

# CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

# Last Lecture: String Algorithms

---

- String Matching Problem
- Hash Table
- Knuth-Morris-Pratt (KMP) Algorithm
- Suffix Trie
- Suffix Array
- Note on String Problems

# String Matching Problem

- Given a text  $T$  and a pattern  $P$ , find all the occurrences of  $P$  within  $T$
- Notations:
  - ▣  $n$  and  $m$ : lengths of  $P$  and  $T$
  - ▣  $\Sigma$ : set of alphabets
    - Constant size
  - ▣  $P_i$ :  $i$ th letter of  $P$  (1-indexed)
  - ▣  $a, b, c$ : single letters in  $\Sigma$
  - ▣  $x, y, z$ : strings

# String Matching Example

□  $T = \text{AGCATGCTGCAGTCATGCTTAGGCTA}$

□  $P = \text{GCT}$

□ A naïve method takes  $O(nm)$  time

▣ We initiate string comparison at every starting point

▣ Each comparison takes  $O(m)$  time

□ We can certainly do better!

# Hash Function

- A function that takes a string and outputs a number
- A good hash function has few collisions
  - ▣ i.e. If  $x \neq y$ ,  $H(x) \neq H(y)$  with high probability
- An easy and powerful hash function is a polynomial mod some prime  $p$ 
  - ▣ Consider each letter as a number (ASCII value is fine)
  - ▣  $H(x_1 \dots x_k) = x_1 a^{k-1} + x_2 a^{k-2} + \dots + x_{k-1} a + x_k$
  - ▣ How do we find  $H(x_2 \dots x_{k+1})$  from  $H(x_1 \dots x_k)$ ?

# Hash Table

- Main idea: preprocess  $T$  to speedup queries
  - ▣ Hash every substring of length  $k$
  - ▣  $k$  is a small constant
- For each query  $P$ , hash the first  $k$  letters of  $P$  to retrieve all the occurrences of it within  $T$
- Don't forget to check collisions!

# Hash Table

## □ Pros:

- Easy to implement
- Significant speedup in practice

## □ Cons:

- Doesn't help the asymptotic efficiency
  - Can take  $\Theta(nm)$  time if hashing is terrible
- A lot of memory consumption

# Knuth-Morris-Pratt (KMP) Matcher

- A linear time (!) algorithm that solves the string matching problem by preprocessing  $P$  in  $\Theta(m)$  time
  - ▣ Main idea is to skip some comparisons by using the previous comparison result
- Uses an auxiliary array  $\pi$  that is defined as the following:
  - ▣  $\pi[i]$  is the largest integer smaller than  $i$  such that  $P_1 \dots P_{\pi[i]}$  is a suffix of  $P_1 \dots P_i$
- ... It's better to see an example than the definition



# $\pi$ Table Example (from CLRS)

$i$	1	2	3	4	5	6	7	8	9	10
$P_i$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

- $\pi[i]$ : the largest integer smaller than  $i$  such that  $P_1 \dots P_{\pi[i]}$  is a suffix of  $P_1 \dots P_i$ 
  - ▣ e.g.  $\pi[6] = 4$  since abab is a suffix of ababab
  - ▣ e.g.  $\pi[9] = 0$  since no prefix of length  $\leq 8$  ends with c
- Let's see why this is useful

# Using the $\pi$ Table

- $T = \text{ABC ABCDAB ABCDABCDABDE}$
- $P = \text{ABCDABD}$
- $\pi = (0, 0, 0, 0, 1, 2, 0)$
- Start matching at the first position of  $T$ :

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
**ABC****D****ABD**  
1234567

- Mismatch at the 4<sup>th</sup> letter of  $P$ !

# Using the $\pi$ Table

- There is no point in starting the comparison at  $T_2, T_3$ 
  - ▣ We matched  $k = 3$  letters so far
  - ▣ Shift  $P$  by  $k - \pi[k] = 3$  letters

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
**A**BCDABD  
1234567

- Mismatch at  $T_4$  again!

# Using the $\pi$ Table

- We define  $\pi[0] = -1$ 
  - We matched  $k = 0$  letters so far
  - Shift  $P$  by  $k - \pi[k] = 1$  letter

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
**ABCDABD**  
1234567

- Mismatch at  $T_{11}$ !

# Using the $\pi$ Table

- $\pi[6] = 2$  says  $P_1P_2$  is a suffix of  $P_1 \dots P_6$
- Shift  $P$  by  $6 - \pi[6] = 4$  letters

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
ABCDABD  
| |  
**ABCDABD**  
1234567

- Again, no point in shifting  $P$  by 1, 2, or 3 letters

# Using the $\pi$ Table

- ❑ Mismatch at  $T_{11}$  again!

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
**ABCDABD**  
1234567

- ❑ Currently 2 letters are matched
- ❑ We shift  $P$  by  $2 = 2 - \pi[2]$  letters

# Using the $\pi$ Table

- ❑ Mismatch at  $T_{11}$  yet again!

12345678901234567890123  
**ABC ABCDAB ABCDABCDABDE**  
**A**BCDABD  
1234567

- ❑ Currently no letters are matched
- ❑ We shift  $P$  by  $1 = 0 - \pi[0]$  letters

# Using the $\pi$ Table

- ❑ Mismatch at  $T_{18}$

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3  
**ABC ABCDAB ABCDABCDABDE**  
**ABCDABD**  
1 2 3 4 5 6 7

- ❑ Currently 6 letters are matched
- ❑ We shift  $P$  by  $4 = 6 - \pi[6]$  letters



# Using the $\pi$ Table

- Finally, there it is!

12345678901234567890123  
**ABC ABCDAB ABCD****ABCDABDE**  
**ABCDABD**  
1234567

- Currently all 7 letters are matched
- After recording this match (match at  $T_{16} \dots T_{22}$ ), we shift  $P$  again in order to find other matches
  - ▣ Shift by  $7 = 7 - \pi[7]$  letters

# Computing $\pi$

- Observation 1: if  $P_1 \dots P_{\pi[i]}$  is a suffix of  $P_1 \dots P_i$ , then  $P_1 \dots P_{\pi[i]-1}$  is a suffix of  $P_1 \dots P_{i-1}$ 
  - ▣ Well, obviously...
- Observation 2: all the prefixes of  $P$  that are a suffix of  $P_1 \dots P_i$  can be obtained by recursively applying  $\pi$  to  $i$ 
  - ▣ e.g.  $P_1 \dots P_{\pi[i]}$ ,  $P_1 \dots P_{\pi[\pi[i]]}$ ,  $P_1 \dots P_{\pi[\pi[\pi[i]]]}$  are all suffixes of  $P_1 \dots P_i$

# Computing $\pi$

- A non-obvious conclusion:
  - ▣ First, let's write  $\pi^{(k)}[i]$  as  $\pi[\cdot]$  applied  $k$  times to  $i$
  - ▣ e.g.  $\pi^{(2)}[i] = \pi[\pi[i]]$
  - ▣  $\pi[i]$  is equal to  $\pi^{(k)}[i - 1] + 1$ , where  $k$  is the smallest integer that satisfies  $P_{\pi^{(k)}[i-1]+1} = P_i$ 
    - If there is no such  $k$ ,  $\pi[i] = 0$
- Intuition: we look at all the prefixes of  $P$  that are suffixes of  $P_1 \dots P_{i-1}$  and find the longest one whose next letter matches  $P_i$  too

# Implementation

```
pi[0] = -1;
int k = -1;
for(int i = 1; i <= m; i++) {
    while(k >= 0 && P[k+1] != P[i])
        k = pi[k];
    pi[i] = ++k;
}
```

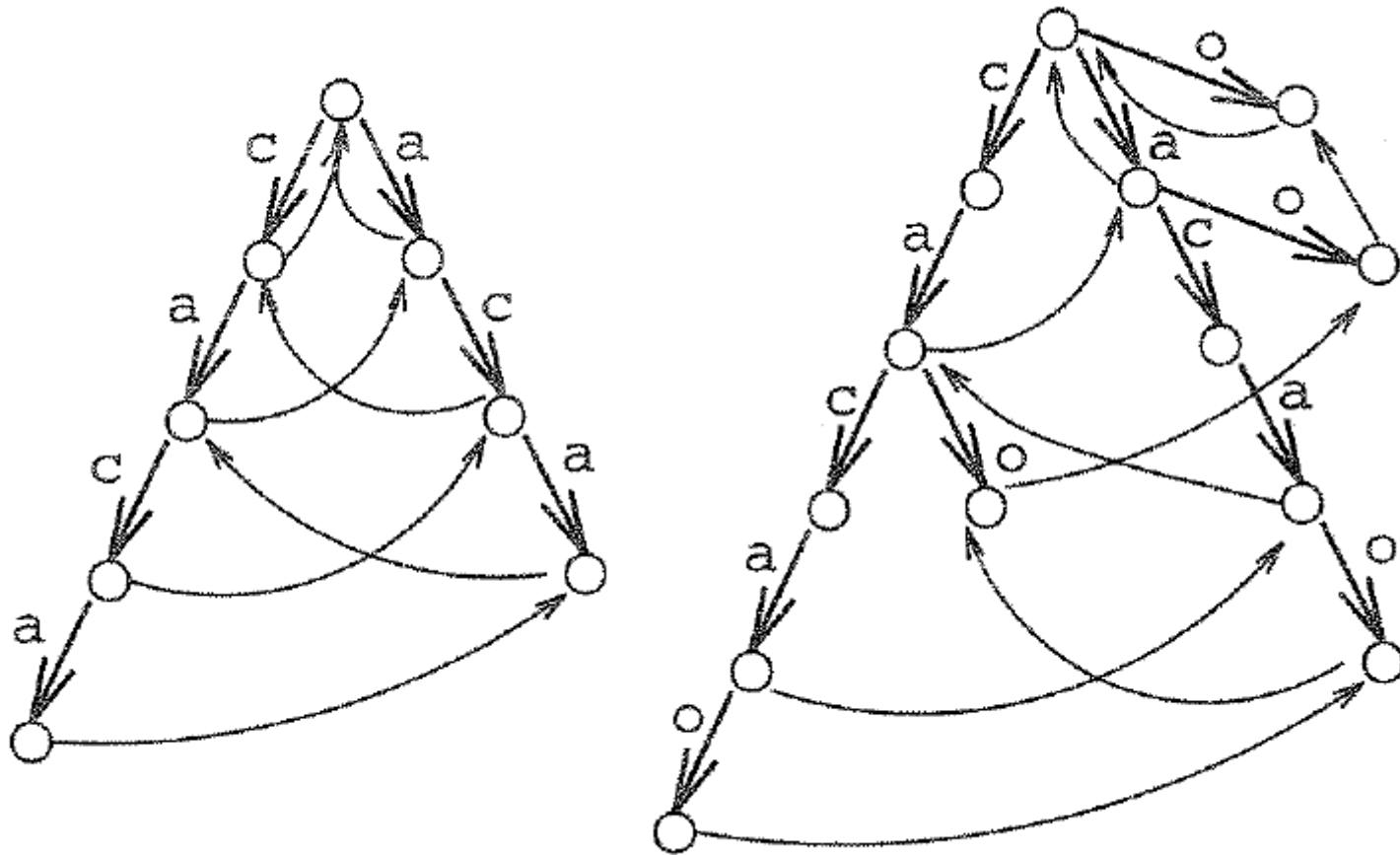
# Pattern Matching Implementation

```
int k = 0;
for(int i = 1; i <= n; i++) {
    while(k >= 0 && P[k+1] != T[i])
        k = pi[k];
    k++;
    if(k == m) {
        // P matches T[i-m+1..i]
        k = pi[k];
    }
}
```

# Suffix Trie

- Suffix trie of a string  $T$  is a rooted tree that stores all the suffixes (thus all the substrings)
- Each node corresponds to some substring of  $T$
- Each edge is associated with an alphabet
- For each node that corresponds to  $ax$ , there is a special pointer called *suffix link* that leads to the node corresponding to  $x$
- Surprisingly easy to implement!

# Suffix Trie Example



(Figure modified from Ukkonen's original paper)

# Incremental Construction

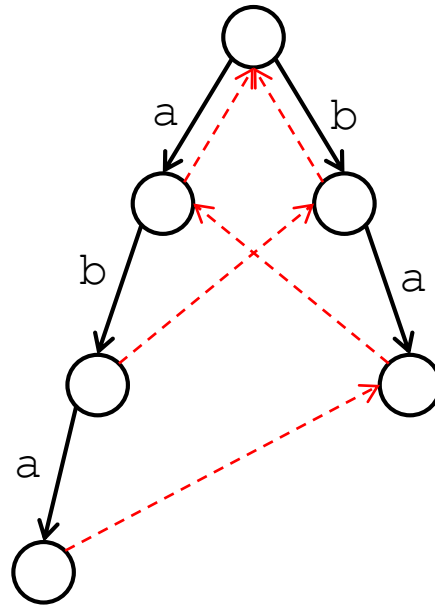
- Given the suffix tree for  $T_1 \dots T_n$ 
  - ▣ Then we append  $T_{n+1} = a$  to  $T$ , creating necessary nodes
- Start at node  $u$  corresponding to  $T_1 \dots T_n$ 
  - ▣ Create an  $a$ -transition to a new node  $v$
- Take the suffix link at  $u$  to go to  $u'$ , corresponding to  $T_2 \dots T_n$ 
  - ▣ Create an  $a$ -transition to a new node  $v'$
  - ▣ Create a suffix link from  $v$  to  $v'$



# Incremental Construction

- We repeat the previous process:
  - ▣ Take the suffix link at the current node
  - ▣ Make a new  $a$ -transition there
  - ▣ Create the suffix link from the previous node
- We stop if the node already has an  $a$ -transition
  - ▣ Because from this point, all nodes that are reachable via suffix links already have an  $a$ -transition

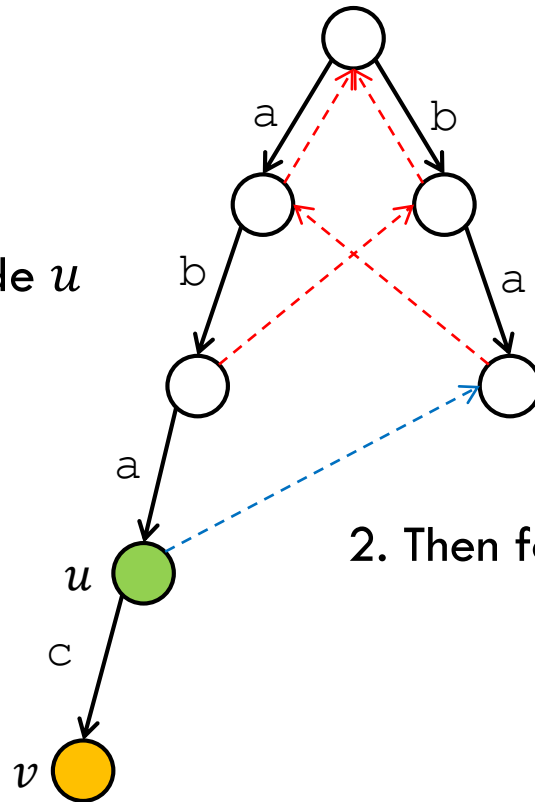
# Construction Example



Given the suffix trie for aba  
We want to add a new letter c

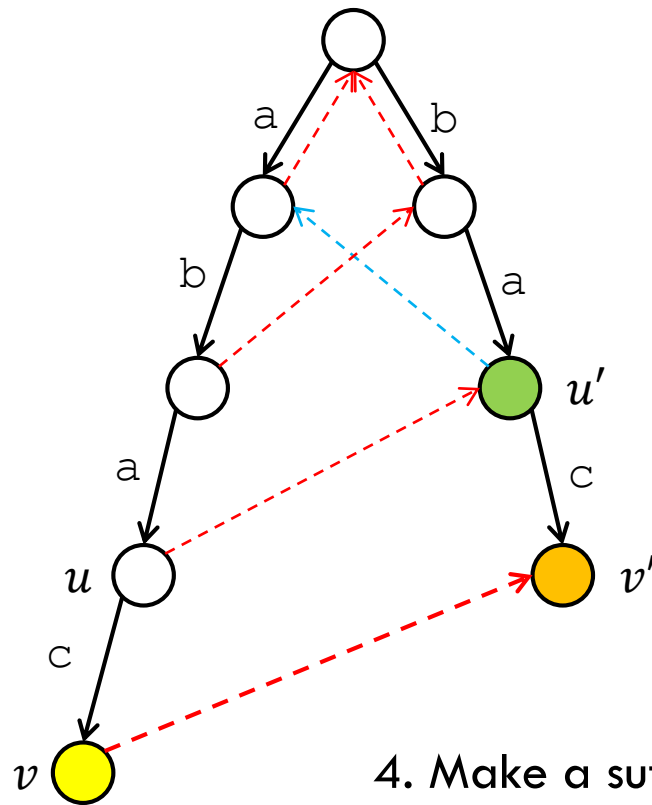
# Construction Example

1. Start at the green node  $u$  and make a  $c$ -transition



2. Then follow the suffix link

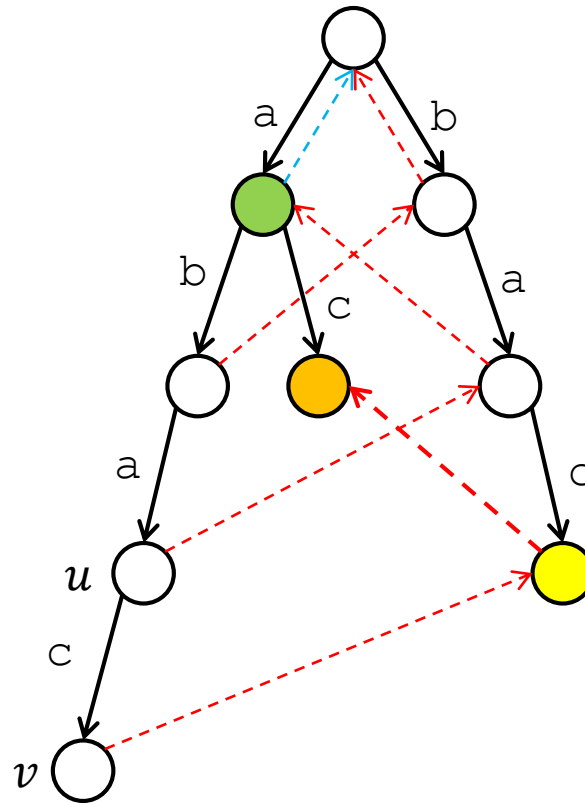
# Construction Example



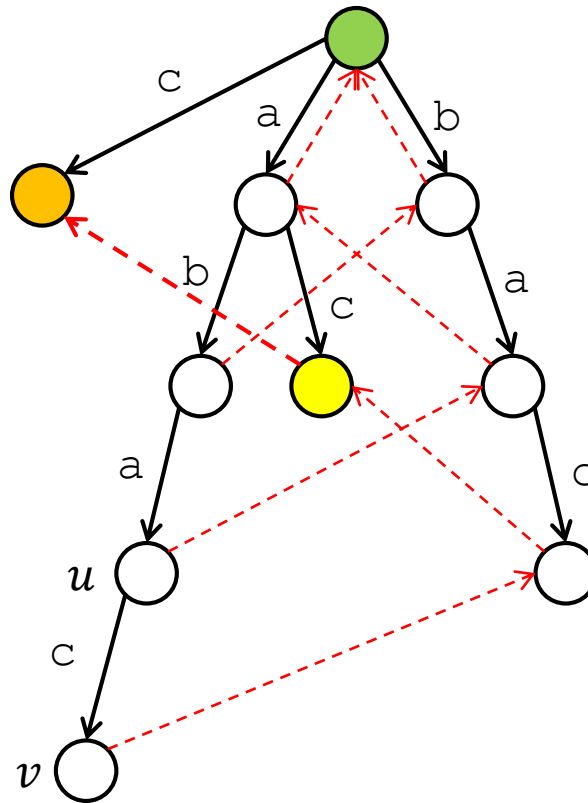
3. Make a  $c$ -transition at  $u'$

4. Make a suffix link from  $v$

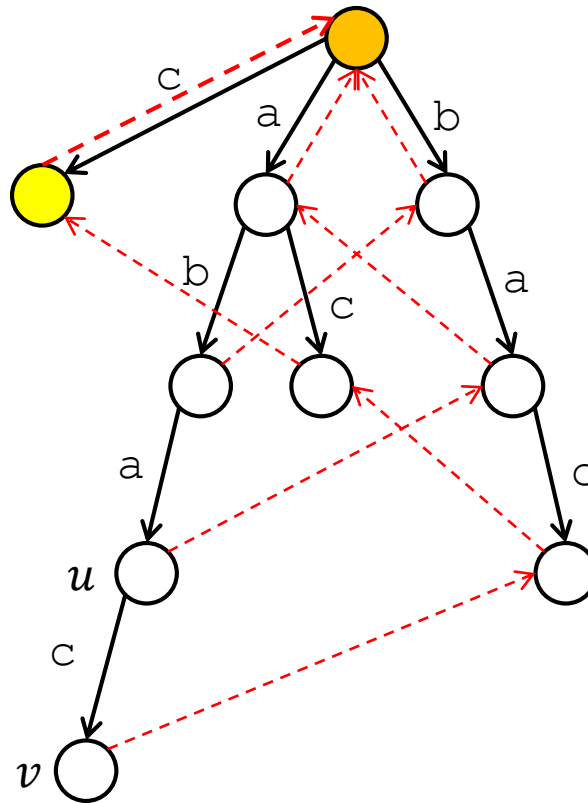
# Construction Example



# Construction Example



# Construction Example



# Suffix Trie Analysis

- Construction time is linear in the tree size
  - ▣ But the tree size can be quadratic in  $n$ 
    - e.g.  $T = aa...abb...b$



# Pattern Matching

- To find  $P$ , start at the root and keep following edges labeled with  $P_1, P_2$ , etc.
- Got stuck? Then  $P$  doesn't exist in  $T$

# Suffix Array

Input string	Get all suffixes	Sort the suffixes	Take the indices
BANANA	1 BANANA	6 A	6, 4, 2, 1, 5, 3
	2 ANANA	4 ANA	
	3 NANA	2 ANANA	
	4 ANA	1 BANANA	
	5 NA	5 NA	
	6 A	3 NANA	

# Suffix Array

- Memory usage is  $O(n)$
- Has the same computational power as suffix trie
- Can be constructed in  $O(n)$  time (!)
  - ▣ But it's hard to implement
- There is an approachable  $O(n \log^2 n)$  algorithm
  - ▣ If you want to see how it works, read the paper on the course website
  - ▣ <http://cs97si.stanford.edu/suffix-array.pdf>

# Note on String Problems

- Always be aware of the null-terminators
- Simple hash works so well in many problems
  - ▣ Even for problems that aren't supposed to be solved by hashing
- If a problem involves rotations of a string, consider concatenating it with itself and see if it helps
- Stanford team notebook has implementations of suffix arrays and the KMP matcher