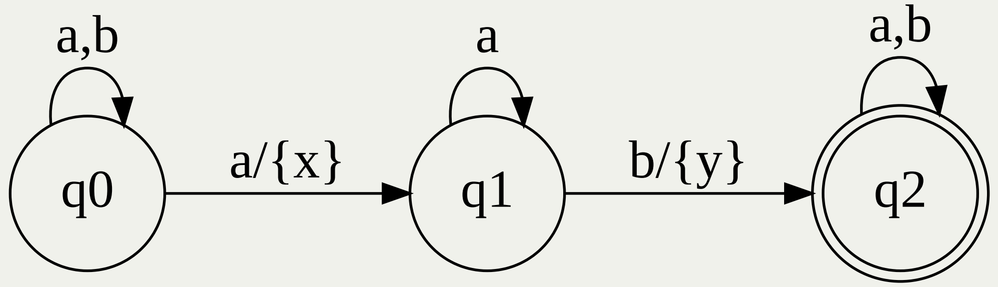
Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12
Run 1		x		y								
Run 2					x			y				
Run 3									x	y		

$[\![A]\!](w)$	x	y
(run 1)	2	4
(run 2)	5	8
(run 3)	9	10

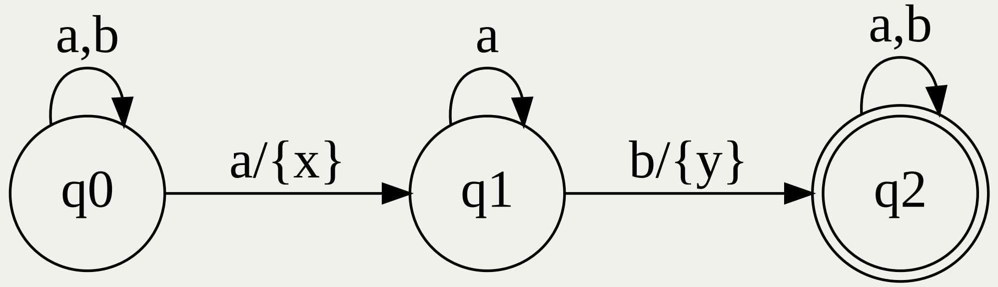
Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12
Run 1		x		y								
Run 2					x			y				
Run 3									x	y		

$[\![A]\!](w)$	x	y
(run 1)	2	4
	3	4
(run 2)	5	8
	6	8
	7	8
(run 3)	9	10

Tagging positions in words



w	b	a	a	b	a	a	a	b	a	b	a	a
Positions	1	2	3	4	5	6	7	8	9	10	11	12
Run 1		x		y								
Run 2					x			y				
Run 3									x	y		

$[\![A]\!](w)$	x	y
(run 1)	2	4
	3	4
(run 2)	5	8
	6	8
	7	8
(run 3)	9	10

Build a data structure allowing to access each tuple in $[\![A]\!](w)$ efficiently.

Related formalisms

VSet Automata are akin to:

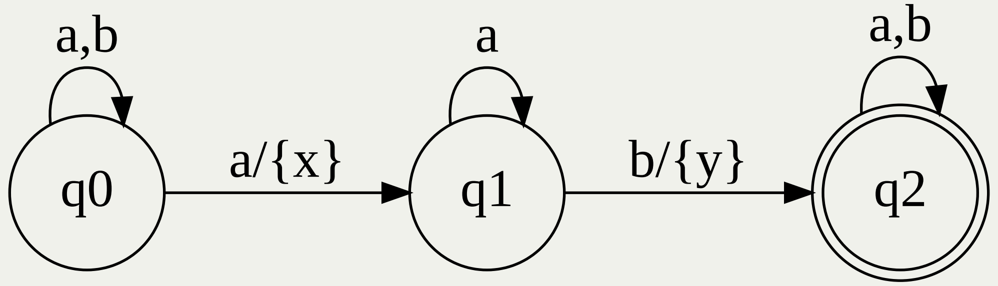
- **Document spanners**: “span” can be encoded as two variables s_x (start x) and e_x (end x)

Related formalisms

VSet Automata are akin to:

- **Document spanners**: “span” can be encoded as two variables s_x (start x) and e_x (end x)
- **MSO over words**:
    - relation $a(x)$ for each letter a: “there is an a at position x
    - order $<$ on positions
    - first order and monadic second order quantifications
    - **Theorem**: for each such formula φ, there exists a vset automate A_φ such that $[\![A_\varphi]\!] = [\![\varphi]\!]$.

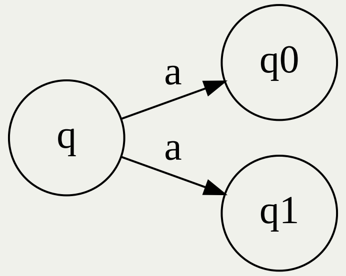
$$\varphi(x,y) \equiv a(x) \wedge b(y) \wedge \forall z((x < z \wedge z < y) \Rightarrow a(z))$$



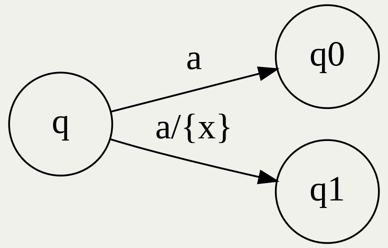
vset automata A_φ for φ

Desirable properties

- **Determinism**: two edges going out of state q have distinct labels.



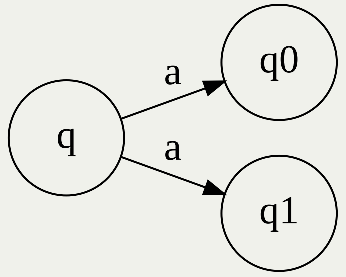
Forbidden



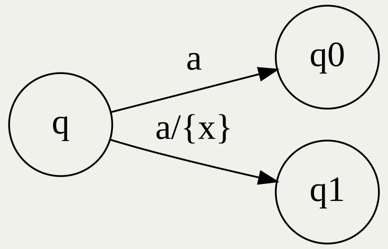
Allowed

Desirable properties

- **Determinism**: two edges going out of state q have distinct labels.

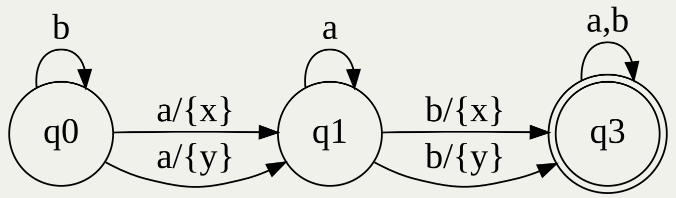


Forbidden

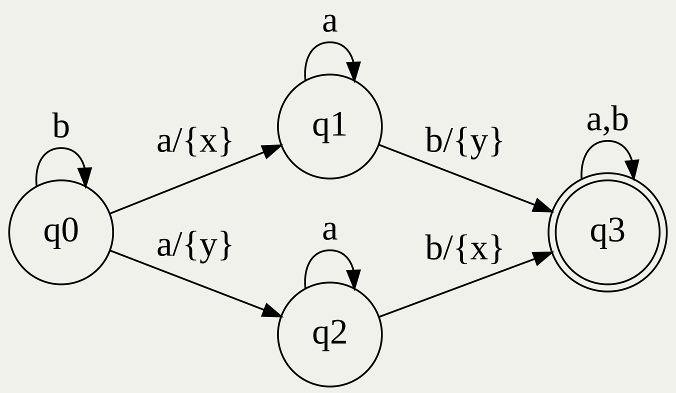


Allowed

- **Functionality**: every path from q_0 to a final state tags each variable exactly once.



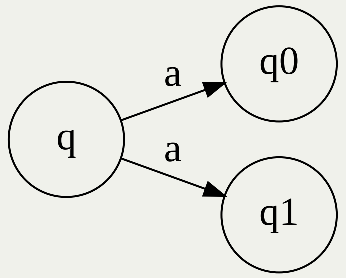
Not functional



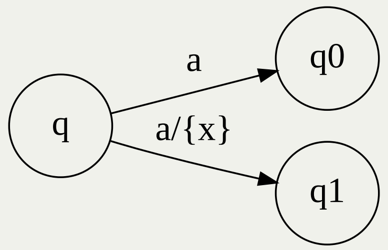
Functional version

Desirable properties

- **Determinism**: two edges going out of state q have distinct labels.

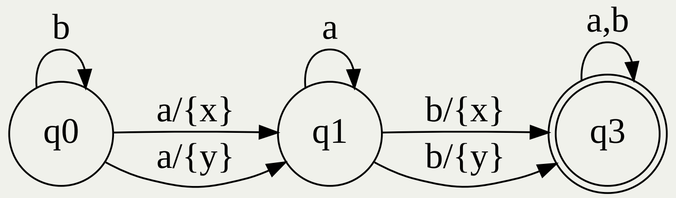


Forbidden

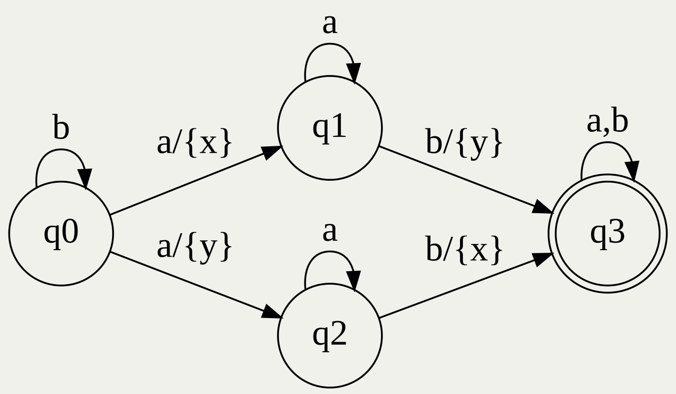


Allowed

- **Functionality**: every path from q_0 to a final state tags each variable exactly once.



Not functional



Functional version

Functionnality: every q is mapped with X_q, variables tagged before reaching q.

Normalization

Let A be an vset automaton. One can construct a deterministic function vset automaton A' such that $[\![A]\!] = [\![A']\!]$.

- Intuition: automata over states $2^{Q \cup X}$.
- A' may be of size $exp(A)$

In this talk: **data complexity model** where w is the *data* and A is the *query*.

$\Rightarrow$ A is considered constant, hence assumed to be deterministic and functional.

Direct Access for vset automata

Direct access queries

Fix a (deterministic functional) automaton $A(x_1, \ldots, x_k)$ (considered constant).

A **direct access query** for a word $w \in \Sigma^*$:

- input integer k,
- output the kth tuple of $[\![A]\!](w)$ or,
- **fails** if $k > \#[\![A]\!](w)$.

where $[\![A]\!](w)$ is ordered by lexicographical ordering on $[n]^X$.

Direct access queries

Fix a (deterministic functional) automaton $A(x_1, \dots, x_k)$ (considered constant).

A **direct access query** for a word $w \in \Sigma^*$:

- input integer k,
- output the kth tuple of $[\![A]\!](w)$ or,
- **fails** if $k > \#[\![A]\!](w)$.

where $[\![A]\!](w)$ is ordered by lexicographical ordering on $[n]^X$.

$[\![A]\!](w)$	x	y
	2	4
	3	4
	5	8
	6	8
	7	8
	9	10

Direct access queries

Fix a (deterministic functional) automaton $A(x_1, \ldots, x_k)$ (considered constant).

A **direct access query** for a word $w \in \Sigma^*$:

- input integer k,
- output the kth tuple of $[\![A]\!](w)$ or,
- **fails** if $k > \#[\![A]\!](w)$.

where $[\![A]\!](w)$ is ordered by lexicographical ordering on $[n]^X$.

$[\![A]\!](w)$	x	y
	2	4
	3	4
	5	8
	6	8
	7	8
	9	10

$[\![A]\!][3]$

Direct access queries

Fix a (deterministic functional) automaton $A(x_1, \ldots, x_k)$ (considered constant).

A **direct access query** for a word $w \in \Sigma^*$:

- input integer k,
- output the kth tuple of $[\![A]\!](w)$ or,
- **fails** if $k > \#[\![A]\!](w)$.

where $[\![A]\!](w)$ is ordered by lexicographical ordering on $[n]^X$.

$[\![A]\!](w)$	x	y
	2	4
	3	4
	5	8
	6	8
	7	8
	9	10

$[\![A]\!][10]$

Direct access queries

Fix a (deterministic functional) automaton $A(x_1, \ldots, x_k)$ (considered constant).

A **direct access query** for a word $w \in \Sigma^*$:

- input integer k,
- output the kth tuple of $[\![A]\!](w)$ or,
- **fails** if $k > \#[\![A]\!](w)$.

where $[\![A]\!](w)$ is ordered by lexicographical ordering on $[n]^X$.

$[\![A]\!](w)$	x	y
	2	4
	3	4
	5	8
	6	8
	7	8
	9	10

$[\![A]\!][5]$

Direct access complexity

Given $w \in \Sigma^*$ of length n:

- **Precomputation phase**: construct a data structure D_w in time $p(n)$.
- **Access phase**: given k, output $[\![A]\!](w)[k]$ in time $a(n)$ using D_w.

Direct access complexity

Given $w \in \Sigma^*$ of length n:

- **Precomputation phase**: construct a data structure D_w in time $p(n)$.
- **Access phase**: given k, output $[\![A]\!](w)[k]$ in time $a(n)$ using D_w.

Naive approach:

- **Precomputation**: D_w is a materialization of $[\![A]\!](w)$ in an array.
- **Access phase**: read the kth entry of D_w.

Direct access complexity

Given $w \in \Sigma^*$ of length n:

- **Precomputation phase**: construct a data structure D_w in time $p(n)$.
- **Access phase**: given k, output $[\![A]\!](w)[k]$ in time $a(n)$ using D_w.

Naive approach:

- **Precomputation**: D_w is a materialization of $[\![A]\!](w)$ in an array. $p(n) \geq O(\#[\![A]\!](w))$
- **Access phase**: read the kth entry of D_w.

Direct access complexity

Given $w \in \Sigma^*$ of length n:

- **Precomputation phase**: construct a data structure D_w in time $p(n)$.
- **Access phase**: given k, output $[\![A]\!](w)[k]$ in time $a(n)$ using D_w.

Naive approach:

- **Precomputation**: D_w is a materialization of $[\![A]\!](w)$ in an array. $p(n) \geq O(\#[\![A]\!](w))$
- **Access phase**: read the kth entry of D_w. $a(n) = O(1)$

Direct access complexity

Given $w \in \Sigma^*$ of length n:

- **Precomputation phase**: construct a data structure D_w in time $p(n)$.
- **Access phase**: given k, output $[\![A]\!](w)[k]$ in time $a(n)$ using D_w.

Naive approach:

- **Precomputation**: D_w is a materialization of $[\![A]\!](w)$ in an array. $p(n) \geq O(\#[\![A]\!](w))$
- **Access phase**: read the kth entry of D_w. $a(n) = O(1)$

Can we have better preprocessing without hurting access time too much?

Contribution

We show that we can solve direct access for $[\![A]\!](w)$ in time:

- Linear time precomputation $O(|w|)$,
- Polylogarithmic access time $O(\log^2 |w|)$.

$O(\cdot)$ notation hides constants depending on $|A|$ but they are all polynomially bounded if A is deterministic and unambiguous.

Data structure idea

Counting reduction

Let $\tau = [\![A]\!](w)[k]$.

$\tau(x_1)$ is the first position p_1 for which
$\#[\![A]\!]_{x_1 \leq p_1}(w) \geq k$

	x_1	$\dots$	x_k
	$< p_1$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
k	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$> p_1$	$\dots$	$\dots$

Counting reduction

Let $\tau = [\![A]\!](w)[k]$.

$\tau(x_1)$ is the first position p_1 for which
$\#[\![A]\!]_{x_1 \leq p_1}(w) \geq k$

	x_1	$\dots$	x_k
	$< p_1$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
k	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$> p_1$	$\dots$	$\dots$

Counting reduction

Let $\tau = [\![A]\!](w)[k]$.

	x_1	$\dots$	x_k
	$< p_1$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
k	p_1	$\dots$	$\dots$
	$\dots$	$\dots$	$\dots$
	p_1	$\dots$	$\dots$
	$> p_1$	$\dots$	$\dots$

$\tau(x_1)$ is the first position p_1 for which

$$\#[\![A]\!]_{x_1 \leq p_1}(w) \geq k$$

- Binary search to find p_1: $O(\log n)$ calls to computing $\#[\![A]\!]_{x_1 \leq p}(w) \geq k$ by changing p
- We proceed recursively to find $\tau(x_2)$: first value p such that

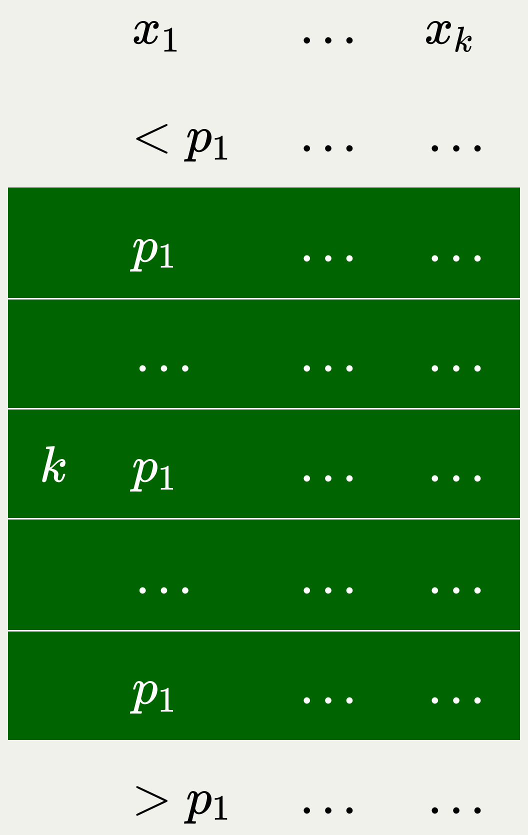
$$\#[\![A]\!]_{x_1 = p_1, x_2 \leq p}(w) \geq k$$

Counting reduction

Let $\tau = [\![A]\!](w)[k]$.

$\tau(x_1)$ is the first position p_1 for which

$\#[\![A]\!]_{x_1 \leq p_1}(w) \geq k$



	x_1	…	x_k
	$< p_1$	…	…
	p_1	…	…
	…	…	…
k	p_1	…	…
	…	…	…
	p_1	…	…
	$> p_1$	…	…

- Binary search to find p_1: $O(\log n)$ calls to computing $\#[\![A]\!]_{x_1 \leq p}(w) \geq k$ by changing p
- We proceed recursively to find $\tau(x_2)$: first value p such that

$$\#[\![A]\!]_{x_1 = p_1, x_2 \leq p}(w) \geq k - \#[\![A]\!]_{x_1 < p_1}(w)$$

Maintaining matrix products

We can express $\#[\![A]\!]_{x \leq p}(w)$ as a matrix product (transition matrices):

$$P \cdot M_1 \dots M_n \cdot R$$

Moreover: $\#[\![A]\!]_{x \leq p}$ and $\#[\![A]\!]_{x \leq r}$ are the same product but for M_p and M_r.

Final data structure D: represents a matrix product $A_1 \cdot \dots \cdot A_r$ such that one can quickly update D so that it represents the product where A_i is replaced by B_i.

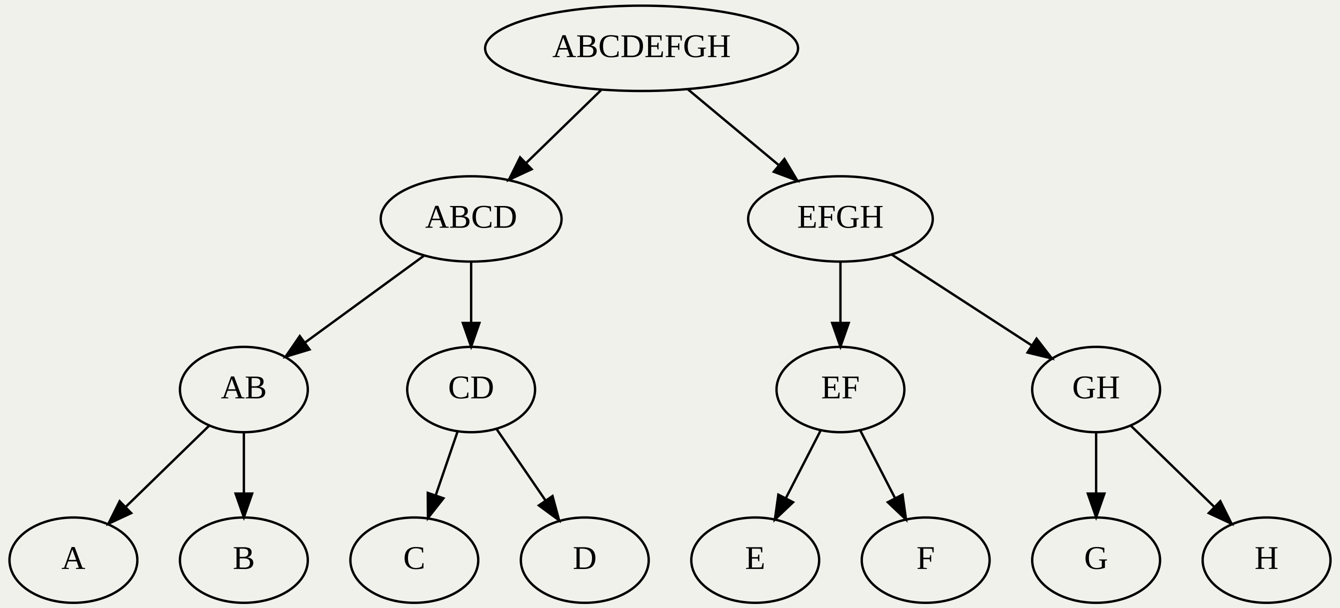
Maintaining matrix products

We can express $\#[\![A]\!]_{x \leq p}(w)$ as a matrix product (transition matrices):

$$P \cdot M_1 \dots M_n \cdot R$$

Moreover: $\#[\![A]\!]_{x \leq p}$ and $\#[\![A]\!]_{x \leq r}$ are the same product but for M_p and M_r.

Final data structure D: represents a matrix product $A_1 \cdot \dots \cdot A_r$ such that one can quickly update D so that it represents the product where A_i is replaced by B_i.



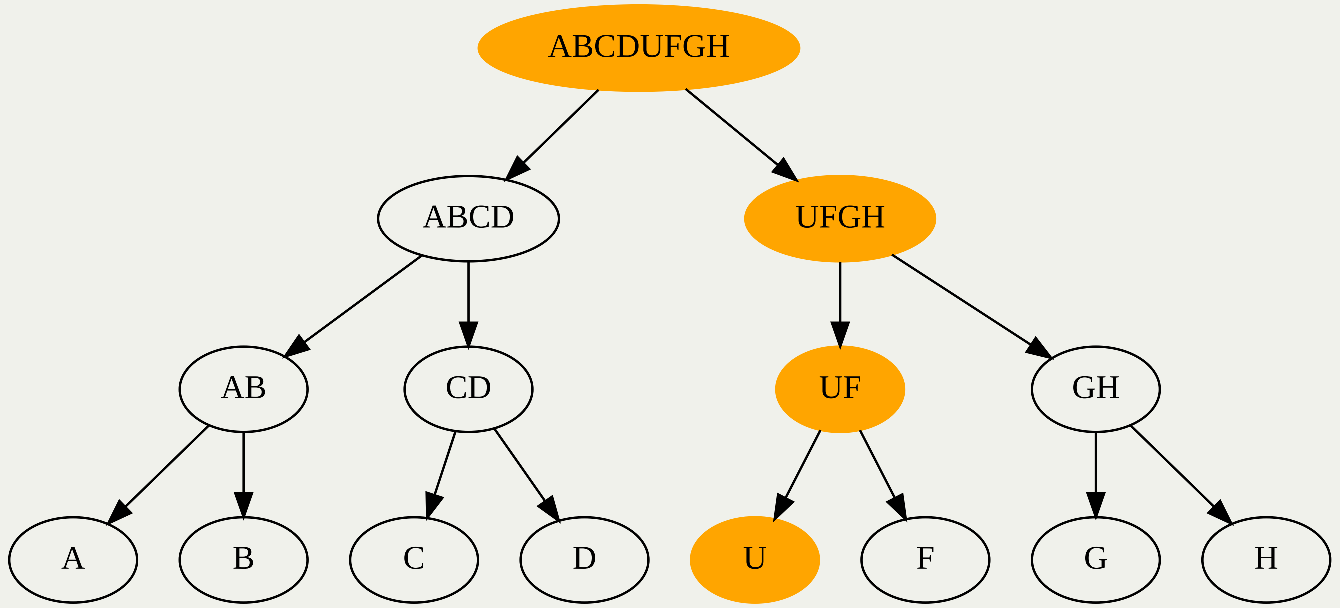
Maintaining matrix products

We can express $\#[\![A]\!]_{x \leq p}(w)$ as a matrix product (transition matrices):

$$P \cdot M_1 \dots M_n \cdot R$$

Moreover: $\#[\![A]\!]_{x \leq p}$ and $\#[\![A]\!]_{x \leq r}$ are the same product but for M_p and M_r.

Final data structure D: represents a matrix product $A_1 \cdot \dots \cdot A_r$ such that one can quickly update D so that it represents the product where A_i is replaced by B_i.



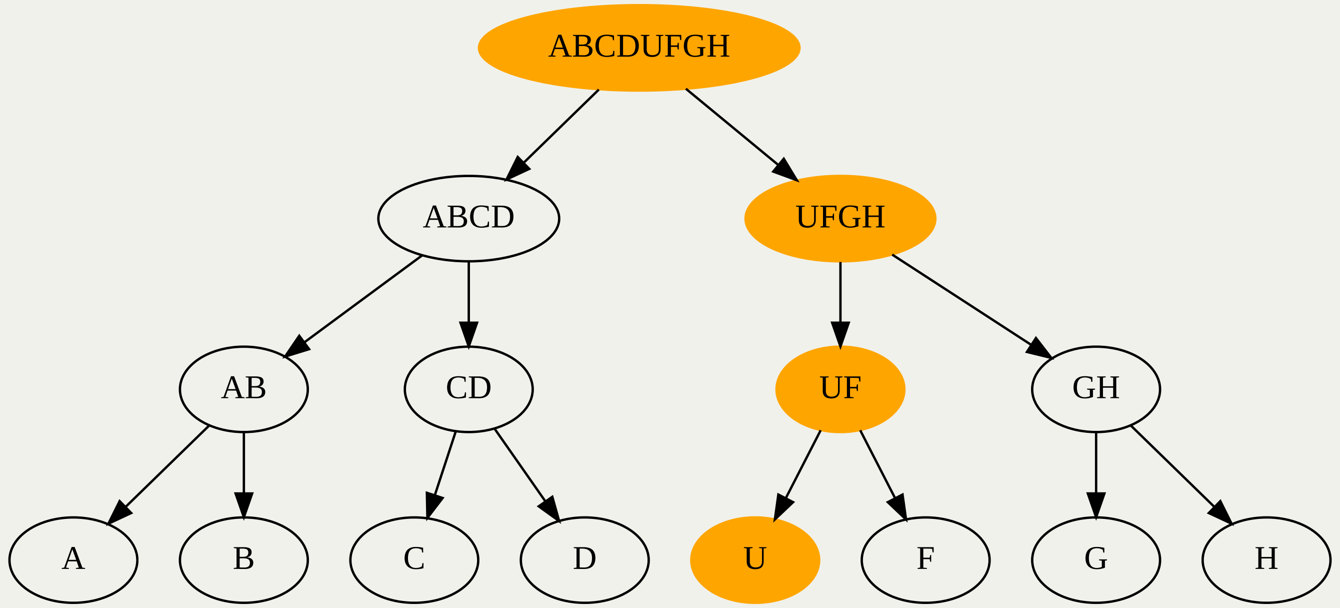
Maintaining matrix products

We can express $\#[\![A]\!]_{x \leq p}(w)$ as a matrix product (transition matrices):

$$P \cdot M_1 \dots M_n \cdot R$$

Moreover: $\#[\![A]\!]_{x \leq p}$ and $\#[\![A]\!]_{x \leq r}$ are the same product but for M_p and M_r.

Final data structure D: represents a matrix product $A_1 \cdot \dots \cdot A_r$ such that one can quickly update D so that it represents the product where A_i is replaced by B_i.



Update time: $O(\log n)$.

Dynamic words

Given data structures D_1 for w_1 and D_2 for w_2,
$k \leq |w_1|, i, j \leq |w_2|$:

$w_1 = a_1 \ldots a_n$, $w_2 = b_1 \ldots b_n$

Construct D_1' for w_1' and D_2' for w_2':

$w_1' = a_1 \ldots a_k \; b_i \ldots b_j \; a_{k+1} \ldots a_n$

$w_2' = b_1 \ldots b_{j-1} b_{j+1} \ldots b_m$

Dynamic words

Given data structures D_1 for w_1 and D_2 for w_2, $k \leq |w_1|, i, j \leq |w_2|$:

$w_1 = a_1 \ldots a_n$, $w_2 = b_1 \ldots b_n$

Construct D_1' for w_1' and D_2' for w_2':

$w_1' = a_1 \ldots a_k \; b_i \ldots b_j \; a_{k+1} \ldots a_n$

$w_2' = b_1 \ldots b_{j-1} b_{j+1} \ldots b_m$

- Concatenation of words (cut w_2 completely and paste at the end of w_1)
- Insertion of letter (create data structure for $w_2 = a$ in $O(1)$, cut it and paste it in w_1)
- Remove substring (cut the substring)
- Update letter (cut the letter and insert a new one in its place)

Dynamic words

Given data structures D_1 for w_1 and D_2 for w_2, $k \leq |w_1|, i, j \leq |w_2|$:

$w_1 = a_1 \dots a_n$, $w_2 = b_1 \dots b_n$

Construct D_1' for w_1' and D_2' for w_2':

$$w_1' = a_1 \dots a_k \; b_i \dots b_j \; a_{k+1} \dots a_n$$

$$w_2' = b_1 \dots b_{j-1} b_{j+1} \dots b_m$$

- Concatenation of words (cut w_2 completely and paste at the end of w_1)
- Insertion of letter (create data structure for $w_2 = a$ in $O(1)$, cut it and paste it in w_1)
- Remove substring (cut the substring)
- Update letter (cut the letter and insert a new one in its place)

Naive approach: in $O(n + m)$ by doing it from scratch.

Dynamic words

Given data structures D_1 for w_1 and D_2 for w_2, $k \leq |w_1|, i, j \leq |w_2|$:

$w_1 = a_1 \dots a_n$, $w_2 = b_1 \dots b_n$

Construct D_1' for w_1' and D_2' for w_2':

$w_1' = a_1 \dots a_k \; b_i \dots b_j \; a_{k+1} \dots a_n$

$w_2' = b_1 \dots b_{j-1} b_{j+1} \dots b_m$

- Concatenation of words (cut w_2 completely and paste at the end of w_1)
- Insertion of letter (create data structure for $w_2 = a$ in $O(1)$, cut it and paste it in w_1)
- Remove substring (cut the substring)
- Update letter (cut the letter and insert a new one in its place)

Naive approach: in $O(n+m)$ by doing it from scratch.

Possible $O(\log(n+m))$: use AVL tree operations to keep matrix product tree from balanced.

Future work

- Implementations:
    - reasonable data structures
    - updates may be cheap enough to maintain a query on a code base.
- Generalizations:
    - Orders that are not lexicographical
    - MSO over trees.