"Fully Incremental LCS Computation"

15th International Symposium on Fundamentals on Computing Theory (FCT'05), 17-20 August 2005, Luebeck, Germany

Yusuke Ishida, Shunsuke Inenaga, Masayuki Takeda
Kyushu Univ., Japan
&
Ayumi Shinohara
Tohoku Univ., Japan

Longest Common Subsequence

- A string obtained by removing 0 or more characters from string ***A*** is called a ***subsequence*** of ***A***.
- The longest subsequence that occurs in both strings ***A*** and ***B*** is called the ***longest common subsequence*** (***LCS***) of ***A*** and ***B***.

A: ~~c~~ b ~~a~~ c ~~b~~ ~~a~~ a b a
B: b c ~~d~~ a b a

⇨ LCS(***A***,***B***) = b c a b a

- LCS is a common metric for sequence comparison.

Dynamic Programming

- LCS (and its length) of strings ***A*** and ***B*** can be computed by dynamic programming approach.

$$DP[i,j] = \begin{cases} 0, & \text{if } i=0 \text{ or } j=0 \\ \max\{DP[i-1,j], DP[i,j-1]\}, & \text{if } A[j] \neq B[i] \text{ and } i,j > 0 \\ DP[i-1,j-1]+1, & \text{if } A[j] = B[i] \text{ and } i,j > 0 \end{cases}$$

A

B

		~~c~~	b	~~a~~	c	~~b~~	~~a~~	a	b	a
	0	0	0	0	0	0	0	0	0	0
b	0	0	1	1	1	1	1	1	1	1
c	0	1	1	1	2	2	2	2	2	2
~~d~~	0	1	1	1	2	2	2	2	2	2
a	0	1	1	2	2	2	3	3	3	3
b	0	1	2	2	2	3	3	3	4	4
a	0	1	2	3	3	3	4	4	4	5

***O*(*mn*) time & space**

***n* = |*A*|**
***m* = |*B*|**

LCS(***A***,***B***) = 5

Fully Incremental LCS Problem

- Given LCS(***A***,***B***) and character ***c***, compute LCS(***cA***,***B***), LCS(***Ac***,***B***), LCS(***A***,***cB***) and LCS(***A***,***Bc***).
  - So we are able to e.g. process log files backdating to the past, and compute alignments between suffixes of one and the other.
- Naïve use of ***DP*** table takes $O(mn)$ time for computing LCS(***cA***,***B***) and LCS(***A***,***cB***) from LCS(***A***,***B***).
  - More efficiently!?
- Landau et al. presented an algorithm that computes LCS(***cA***,***B***) in ***O(L) time***, where L = LCS(***A***,***B***).
- This work: ***efficient computation for*** LCS(***A***,***cB***), LCS(***Ac***,***B***) ***and*** LCS(***A***,***Bc***)

Fully Incremental LCS Problem [cont.]

a***B***

		a	b	b
	0	0	0	0
a	0	1	1	1
b	0	1	2	2
a	0	1	2	2

$O(\boldsymbol{n})$

A, ***B***

		a	b	b
	0	0	0	0
b	0	0	1	1
a	0	1	1	1

$O(\boldsymbol{L})$

b***A***

		b	a	b	b
	0	0	0	0	0
b	0	1	1	1	1
a	0	1	2	2	2

$O(\boldsymbol{L})$

Ac

		a	b	b	c
	0	0	0	0	0
b	0	0	1	1	1
a	0	1	1	1	1

$O(\boldsymbol{n})$

Bb

		a	b	b
	0	0	0	0
b	0	0	1	1
a	0	1	1	1
b	0	1	2	2

Fully Incremental LCS Problem [cont.]

Time and Space Comparison (fixed alphabet)

	Naïve DP	Modified algo. of Kim & Park	Our algorithm
LCS(cA,B)	$O(mn)$	$O(m+n)$	$O(L)$
LCS(Ac,B)	$O(m)$	$O(m)$	$O(L)$
LCS(A,cB)	$O(mn)$	$O(m+n)$	$O(n)$
LCS(A,Bc)	$O(n)$	$O(n)$	$O(n)$
Total space	$O(mn)$	$O(mn)$	$O(nL+m)$

L = LCS(A,B) $\leq$ min(m,n)

Our Approach

- The algorithm of Laudau et al. computes LCS($\boldsymbol{cA}$,$\boldsymbol{B}$) in $O(\boldsymbol{L})$ time.
- Their algorithm does not compute the whole DP matrix – it only considers the set P of ***partition points***.

- Based on their algorithm, we compute LCS($\boldsymbol{A}$,$\boldsymbol{cB}$) in $O(\boldsymbol{n})$ time by considering ***partition points*** only.
- Suppose we have computed DP for strings $\boldsymbol{A}$ and $\boldsymbol{B}$. Let us denote by DP^{Bh} the DP matrix that is obtained from DP after we add a new character to the head (left) of $\boldsymbol{B}$.
- Same for P^{Bh} and P.

Match Point & Partition Point

- Pair (i, j) is said to be a ***match point*** if $A[j] = B[i]$.
- Pair (i, j) is said to be a ***partition point*** if $DP[i, j] = DP[i-1, j] + 1$.

A

B

		c	b	a	c	b	a	a	b	a
	0	0	0	0	0	0	0	0	0	0
b	0	0	1	1	1	1	1	1	1	1
c	0	1	1	1	2	2	2	2	2	2
d	0	1	1	1	2	2	2	2	2	2
a	0	1	1	2	2	2	3	3	3	3
b	0	1	2	2	2	3	3	3	4	4
a	0	1	2	3	3	3	4	4	4	5

○ *match point*

□ *partition point*

Match Point & Partition Point [cont.]

- The set of partition points of *DP* is denoted by *P*.
- If (i, j) is a partition point with score v, we write as $P[v, j] = i$.

A

B

		c	b	a	c	b	a	a	b	a
	0	0	0	0	0	0	0	0	0	0
b	0	0	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$
c	0	$\boxed{1}$	1	1	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$
d	0	1	1	1	2	2	2	2	2	2
a	0	1	1	$\boxed{2}$	2	2	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$	$\boxed{3}$
b	0	1	$\boxed{2}$	2	2	$\boxed{3}$	3	3	$\boxed{4}$	$\boxed{4}$
a	0	1	2	$\boxed{3}$	$\boxed{3}$	3	$\boxed{4}$	$\boxed{4}$	4	$\boxed{5}$

$P[2, 3] = 4$

$P[4, 7] = 6$

Computing LCS(*A*,*cB*)

DP — rows: *B*; columns: *A*

		a	a	a	a	b	a	c	b	a
	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	1	1	1
b	0	0	0	0	0	1	1	1	2	2
a	0	1	1	1	1	1	2	2	2	3
b	0	1	1	1	1	2	2	2	3	3
a	0	1	2	2	2	2	3	3	3	4

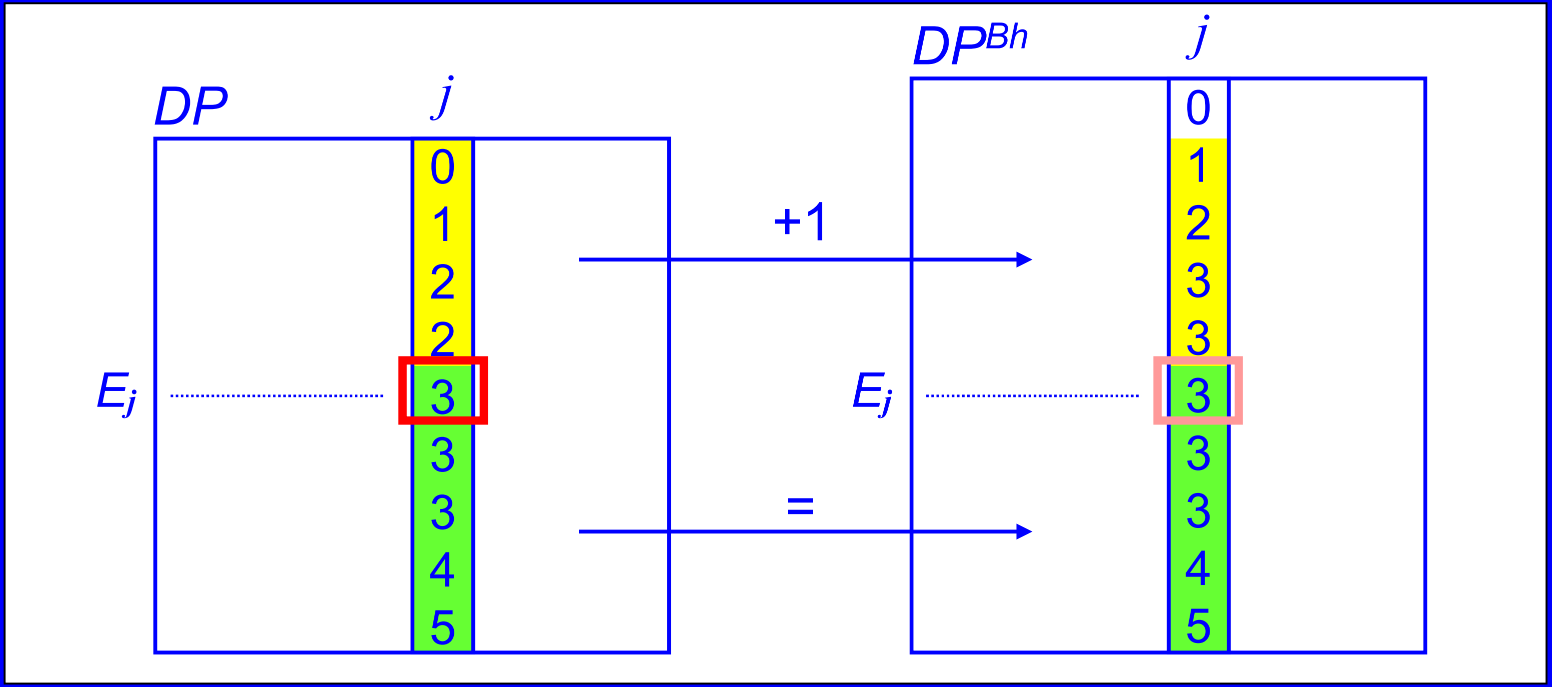
DP^{Bh} — rows: b*B*; columns: *A*

		a	a	a	a	b	a	c	b	a
	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	1	1	1	1	1
c	0	0	0	0	0	1	1	2	2	2
b	0	0	0	0	0	1	1	2	3	3
a	0	1	1	1	1	1	2	2	3	4
b	0	1	1	1	1	2	2	2	3	4
a	0	1	2	2	2	2	3	3	3	4

- There are no changes to the partition points until the 1st occurrence of “b” in *A*.
- All the cells in the 1st row of DP^{Bh} after the first occurrence of “b” get score 1.
- At most one partition point is eliminated at each column.

Eliminated Partition Point

- Lemma 1. For any column j, there exists row index E_j s.t.
  $DP^{Bh}[i, j] = DP[i, j] + 1$ for $i < E_j$,
  $DP^{Bh}[i, j] = DP[i, j]$ for $i \geq E_j$.

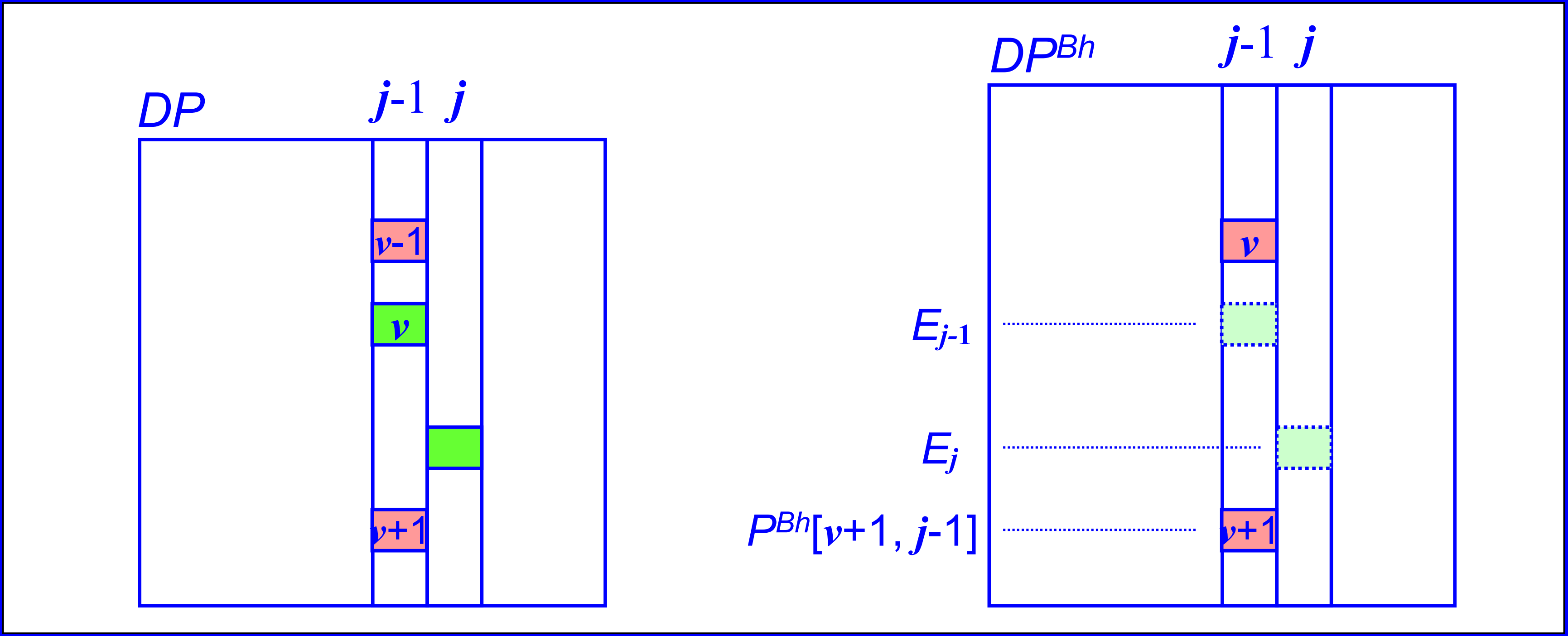


- (E_j, j) is the partition point to be eliminated in DP^{Bh}.

Eliminated Partition Point [cont.]

- Lemma 2. Let $(E_{j-1}, j\text{-}1)$ and (E_j, j) be the partition points eliminated at columns j-1 and j, resp. Let $DP[E_{j-1}, j\text{-}1] = v$. Then,

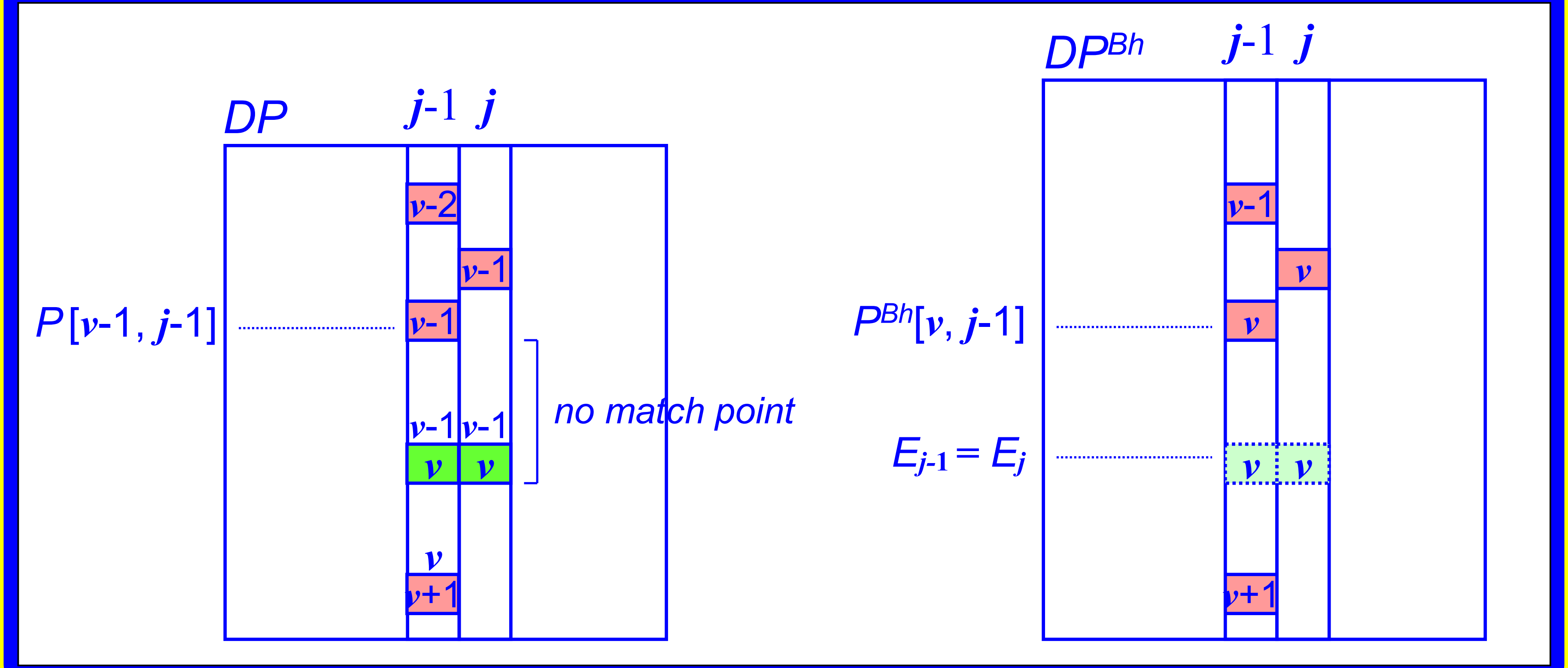
$$E_{j-1} \leq E_j \leq P^{Bh}[v+1, j\text{-}1].$$



Eliminated Partition Point [cont.]

- Lemma 3-1. If there is no match point (x, j) such that $P^{Bh}[v, j\text{-}1] < x \leq E_{j\text{-}1}$,

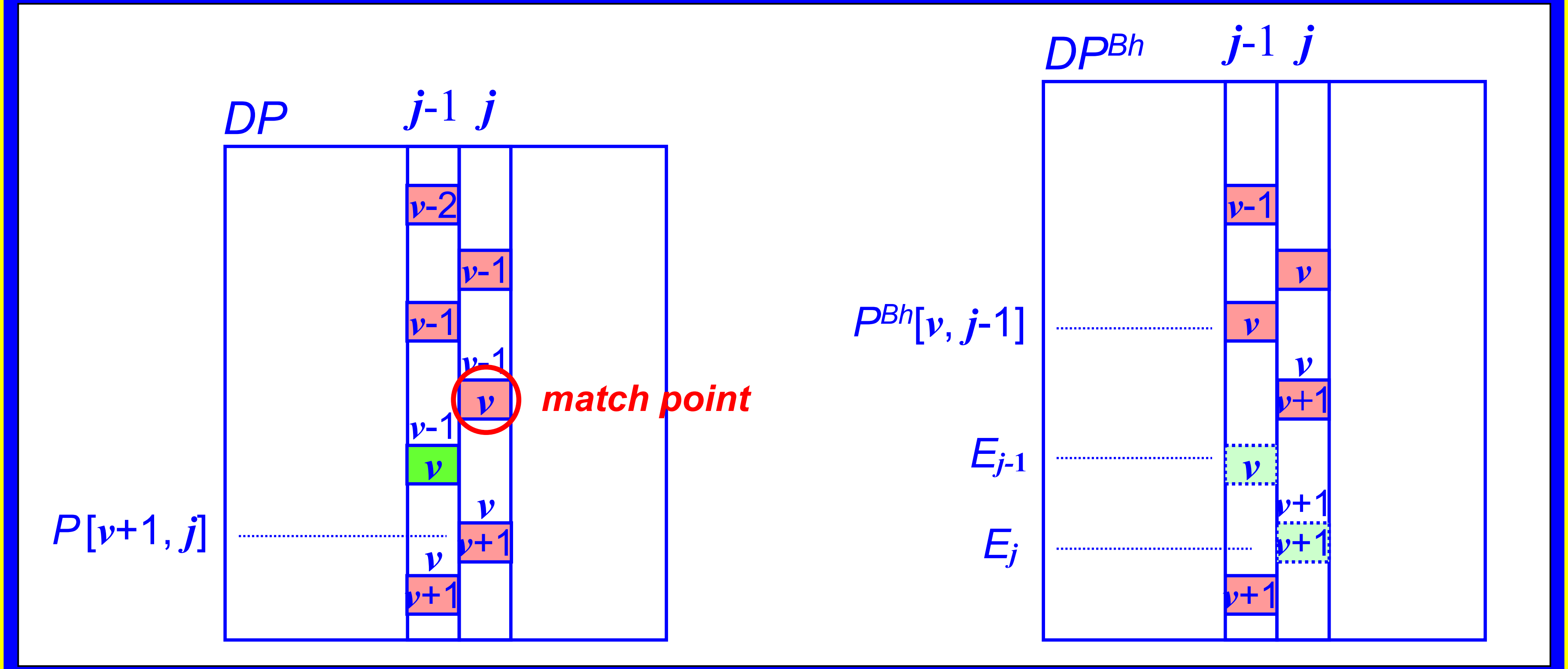
$$E_j = E_{j\text{-}1}$$



Eliminated Partition Point [cont.]

- Lemma 3-2. Otherwise,

$$E_j = P[v+1, j].$$



Eliminated Partition Point [cont.]

- Due to Lemma 3-1 and 3-2, the partition points to be eliminated in DP^{Bh} can be computed by processing the columns of DP from left to right.

- The remaining thing is how to judge whether there exists a partition point (x, j) such that $P^{Bh}[v, j\text{-}1] < x \leq E_{j\text{-}1}$ at each column j. ⇒ ***Next Match Table***

Next Match Table

- *NextMatch*[i, ***c***] returns the first occurrence of "***c***" after position i in string ***B***, if such exists. Otherwise, it returns ***null***.

			Σ			
			a	b	c	d
		0	***null***	1	2	4
	b	1	***null***	3	2	4
B	c	2	***null***	3	***null***	4
	b	3	***null***	***null***	***null***	4
	d	4	***null***	***null***	***null***	***null***

- Using *NextMatch* table we can check $P^{Bh}[v, j\text{-}1] < x \leq E_{j\text{-}1}$ in constant time.

Update Next Match Table

- When we get a new character to the head of ***B***…

Σ

			a	b	c	d
		-1	0	1	2	4
	a	0	*null*	1	2	4
	b	1	*null*	3	2	4
a***B***	c	2	*null*	3	*null*	4
	b	3	*null*	*null*	*null*	4
	d	4	*null*	*null*	*null*	*null*

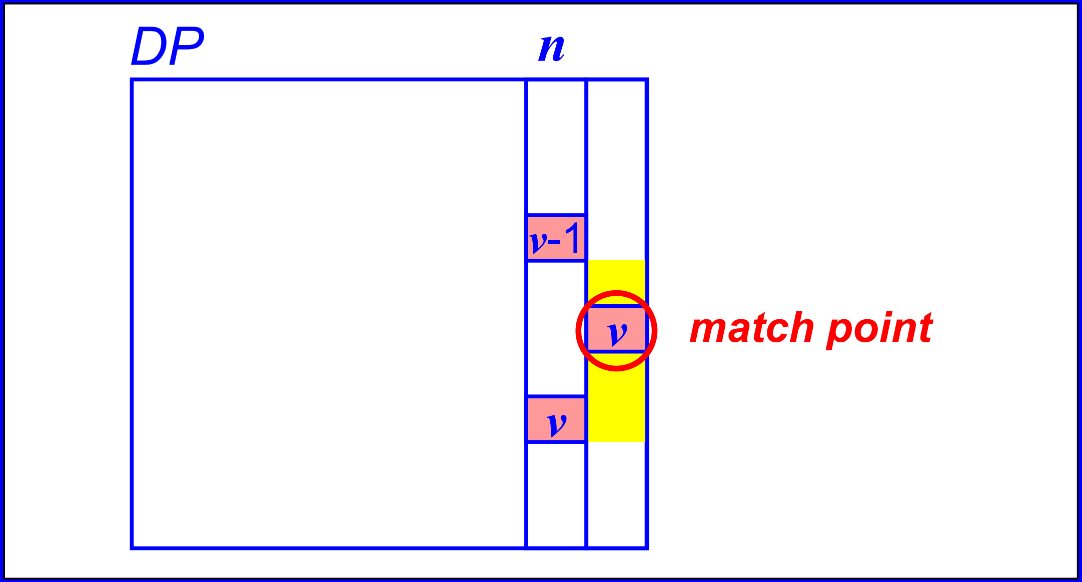
- For fixed alphabet Σ it takes constant time.

Complexity for Computing LCS(*A*, *cB*)

- When updating ***DP*** to $\boldsymbol{DP^{Bh}}$, at most ***n*** partition points are newly added, and at most ***n*** partition points are eliminated.
- Using *NextMatch* Table, each eliminated partition point can be found in $O(1)$ time.
- *NextMatch* table can be updated in $O(1)$ time.
- Conclusion: LCS(***A***, ***cB***) ***can be computed from*** LCS(***A***, ***B***) ***in*** $O(\boldsymbol{n})$ ***time***.

Computing LCS(*Ac*,*B*)

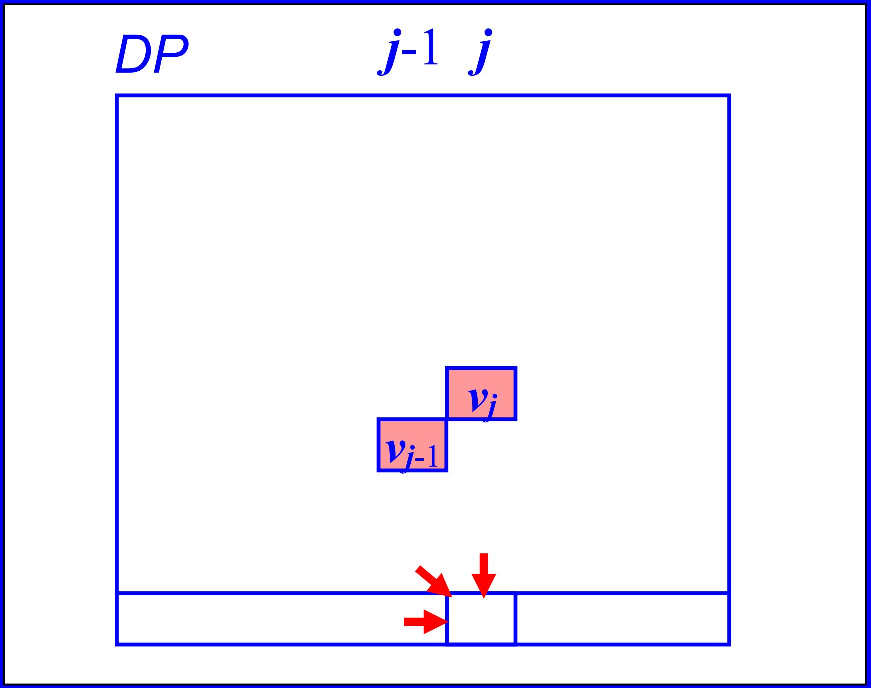
- If there exist match points between *P*[v-1, n] and *P*[v,n], the uppermost match point becomes the new partition point of score v at column $n+1$.



- Since there are *L* intervals to be checked at column $n+1$, it takes *O*(*L*) ***time*** (we can use *NextMatch* table).

Computing LCS(A,Bc)

- New partition points at row m+1 can be computed in the same way as the standard DP approach.



- There are n columns to be checked at row m+1. Therefore $O(n)$ *time*.

Update Next Match Table

- When we get a new character to the tail of ***B***…

Σ

B

		a	b	c	d
	0	*null*	1	2	4
b	1	*null*	3	2	4
c	2	*null*	3	*null*	4
b	3	*null*	*null*	*null*	4
d	4	*null*	*null*	*null*	*null*

Σ

***B*a**

		a	b	c	d
	0	5	1	2	4
b	1	5	3	2	4
c	2	5	3	*null*	4
b	3	5	*null*	*null*	4
d	4	5	*null*	*null*	*null*
a	5	*null*	*null*	*null*	*null*

- There can be at most ***m*** entries to be updated in *NextMatch* table. But the amortized time complexity for each new character is ***constant***.

Conclusion & Future Work

- Given LCS($\boldsymbol{A}$,$\boldsymbol{B}$), the proposed algorithm computes
  - LCS($\boldsymbol{cA}$, $\boldsymbol{B}$) in $O(\boldsymbol{L})$ time,
  - LCS($\boldsymbol{Ac}$, $\boldsymbol{B}$) in $O(\boldsymbol{L})$ time,
  - LCS($\boldsymbol{A}$, $\boldsymbol{cB}$) in $O(\boldsymbol{n})$ time, and
  - LCS($\boldsymbol{A}$, $\boldsymbol{Bc}$) in $O(\boldsymbol{n})$ time,

  including (amortized) constant time update of *NextMatch*.

- Possible future work would be to extend our algorithm to compressed strings - fully incremental LCS computation ***without decompression***. Run-length encoding?