A Linear Time Pattern Matching Algorithm Between a String and a Tree

Tatsuya AKUTSU *

Mechanical Engineering Laboratory,
1-2 Namiki, Tsukuba, Ibaraki, 305 Japan.

Abstract. In this paper, we describe a linear time algorithm for testing whether or not there is a path of a tree $T (|V(T)| = n)$ that coincides with a string $s (|s| = m)$. In the algorithm, $O(n/m)$ vertices are selected from $V(T)$ such that any path of length more than $m - 2$ must contain at least one of the selected vertices. A search is performed using the selected vertices as 'bases.' A suffix tree is used effectively in the algorithm. Although the size of the alphabet is assumed to be bounded by a constant in this paper, the algorithm can be applied to the case of unbounded alphabets by increasing the time complexity to $O(n \log n)$.

Keywords: subtree, subgraph isomorphism, string matching, suffix tree, graph algorithms

1 Introduction

The subgraph isomorphism problem is famous and important in computer science. It is the problem of testing whether or not there is a subgraph of $T$ isomorphic to $S$ when the graphs of $S$ and $T$ are given. It is important for practical applications as well. In particular, many heuristic algorithms have been developed for database systems in chemistry [13, 14].

In general, the problem was proved to be NP-complete [6]. However, polynomial time algorithms have been developed in special cases [10, 12]. When the graphs are simple paths, the problem is reduced to the string matching problem, for which several linear time algorithms have been developed [3, 7]. When the graphs are restricted to trees, the problem is solved in $O(n^{2.5})$ time [4]. Moreover, if the vertex degree of two input trees is bounded by a constant, the problem is solved in $O(n^2)$ time. If the graphs are rooted trees such that the labels of children of each node are

* The author thanks to Prof. Tomio Hirata in Nagoya University for helpful comments. This research was partially supported by the Grand-in-Aid for Scientific Research on Priority Areas, "Genome Informatics", of the Ministry of Education, Science and Culture of Japan.
distinct, whether there is an \( o(n^2) \) time algorithm or not had been an open problem for a long time. However, it was solved confirmatively by Kosaraju [8] and the result was improved by Dubiner et.al. [5]. However, as far as we know, there is no \( o(n^2) \) time algorithm for undirected trees or rooted trees such that children of a node may have identical labels even if the vertex degree is bounded. In this paper, we show a linear time algorithm for a special case of the problem, that is, the case where \( S \) is a path and \( T \) is an undirected tree. Moreover, \( T \) may have a node such that adjacent vertices have identical labels.

In this paper, \( T \) denotes an input undirected tree with labeled vertices and \( s = s_1s_2 \ldots s_m \) denotes an input string of length \( m \). We do not assume that labels of vertices adjacent to the same vertex are different. Although vertices are assumed to be labeled, the result can also be applied to the case of labeled edges. For a graph \( G \), \( V(G) \) denotes the set of vertices and \( E(G) \) denotes the set of edges. \( n \) denotes the number of the vertices of the tree \( T \) (i.e., \( n = |V(T)| \)). For a vertex \( v \), \( label(v) \) denotes the label associated with \( v \). We assume that the size of the alphabet is bounded by a constant. The problem is to test whether or not there is a vertex disjoint path \( (v_1, v_2, \ldots, v_m) \) in \( T \) such that \( label(v_1)label(v_2) \ldots label(v_m) = s \).

This paper describes an \( O(n) \) time algorithm for this problem.

Of course, a rooted tree version of the problem, that is, the case where \( T \) is a rooted tree and only the paths which do not connect sibling nodes are allowed, is trivially solved in linear time [5] by using a linear time substring matching algorithm [3, 7] with backtracking. However, this method does not seem to work for the problem of this paper. The linear time algorithm which we developed here is based on a different idea: \( O(n/m) \) vertices are selected from \( V(T) \) such that any path of length more than \( m - 2 \) must contain at least one of the selected vertices. From each of the selected vertices, a search is performed with traversing the suffix tree associated with \( s \).

## 2 Suffix Tree

In this section, we give an overview of the well-known data structure, suffix tree [2, 11]. The suffix tree is used in on-line string matching for a large fixed text. Moreover, it is applied to a variety of pattern matching problems [1, 5, 9].

Let \( s = s_1s_2 \ldots s_m \) be a string. \( |s| \) denotes the length of \( s \). \( s_i \) denotes the suffix of \( s \) which starts from \( s^i \). \( s^{-1} \) denotes the reversed string of \( s \) and \( s_i^{-1} \) denotes \( (s^{-1})_i \). For a string \( s \) and a character \( x \), \( sx \) denotes the concatenation of \( s \) and "\( x \)". We assume without loss of generality that the special character '\#' does not appear in \( s \). The suffix tree \( SUF_s \) associated with \( s \) is the rooted tree with \( m \) leaves and at most \( m - 1 \) internal nodes such that

- Each edge is associated with a substring of \( s\# \).
- Sibling edges must have labels whose first character is distinct.
- Each leaf is associated with a distinct position of \( s \).
- The concatenation of the labels on the path from the root to a leaf \( l_i \) describes \( (s\#)_i \).

It is known that the size of a suffix tree is \( O(m) \) and a suffix tree can be constructed in \( O(m) \) time if the size of the alphabet is bounded by a constant [11]. It is easy to