CS481: Bioinformatics Algorithms

Can Alkan
EA224
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/

The *Shift-And* Method

- Define M to be a binary n by m matrix such that:

  $M(i,j) = 1$ iff the first i characters of P exactly match the i characters of T ending at character j.

  $M(i,j) = 1$ iff $P[1 .. i] \equiv T[j-i+1 .. j]$

The *Shift-And* Method

- **Let T = california**
- **Let P = for**

M =

	1	2	3	4	5	6	7	8	9	m = 10
1	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0

- **M(*i,j*) = *1* iff the first *i* characters of P exactly match the *i* characters of T ending at character *j.***

How to construct M

- We will construct M column by column.
- Two definitions:
- *Bit-Shift(j-1)* is the vector derived by *shifting* the vector for column *j-1* down by one and setting the first bit to *1*.
- Example:

$$\boldsymbol{BitShift}\left(\begin{pmatrix}0\\1\\1\\0\\1\end{pmatrix}\right)=\begin{pmatrix}1\\0\\1\\1\\0\end{pmatrix}$$

How to construct M

- We define the n-length binary vector U(*x*) for each character *x* in the alphabet. U(*x*) is set to *1* for the positions in P where character *x* appears.
- Example:

P = abaac

$$\boldsymbol{U}(\boldsymbol{a}) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \qquad \boldsymbol{U}(\boldsymbol{b}) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \boldsymbol{U}(\boldsymbol{c}) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

How to construct M

- **Initialize column 0 of M to all zeros**
- **For *j* > 1 column *j* is obtained by**

$$\boldsymbol{M(j) = BitShift(j - \quad) \wedge U(T(j))}$$

An example $j = 1$

1 2 3 4 5 6 7 8 9 10

T = x a b x a b a a c a

1 2 3 4 5

P = a b a a c

	1	2	3	4	5	6	7	8	9	10
1	0									
2	0									
3	0									
4	0									
5	0									

$$U(x) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$BitShift(0)\,\&\,U(T(1)) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example $j = 2$

1 2 3 4 5 6 7 8 9 10

T = x a b x a b a a c a

1 2 3 4 5

P = a b a a c

	1	2	3	4	5	6	7	8	9	10
1	0	1								
2	0	0								
3	0	0								
4	0	0								
5	0	0								

$$U(a) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

$$BitShift(1) \,\&\, U(T(2)) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example $j = 3$

1 2 3 4 5 6 7 8 9 10

T = x a b x a b a a c a

1 2 3 4 5

P = a b a a c

	1	2	3	4	5	6	7	8	9	10
1	0	1	0							
2	0	0	1							
3	0	0	0							
4	0	0	0							
5	0	0	0							

$$U(b) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$BitShift(2) \,\&\, U(T(3)) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example $j = 8$

1 2 3 4 5 6 7 8 9 10

T = x a b x a b a a c a

1 2 3 4 5

P = a b a a c

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	1	0	1	1		
2	0	0	1	0	0	1	0	0		
3	0	0	0	0	0	0	1	0		
4	0	0	0	0	0	0	0	1		
5	0	0	0	0	0	0	0	0		

$$U(\boldsymbol{a}) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

$$\boldsymbol{BitShift}(7)\,\&\,\boldsymbol{U}(\boldsymbol{T}(8)) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \& \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Correctness

- For $i > 1$, Entry $M(i,j) = 1$ iff

1) The first $i-1$ characters of P match the $i-1$characters of T ending at character $j-1$.
2) Character $P(i) \equiv T(j)$.

- 1) is true when $M(i-1,j-1) = 1$.
- 2) is true when the i'th bit of $U(T(j)) = 1$.

- The algorithm computes the ***and*** of these two bits.

Correctness

1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
a b a a c

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	1	0	1	1	0	1
2	0	0	1	0	0	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0	1	0

- M(4,8) = *1*, this is because a b a a is a prefix of P of length 4 that ends at position 8 in T.
- Condition 1) – We had a b a as a prefix of length 3 that ended at position 7 in T ↔ M(3,7) = 1.
- Condition 2) – The fourth bit of P is the eighth bit of T ↔ The fourth bit of U(T(8)) = 1.

How much did we pay?

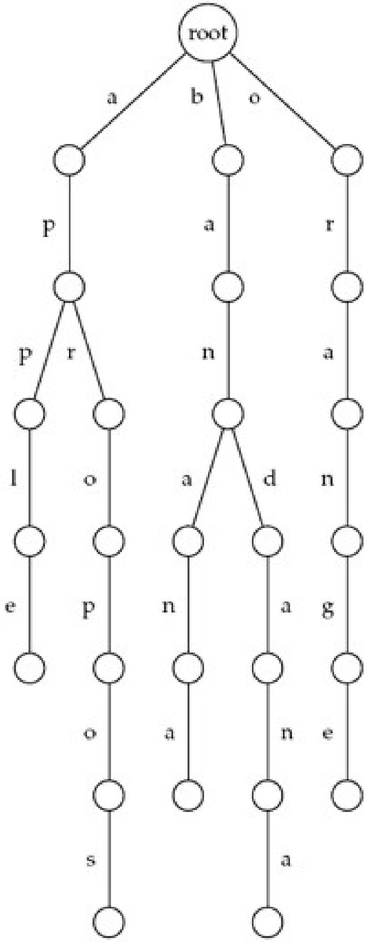
- Formally the running time is $\Theta(mn)$.
- However, the method is very efficient if n is the size of a single or a few computer words.
- Furthermore only two columns of M are needed at any given time. Hence, the space used by the algorithm is $O(n)$.

Slides from Charles Yan

AHO-CORASICK

Search in keyword trees

- Naïve threading in keyword trees do not *remember* the partial matches
- P={apple, appropos}
- T=appappropos
- When threading
  - *app* is a partial match
  - But naïve threading will go back to the root and re-thread *app*
- Define *failure links*



Failure Link

v: a node in keyword tree K

L(v): the label on v, that is, the concatenation of characters on the path from the root to v.

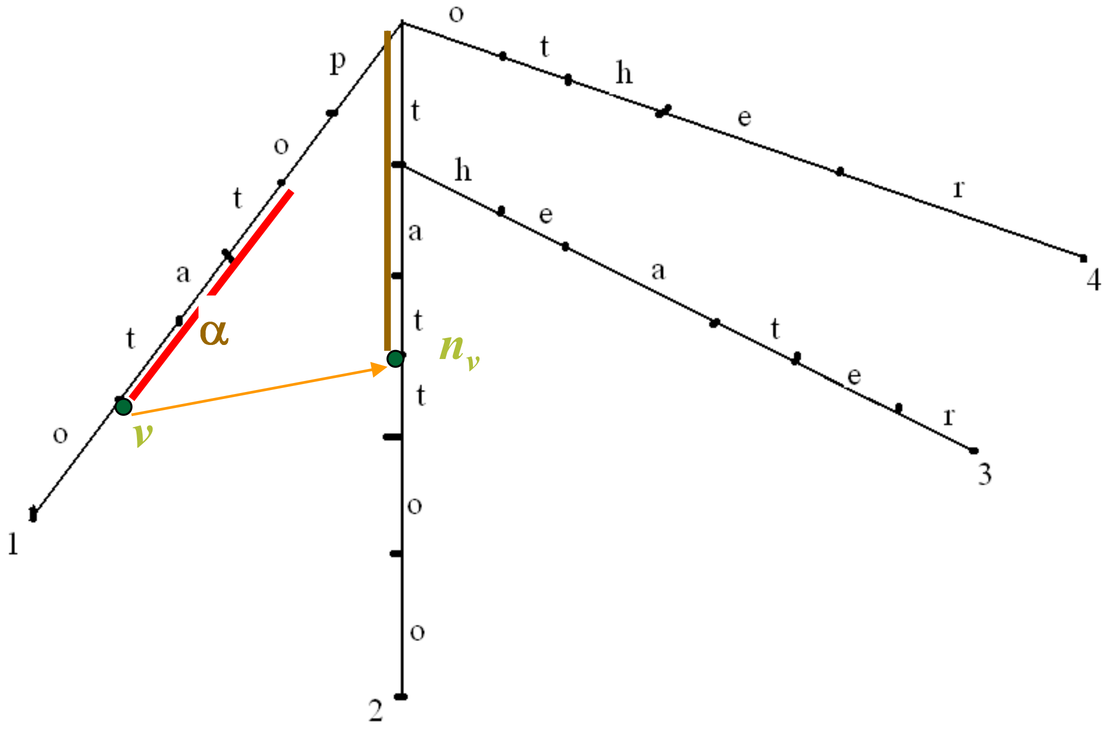
lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be α.

Lemma. There is a unique node in the keyword tree that is labeled by string α. Let this node be n_v. Note that n_v can be the root.

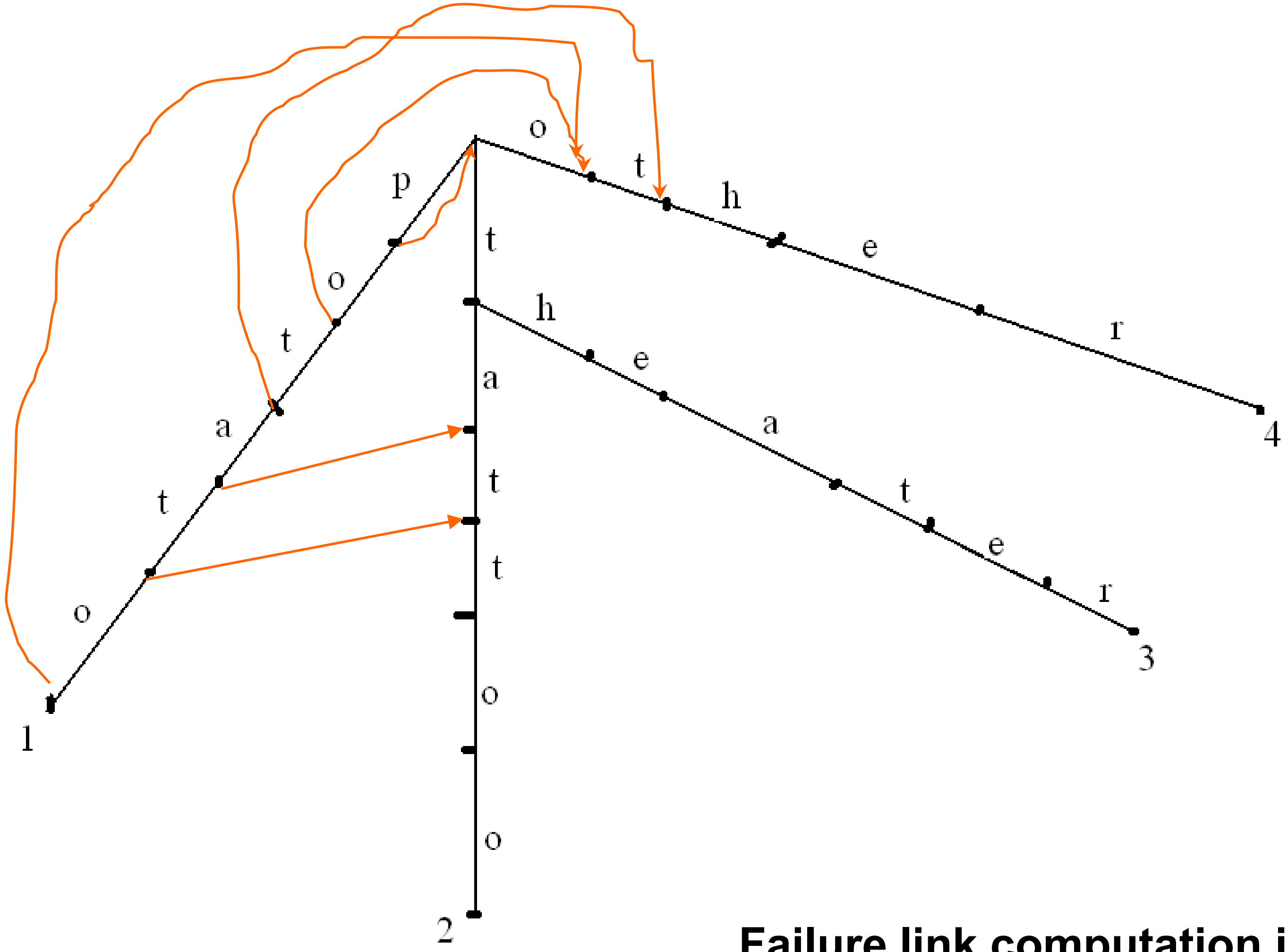
The ordered pair (v, n_v) is called a **failure link**.

Failure Link

P={potato, tattoo, theater, other}



Failure Link



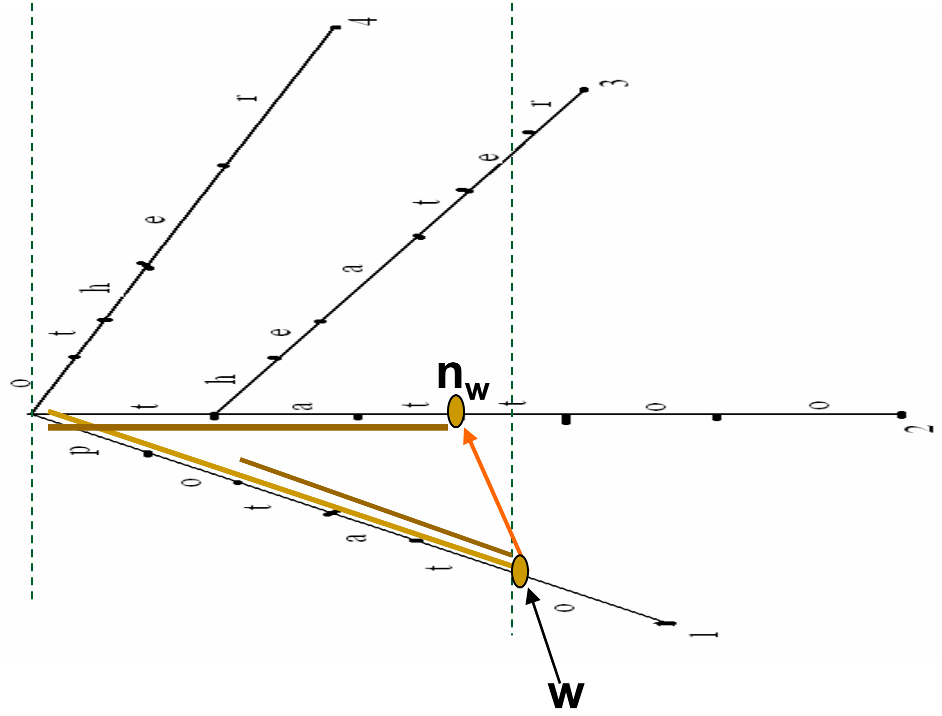
Failure link computation is O(n)

Failure Link

l=3

c=8

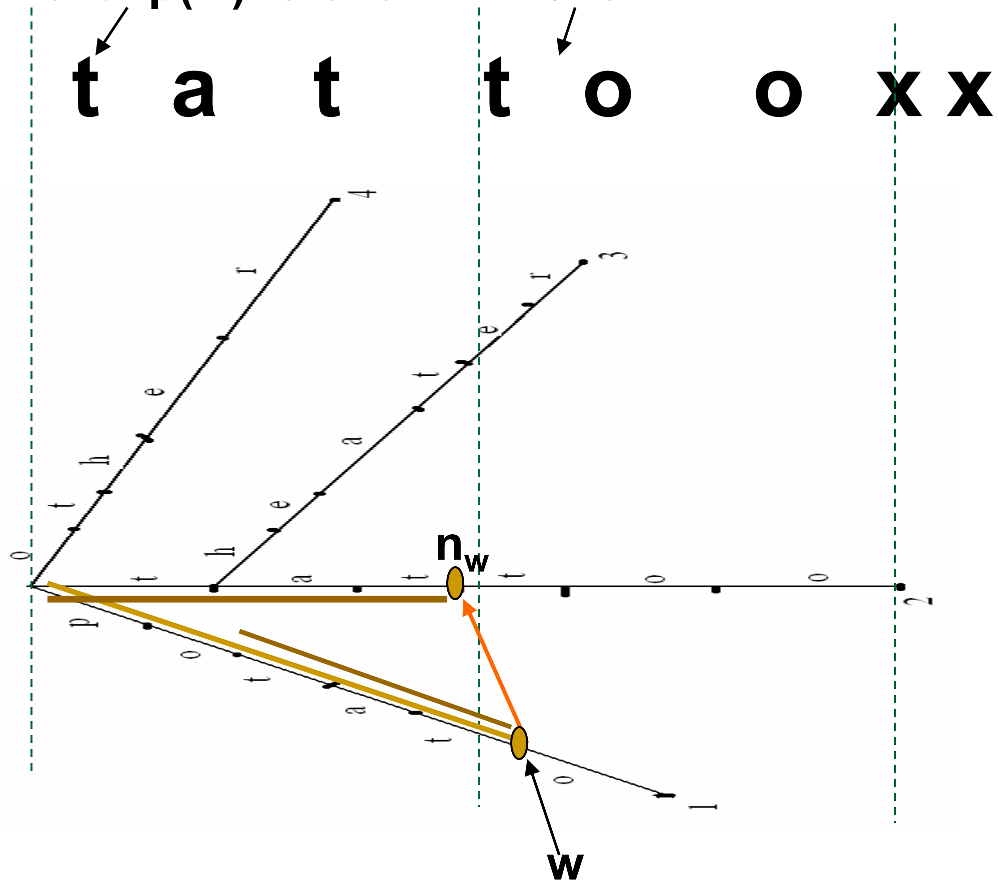
x x p o t a t t o o x x



Failure Link

l=c-lp(w)=8-3=5 c=8

x x p o t a t t o o x x



Failure Link

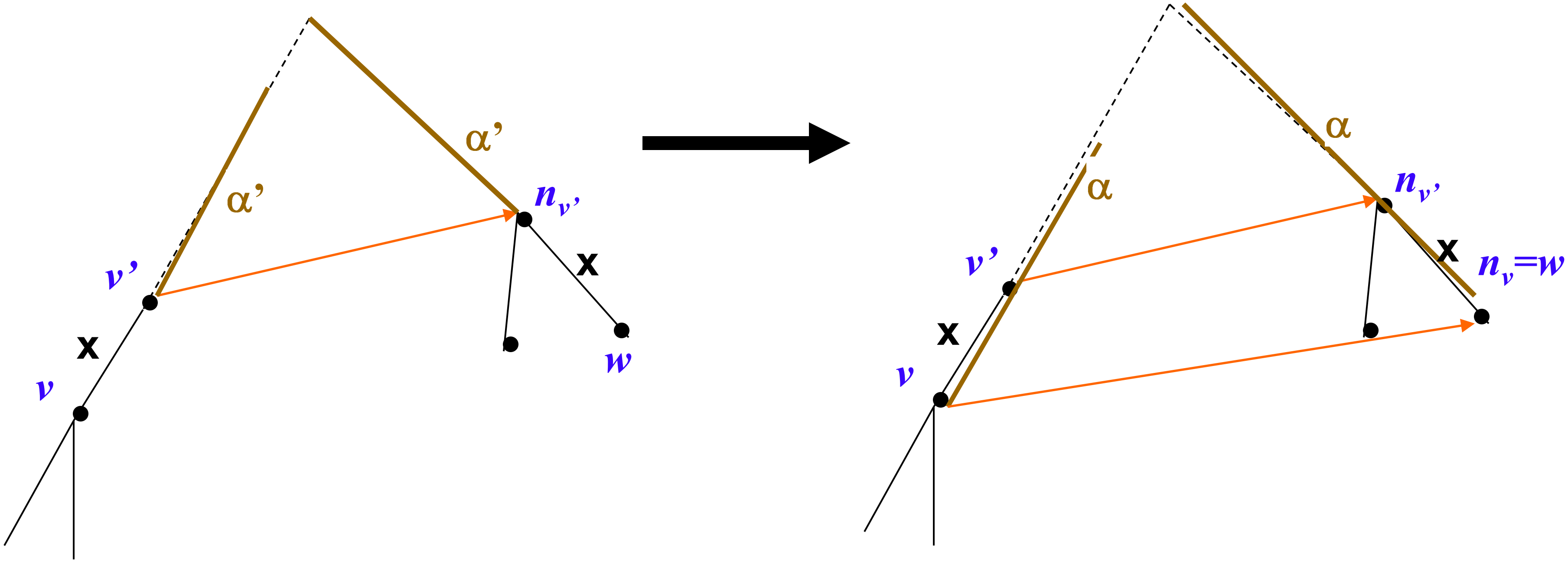
How to construct failure links for a keyword tree in a linear time?

Let d be the distance of a node (v) from the root r.

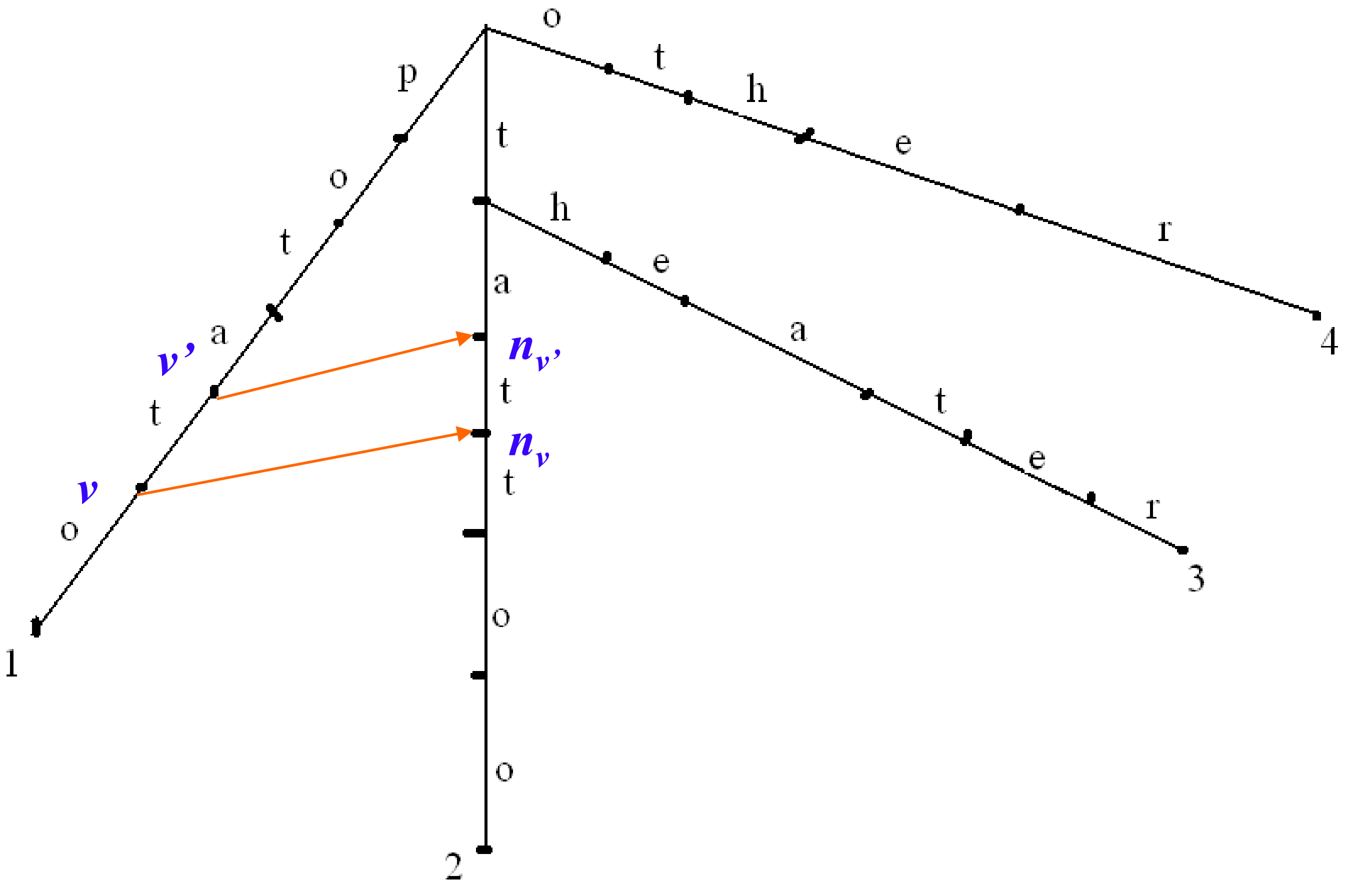
- When $d \leq 1$, i.e., v is the root or v is one character away from r, then $n_v = r$.
- Suppose n_v has been computed for every node (v) with $d \leq k$, we are going to compute n_v for every node with $d = k+1$.
  - v`: parent of v, then v` is k characters from r, that is $d = k$
    - thus the failure link for v` has been computed. $n_{v`}$
  - x: the character on edge (v`, v)

Failure Link

(1) If there is an edge ($n_{v`}$, w) out of $n_{v`}$ labeled with x, then n_v=w.

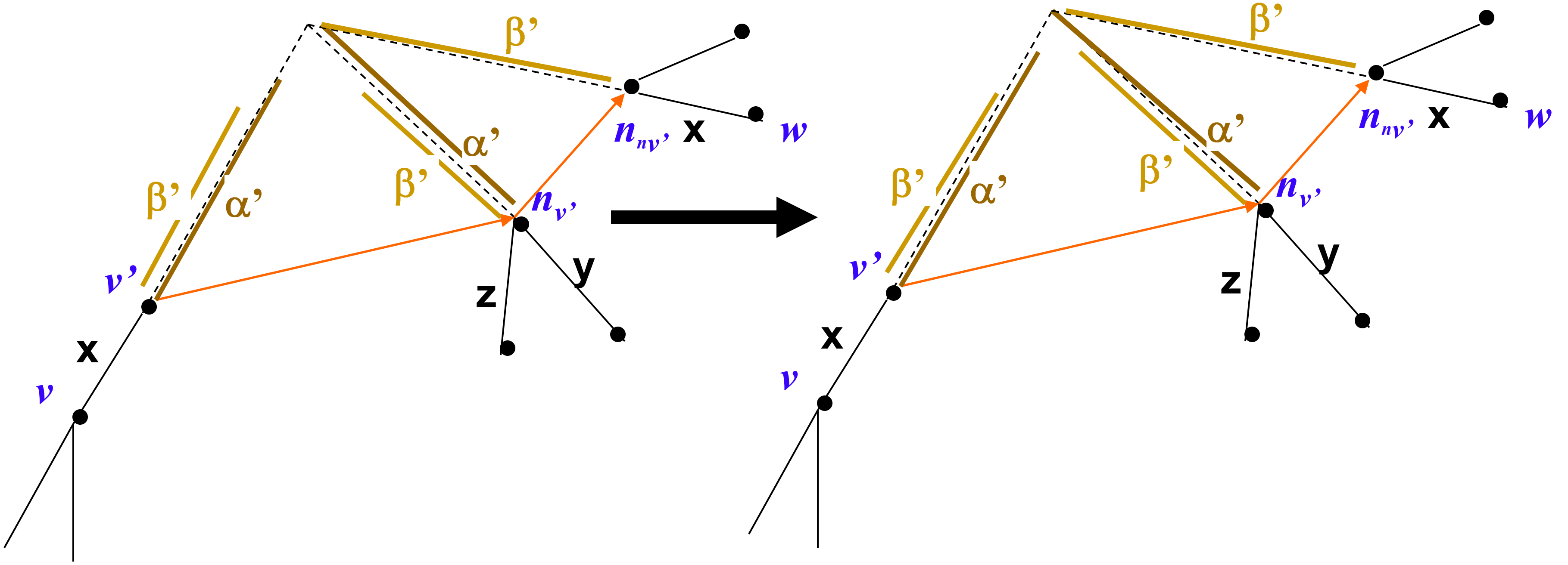


Failure Link



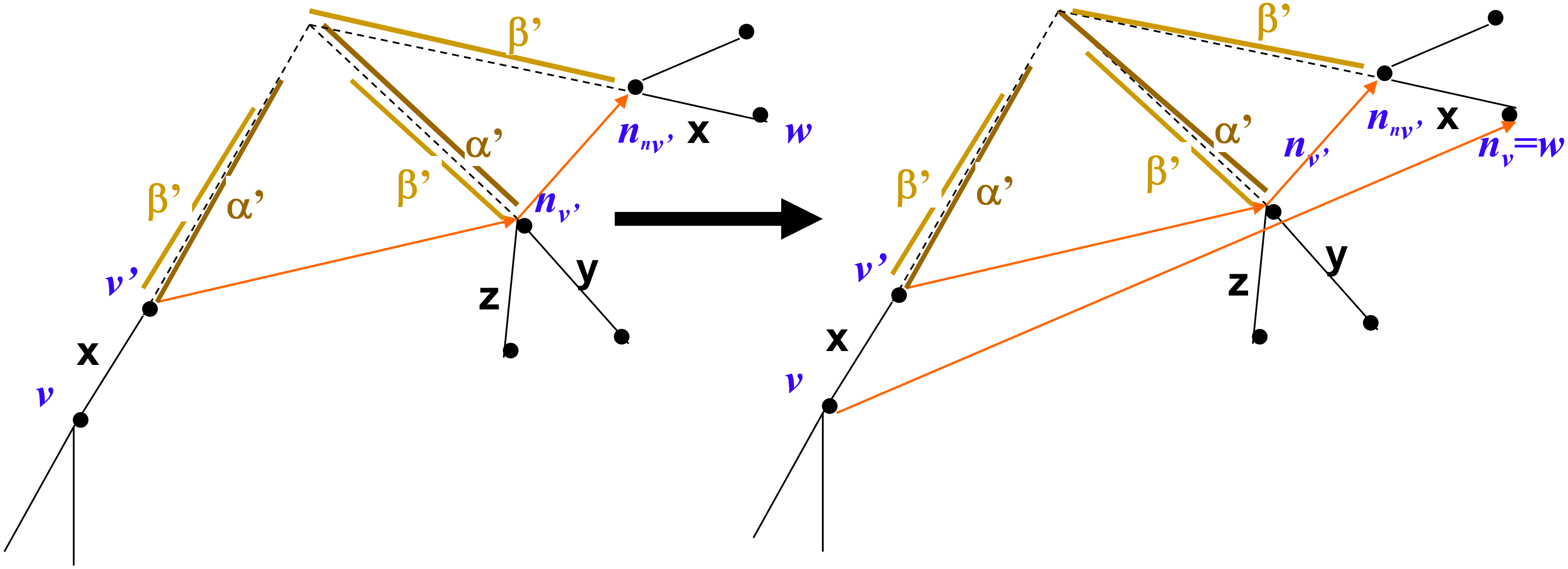
Failure Link

(2) If such an edge does not exist, examine $n_{nv'}$ to see if there is an edge out of it labeled with x. Continue until the root.

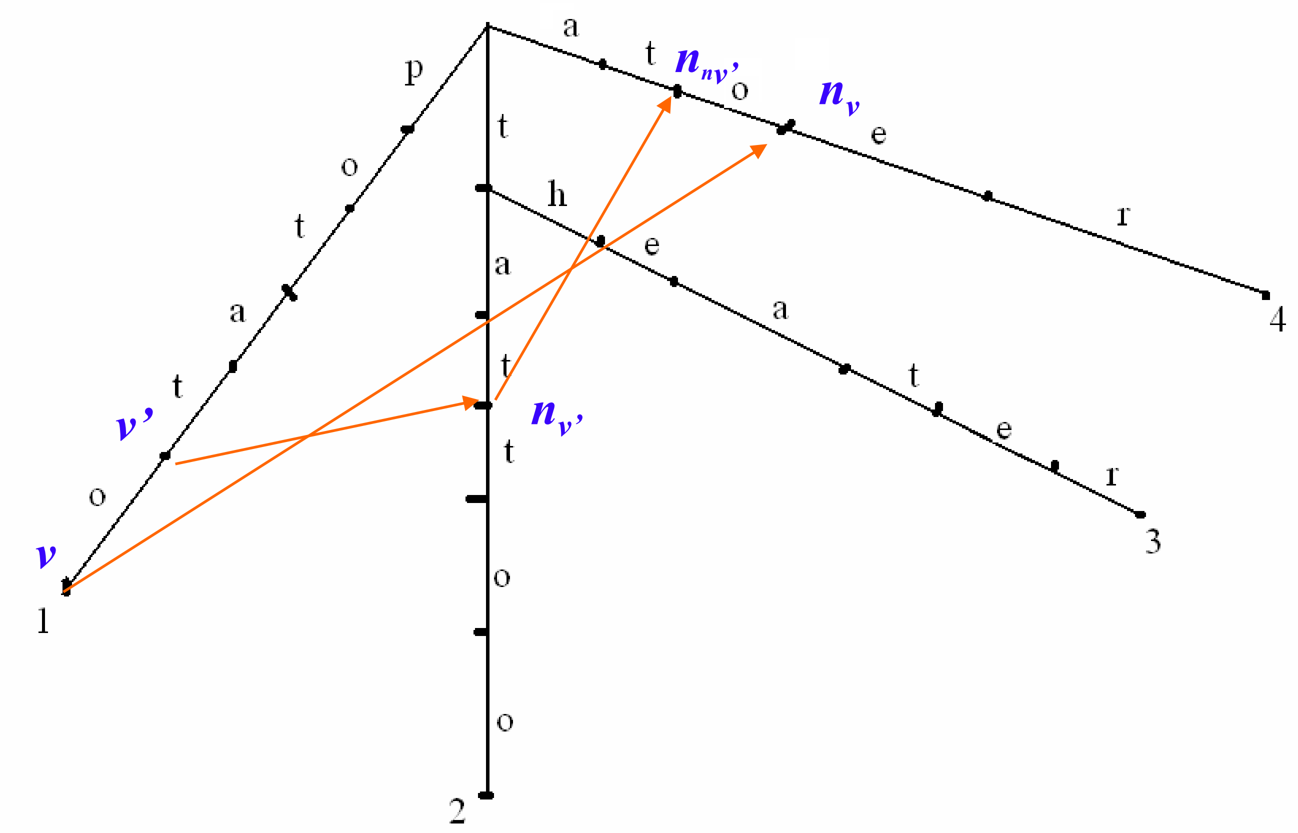


Failure Link

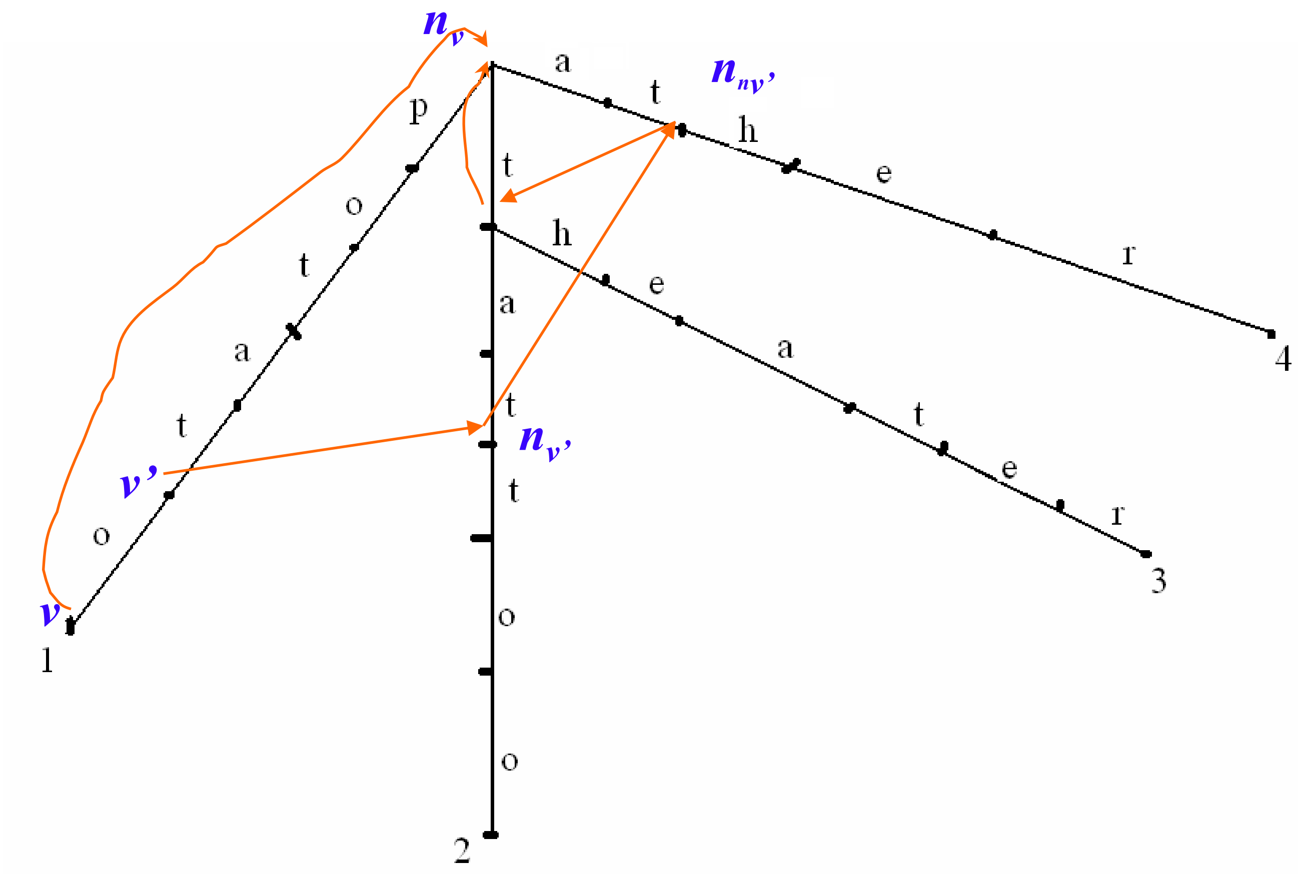
(2) If such an edge does not exist, examine $n_{nv`}$ to see if there is an edge out of it labeled with x. Continue until the root.



Failure Link



Failure Link



Failure Link

Output: calculate n_v for v

Algorithm n_v

```
v` is the parent of v in K
x is the character on edge (v`, v)
$w=n_{v`}$
while there is no edge out of w labeled with x and $w \neq r$
        $w=n_w$
If there is an edge (w, w`) out of w labeled x then
        $n_v=w`$
else $n_v=r$
```

Aho-Corasick Algorithm

Input: Pattern set P and text T

Output: all occurrences in T any pattern from P

```
Algorithm AC
$l$=1;
c=1;
w=root of K
Repeat
    while there is an edge (w, w') labeled with T(c)
        if w` is numbered by pattern $i$ then
            report that $p_i$ occurs in T starting at $l$;
        w=w'; c++;
    w=$n_w$ and $l$=c-lp(w);
Until c>m
```

Slides from Tolga Can

SUFFIX ARRAYS

Suffix arrays

- Suffix arrays were introduced by Manber and Myers in 1993
- More space efficient than suffix trees
- A suffix array for a string x of length *m* is an array of size *m* that specifies the lexicographic ordering of the suffixes of x.

Suffix arrays

Example of a suffix array for acaaacatat$

0	aaacatat$	3
1	aacatat$	4
2	acaaacatat$	1
3	acatat$	5
4	atat$	7
5	at$	9
6	caaacatat$	2
7	catat$	6
8	tat$	8
9	t$	10
10	$	11

Suffix array construction

- Naive in place construction
  - Similar to insertion sort
  - Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
  - Running time complexity:
    - O(m^2) where m is the length of the string
- Manber and Myers give a O($m \log m$) construction.

Suffix arrays

- O(*n*) space where *n* is the size of the database string
- Space efficient. However, there’s an increase in query time
- Lookup query
  - Based on binary search
  - O(m log n) time; m is the size of the query
  - Can reduce time to O(m + log n) using a more efficient implementation

Searching for a pattern in Suffix Arrays

```
find(Pattern P in SuffixArray A):
    i = 0
    lo = 0, hi = length(A)
    for 0<=i<length(P):
        Binary search for x,y
        where P[i]=S[A[j]+i] for
  lo<=x<=j<y<=hi
        lo = x, hi = y
    return {A[lo],A[lo+1],...,A[hi-1]}
```

Search example

- Search *is* in *mississippi$*

Examine the pattern letter by letter, reducing the range of occurrence each time.

First letter *i*:

occurs in indices from 0 to 3

So, pattern should be between these indices.

Second letter *s*:

occurs in indices from 2 to 3

Done.

Output: issippi$ and ississippi$

0	11	i$
1	8	ippi$
2	5	issippi$
3	2	ississippi$
4	1	mississippi$
5	10	pi$
6	9	ppi$
7	7	sippi$
8	4	sissippi$
9	6	ssippi$
10	3	ssissippi$
11	12	$

Suffix Arrays

- It can be built very fast.
- It can answer queries very fast:
    - How many times ATG appears?
- Disadvantages:
    - Can’t do approximate matching
    - Hard to insert new stuff (need to rebuild the array) dynamically.