공학석사학위논문

Cartesian Tree Matching and Indexing

Cartesian 트리에 기반한 문자열 매칭 및 인덱싱

2020 년 8 월

서울대학교 대학원

컴퓨터공학부

박 성 관

Cartesian Tree Matching and Indexing

Cartesian 트리에 기반한 문자열 매칭 및 인덱싱

지도교수 박 근 수

이 논문을 공학석사 학위논문으로 제출함

2020 년 6 월

서울대학교 대학원

컴퓨터공학부

박 성 관

박성관의 석사 학위논문을 인준함

2020 년 7 월

위 원 장	Srinivasa Rao Satti	(인)
부위원장	박 근 수	(인)
위 원	Bernhard Egger	(인)

Abstract

Cartesian Tree Matching and Indexing

Sung Gwan Park
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

We introduce a new metric of match, called Cartesian tree matching, which means that two strings match if they have the same Cartesian trees. Based on Cartesian tree matching, we define single pattern matching for a text of length n and a pattern of length m, and multiple pattern matching for a text of length n and k patterns of total length m. We present an $O(n + m)$ time algorithm for single pattern matching, and an $O((n + m) \log k)$ deterministic time or $O(n+m)$ randomized time algorithm for multiple pattern matching. We also define an index data structure called Cartesian suffix tree, and present an $O(n)$ randomized time algorithm to build the Cartesian suffix tree. Our efficient algorithms for Cartesian tree matching use a representation of the Cartesian tree, called the parent-distance representation.

Keywords: Cartesian tree matching, Pattern matching, Indexing, Parent-distance representation
Student Number: 2018-26744

Contents

List of Figures

Chapter 1

Introduction

String matching is one of fundamental problems in computer science, and it can be applied to many practical problems. In many applications string matching has variants derived from exact matching (which can be collectively called *generalized matching*), such as order-preserving matching [20, 21, 23], parameterized matching [4, 7, 8], jumbled matching [9], overlap matching [3], pattern matching with swaps [2], and so on. These problems are characterized by the way of defining a *match*, which depends on the application domains of the problems. In financial markets, for example, people want to find some patterns in the time series data of stock prices. In this case, they would like to know more about some pattern of price fluctuations than exact prices themselves [15]. Therefore, we need a definition of match which is appropriate to handle such cases.

The Cartesian tree [29] is a tree data structure that represents an array, only focusing on the results of comparisons between numeric values in the array. The Cartesian tree has been used in many topics such as two-dimensional searching, rank-select data structures, and range minimum queries [16, 29].

In this thesis we introduce a new metric of match, called *Cartesian tree*

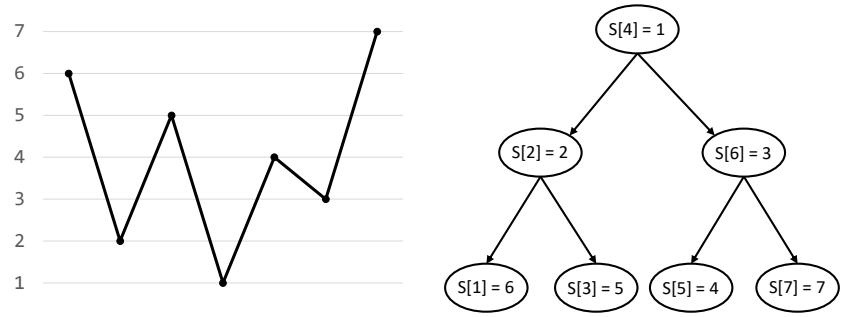


Figure 1.1 Example pattern $S = (6, 2, 5, 1, 4, 3, 7)$ and its corresponding Cartesian tree

matching, which means that two strings match if they have the same Cartesian trees. If we model the time series stock prices as a numerical string, we can find a desired pattern from the data by solving a Cartesian tree matching problem. For example, let's assume that the pattern we want to find looks like the picture on the left of Figure 1.1, which is a common pattern called the head-and-shoulder [15] (in fact there are two versions of the head-and-shoulder: one is the picture in Figure 1.1 and the other is the picture reversed). The picture on the right of Figure 1.1 is the Cartesian tree corresponding to the pattern on the left. Cartesian tree matching finds every position of the text which has the same Cartesian tree as the picture on the right of Figure 1.1.

Even though order-preserving matching [20, 21, 23] can also be applied to finding patterns in time series data, Cartesian tree matching may be more appropriate than order-preserving matching in finding patterns. For instance, let's assume that we are looking for the pattern in Figure 1.1 in time series stock prices. An important characteristic of the pattern is that the price hits the bottom (head), and it has a shoulder before and one after the head. But the relative order between the two shoulders (i.e., which one is higher) does not matter. If we model this pattern into order-preserving matching, then order-preserving matching imposes a relative order between two shoulders $S[2]$ and

$S[6]$. Moreover, it imposes an unnecessary order between two valleys $S[3]$ and $S[5]$. If we try to generate all possible patterns in terms of order-preserving matching, the number of patterns can be exponential in the pattern length. Hence, order preserving matching may not be able to find such a pattern in time series data efficiently. In contrast, the pattern in Figure 1.1 can be represented by a single Cartesian tree, and therefore Cartesian tree matching is a more appropriate metric in such cases.

In this thesis we define string matching problems based on Cartesian tree matching: single pattern matching for a text of length n and a pattern of length m, and multiple pattern matching for a text of length n and k patterns of total length m, and we present efficient algorithms for them. We also define an index data structure called Cartesian suffix tree as in the cases of parameterized matching and order-preserving matching [8, 13], and present an efficient algorithm to build the Cartesian suffix tree. To obtain efficient algorithms for Cartesian tree matching, we define a representation of the Cartesian tree, called the *parent-distance representation*.

In Chapter 2 we give basic definitions for Cartesian tree matching. In Chapter 3 we propose an $O(n + m)$ time algorithm for single pattern matching. In Chapter 4 we present an $O((n + m) \log k)$ deterministic time or $O(n + m)$ randomized time algorithm for multiple pattern matching. In Chapter 5 we define the Cartesian suffix tree, and present an $O(n)$ randomized time algorithm to build the Cartesian suffix tree of a string of length n. We conclude in Chapter 6.

Chapter 2

Problem Definition

2.1 Basic notations

A *string* is a sequence of characters in an alphabet Σ, which is a set of integers. We assume that the comparison between any two characters can be done in $O(1)$ time. For a string S, $S[i]$ represents the i-th character of S, and $S[i..j]$ represents a substring of S starting from i and ending at j. Hence, $S[1..i]$ is the prefix of S ending at i, and $S[i..n]$ is the suffix of S starting from i, assuming that n is the length of S.

2.2 Cartesian tree matching

A string S can be associated with its corresponding Cartesian tree $CT(S)$ according to the following rules [29]:

- If S is an empty string, $CT(S)$ is an empty tree.
- If $S[1..n]$ is not empty and $S[i]$ is the minimum value among S, $CT(S)$ is the tree with $S[i]$ as the root, $CT(S[1..i-1])$ as the left subtree, and $CT(S[i+1..n])$ as the right subtree. If there are two or more minimum

values, we choose the leftmost one as the root.

Since each character in string S corresponds to a node in Cartesian tree $CT(S)$, we can treat each character as a node in the Cartesian tree.

Cartesian tree matching is the problem to find all the matches in the text which have the same Cartesian tree as a given pattern. Formally, we define the problem as follows:

Definition 2.1. (Cartesian tree matching) We say that two strings S_1 and S_2 match (which is denoted by $S_1 \approx S_2$) if their Cartesian trees are the same. Given two strings text $T[1..n]$ and pattern $P[1..m]$, find every $1 \leq i \leq n-m+1$ such that $T[i..i+m-1] \approx P[1..m]$ (i.e., $CT(T[i..i+m-1]) = CT(P[1..m])$).

For example, let's consider a sample text $T = (41, 36, 15, 8, 41, 23, 28, 16, 26, 22, 56, 29, 12, 61)$. If we find the pattern in Figure 1.1, which is $P = (6, 2, 5, 1, 4, 3, 7)$, we can find a match at position 5 of the text, i.e., $CT(T[5..11]) = CT(P[1..7])$.

Note that Cartesian tree matching is a transitive relation. That is, if there are three strings S_1, S_2 and S_3 such that $S_1 \approx S_2$ and $S_2 \approx S_3$, then $S_1 \approx S_3$ always holds.

Order-preserving matching [21] is the problem to find all the matches in the text which have the same *order relations* as a given pattern.

Definition 2.2. (Natural representation of order relations) Given a string $S[1..n]$ and a character c, the *rank* of c in S is defined as $rank_S(c) = 1 + |\{i : S[i] < c \text{ for } 1 \leq i \leq n\}|$. For a string S, we define the natural representation of the order relations as an integer string $\sigma(S)[1..n]$, where $\sigma(S)[i] = rank_S(S[i])$.

Definition 2.3. (Order-preserving matching) Given two strings text $T[1..n]$ and pattern $P[1..m]$, find every $1 \leq i \leq n-m+1$ such that $\sigma(S[i..i+m-1]) = \sigma(P)$.

Every order-preserving matching is also a Cartesian tree matching, but the converse is not true as follows. If two strings are matched in the metric of

order-preserving matching, the indices of minimum values of the two strings are located in the same positions. Furthermore, this property holds recursively in the substrings to the left and to the right of the minimum values. Hence, both strings have the same Cartesian trees. For the other direction, let's consider the above example. In the metric of Cartesian tree matching, $T[5..11]$ matches $P[1..7]$. In the metric of order-preserving matching they does not match, because the relative order between $T[6] = 23$ and $T[10] = 22$ is different from that between $P[2] = 2$ and $P[6] = 3$. That is, $T[5..11]$ shows that a Cartesian tree matching may not be an order-preserving matching. Therefore, Cartesian tree matching has a weaker matching condition than order-preserving matching. Both order-preserving matching and Cartesian tree matching satisfy the properties of a *substring consistent equivalence relation (SCER)* [25], which is defined as follows.

Definition 2.4. (Substring consistent equivalence relation) If a relation $\sim$ on two strings S_1 and S_2 satisfies the following properties, we call the relation a *substring consistent equivalence relation.*

1. S_1 and S_2 have the same length n.
2. For all $1 \leq i \leq j \leq n$, $S_1[i..j] \sim S_2[i..j]$.

We will show that Cartesian tree matching satisfies the properties of a substring consistent equivalence relation in Section 3.2.

Chapter 3

Single Pattern Matching in $O(n+m)$ Time

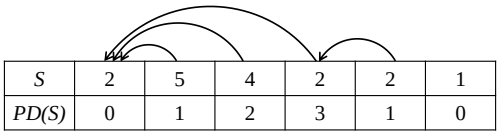
3.1 Parent-distance representation

In order to solve Cartesian tree matching without building every possible Cartesian tree, we propose an efficient representation to store the information about Cartesian trees, called the *parent-distance representation*.

Definition 3.1. (Parent-distance representation) Given a string $S[1..n]$, the *parent-distance representation* of S is an integer string $PD(S)[1..n]$, which is defined as follows:

$$PD(S)[i] = \begin{cases} i - \max_{1 \le j < i}\{j : S[j] \le S[i]\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.1 shows the parent-distance representation of string $S = (2, 5, 4, 2, 2, 1)$. The arrows in Figure 3.1 starting from each $S[i]$ point to their corresponding $S[j]$ in Definition 3.1. For example, $S[4]$ points to $S[1]$ and the distance between them is 3, thus $PD(S)[4] = 3$. Note that $S[j]$ in Definition 3.1 represents the parent of $S[i]$ in Cartesian tree $CT(S[1..i])$. Furthermore, if there is no such



S	2	5	4	2	2	1
$PD(S)$	0	1	2	3	1	0

Figure 3.1 Parent-distance representation of string $S = (2, 5, 4, 2, 2, 1)$

j, $S[i]$ is the root of Cartesian tree $CT(S[1..i])$ (and there is no arrow from $S[i]$ in Figure 3.1).

Theorem 3.1 shows that the parent-distance representation has a one-to-one mapping to the Cartesian tree, so it can substitute the Cartesian tree without any loss of information.

Theorem 3.1. Two strings S_1 and S_2 have the same Cartesian trees if and only if S_1 and S_2 have the same parent-distance representations.

Proof. If two strings have different lengths, they have different Cartesian trees and different parent-distance representations, so the theorem holds. Therefore, we can only consider the case where S_1 and S_2 have the same length. Let n be the length of S_1 and S_2. We prove the theorem by an induction on n.

If $n = 1$, S_1 and S_2 will always have the same Cartesian trees with only one node. Furthermore, they will have the same parent-distance representation (0). Therefore, the theorem holds when $n = 1$.

Let's assume that the theorem holds when $n = k$, and show that it holds when $n = k + 1$.

($\Longrightarrow$) Assume that $S_1[1..k + 1]$ and $S_2[1..k + 1]$ have the same Cartesian trees (i.e., $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$). There are two cases.

- If $S_1[k + 1]$ and $S_2[k + 1]$ are not roots of the Cartesian trees, let $S_1[j]$ be the parent of $S_1[k+1]$, and $S_2[l]$ the parent of $S_2[k+1]$. Since $CT(S_1[1..k+1]) = CT(S_2[1..k+1])$, we have $j = l$. If we remove $S_1[k+1]$ from Cartesian tree $CT(S_1[1..k + 1])$, we obtain the tree $CT(S_1[1..k])$, where the left subtree of $S_1[k + 1]$ is attached to its parent $S_1[j]$. If we remove $S_2[k + 1]$

from $CT(S_2[1..k+1])$, we obtain $CT(S_2[1..k])$ in the same way. Since $CT(S_1[1..k+1]) = CT(S_2[1..k+1])$, we get $CT(S_1[1..k]) = CT(S_2[1..k])$, and therefore $PD(S_1)[1..k] = PD(S_2)[1..k]$ by induction hypothesis. Since $PD(S_1)[k+1] = k+1-j$ and $PD(S_2)[k+1] = k+1-l$ (and $j = l$), we have $PD(S_1) = PD(S_2)$.

- If $S_1[k+1]$ and $S_2[k+1]$ are roots, we remove $S_1[k+1]$ and $S_2[k+1]$ to get $CT(S_1[1..k])$ and $CT(S_2[1..k])$. Since $CT(S_1[1..k+1]) = CT(S_2[1..k+1])$, we have $CT(S_1[1..k]) = CT(S_2[1..k])$, and therefore $PD(S_1)[1..k] = PD(S_2)[1..k]$ by induction hypothesis. Since $PD(S_1)[k+1] = PD(S_2)[k+1] = 0$ in this case, we get $PD(S_1) = PD(S_2)$.

($\Longleftarrow$) Assume that $S_1[1..k+1]$ and $S_2[1..k+1]$ have the same parent-distance representations (i.e., $PD(S_1)[1..k+1] = PD(S_2)[1..k+1]$). Since $PD(S_1)[1..k] = PD(S_2)[1..k]$, we have $CT(S_1[1..k]) = CT(S_2[1..k])$ by induction hypothesis. From $CT(S_1[1..k])$, we can derive $CT(S_1[1..k+1])$ as follows. If $PD(S_1)[k+1] > 0$, let x be $S_1[k+1-PD(S_1)[k+1]]$. We insert $S_1[k+1]$ into $CT(S_1[1..k])$ so that the parent of $S_1[k+1]$ is x and the original right subtree of x becomes the left subtree of $S_1[k+1]$. If $PD(S_1)[k+1] = 0$, $S_1[k+1]$ is the root of $CT(S_1[1..k+1])$ and $CT(S_1[1..k])$ becomes the left subtree of $S_1[k+1]$. We derive $CT(S_2[1..k+1])$ from $CT(S_2[1..k])$ in the same way. Since $CT(S_1[1..k]) = CT(S_2[1..k])$ and $PD(S_1)[k+1] = PD(S_2)[k+1]$, we can conclude that $CT(S_1[1..k+1]) = CT(S_2[1..k+1])$.

Therefore, we have proved that there is a one-to-one mapping between Cartesian trees and parent-distance representations. □

3.2 Computing parent-distance representation

Given a string $S[1..n]$, we can compute the parent-distance representation in linear time using a stack, as in [13, 14]. The main idea is that if two characters $S[i]$ and $S[j]$ for $i < j$ satisfy $S[i] > S[j]$, $S[i]$ cannot be the parent of $S[k]$

Algorithm 1 Computing parent-distance representation of a string

```
1:  procedure PARENT-DIST-REP(S[1..n])
2:      ST ← an empty stack
3:      for i ← 1 to n do
4:          while ST is not empty do
5:              (value, index) ← ST.top
6:              if value ≤ S[i] then
7:                  break
8:              ST.pop
9:          if ST is empty then
10:             PD(S)[i] ← 0
11:         else
12:             PD(S)[i] ← i − index
13:         ST.push((S[i], i))
14:     return PD(S)
```

for any $k > j$. Therefore, we will only store $S[i]$ which does not have such $S[j]$ while scanning from left to right. If we store such $S[i]$ only, they form a non-decreasing subsequence of S. When we consider a new value, therefore, we can pop values that are larger than the new value, find its parent, and push the new value and its index into the stack. Algorithm 1 describes the algorithm to compute $PD(S)$.

Furthermore, given the parent-distance representation of string S, we can compute the parent-distance representation of any substring $S[i..j]$ easily. To compute $PD(S[i..j])[k]$, we need only check whether the parent of $S[i+k-1]$ is within $S[i..j]$ or not (i.e., the parent is outside if $PD(S)[i+k-1] \geq k$).

$$PD(S[i..j])[k] = \begin{cases} 0 & \text{if } PD(S)[i+k-1] \geq k \\ PD(S)[i+k-1] & \text{otherwise.} \end{cases} \tag{3.1}$$

For example, the parent-distance representation of string $S = (2, 7, 5, 6, 4, 3, 1)$ is $PD(S) = (0, 1, 2, 1, 4, 5, 0)$. For $PD(S[2..7])$, we can use the above equation and compute the value at each position in constant time, getting $PD(S[2..7]) = (0, 0, 1, 0, 0, 0)$.

By using Theorem 3.1 and Equation 3.1, we can show that Cartesian tree matching satisfies the properties of a substring consistent equivalence relation.

Theorem 3.2. Cartesian tree matching is a substring consistent equivalence relation.

Proof. Given two strings S_1 and S_2, let's assume that $S_1 \approx S_2$. Clearly, S_1 and S_2 have a same length n. Furthermore, by Theorem 3.1, $PD(S_1) = PD(S_2)$ holds. By Equation (3.1), for every $1 \leq i \leq j \leq n$ and $1 \leq k \leq j - i + 1$, $PD(S_1[i..j])[k] = PD(S_2[i..j])[k]$ holds. Therefore, for every $1 \leq i \leq j \leq n$, $PD(S_1[i..j]) = PD(S_2[i..j])$ holds, and thus we have $S_1[i..j] \approx S_2[i..j]$. □

3.3 Failure function

We can define a failure function similar to the one used in the KMP algorithm [22].

Definition 3.2. (Failure function) The *failure function* π of string P is an integer string such that:

$$\pi[q] = \begin{cases} \max\{k : CT(P[1..k]) = CT(P[q-k+1..q]) \text{ for } 1 \leq k < q\} & \text{if } q > 1 \\ 0 & \text{if } q = 1 \end{cases}$$

That is, $\pi[q]$ is the largest k such that the prefix and the suffix of $P[1..q]$ of length k have the same Cartesian trees. For example, assuming that $P =$

$(5, 7, 4, 6, 1, 3, 2)$, the corresponding failure function is $\pi = (0, 1, 1, 2, 3, 4, 1)$. We can see that $CT(P[1..4]) = CT(P[3..6])$ from $\pi[6] = 4$. We will present an algorithm to compute the failure function of a given string in Section 3.5.

Algorithm 2 Text search of Cartesian tree matching

```
1: procedure CARTESIAN-TREE-MATCH(T[1..n], P[1..m])
2:     PD(P) ← PARENT-DIST-REP(P)
3:     π ← FAILURE-FUNC(P)
4:     len ← 0
5:     DQ ← an empty deque
6:     for i ← 1 to n do
7:         Pop elements (value, index) from back of DQ such that value > T[i]
8:         while len ≠ 0 do
9:             if PD(T[i − len..i])[len + 1] = PD(P)[len + 1] then
10:                break
11:            else
12:                len ← π[len]
13:                Delete elements (value, index) from front of DQ such that
   index < i − len
14:        len ← len + 1
15:        DQ.push_back((T[i], i))
16:        if len = m then
17:            print "Match occurred at i − m + 1"
18:            len ← π[len]
19:            Delete elements (value, index) from front of DQ such that
   index ≤ i − len
```

3.4 Text search

As in the original KMP text search algorithm, we can use the failure function in order to achieve linear time text search: scan the text from left to right, and use the failure function every time we find a mismatch between the text and the pattern. We apply this idea to Cartesian tree matching.

In order to perform a text search using $O(m)$ space, we compute the parent-distance representation of the text *online* as we read the text, so that we don't need to store the parent-distance representation of the whole text, which would cost $O(n)$ space. Furthermore, among the text characters which are matched with the pattern, we only have to store elements that form a non-decreasing subsequence by using a *deque* (instead of a stack in Section 3.2) in order to delete elements in front. Using this idea, we can keep the size of the deque to be always smaller than or equal to m. Therefore, we can perform the text search using $O(m)$ space. Algorithm 2 shows the text search algorithm of Cartesian tree matching. In line 9 we need to compute $x = PD(T[i - len..i])[len + 1]$. If the deque is empty, then $x = 0$. Otherwise, let $(value, index)$ be the element at the back of the deque. Then $x = i - index$. This computation takes constant time. Just before line 14, we do not compare $PD(T[i])$ and $PD(P)[1]$ when $len = 0$, because they always match. Therefore, we can safely perform line 14.

3.5 Computing failure function

We compute the failure function π in a way similar to the text search, as in the KMP algorithm. However, we can compute the parent-distance representation of the pattern in $O(m)$ time before we compute the failure function. Hence we don't need a deque and the computation is slightly simpler than text search. Algorithm 3 shows the procedure to compute the failure function.

Algorithm 3 Computing failure function in Cartesian tree matching

```
1:  procedure FAILURE-FUNC(P[1..m])
2:      PD(P) ← PARENT-DIST-REP(P)
3:      len ← 0
4:      π[1] ← 0
5:      for i ← 2 to m do
6:          while len ≠ 0 do
7:              if PD(P[i − len..i])[len + 1] = PD(P[1..len + 1])[len + 1] then
8:                  break
9:              else
10:                 len ← π[len]
11:         len ← len + 1
12:         π[i] ← len
```

3.6 Correctness and time complexity

Since our algorithm for Cartesian tree matching including text search and the computation of the failure function follow the KMP algorithm, it is easy to see that our algorithm correctly finds all occurrences (in the sense of Cartesian tree matching) of the pattern in the text. Since our algorithm checks one character of the parent-distance representation in constant time, it takes $O(n)$ time for text search and $O(m)$ time to compute the failure function, as in KMP algorithm.

Therefore, our algorithm requires $O(m+n)$ time for Cartesian tree matching using $O(m)$ space.

Theorem 3.3. Given two strings text $T[1..n]$ and pattern $P[1..m]$, single pattern Cartesian tree matching can be done in $O(n+m)$ time using $O(m)$ space.

3.7 Cartesian tree signature

There is an alternative representation of Cartesian trees, called *Cartesian tree signature* [14]. The Cartesian tree signature of $S[1..n]$ is an array $L[1..n]$ such that $L[i]$ equals the number of the elements popped from the stack in the i-th iteration of Algorithm 1. Furthermore, the Cartesian tree signature can be represented as a bit string $1^{L[1]}01^{L[2]}0 \cdots 1^{L[n]}0$ of length less than $2n$, which is a succinct representation of a Cartesian tree. For example, the Cartesian tree signature of string $S = (2, 7, 5, 6, 4, 3, 1)$ is $L = (0, 0, 1, 0, 2, 1, 2)$, and its corresponding bit string is 0010011010110.

We can use this representation to perform Cartesian tree matching. While we compute the Cartesian tree signature, we store one more array $D[1..n]$, which is defined as follows: If $S[i]$ is never popped out from the stack, $D[i] = 0$. Otherwise, let $S[j]$ be the value which popped $S[i]$ out from the stack, and $D[i] = j - i$. For string $S = (2, 7, 5, 6, 4, 3, 1)$, we have $D = (6, 1, 2, 1, 1, 1, 0)$.

Using array D, we can delete one character at the front of string $S[1..n]$ in constant time. In order to get Cartesian tree signature L' and its corresponding D' for $S[2..n]$, we do the following: If $D[1] > 0$, we decrease $L[D[1] + 1]$ by one and erase $L[1]$ from L. If $D[1] = 0$, we just erase $L[1]$. After that, we delete $D[1]$ from D to get D'. For example, if we want to delete one character at the front of $S = (2, 7, 5, 6, 4, 3, 1)$, we decrease $L[D[1] + 1] = L[7]$ by one, and delete $L[1]$ and $D[1]$. This results in $L' = (0, 1, 0, 2, 1, 1)$ and $D' = (1, 2, 1, 1, 1, 0)$. These arrays are the correct Cartesian tree signature and its corresponding array D of $S[2..7] = (7, 5, 6, 4, 3, 1)$. In this way, we can perform Algorithm 2 using the Cartesian tree signature. Computing the failure function can also be done in a similar way.

Note that the Cartesian tree signature can represent a Cartesian tree using less space than the parent-distance representation, but it needs an auxiliary array D to perform string matching, which uses the same space as the parent-

distance representation. For Cartesian tree matching, therefore, it uses more space than Algorithm 2.

Chapter 4

Multiple Pattern Matching in $O((n+m)\log k)$ Time

In this chapter we extend Cartesian tree matching to the case of multiple patterns. Definition 4.1 gives the formal definition of multiple pattern matching.

Definition 4.1. (Multiple pattern Cartesian tree matching) Given a text $T[1..n]$ and patterns $P_1[1..m_1], P_2[1..m_2], ..., P_k[1..m_k]$, where $m = m_1 + m_2 + \cdots + m_k$, *multiple pattern Cartesian tree matching* is to find every position in the text which matches at least one pattern, i.e., it has the same Cartesian tree as that of at least one pattern.

We modify the Aho-Corasick algorithm [1] using the parent-distance representation defined in Section 3.1 to do multiple pattern matching in $O((n + m)\log k)$ time.

4.1 Constructing the Aho-Corasick automaton

Instead of using the patterns themselves in the Aho-Corasick automaton, we use their parent-distance representations to make an automaton. Each node in the automaton corresponds to the prefix of the parent-distance representation

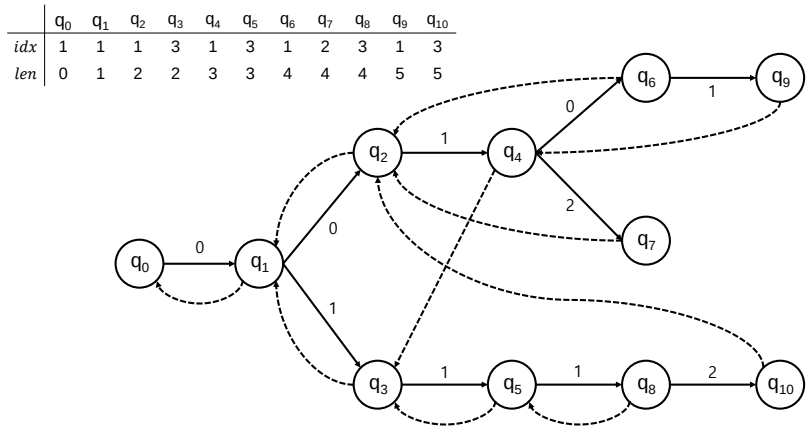


Figure 4.1 Aho-Corasick automaton for $P_1 = (4, 2, 3, 1, 5), P_2 = (3, 1, 4, 2), P_3 = (1, 2, 3, 5, 4)$

of some pattern. We maintain two integers idx and len for every node such that the node corresponds to the parent-distance representation of the pattern prefix $P_{idx}[1..len]$. If there are more than one possible indexes, we store the smallest one. Each node also has a state transition function $trans(x)$, which gets an integer x as an input and returns the next node, or report that there is no such node. We can construct the trie and the state transition function for every node in $O(m \log k)$ time, assuming that we use a balanced binary search tree to implement the transition function. Figure 4.1 shows an automaton for three patterns $P_1 = (4, 2, 3, 1, 5), P_2 = (3, 1, 4, 2), P_3 = (1, 2, 3, 5, 4)$, where we use the parent-distance representations of the patterns, $PD(P_1) = (0, 0, 1, 0, 1), PD(P_2) = (0, 0, 1, 2), PD(P_3) = (0, 1, 1, 1, 2)$ to construct the automaton.

The failure function π of the Aho-Corasick automaton is defined as follows: Let q_i be a node in the automaton, and s_i be the substring that node q_i represents in the trie. Let s_j be the longest proper suffix of s_i which matches (in the sense of Cartesian tree matching) prefix s_k of some pattern P_k. The failure

Algorithm 4 Computing failure function in multiple pattern matching

```
1:  procedure MULTIPLE-FAILURE-FUNC(P_1, P_2, ..., P_k)
2:      for i ← 1 to k do
3:          PD(P_i) ← PARENT-DIST-REP(P_i)
4:      TR ← Build trie with PD(P_i)'s
5:      for node ← breadth-first traversal of the trie do
6:          len ← len[node]
7:          idx ← idx[node]
8:          π[node] ← TR.root
9:          ptr ← parent of node in the trie
10:         while ptr ≠ TR.root do
11:             ptr ← π[ptr]
12:             plen ← len[ptr]
13:             x ← PD(P_idx[len − plen..len])[plen + 1]
14:             if ptr.trans(x) exists then
15:                 π[node] ← ptr.trans(x)
16:                 break
```

function of q_i is defined as node q_k (i.e., $\pi[q_i] = q_k$).

Example 4.1. A dotted line in Figure 4.1 shows the failure function of each node. For instance, node q_7 represents $P_2[1..4]$, and its failure function q_2 represents $P_2[1..2]$. We can see that $P_2[1..2]$ matches $P_2[3..4]$ (i.e., $PD(P_2[1..2]) = PD(P_2[3..4]) = (0, 0)$), which is the longest proper suffix of $P_2[1..4]$ that matches a prefix of some pattern. Note that the parent-distance representation of s_k may not be the suffix of the parent-distance representation of s_i. For example, q_7 has the parent-distance representation $(0, 0, 1, 2)$, but its failure function q_2 has the parent-distance representation $(0, 0)$ which is not a suffix of $(0, 0, 1, 2)$.

Algorithm 4 computes the failure function of the trie. As in the original Aho-Corasick algorithm, we traverse the trie with breadth-first order (except

the root) and compute the failure function. The main difference between Algorithm 4 and the Aho-Corasick algorithm is at line 13, where we decide the next character to match. According to the definition of the trie, *node* corresponds to the parent-distance representation of $P_{idx}[1..len]$, and so the parent of *node* corresponds to the parent-distance representation of $P_{idx}[1..len-1]$. In the while loop from line 10 to 16, *ptr* corresponds to the parent-distance representation of some suffix of $P_{idx}[1..len-1]$, because *ptr* is a node that can be reached from the parent of *node* following the failure links. Since *ptr* corresponds to some string of length *plen*, we can conclude that *ptr* represents $P_{idx}[len-plen..len-1]$. We want to check whether $P_{idx}[len-plen..len]$ matches some node in the trie, so we should check whether *ptr* has the transition using $x = PD(P_{idx}[len-plen..len])[plen+1]$. If *ptr* has the transition $ptr.trans(x)$, it corresponds to $P_{idx}[len-plen..len]$, and we can conclude that $\pi[node] = ptr.trans(x)$. If *ptr* doesn't have such a transition, there is no node that represents $P_{idx}[len-plen..len]$, and thus we have to continue the loop.

Example 4.2. Suppose that we compute the failure function of q_7 in Figure 4.1. From $idx[q_7] = 2$ and $len[q_7] = 4$, we know that q_7 represents $P_2[1..4]$, and so q_4, which is the parent of q_7, represents $P_2[1..3]$. We begin the while loop starting from $ptr = \pi[q_4] = q_3$. Since $len[q_3] = 2$, we know that q_3, which represents $P_3[1..2]$, matches $P_2[2..3]$. In order to check whether $P_2[2..4]$ matches some node in the trie, we compute $x = PD(P_2[2..4])[3] = 2$ and check whether $q_3.trans(x)$ exists. Since there is no such transition, we continue the while loop with $ptr = \pi[q_3] = q_1$. We know that q_1, which represents $P_1[1..1]$, matches $P_2[3..3]$ from $len[q_1] = 1$. In order to check whether $P_2[3..4]$ matches some node, we compute $x = PD(P_2[3..4])[2] = 0$ and check whether $q_1.trans(x)$ exists. Since there is such a transition, we conclude that $\pi[q_7] = q_1.trans(0) = q_2$. Note that x may change during the while loop, which is not the case in the Aho-Corasick algorithm.

While computing the failure function, we can also compute the output function in the same way as the Aho-Corasick algorithm. The output function of node q_i is the set of patterns which match some suffix of s_i. This function is used to output all possible matches at the node.

4.2 Multiple pattern matching

Using the automaton defined above, we can solve multiple pattern Cartesian tree matching in $O(n \log k)$ time. The text search algorithm is essentially the same as that of the Aho-Corasick algorithm, following the trie and using the failure links in case of any mismatches. As in the single pattern case, we compute the parent-distance representation of the text online in the same way as Algorithm 2 (using a deque) to ensure $O(m)$ space. The time complexity of our multiple pattern Cartesian tree matching is $O((n+m) \log k)$ using $O(m)$ space, where the $\log k$ factor is included due to the binary search tree in each node. Since there can be at most k outgoing edges from each node, we can perform an operation in the binary search tree in $O(\log k)$ time. Combined with the time-complexity analysis of the Aho-Corasick algorithm, this shows that our algorithm has the time complexity of $O((n + m) \log k)$. Furthermore, by using the output function, we can report all the matches in $O((n + m) \log k + occ)$ time, where occ is a total number of occurrences of the patterns. We can reduce the time complexity further to randomized $O(n + m)$ time by using a hash instead of a binary search tree [12].

Theorem 4.1. Given a text $T[1..n]$ and patterns $P_1[1..m_1], P_2[1..m_2], ..., P_k[1..m_k]$, where $m = m_1 + m_2 + \cdots + m_k$, multiple pattern Cartesian tree matching can be done in worst-case $O((n+m) \log k)$ time or randomized $O(n+m)$ time using $O(m)$ space.

Chapter 5

Cartesian Suffix Tree in Randomized $O(n)$ Time

In this chapter we apply the notion of Cartesian tree matching to the suffix tree as in the cases of parameterized matching and order-preserving matching [8, 13]. We first define the Cartesian suffix tree, and show that it can be built in randomized $O(n)$ time or worst-case $O(n \log n)$ time using the result from Cole and Hariharan [12], where n is the length of the text.

5.1 Defining Cartesian suffix tree

The Cartesian suffix tree is an index data structure that allows us to find an occurrence of a given pattern $P[1..m]$ in randomized $O(m)$ time or worst-case $O(m \log n)$ time. In order to store the information of Cartesian suffix trees efficiently, we again use the parent-distance representation from Section 3.1. Definition 5.1 gives the formal definition of the Cartesian suffix tree.

Definition 5.1. (Cartesian suffix tree) Given a string $T[1..n]$, the *Cartesian suffix tree* of T is a compacted trie built with $PD(T[i..n]) \cdot (-1)$ for every $1 \leq i \leq n$ (where the special character -1 is concatenated to the end of

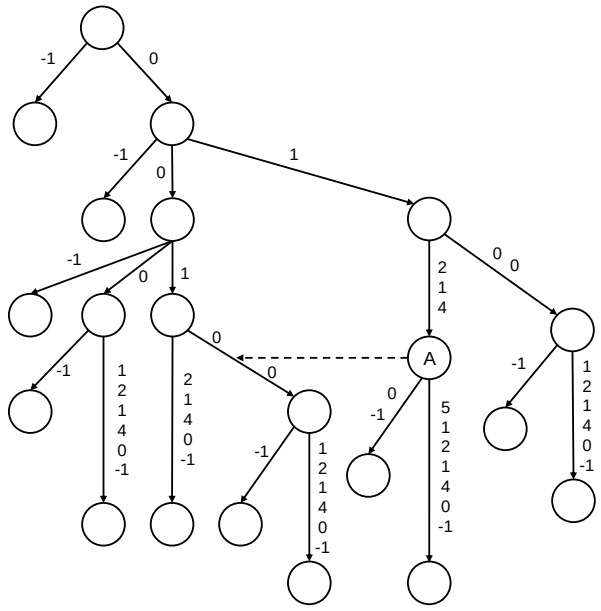


Figure 5.1 Cartesian suffix tree of $S = (2, 7, 5, 6, 4, 3, 11, 9, 10, 8, 1)$

$PD(T[i..n])$) and string (-1).

Note that we append a special character -1 to the end of each parent-distance representation to ensure that no string is a prefix of another string.

Figure 5.1 shows an example Cartesian suffix tree of $T = (2, 7, 5, 6, 4, 3, 11, 9, 10, 8, 1)$. Each edge actually stores the suffix number, start position, and end position instead of the parent-distance representation itself. For example, node A corresponds to substring $T[1..5]$ or $T[6..10]$, whose parent-distance representation is $PD(T[1..5]) = PD(T[6..10]) = (0, 1, 2, 1, 4)$. Hence, the edge that goes into node A stores the suffix number 1 or 6, start and end positions 3 and 5.

5.2 Constructing Cartesian suffix tree

There are several algorithms efficiently constructing the suffix tree, such as McCreight's algorithm [26] and Ukkonen's algorithm [28]. However, the *distinct right context property* [17, 8] should hold in order to apply these algorithms, which is defined as follows.

Definition 5.2. (Distinct right context property) For every internal node v of a suffix tree T which represents $T[i..j]$, there exists a node in T that represents $T[i+1..j]$. (The suffix link of v is defined as the node that represents $T[i+1..j]$.)

The Cartesian suffix tree does not have the distinct right context property. In Figure 5.1, the internal node marked with A does not satisfy this property because $PD(T[2..6]) = PD(T[7..11]) = (0, 0, 1, 0, 0)$ and thus there is no explicit node corresponding to parent-distance representation $(0, 0, 1, 0)$.

In order to handle this issue, we use an algorithm due to Cole and Hariharan [12]. This algorithm can construct a compacted trie for a *quasi-suffix collection*, which satisfies the following properties:

1. A quasi-suffix collection is a set of n strings $s_1, s_2, ..., s_n$, where the length of s_i is $n + 1 - i$.

2. For any two different strings s_i and s_j, s_i should not be a prefix of s_j.

3. For any i and j, if s_i and s_j have a common prefix of length l, s_{i+1} and s_{j+1} should have a common prefix of length at least $l - 1$.

A collection of parent-distance representations for the Cartesian suffix tree satisfies all of the above properties. The first two properties are trivial. Furthermore, if $s_i = PD(T[i..n]) \cdot (-1)$ and $s_j = PD(T[j..n]) \cdot (-1)$ have a common prefix of length l, i.e., $PD(T[i..i + l - 1]) = PD(T[j..j + l - 1])$, we can show that $PD(T[i+1..i+l-1]) = PD(T[j+1..j+l-1])$ by Equation 3.1. Therefore, $s_{i+1} = PD(T[i+1..n]) \cdot (-1)$ and $s_{j+1} = PD(T[j+1..n]) \cdot (-1)$ have a common prefix of length $l - 1$ or more, showing the third property holds.

One more property we need to perform Cole and Hariharan's algorithm is a *character oracle*, which returns the i-th character of s_j in constant time. We can do this in constant time using Equation 3.1, once the parent-distance representation of T is computed.

Since we have all properties needed to perform Cole and Hariharan's algorithm, we can construct a Cartesian suffix tree in randomized $O(n)$ time using $O(n)$ space [12]. In the worst case, it can be built in $O(n \log n)$ time by using a binary search tree instead of a hash table to store the children of each node in the suffix tree, because the alphabet size $|\Sigma|$ is $O(n)$. We can also modify our algorithm to construct a Cartesian suffix tree online, using the idea in [24, 27].

Theorem 5.1. Given a string $T[1..n]$, the Cartesian suffix tree of T can be built in randomized $O(n)$ time or worst-case $O(n \log n)$ time using $O(n)$ space.

Chapter 6

Conclusion

We have defined Cartesian tree matching and the parent-distance representation of a Cartesian tree. We developed a linear time algorithm for single pattern matching and an $O((n+m)\log k)$ deterministic time or $O(n+m)$ randomized time algorithm for multiple pattern matching. Finally, we defined an index data structure called Cartesian suffix tree, and showed that it can be constructed in $O(n)$ randomized time. We believe that the notion of Cartesian tree matching, which is a new metric on string matching and indexing over numeric strings, can be used in many applications.

There have been many works on approximate generalized matching. For example, there are results for approximate order-preserving matching [11], approximate jumble matching [10], approximate swapped matching [5], and approximate parameterized matching [6, 19]. There are also results on computing the period of a generalized string, such as computing the period in the order-preserving model [18]. Many problems including approximate matching and computing the period in the Cartesian tree matching model are future research topics.

Bibliography

[1] A. V. Aho and M. J. Corasick, Efficient String Matching: An aid to bibliographic search. Communications of the ACM, 18(6) (1975) 333–340.

[2] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. Journal of Algorithms, 37(2) (2000) 247–266.

[3] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. Information and Computation, 181(1) (2003) 57–74.

[4] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. Information Processing Letters, 49(3) (1994) 111–115.

[5] A. Amir, M. Lewenstein, and E. Porat. Approximate swapped matching. Information Processing Letters, 83(1) (2002) 33-39.

[6] A. Apostolico, P. L. Erdos, and M. Lewenstein. Parameterized matching with mismatches. Journal of Discrete Algorithms, 5(1) (2007) 135–140.

[7] B. S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, 1993, pp. 71-80.

[8] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM Journal on Computing, 26(5) (1997) 1343–1362.

[9] P. Burcsi, F. Cicalese, G. Fici, and Z. Liptak. Algorithms for jumbled pattern matching in strings. International Journal of Foundations of Computer Science, 23(2) (2012) 357–374.

[10] P. Burcsi, F. Cicalese, G. Fici, and Z. Liptak. On approximate jumbled pattern matching in strings. Theory of Computing Systems, 50(1) (2012) 35-51.

[11] T. Chhabra, E. Giaquinta, and J. Tarhio. Filtration algorithms for approximate order-preserving matching. In: International Symposium on String Processing and Information Retrieval, 2015, pp. 177-187.

[12] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. SIAM Journal on Computing, 33(1) (2003) 26-42.

[13] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving indexing. Theoretical Computer Science, 638 (2016) 122–135.

[14] E. D. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. Algorithmica, 68(3) (2014) 610–625.

[15] T. Fu, F. Chung, R. Luk, and C. Ng. Stock time series pattern matching: Template-based vs. rule-based approaches. Engineering Applications of Artificial Intelligence, 20(3) (2007) 347–364.

[16] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, 1984, pp. 135-143.

[17] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. SIAM Journal on Computing, 24(3) (1995) 520–562.

[18] G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. M. Shur, and T. Walen. String periods in the order-preserving model. Information and Computation, 270 (2020) 104463.

[19] C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. ACM Transactions on Algorithms, 3(3) (2007) Article 29.

[20] J. Kim, A. Amir, J. C. Na, K. Park, and J. S. Sim. On representations of ternary order relations in numeric strings. Mathematics in Computer Science, 11(2) (2017) 127–136.

[21] J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. Theoretical Computer Science, 525 (2014) 68-79.

[22] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2) (1977) 323–350.

[23] M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters, 113(12) (2013) 430–433.

[24] T. Lee, J. C. Na, and K. Park. On-line construction of parameterized suffix trees for large alphabets. Information Processing Letters, 111(5) (2011) 201–207.

[25] Y. Matsuoka, T. Aoki, S. Inenaga, H. Bannai, and M. Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations, Theoretical Computer Science, 656 (2016) 225-233.

[26] E. M. McCreight. A space-economical suffix tree construction algorithm. Journal of the ACM, 23(2) (1976) 262–272.

[27] J. C. Na, R. Giancarlo, and K. Park. On-line construction of two dimensional suffix trees in $O(n^2 \log n)$ time. Algorithmica, 48(2) (2007) 173–186.

[28] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3) (1995) 249–260.

[29] J. Vuillemin. A unifying look at data structures. Communications of the ACM, 23(4) (1980) 229–239.

요약

본 논문에서는 Cartesian 트리에 기반한 새로운 매칭 기준인 Cartesian 트리 매칭을 제안한다. 이는 두 문자열의 Cartesian 트리가 서로 같을 때, 두 문자열을 매칭된 것으로 정의하는 문제이다.

Cartesian 트리 매칭의 기준 하에서, 본 연구에서는 길이 n인 텍스트와 길이 m인 패턴 사이의 단일패턴매칭 문제와 길이 n인 텍스트와 길이의 합이 m인 여러 개의 패턴 사이의 다중패턴매칭 문제를 정의하고, 단일패턴매칭 문제를 해결하는 $O(n+m)$ 시간 알고리즘과 다중패턴매칭 문제를 해결하는 $O((n+m)\log k)$ 시간 결정론적 알고리즘 및 $O(n+m)$ 시간 무작위 알고리즘을 제시한다. 또한, Cartesian 트리 매칭에 대한 인덱스 자료구조인 Cartesian 접미사트리를 정의하고, 이를 구축하는 $O(n)$ 시간 무작위 알고리즘을 제시한다.

본 논문에서는 Cartesian tree를 표현하는 방식인 부모거리표현 (parent-distance representation)을 정의하고, 이를 이용하여 위 문제들을 해결하는 효율적인 알고리즘들을 제시한다.

주요어: Cartesian 트리 매칭, 패턴매칭, 인덱싱, 부모거리표현

학번: 2018-26744