Succincter text indexing with wildcards

Chris Thachuk
University of British Columbia

CPM 2011
June 27, 2011

Problem overview

T: *aa*aca*a*aa*cacc*ac

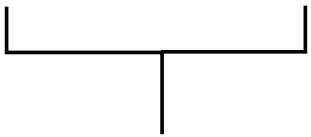
Problem overview

T: *aa*aca*a*aa*cacc*ac

wildcard

Problem overview

T: *aa*aca*a*aa*cacc*ac



text segment

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

Problem overview

T: *aa*aca*a*aa*cacc*ac

P: aca

4 matches

Problem overview

Problem

Create a succinct index for:

- a text **T** of length **n**
- over an alphabet of size σ
- containing **d** wildcard positions

to support efficient pattern matching queries.

Problem overview

Problem

Create a succinct index for:

- a text **T** of length **n**
- over an alphabet of size σ
- containing **d** wildcard positions

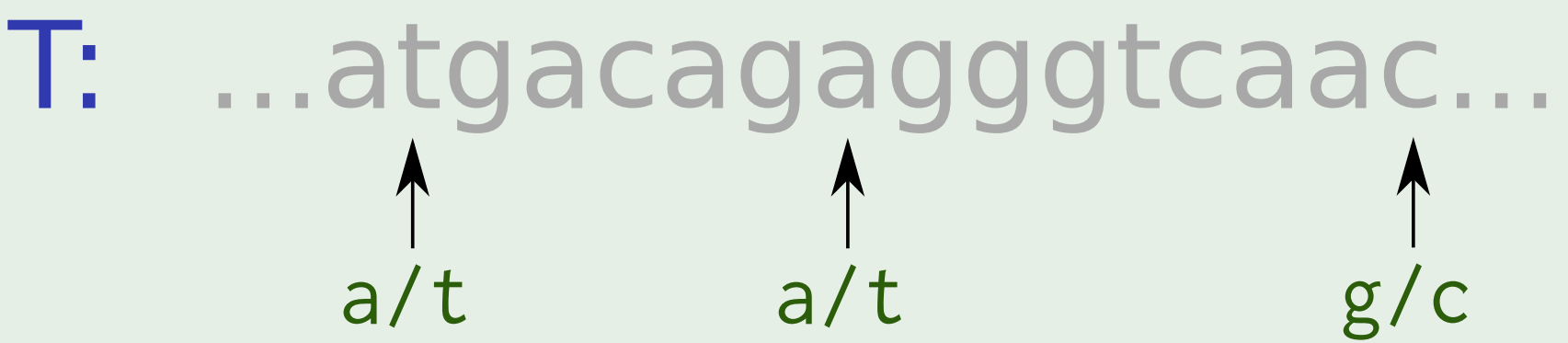
to support efficient pattern matching queries.

Assumption (in presentation)

No adjacent wildcards in **T**.

Motivation

SNPs in the human genome



$|\mathbf{T}| \approx$ 3 billion bases

$\sigma = 4$

$\mathbf{d} \approx$ 3 million positions

$\rightsquigarrow$ also protein collections ($\sigma = 20$)

Our results in context

Non succinct indexes

Cole, Gottlieb, Lewenstein (2004)	$O(n \log^k n)$ words
Lam *et al.*, (2007)	$O(n)$ words

Our results in context

Non succinct indexes

Cole, Gottlieb, Lewenstein (2004)	$O(n \log^k n)$ words
Lam *et al.*, (2007)	$O(n)$ words

Succint indexes

	Index space (bits)	**Query working space (bits)**
Tam *et al.*, 2009	$(3 + o(1))n \log \sigma$	$O(n \log d)$

Our results in context

Non succinct indexes

Cole, Gottlieb, Lewenstein (2004)	$O(n \log^k n)$ words
Lam *et al.*, (2007)	$O(n)$ words

Succint indexes

	Index space (bits)	**Query working space (bits)**
Tam *et al.*, 2009	$(3 + o(1))n \log \sigma$	$O(n \log d)$
	(reduced working space by increasing query time)	
Tam *et al.*, 2009	$(3 + o(1))n \log \sigma$	$O(n \log \sigma)$

Our results in context

Non succinct indexes

Cole, Gottlieb, Lewenstein (2004)	$O(n \log^k n)$ words
Lam *et al.*, (2007)	$O(n)$ words

Succint indexes

	Index space (bits)	**Query working space (bits)**
Tam *et al.*, 2009	$(3 + o(1))n \log \sigma$	$O(n \log d)$
	(reduced working space by increasing query time)	
Tam *et al.*, 2009	$(3 + o(1))n \log \sigma$	$O(n \log \sigma)$
This work	$(\mathbf{2} + o(1))n \log \sigma + \mathbf{O(n)}$	$\mathbf{O(m \log n + md)}$

(Ignoring smaller order terms)

3 Types of matches

Type 1: pattern *contained within* a text segment

T: *aa*aca*a*aa*cacc*ac

P: cac

[Lam *et al.*, 2007, Tam *et al.*, 2009]

3 Types of matches

Type 1: pattern *contained within* a text segment

T: *aa*aca*a*aa*cacc*ac
P: cac

Type 2: pattern *spanning one* wildcard

T: *aa*aca*a*aa*cacc*ac
P: caca

[Lam *et al.*, 2007, Tam *et al.*, 2009]

3 Types of matches

Type 1: pattern *contained within* a text segment

T: *aa*aca*a*aa*cacc*ac
P: cac

Type 2: pattern *spanning one* wildcard

T: *aa*aca*a*aa*cacc*ac
P: caca

Type 3: pattern *contains at least one* text segment

T: *aa*aca*a*aa*cacc*ac
P: acacaca

[Lam *et al.*, 2007, Tam *et al.*, 2009]

Review: SAs and the Burrows Wheeler transform

mississippi

Naïve BWT algorithm

For a string S:

Review: SAs and the Burrows Wheeler transform

mississippi$

Naïve BWT algorithm

For a string S:

1. append \$ to S ($\$ < c, \forall c \in \Sigma$)

Review: SAs and the Burrows Wheeler transform

```
mississippi$
ississippi$m
ssissippi$mi
sissippi$mis
issippi$miss
ssippi$missi
sippi$missis
ippi$mississ
ppi$mississi
pi$mississip
i$mississipp
$mississippi
```

Naïve BWT algorithm

For a string S:

1. append \$ to S ($\$ < c, \forall c \in \Sigma$)
2. list all cyclic rotations of S

Review: SAs and the Burrows Wheeler transform

```
$mississippi
i$mississipp
ippi$mississ
issippi$miss
ississippi$m
mississippi$
pi$mississip
ppi$mississi
sippi$missis
sissippi$mis
ssippi$missi
ssissippi$mi
```

Naïve BWT algorithm

For a string S:

1. append \$ to S ($\$ < c, \forall c \in \Sigma$)
2. list all cyclic rotations of S
3. sort rows of matrix (lexicographically)

Review: SAs and the Burrows Wheeler transform

$	i
i$	p
ippi$	s
issippi$	s
ississippi$	m
mississippi$	
pi$	p
ppi$	i
sippi$	s
sissippi$	s
ssippi$	i
ssissippi$	i

Naïve BWT algorithm

For a string S:

1. append \$ to S ($\$ < c, \forall c \in \Sigma$)
2. list all cyclic rotations of S
3. sort rows of matrix (lexicographically)
4. output last column of matrix

Review: backwards search of pattern P in string S

$	i
i$	p
ippi$	s
issippi$	s
ississippi$	m
mississippi$	$
pi$	p
ppi$	i
sippi$	s
sissippi$	s
ssippi$	i
ssissippi$	i

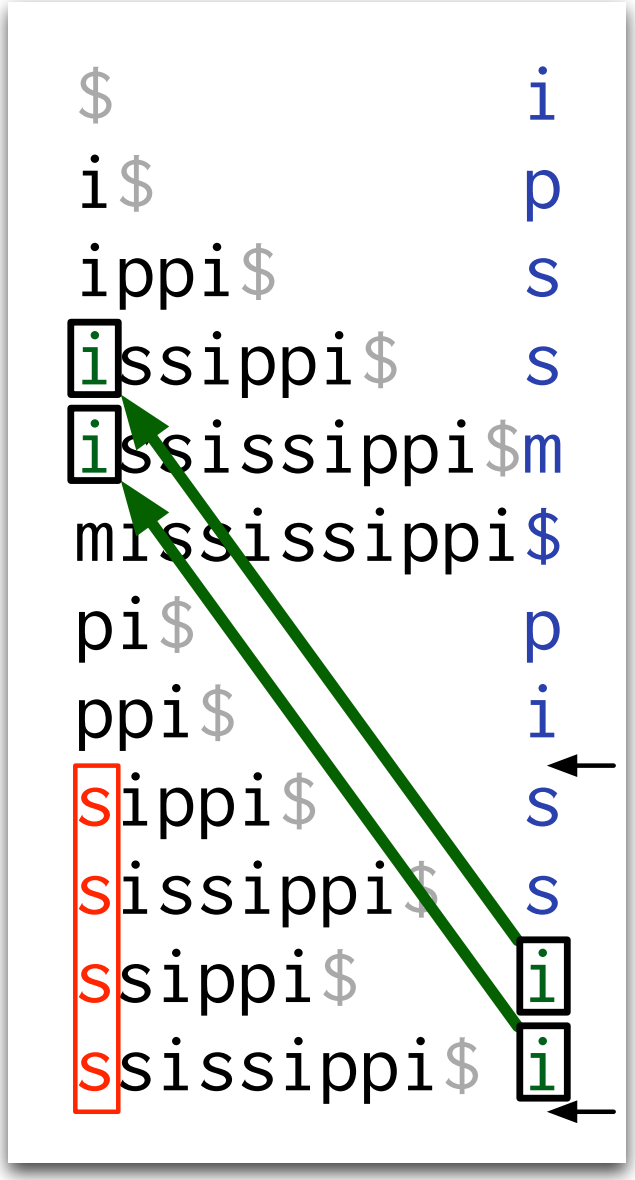
Example

s s i s

$\rightsquigarrow$ SA range: $[8, 11]$

[Ferragina & Manzini 2002]

Review: backwards search of pattern P in string S



Example

s s **i** s

$\rightsquigarrow$ SA range: $[8, 11]$

[Ferragina & Manzini 2002]

Review: backwards search of pattern P in string S

$	i
i$	p
ippi$	s
issippi$	s
ississippi$	m
mississippi$	
pi$	p
ppi$	i
sippi$	s
sissippi$	s
ssippi$	i
ssissippi$	i

Example

s s i s

$\rightsquigarrow$ SA range: $[3, 4]$

[Ferragina & Manzini 2002]

Review: backwards search of pattern P in string S

$	i
i$	p
ippi$	s
issippi$	s
ississippi$	m
mississippi$	
pi$	p
ppi$	i
sippi$	s
sissippi$	s
ssippi$	i
ssissippi$	i

Example

s **s** i s

$\rightsquigarrow$ SA range: $[3, 4]$

[Ferragina & Manzini 2002]

Review: backwards search of pattern P in string S

$	i
i$	p
ippi$	s
issippi$	s
ississippi$	m
mississippi$	$
pi$	p
ppi$	i
sippi$	s
sissippi$	s
ssippi$	i
ssissippi$	i

Example

s s i s

$\rightsquigarrow$ SA range: $[9, 9]$

[Ferragina & Manzini 2002]

Type 1 matching

Build **F**, a compressed suffix array of **T**.

	BWT
*a*aa*cacc*ac	a
*aa*aca*a*aa*cacc*ac	c
*aa*cacc*ac	a
*ac	c
*aca*a*aa*cacc*ac	a
*cacc*ac	a
a*a*aa*cacc*ac	c
a*aa*cacc*ac	*
a*aca*a*aa*cacc*ac	a
a*cacc*ac	a
aa*aca*a*aa*cacc*ac	*
aa*cacc*ac	*
ac	*
aca*a*aa*cacc*ac	*
acc*ac	c
c	a
c*ac	c
ca*a*aa*cacc*ac	a
cacc*ac	*
cc*ac	a

Lemma

All occ_1 Type 1 matches of P can be reported using:

Query time	$O(\|P\| \log \sigma + occ_1 \log^{1+\epsilon} n)$
Index space	$(1 + o(1))n \log \sigma$ *bits*
Work space	$O(\log n)$ *bits*

Matching prefixes of text segments

	BWT
*a*aa*cacc*ac	a
*aa*aca*a*aa*cacc*ac	c
*aa*cacc*ac	a
*ac	c
*aca*a*aa*cacc*ac	a
*cacc*ac	a
a*a*aa*cacc*ac	c
a*aa*cacc*ac	*
a*aca*a*aa*cacc*ac	a
a*cacc*ac	a
aa*aca*a*aa*cacc*ac	*
aa*cacc*ac	*
ac	*
aca*a*aa*cacc*ac	*
acc*ac	c
c	a
c*ac	c
ca*a*aa*cacc*ac	a
cacc*ac	*
cc*ac	a

Matching prefixes of text segments

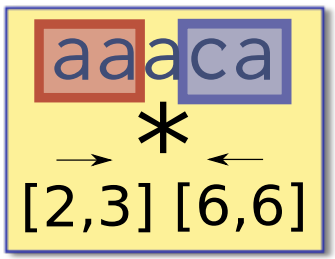
	BWT
*a*aa*cacc*ac	a
*aa*aca*a*aa*cacc*ac	c
*aa*cacc*ac	a
*ac	c
*aca*a*aa*cacc*ac	a
*cacc*ac	a
a*a*aa*cacc*ac	c
a*aa*cacc*ac	*
a*aca*a*aa*cacc*ac	a
a*cacc*ac	a
aa*aca*a*aa*cacc*ac	*
aa*cacc*ac	*
ac	*
aca*a*aa*cacc*ac	*
acc*ac	c
c	a
c*ac	c
ca*a*aa*cacc*ac	a
cacc*ac	*
cc*ac	a

$\text{rank}_*(BWT, i)$ queries determine a lexicographic range of text segments

Type 2 matching

Use **F** and **R** to find candidate text segments

example



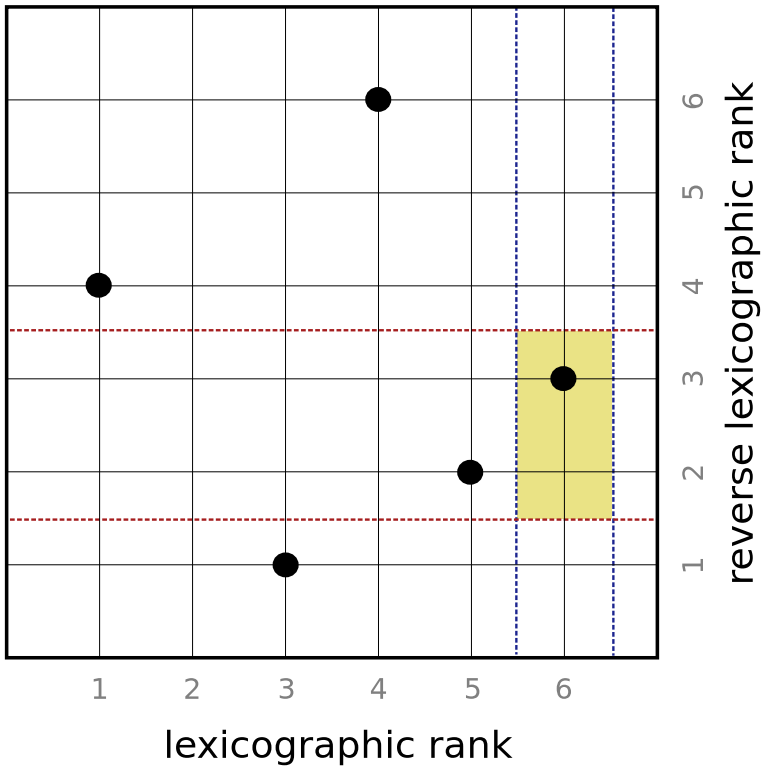
	BWT		BWT^R
*a*aa*cacc*ac	a	*a*aca*aa*	a
*aa*aca*a*aa*cacc*ac	c	*aa*a*aca*aa*	c
*aa*cacc*ac	a	*aa*	a
*ac	c	*aca*aa*	a
*aca*a*aa*cacc*ac	a	*ca*ccac*aa*	a
*cacc*ac	a	*ccac*aa*a*aca*aa*	a
a*a*aa*cacc*ac	c	a*a*aca*aa*	a
a*aa*cacc*ac	*	a*aa*	c
a*aca*a*aa*cacc*ac	a	a*aca*aa*	*
a*cacc*ac	a	a*	a
aa*aca*a*aa*cacc*ac	*	a*ccac*aa*a*aca*aa*	c
aa*cacc*ac	*	aa*a*aca*aa*	* ←
ac	*	aa*	* ←
aca*a*aa*cacc*ac	*	ac*aa*a*aca*aa*	c
acc*ac	c	aca*aa*	*
c	a	c*aa*a*aca*aa*	a
c*ac	c	ca*aa*	a
ca*a*aa*cacc*ac	a ←	ca*ccac*aa*a*aca*aa*	*
cacc*ac	* ←	cac*aa*a*aca*aa*	c
cc*ac	a	ccac*aa*a*aca*aa*	*

Type 2 matching

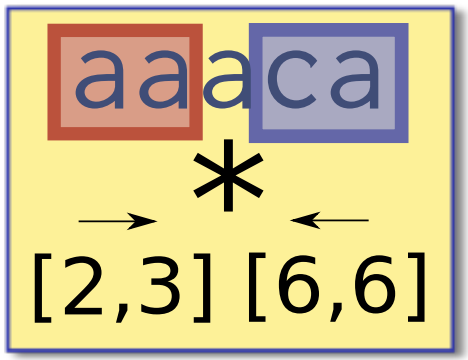
Use **Q** to find compatible candidate [Bose *et al.*, 2009]

*aa*aca*a*aa*cacc*ac

match



example



Type 2 matching

Lemma

All occ_2 Type 2 matches of P can be reported using:

Query time	$O(\|P\| \log \sigma + (\|P\| + occ_2) \frac{\log d}{\log \log d})$
Index space	$(2 + o(1))n \log \sigma$ *bits*
Work space	$O(m \log n)$ *bits*

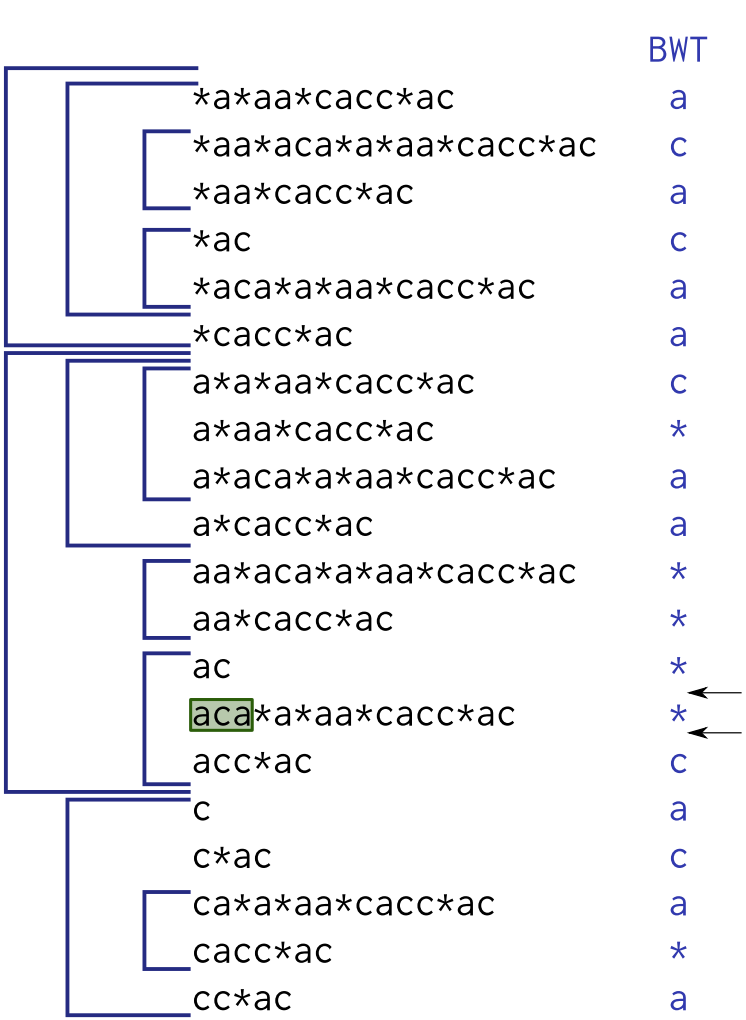
Type 3 matching

T: *aa*aca*a*aa*cacc*ac
P: acacaca

We will enhance **F** to support dictionary queries

- calculate matching statistics
- for each suffix, return text segments contained as a prefix

Review: computing matching statistics



example

~~acacc~~

aca

suffix	length	SA range
c	1	[15,19]
cc	2	[19,19]
acc	3	[14,14]
cacc	4	[18,18]
acacc	3	[13,13]

[Ohlebusch, Gog & Kügel 2010]

Finding dictionary matches

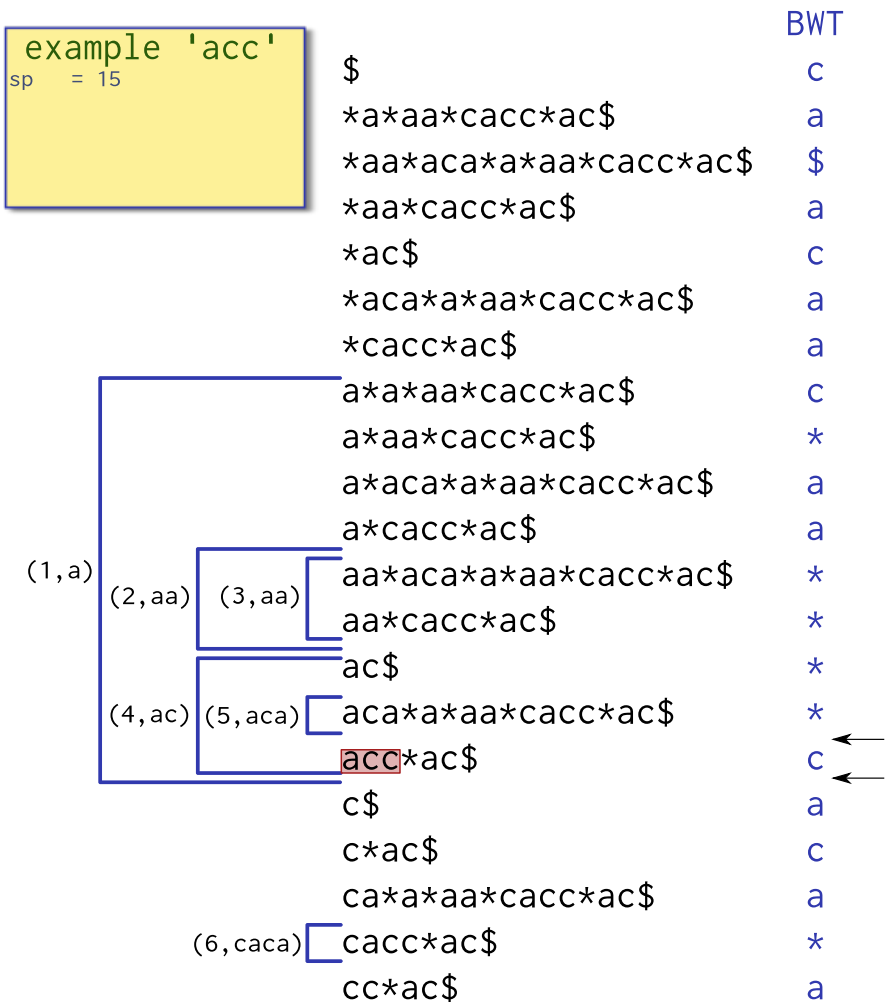
Case 1

example 'acc'
sp = 15

	BWT
$	c
*a*aa*cacc*ac$	a
*aa*aca*a*aa*cacc*ac$	$
*aa*cacc*ac$	a
*ac$	c
*aca*a*aa*cacc*ac$	a
*cacc*ac$	a
a*a*aa*cacc*ac$	c
a*aa*cacc*ac$	*
a*aca*a*aa*cacc*ac$	a
a*cacc*ac$	a
aa*aca*a*aa*cacc*ac$	*
aa*cacc*ac$	*
ac$	*
aca*a*aa*cacc*ac$	* ←
acc*ac$	c ←
c$	a
c*ac$	c
ca*a*aa*cacc*ac$	a
cacc*ac$	*
cc*ac$	a

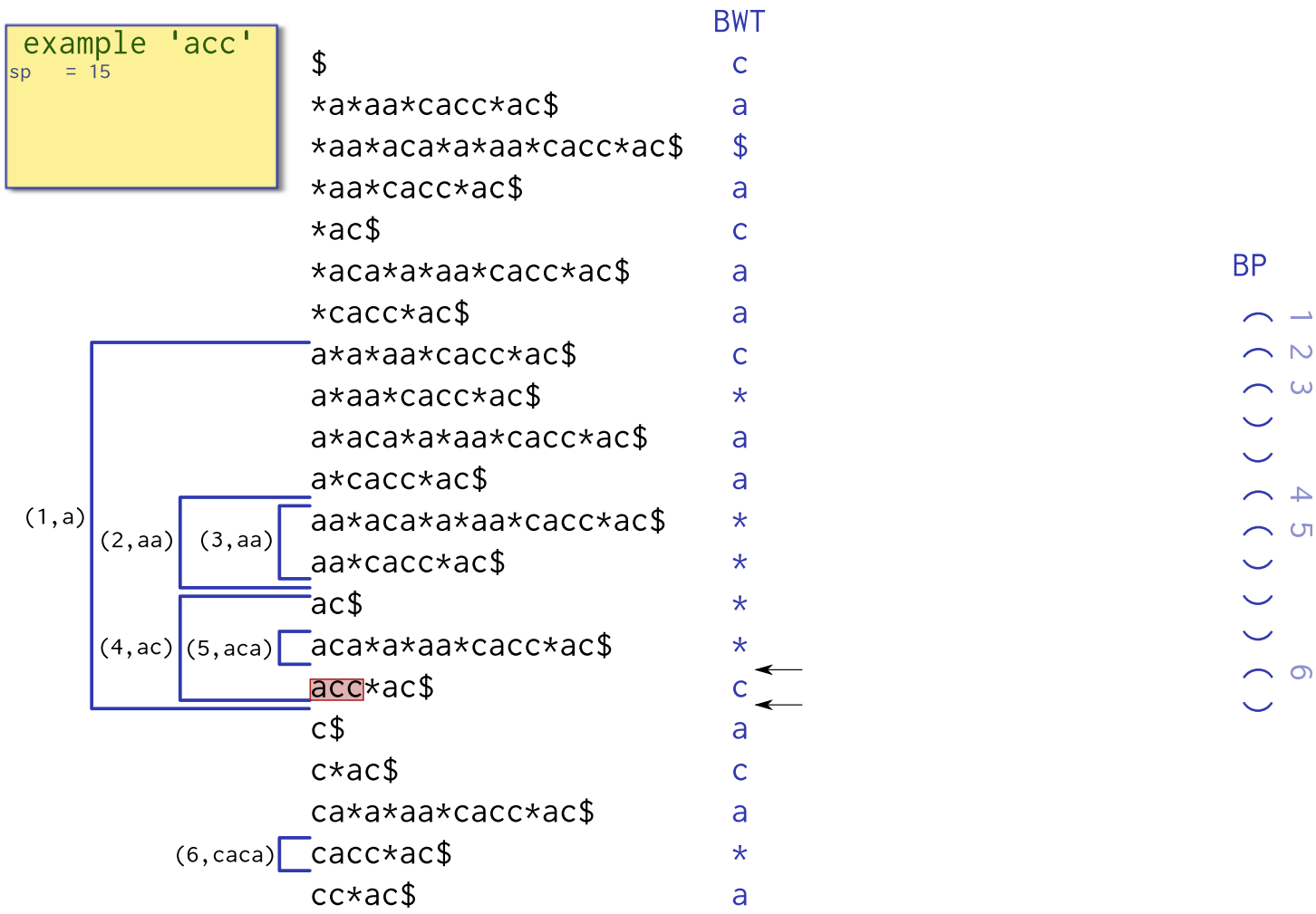
Finding dictionary matches

Case 1



Finding dictionary matches

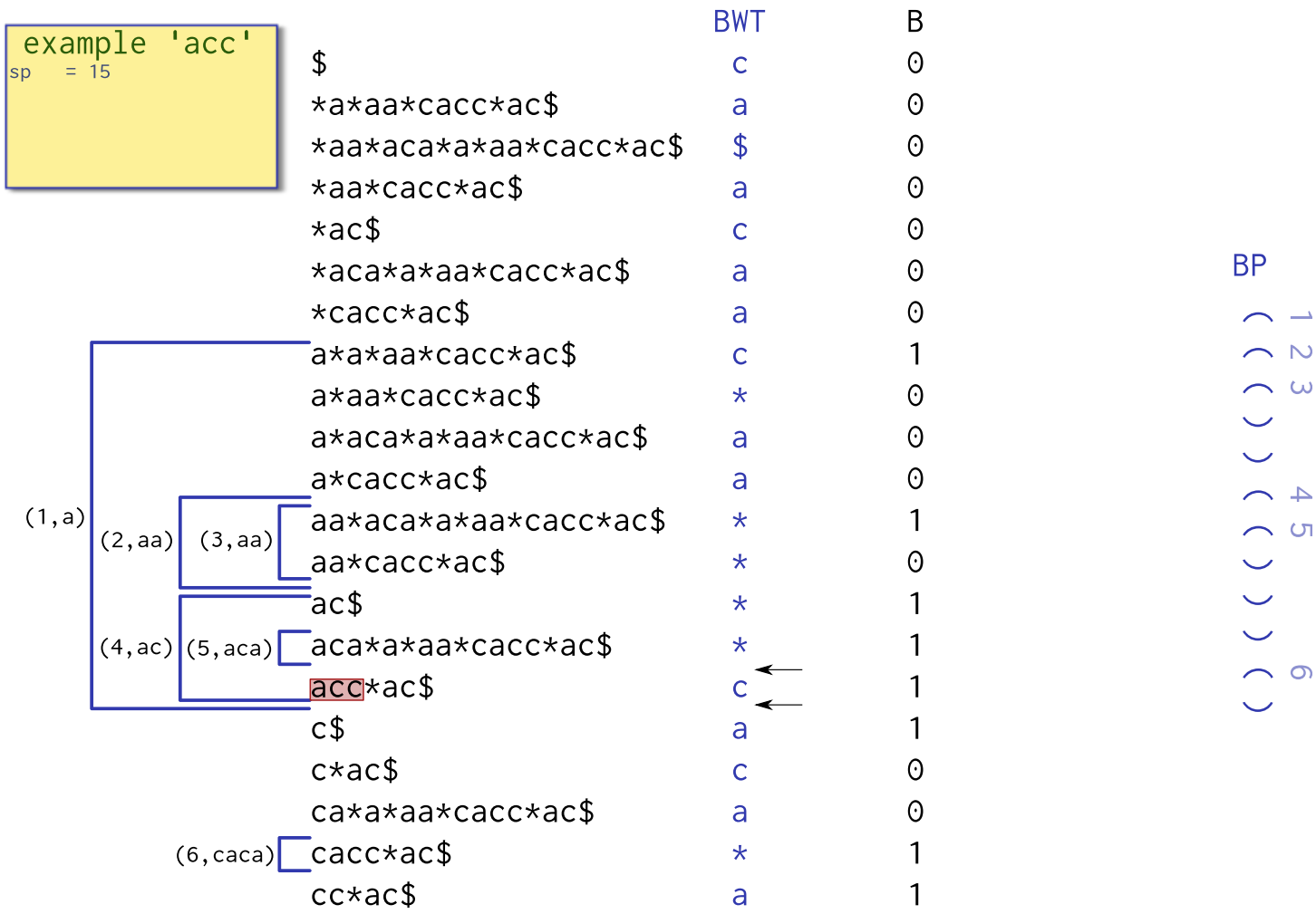
Case 1



[Munro & Ramen 2002]

Finding dictionary matches

Case 1

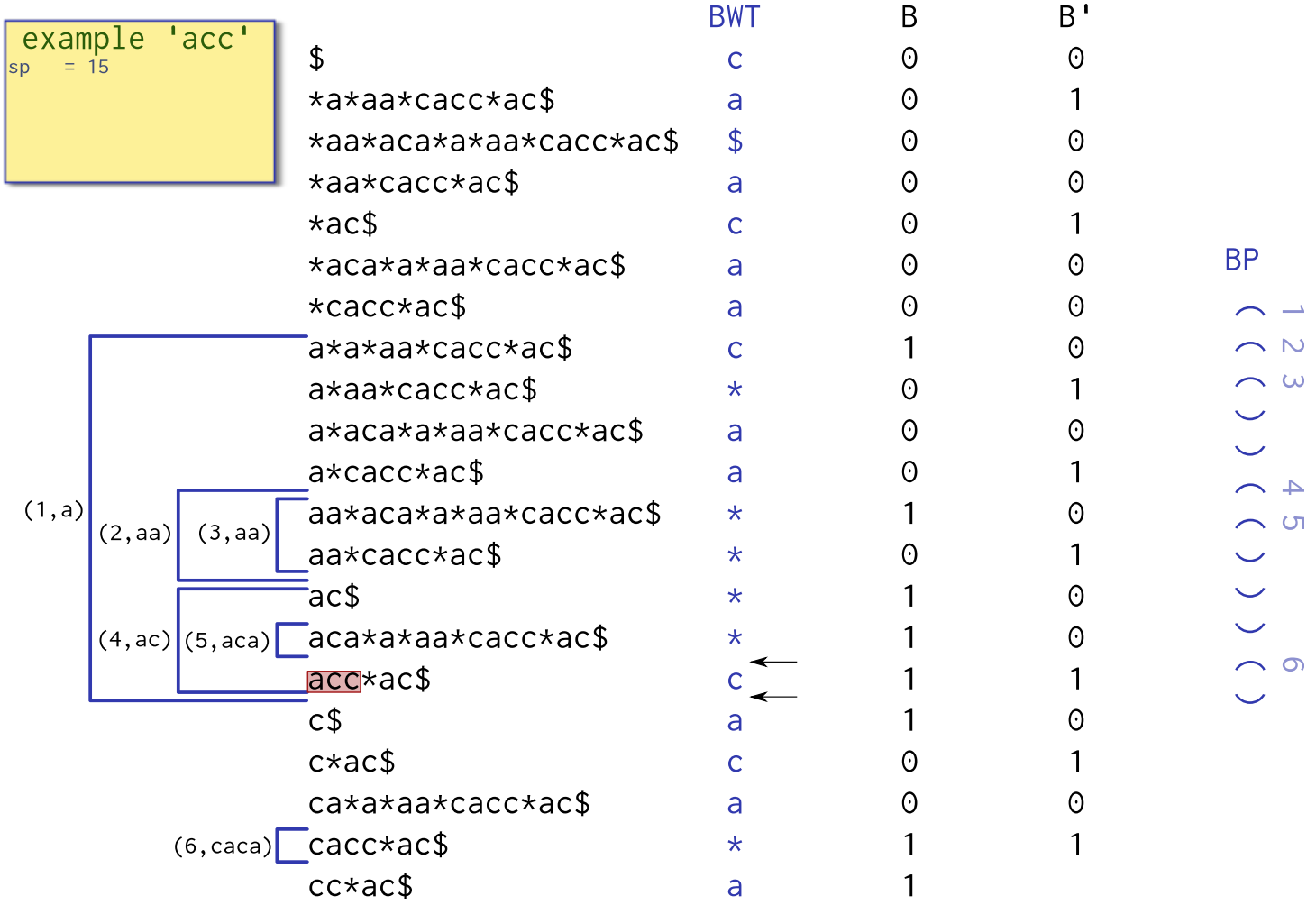


[Ramen *et al.*, 2002]

Finding dictionary matches

Case 1

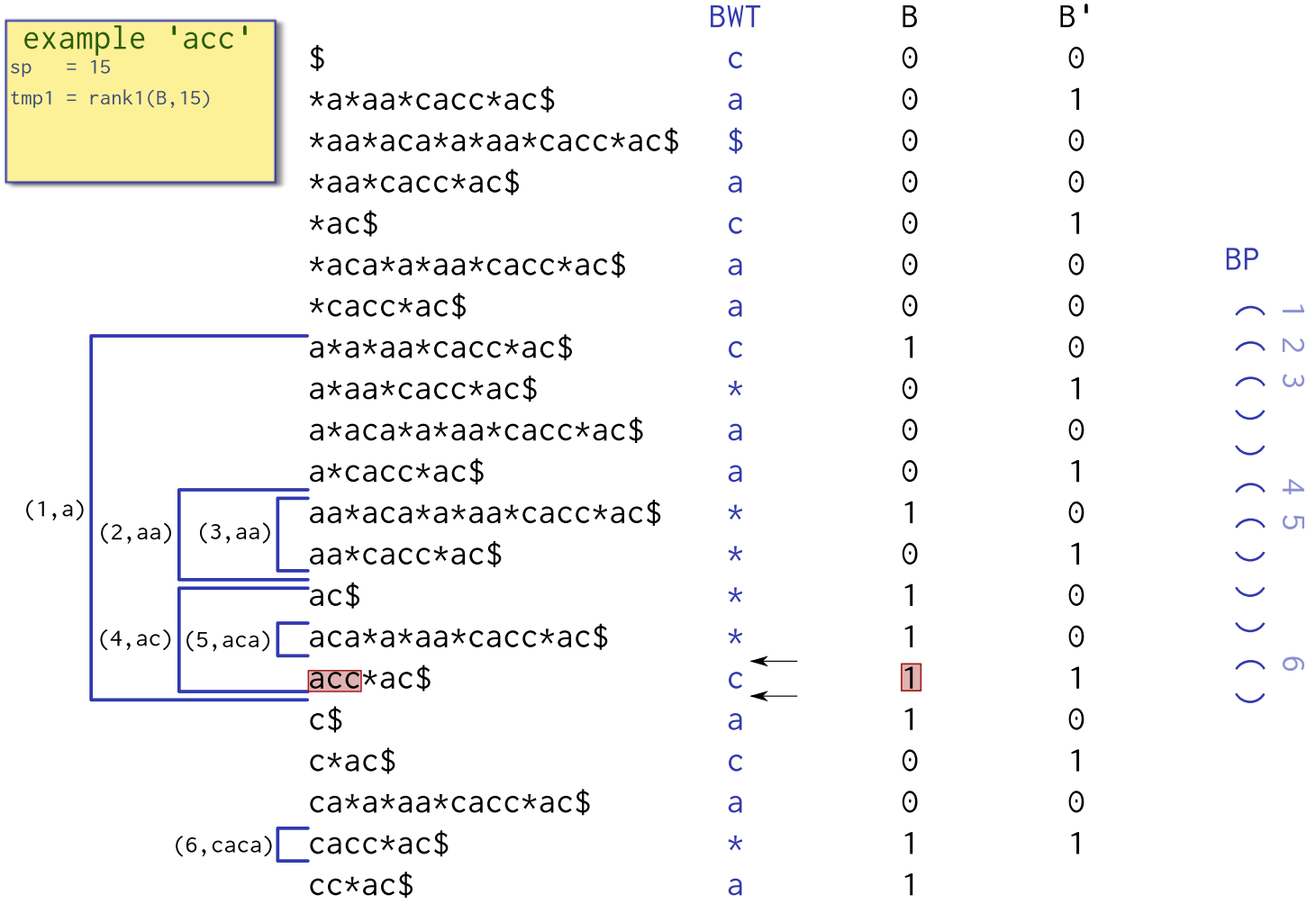
example 'acc'
sp = 15



[Grossi & Vitter 2000]

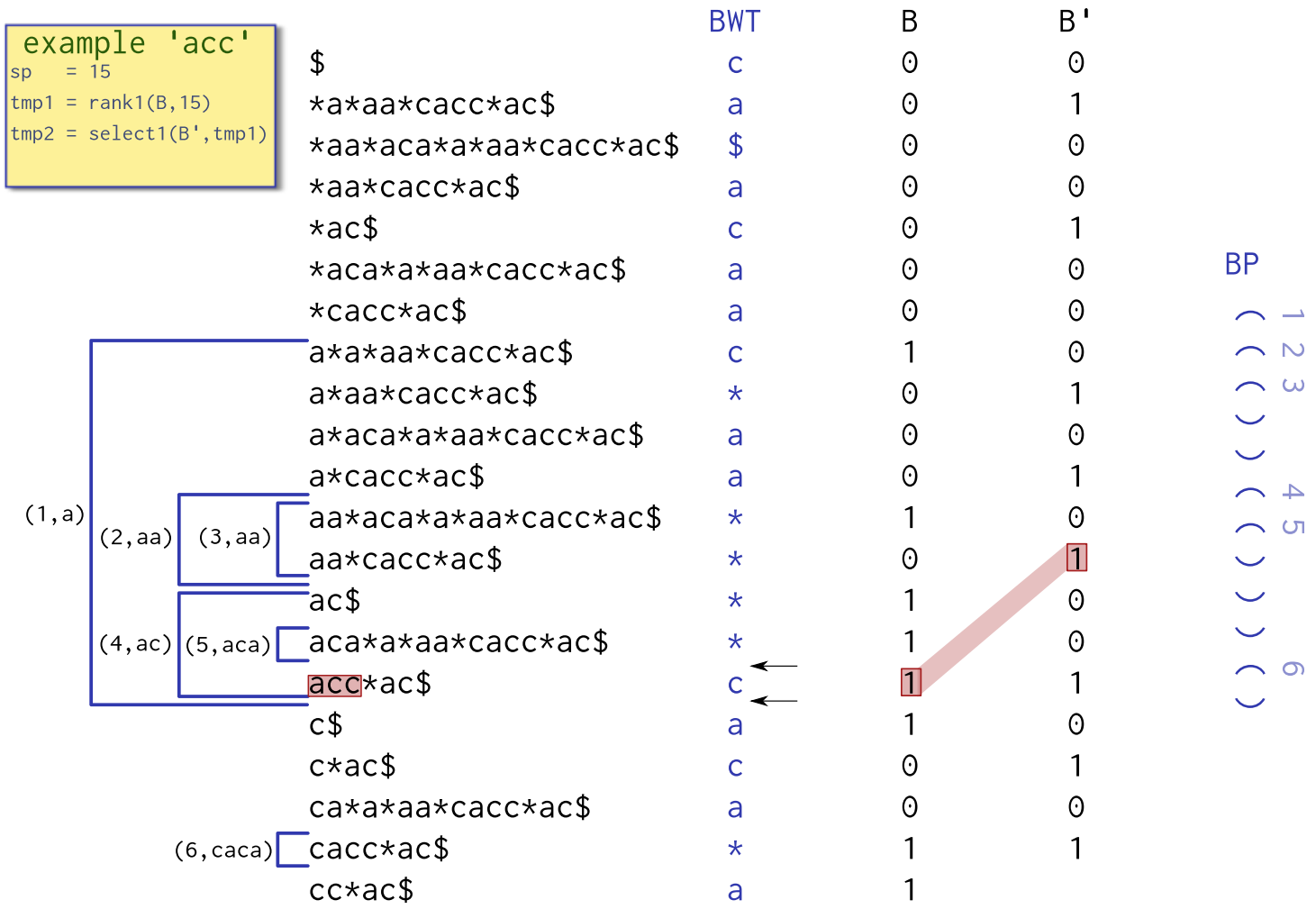
Finding dictionary matches

Case 1



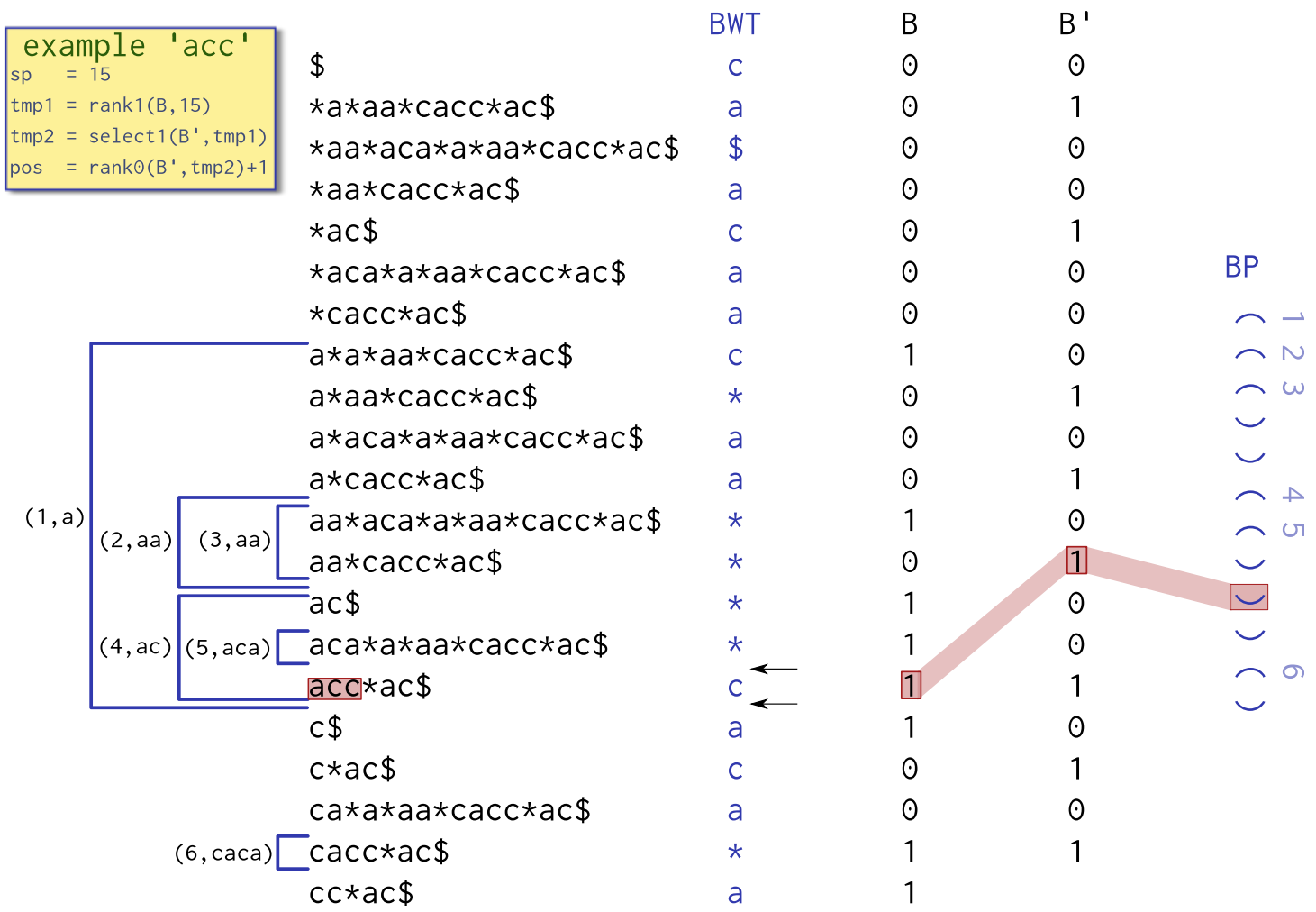
Finding dictionary matches

Case 1



Finding dictionary matches

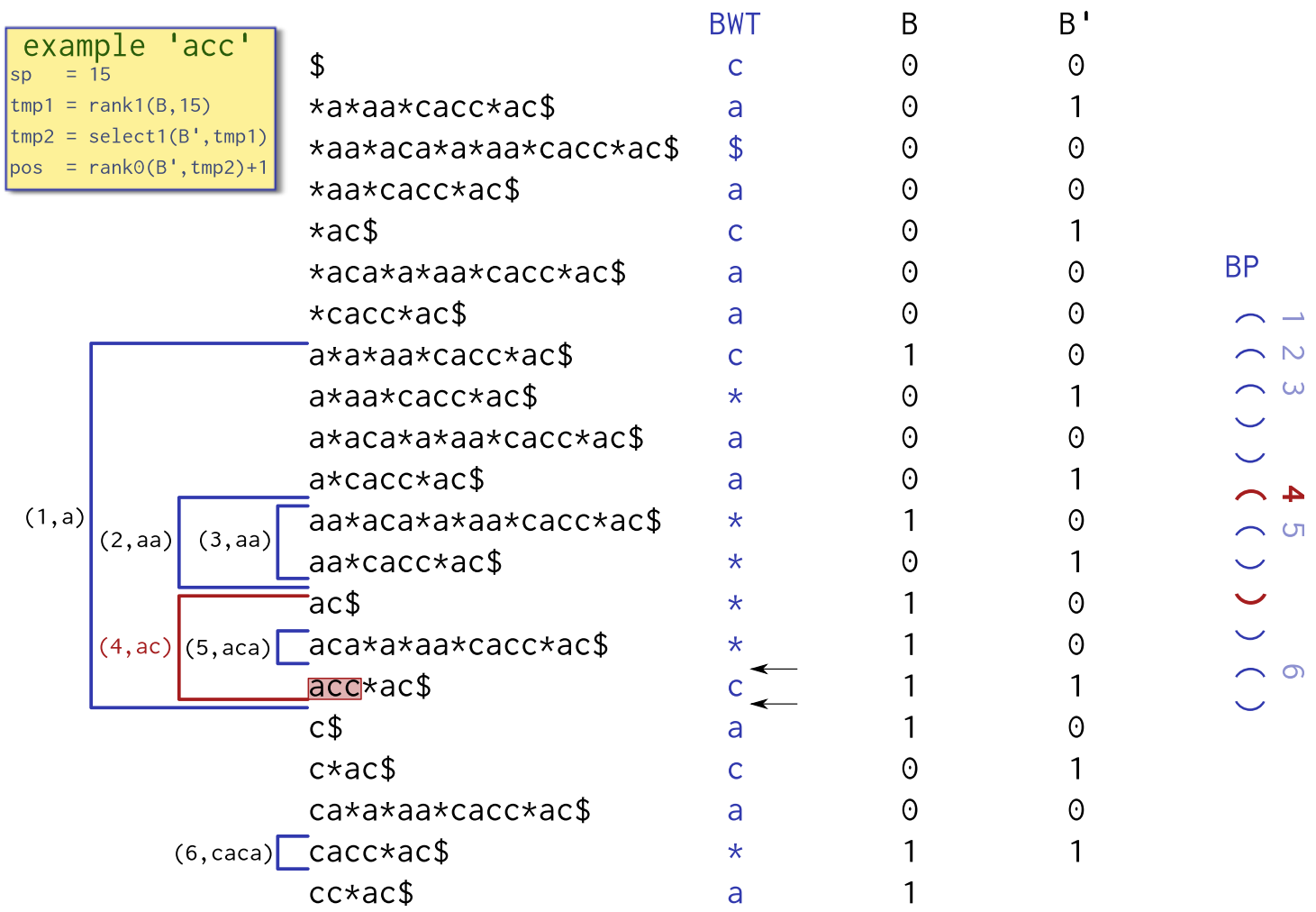
Case 1



if BP[*pos*] $=')'$ **then** find matching $'('$

Finding dictionary matches

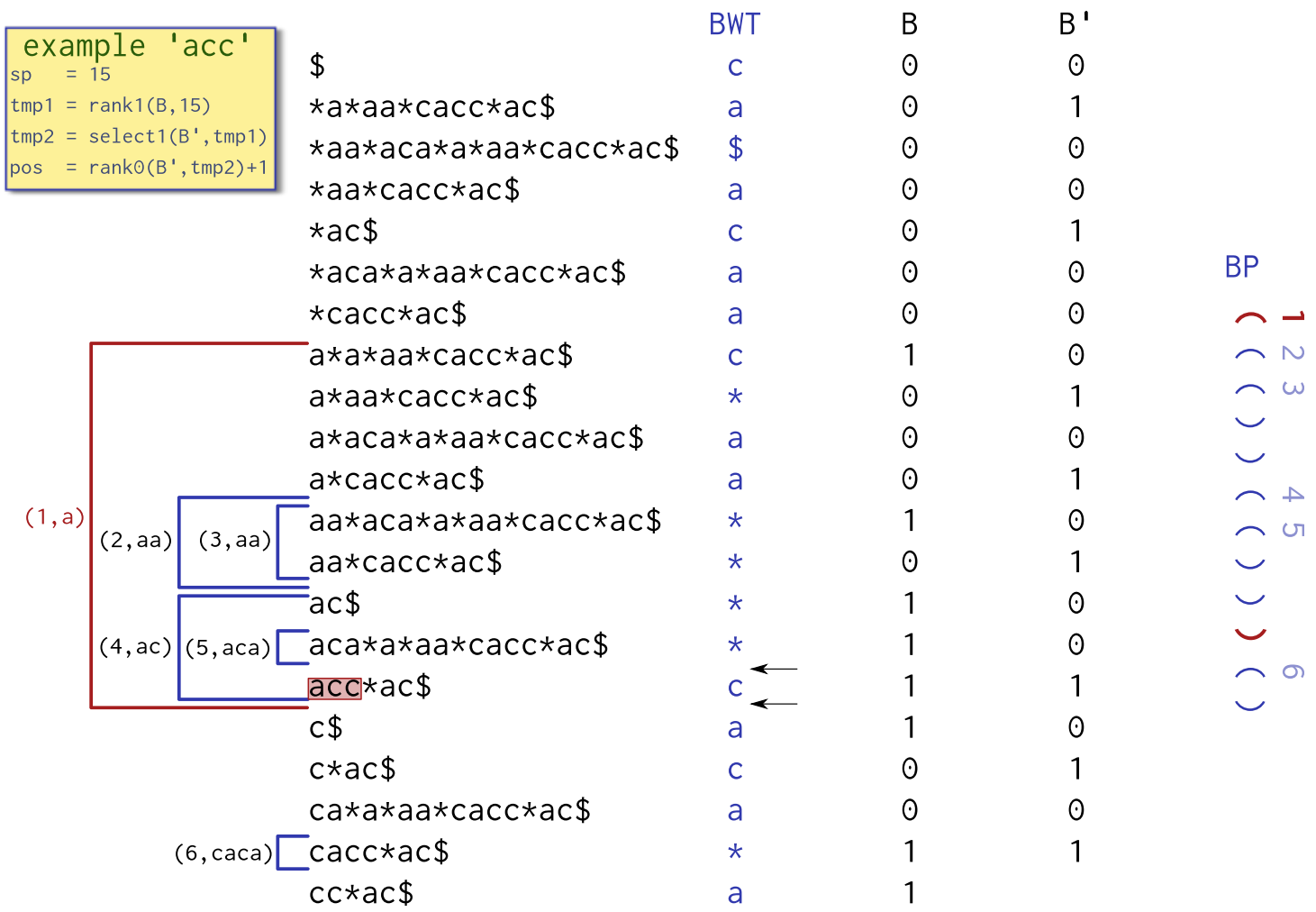
Case 1



if BP[*pos*] $=')'$ **then** find matching $'('$

Finding dictionary matches

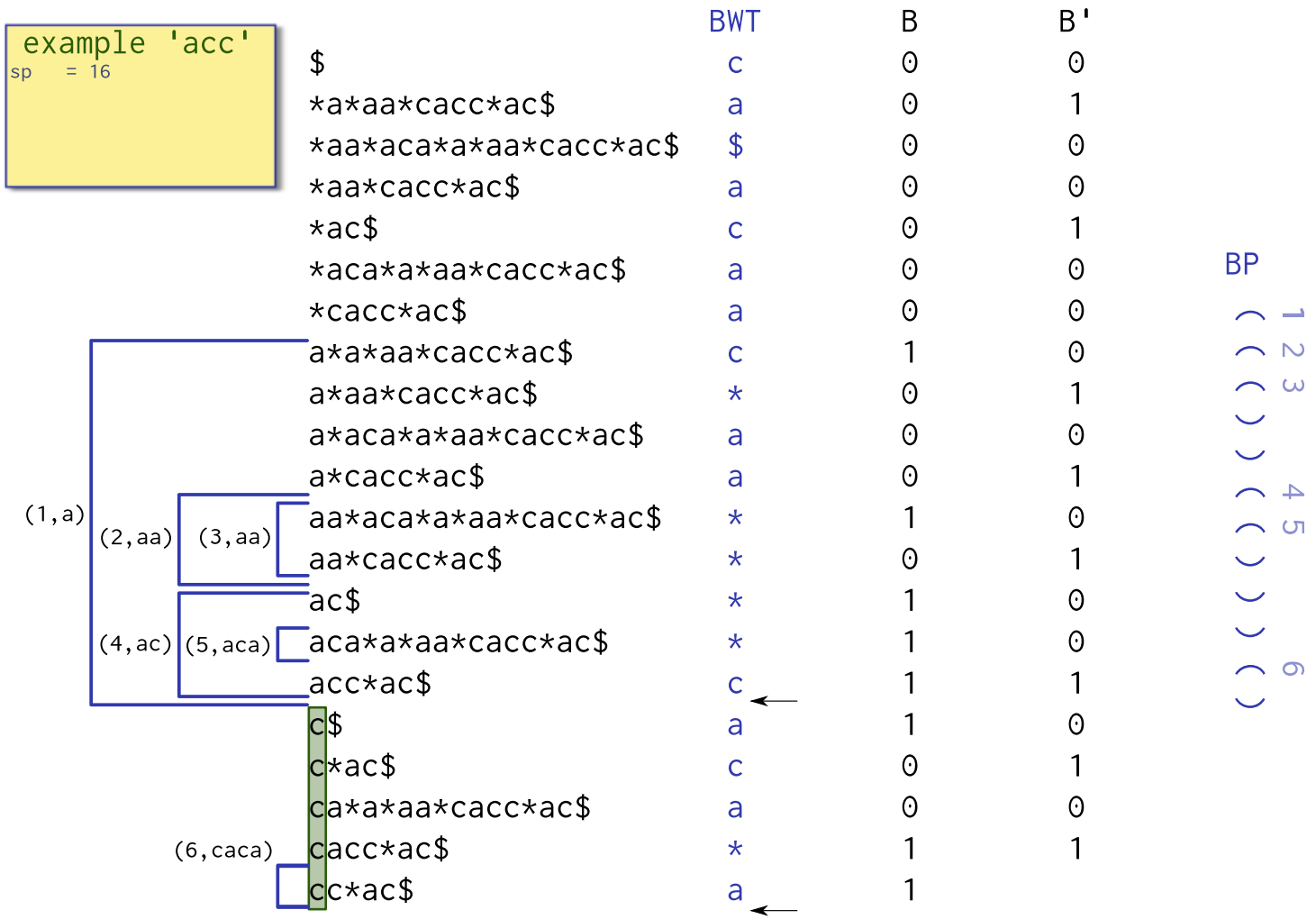
Case 1



if BP[*pos*] $=')'$ **then** find matching $'('$

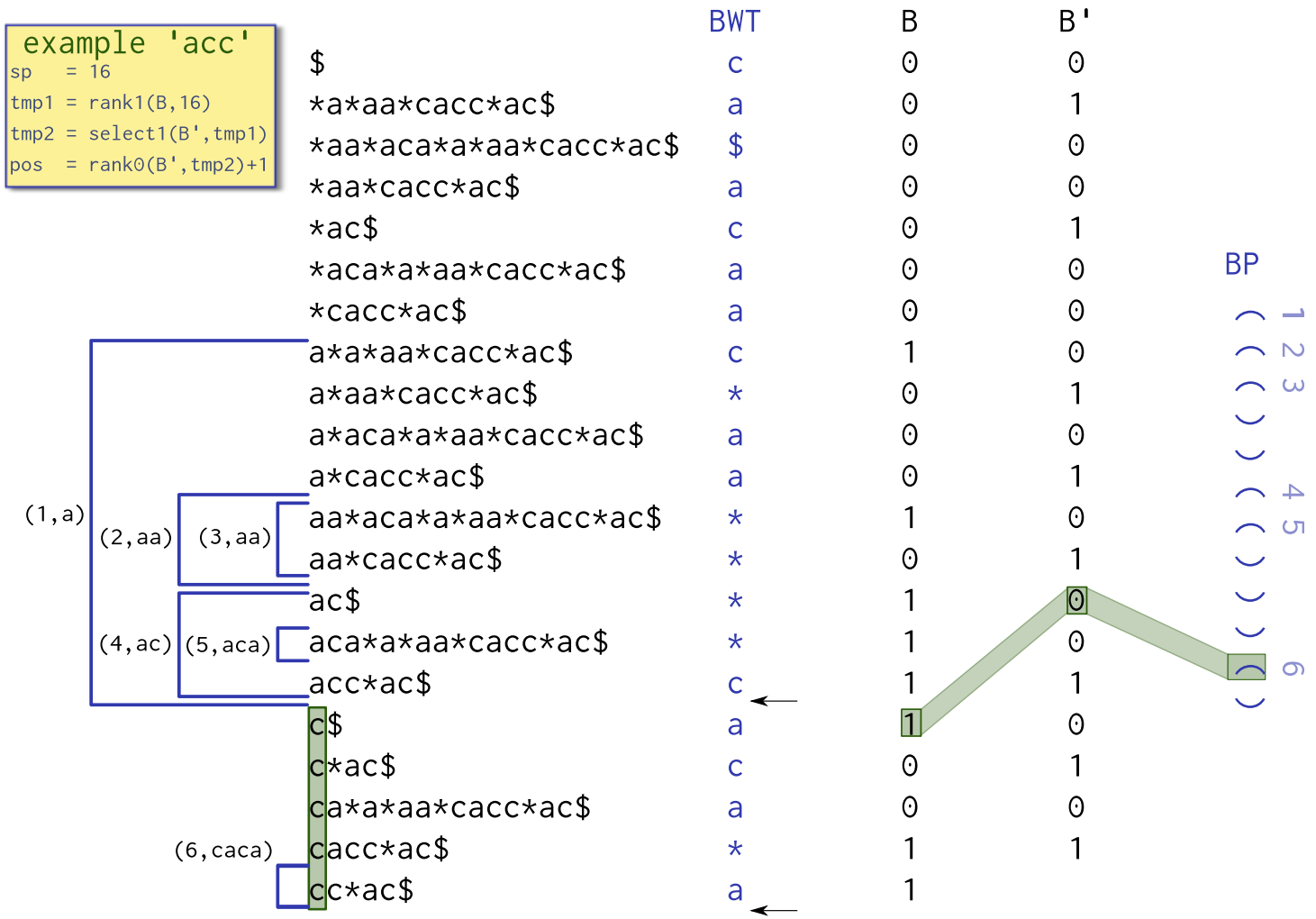
Finding dictionary matches

Case 2



Finding dictionary matches

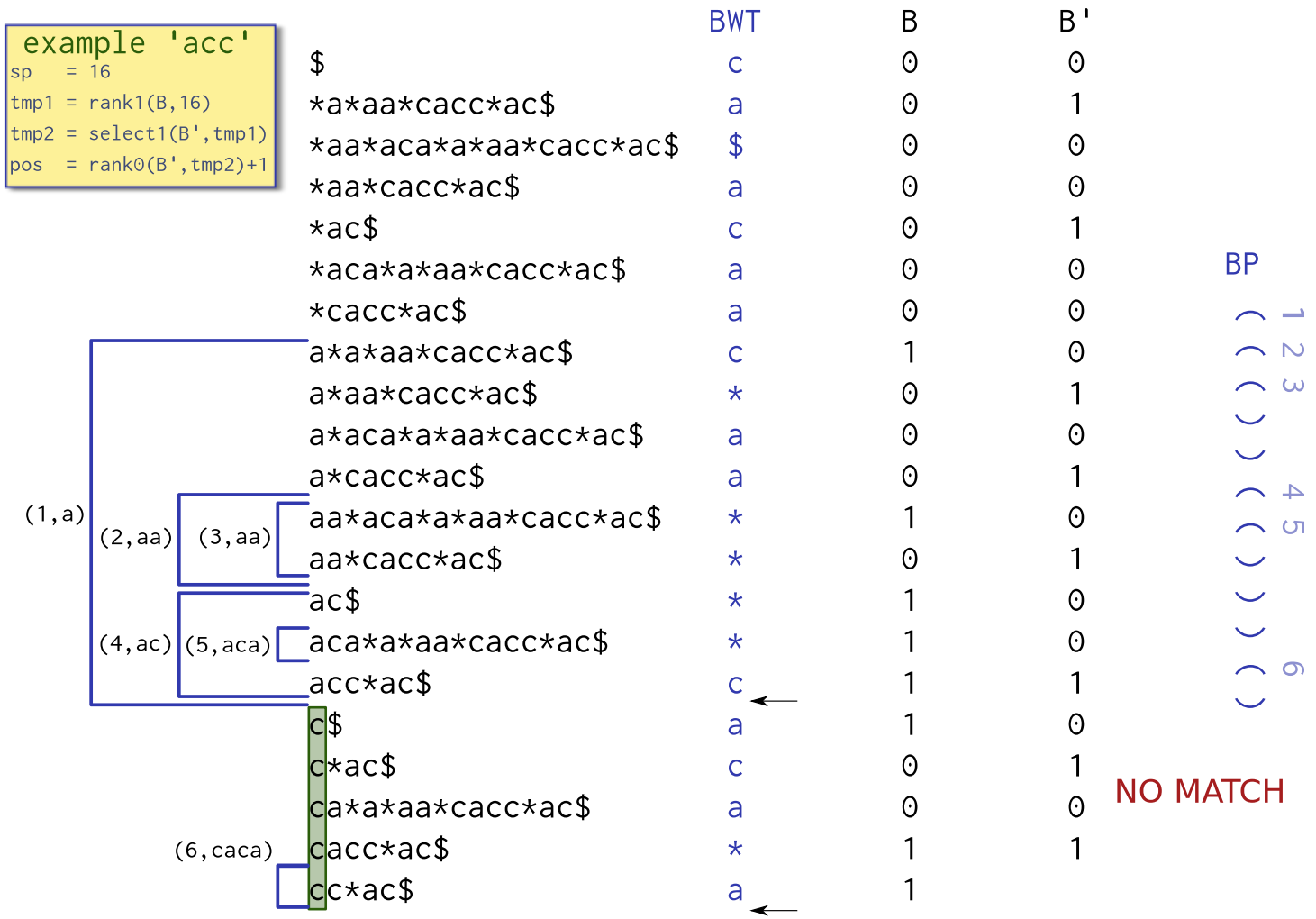
Case 2



if BP$[pos] =$ '(' **then** find parent interval

Finding dictionary matches

Case 2



if BP$[pos] =' ($' **then** find parent interval

Summary of the *full-text* dictionary

Theorem

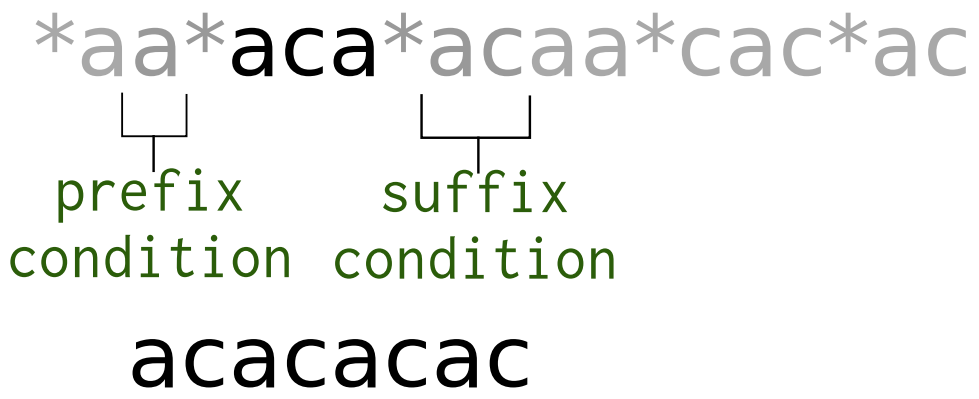
A full-text dictionary using $(1 + o(1))n \log \sigma + O(n) + O(d \log n)$ *bits supports the following operations efficiently:*

- *report/count text segments containing* P
- *report/count text segments prefixed by* P
- *report/count text segments contained in* P

T: *aa*aca*acaa*cac*ac

P: acacacac

Every text segment contained within **P** is a candidate



Every text segment contained within **P** is a candidate

A candidate is valid if it satisfies:

1. suffix condition
2. prefix condition

Algorithm overview

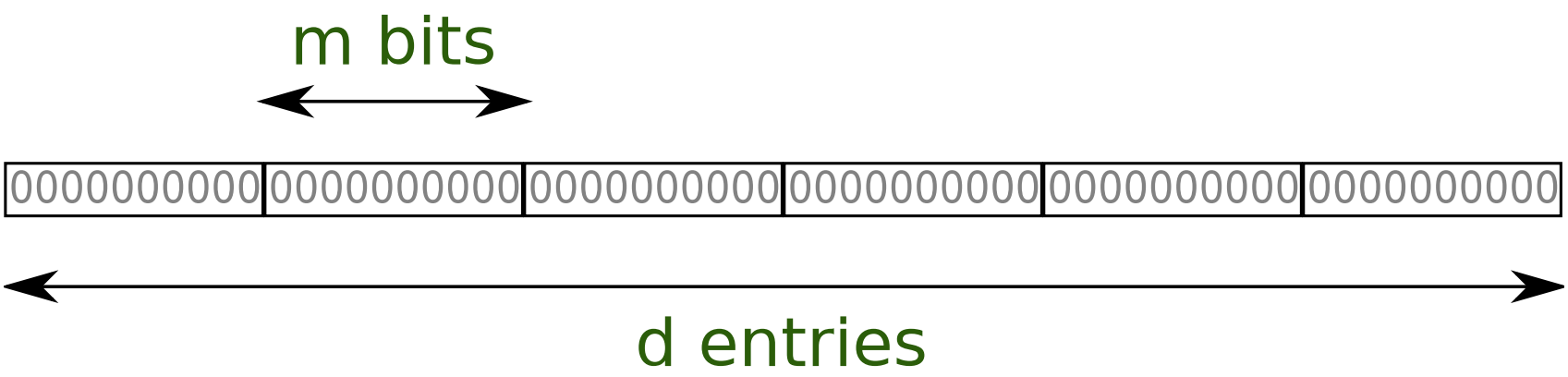
We design a dynamic programming algorithm:

- $|\mathbf{P}|$ phases of algorithm ($i = |\mathbf{P}|, \ldots, 2, 1$)
- phase i considers candidates prefixing $\mathbf{P}[i..|\mathbf{P}|]$

Algorithm overview

We design a dynamic programming algorithm:

- $|\mathbf{P}|$ phases of algorithm ($i = |\mathbf{P}|, \ldots, 2, 1$)
- phase i considers candidates prefixing $\mathbf{P}[i..|\mathbf{P}|]$
- potential matches are tracked in a working array $\mathbf{W}$

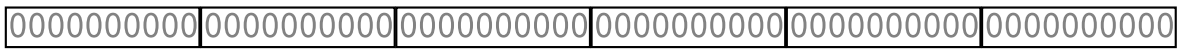


Type 3 matching
Case 1

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 1) P** must end within text segment $j+1$

T: ...*aa*aca*acaa*cac*ac

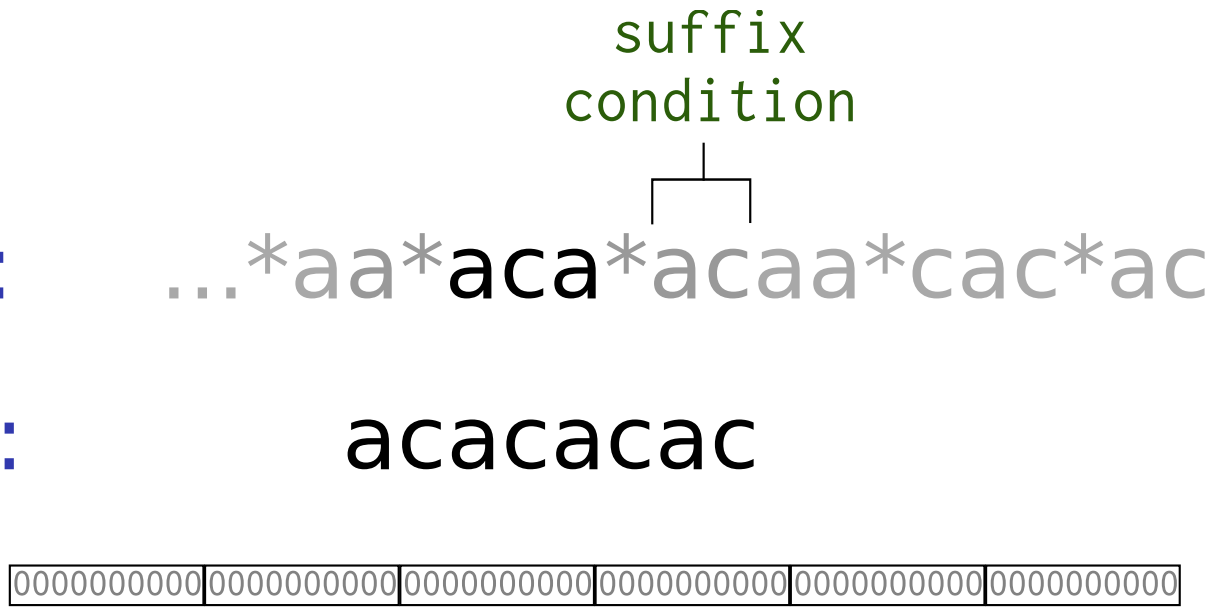
P: acacacac



Type 3 matching
Case 1

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 1) P** must end within text segment $j+1$

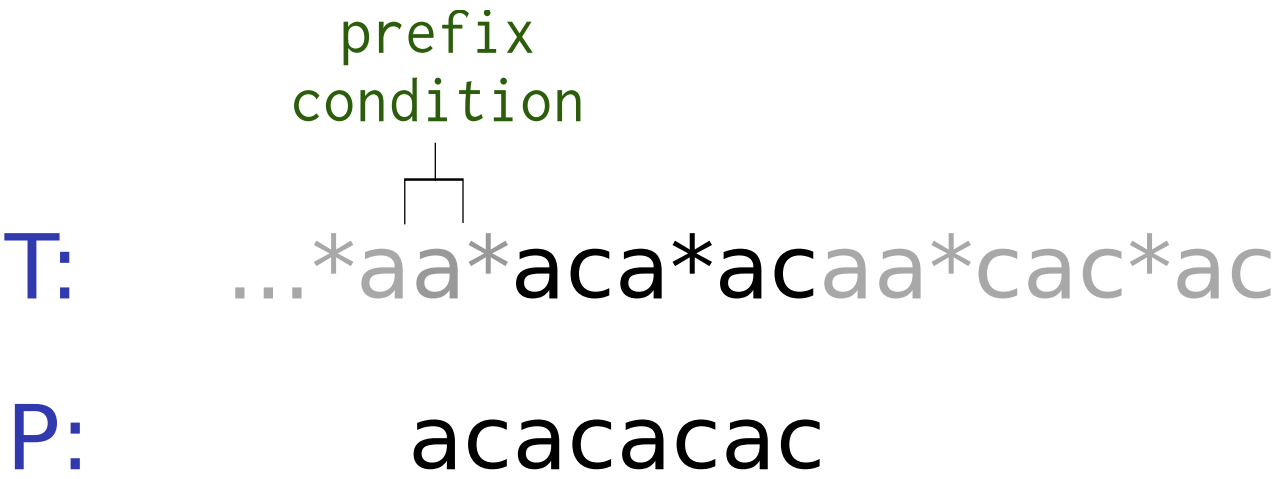
1. verify suffix condition



Type 3 matching
Case 1

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 1) P** must end within text segment $j+1$

1. verify suffix condition
2. **if P** must begin in text segment $j-1$ **then**
   verify prefix condition



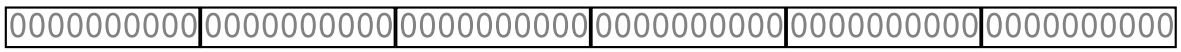
Type 3 matching
Case 1

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 1) P** must end within text segment $j+1$

1. verify suffix condition
2. **if P** must begin in text segment $j-1$ **then**
   verify prefix condition

T: ...*aa*aca*acaa*cac*ac

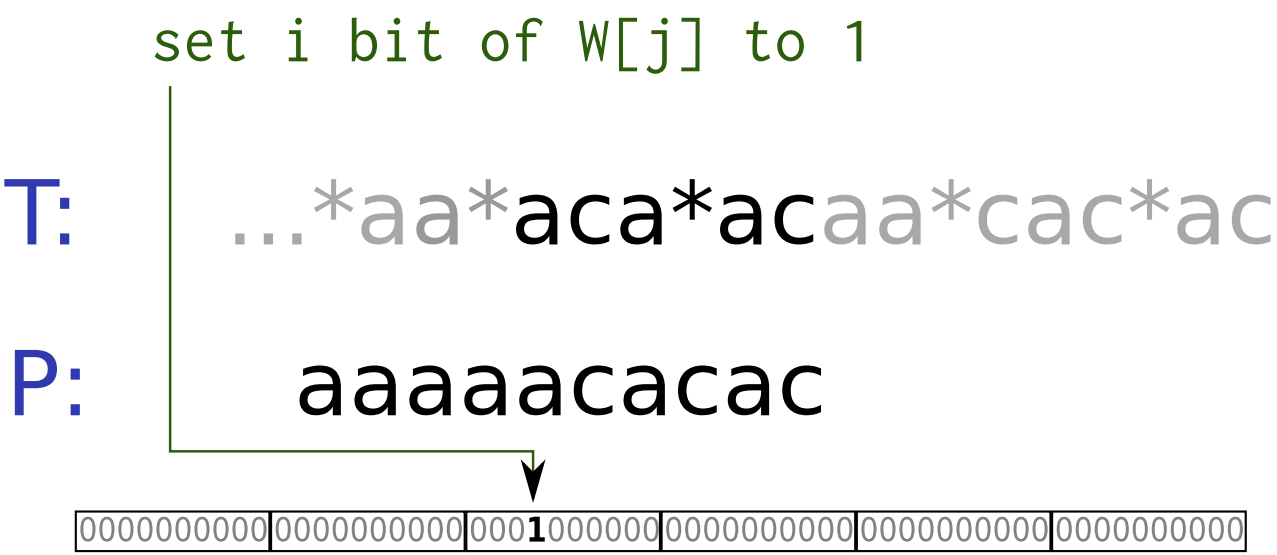
P: acacacac



Type 3 matching
Case 1

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 1) P** must end within text segment $j+1$

1. verify suffix condition
2. **if P** must begin in text segment $j-1$ **then**
   verify prefix condition
3. **else**
   record partial match

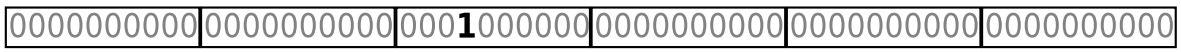


Type 3 matching
Case 2

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 2) P** must contain text segment $j+1$

T: aaa*aa*aca*acaa*cac*ac

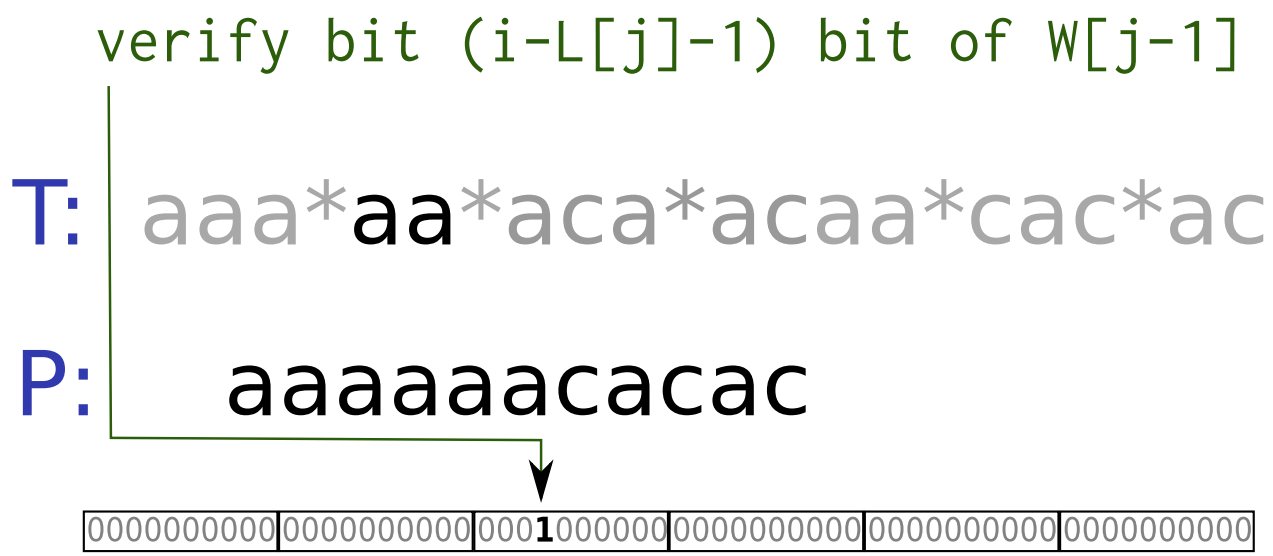
P: aaaaaacacac



Type 3 matching
Case 2

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 2) P** must contain text segment $j+1$

1. verify suffix condition by checking **W**



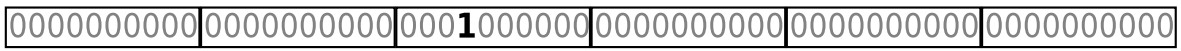
Type 3 matching
Case 2

$\rightsquigarrow$ Text segment j matches $\mathbf{P}[i..|\mathbf{P}|]$
$\rightsquigarrow$ **(case 2) P** must contain text segment $j+1$

1. verify suffix condition by checking **W**
2. **if P** must begin in text segment $j-1$ **then**
   verify prefix condition
3. **else**
   record partial match

T: aaa*aa*aca*acaa*cac*ac

P: aaaaaacacac



Type 3 matching

Lemma

All occ_3 Type 3 matches of P can be reported using:

Query time	$O(\|P\| \log \sigma + \gamma)$
Index space	$(2 + o(1))n \log \sigma$ *bits*
Work space	$O(m \log n + md)$ *bits*

where γ is number of text segments contained in P

Main result

Theorem

Given a text T of length n containing d wildcards, all matches of a pattern P can be reported using:

Query time	$O(\|P\| \log \sigma + m \frac{\log d}{\log \log d} + occ_1 \log n + occ_2 \frac{\log d}{\log \log d} + \gamma)$
Index space	$(2 + o(1))n \log \sigma + O(n) + O(m \log n)$ *bits*
Work space	$O(m \log n + md)$ *bits*

Main result

Theorem

Given a text T of length n containing d wildcards, all matches of a pattern P can be reported using:

Query time	$O(\|P\| \log \sigma + m \frac{\log d}{\log \log d} + occ_1 \log n + occ_2 \frac{\log d}{\log \log d} + \gamma)$
Index space	$(2 + o(1))n \log \sigma + O(n) + O(m \log n)$ *bits*
Work space	$O(m \log n + md)$ *bits*

Main result

Theorem

Given a text T of length n containing d wildcards, all matches of a pattern P can be reported using:

Query time	$O(\|P\| \log \sigma + m \frac{\log d}{\log \log d} + occ_1 \log n + occ_2 \frac{\log d}{\log \log d} + \gamma)$
Index space	$(2 + o(1))n \log \sigma + O(n) + O(m \log n)$ *bits*
Work space	$O(m \log n + md)$ *bits*

Significance of working space reduction

Short read alignment to human genome

$\lvert\mathbf{T}\rvert$	$\approx$ 3 billion bases
$\mathbf{d}$	$\approx$ 3 million bases
$\mathbf{m}$	$\approx 32 - 64$

Working space reduced from GBs [Tam *et al.*, 2009] to 10's of MBs

Questions

Future Work

Can index space be reduced to $(1 + o(1))n \log \sigma$ bits?

$\rightsquigarrow$ **YES, but** with query time:

$O(\max(m^2 \frac{\log \sigma}{\log \log \sigma}, \text{current query time}))$

Thanks to Anne Condon and anonymous reviewers.