RESEARCH ARTICLE
doi: 10.2306/scienceasia1513-1874.2019.45.065

ScienceAsia 45 (2019): **65–73**

A new heuristic for full trie minimization

Meng Zhang[a], **Dequan Chen**[a], **Yi Zhang**[b], **Guohang Song**[a,*]

[a] College of Computer Science and Technology, Jilin University, Changchun 130012 China
[b] College of Electronics and Computer Science, Jilin Jianzhu University, Changchun 130118 China

*Corresponding author, e-mail: songguohang1988@sina.com

Received 25 Feb 2018
Accepted 28 Feb 2019

ABSTRACT: Trie is a data structure with many applications. High space usage is the major drawback of the trie. The order-containing trie optimizes the space usage of the trie by rearranging symbols of strings. We present a new heuristic for building order-containing tries with small space. The heuristic is based on the following observation. For a given string set P, by moving the symbols on a position to the first position of strings in P, the trie of the resulting pattern set may have fewer nodes than that of the trie of P. We present an algorithm to find positions that yield the smallest such trie. The algorithm runs in $O(\|P\|)$ time and uses $O(|P|\log|P|)$ bits space, where $\|P\|$ is the number of total symbols in P, and $|P|$ is the number of patterns in P. By using this method recursively in trie constructions, we can build a trie with fewer nodes than the trie of P. We conduct several experiments that show the new heuristic builds smaller tires than previous work.

KEYWORDS: strings matching, compression, data structure, algorithm

MSC2010: 68W05 68W32

INTRODUCTION

The trie[1] data structure is used in many applications, including bioinformatics, pattern matching, and computer security. High space usage is the major drawback of the trie. Much research focused on reducing the space of tries. Patricia tree[2], also called path compressed trie, compresses the trie by deleting the nodes with one fan-out and labelling edges with a string, the number of nodes in the patricia tree of a string set $P = \{p_1, p_2, \ldots, p_k\}$ over an alphabet Σ is not greater than $2|P| + 1$. Aho-Corasick automaton[3] augments the trie with failure links. The multi-pattern matching can be solved in linear time and space by AC autotomata. Recently, Belazzougui et al[4] used succinct data structures to implement AC automata with small space without slowing down.

De Maine and Rotwitt[5] found that the order in which the characters are tested in building the trie has influences on the size of resulting tries. Using this idea, Comer and Sethi[6] defined the order-containing trie (O-Trie). In an O-Trie the characters are tested in different orders along different paths from the root to leaves. The O-Trie construction reads the patterns in a specific order other than in serial. The orders of testing the characters are contained in the trie nodes. Using alternative orders, the resulting O-Tries have some interesting features, such as with fewer number of nodes. Comer and Sethi[6] proved that to generate an O-Trie with the minimal number of nodes is NP-complete. Comer[7,8] presented heuristic methods to build small O-Tries and gave thorough analysis. These methods may produce a trie larger than the trie for the original pattern set.

We present a new heuristic for the minimal full trie problem. Our approach guarantees that the number of nodes of the resulting O-Trie is not greater than that of the trie. Our method has the same time complexity of the greedy method of Comer[7]. It is based on a solution of the following problem. Assume that the patterns in P are of the same length. By moving the symbol on the position i of each pattern to the first position, we have a new set of patterns, say $P^{(i)}$. The problem is to search for a position i such that the trie of $P^{(i)}$ has the smallest number of nodes. We present an algorithm that solves the problem in $O(\|P\|)$ time and $O(|P|\log|P|)$ bits space, where $\|P\|$ is the sum of lengths of all the patterns in P, and $|P|$ is the number of patterns in P. By applying the algorithm recursively to the trie construction, we can generate O-Tries such that the number of nodes is not greater that of the tries.

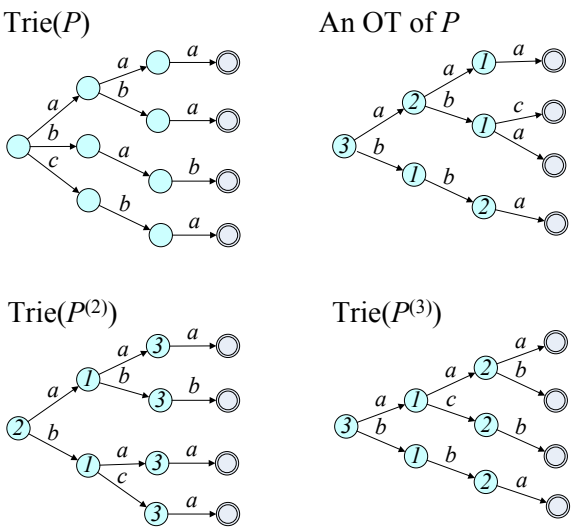

Fig. 1 Examples of an O-Trie and tries of $P = \{aaa, aba, bab, cba\}$, where terminal nodes are double circled. The number in a node q is Pos(q).

PRELIMINARIES

Let Σ denote the alphabet of $|\Sigma|$ symbols and ε the empty string. Let $x = x[1]x[2]\cdots x[l]$ be a string over Σ of length l. Given strings x and y, we say that x is a prefix of xy. Let Pref(P) denote the set of all prefixes of patterns in P. Define the trie of a pattern set as follows.

Definition 1 The trie of a string set P is a tree with edges labelled by a symbol in Σ, the set of nodes is $V = \text{Pref}(P)$, the set of edges is $E = \{(x, a, xa) \mid x, xa \in \text{Pref}(P),\ a \in \Sigma\}$. We use Trie($P$) to denote the trie of P.

The node-depth of a trie node u, denoted by depth(u), is the number of nodes on the path from the root to u. We assume that all strings in the dictionary are of the same length L. The trie nodes are classified into levels according to their depths, the root is in level 1, and children of a node in level l are in level $l+1$. The nodes corresponding to patterns in P are called terminal nodes. The subtree rooted at the node pointed by an a-edge from the root of a tree is called an a-subtree of the tree. Throughout the paper we use $\mathscr{T}$ to denote Trie(P). We denote by Size($\mathscr{T}$) the number of nodes in $\mathscr{T}$. An example of the trie is given in Fig. 1, the upper left tree.

ORDER-CONTAINING-TRIE

Definition 2 A fractional string is a string with some positions that have no symbol, called blank positions.

We use symbol $\square$ to represent a blank position, when $\square$ is not in Σ. A fractional string is thus a string over $\Sigma \cup \{\square\}$. Let p be a fractional string. If u and p are of the same size and for each position i of p, $u[i] = p[i]$ or $u[i] = \square$, we call u a fractional factor of p, or u matches p. Function facIns(u, i, c) sets the blank position i of u to $c \in \Sigma$ and returns the resulting fractional string. For examples, $a\square a$ and $aa\square$ are fractional strings, $a\square a$ matches aaa, and facIns($a\square a, 1, b$) $= aba$.

We introduce the O-Trie using the notion of the fractional string. The set of nodes of an O-Trie, say V, is a subset of fractional factors of patterns in P, the root q_0 is $\square^L$, a string with $\square$ repeats for L times. Each O-Trie involves a unique function Pos that is defined on each non-terminal node q of the O-Trie and returns a blank position of q; if there is no blank position in q, Pos(q) is undefined. For $q \in V$, we denote the set of patterns in P that match q by PatSet(q). The set of symbols on position Pos(q) in each PatSet(q) is denoted by $A(q)$. The set of edges starting from q is defined as

$$E(q) = \{(q, a, \text{facIns}(q, \text{Pos}(q), a)) \mid a \in A(q)\}.$$

The nodes corresponding to patterns in P are terminal. An example of the O-Trie is given in Fig. 1, the upper right trie. Let q be the node pointed by the a-edge from the root of this O-Trie. We have that $q = \square\square a$, PatSet(q) $= \{aaa, aba, cba\}$, and $A(q) = \{a, b\}$, and Pos(q) $= 2$.

An O-Trie is a tree. If not, assume that there exists a node such that there are two paths from the root to this node. Let i be the first position where the nodes are different in the two paths. According to the definition of O-Trie, the fractional string of the two nodes have one mismatch on a position. Thus descendants of the two nodes are all different, which contradicts the assumption.

For an edge (q, a, q') in an O-Trie, node q is the parent node of q', denoted by $q = \text{parent}(q')$. A trie can be viewed as an O-Trie such that Pos(q) $=$ depth(q). Comparing with a trie, O-Tries of the same pattern set may have different shapes and numbers of nodes. This difference is due to the Pos function of the O-Trie.

CONSTRUCTION OF O-TRIES

We give a construction algorithm of an O-Trie. The implementation of function Pos will be given later when we introduce an instance.

The following is the framework of the breadth-first construction algorithm of O-Tries, called BuildOT. The algorithm builds an O-Trie level by level. By definition, a node of an O-Trie corresponds to a fractional factor of some patterns in P. In generating level i nodes from nodes in level $i-1$, each level $i-1$ node is extended by adding edges and children. We select a blank position of the $i-1$ level node and fill the bank position with a letter to form a new fractional factor that is a child. In the algorithm, we use PosSet(s) to denote the set of blank positions of node s and PatSet(s) to denote the set of patterns that match s. Set A is the set of symbols on position pos of each PatSet(s), and $PS[a]$ is the set of patterns in PatSet(s) such that the symbol on the position pos is a.

Algorithm 1 Procedure BuildOT(P, L)
Step 1: Create a node q_0
Step 2: PatSet(q_0) $\leftarrow P$
Step 3: PosSet(q_0) $\leftarrow \{1,\ldots,L\}$
Step 4: Expand(q_0)

Algorithm 2 Procedure Expand(s)
Step 1: `if` PosSet(s) $=\emptyset$ `then` return
Step 2: $pos \leftarrow$ Pos(s)
Step 3: set all elements of PS to $\emptyset$, set A to $\emptyset$
Step 4: `for` each $p \in$ PatSet(s) `do`
  1: $a \leftarrow p[pos]$
  2: `if` $a \neq \varepsilon$ `then`
  add p to $PS[a]$
  `if` $a \notin A$ `then` Add a to A
Step 5: `for` each $a \in A$ `do`
  1: create a node s'
  2: create an edge (s, a, s')
  3: PatSet(s') $\leftarrow PS[a]$
  4: PosSet(s') $\leftarrow$ PosSet(s) $-\{pos\}$
  5: Expand(s')

Construction of O-Tries with fewer nodes

In this section, we present an instance of function Pos such that the corresponding O-Tries have fewer nodes than that of tries.

Let p be a string of length L. For $1 \leqslant l \leqslant L$, we construct a string u such that $u[1] = p[l]$, $u[j] = p[j-1]$ for $2 \leqslant j < l$, and $u[j] = p[j]$ for $j > l$. We denote u by $p^{(l)}$. For P, define $P^{(l)} = \{p_1^{(l)}, p_2^{(l)}, \ldots, p_k^{(l)}\}$.

We define function ML(P) such that for a pattern set P, ML(P) returns a level x of P such that Size(Trie($P^{(x)}$)) is the minimum among Size(Trie($P^{(l)}$)), $1 \leqslant l \leqslant L$.

We use ML to implement Pos in BuildOT(P) as follows. Given a node s, we can build a pattern set from PatSet(s) and PosSet(s). Let PatSet(s) $= \{ps_1, \ldots, ps_t\}$, PosSet(s) $= \{l_1, \ldots, l_c\}$, and $ps_i' = ps_i[l_1]ps_i[l_2]\cdots ps_i[l_c]$. We define the pattern set Pat(s) $= \{ps_1', \ldots, ps_t'\}$. We implement Pos($s$) by ML(Pat($s$)). According to the definition of ML we have the following result.

Theorem 1 *Let P be a pattern set. Let T be the trie generated by* BuildOT(P) *using* ML(Pat($\cdot$)) *as* Pos($\cdot$). *We have* Size(T) $\leqslant$ Size($\mathscr{T}$), *where $\mathscr{T}$ denotes* Trie(P).

When context is clear, we use $\mathscr{T}^l$ to denote Trie($P^{(l)}$). The time and space complexities of computing ML(P) by building each $\mathscr{T}^l$ are $O(\|P\|^2)$ and $O(\|P\|)$. We present here a faster algorithm that only scans P for one pass, which has time and space complexities $O(\|P\|)$ and $O(|P|)$, respectively. We first present a naive algorithm based on a relationship between $\mathscr{T}$ and $\mathscr{T}^l$, and then give a faster method.

A naive algorithm to compute function ML

Given $\mathscr{T}$, we construct the a-subtree of $\mathscr{T}^l$, $1 \leqslant l \leqslant L$, as follows.

Step 1: Mark all the l-level nodes that have an a-edge, and mark the subtrees rooted at the nodes pointed by these a-edges. Mark the paths from marked l-level nodes to the root.
Step 2: For each marked l-level node s, redirect the edge, which points to s, to the destination of the a-edge from s.
Step 3: Delete all marked l-level nodes and edges from them. Delete all un-marked nodes and edges.

Using this method, we generate the a-subtree of $\mathscr{T}^l$ for each symbol a to have $\mathscr{T}^l$, and the function ML is then computed. We give an illustration in Fig. 2.

Next, we give a faster method base on the followed observation. The differences between $\mathscr{T}$ and $\mathscr{T}^l$ lie in the first l levels, for the subgraphs of $\mathscr{T}$ and $\mathscr{T}^l$ from level $l+1$ to level L are isomorphic. Thus the difference of the number of nodes of $\mathscr{T}$ and $\mathscr{T}^l$ equals the difference of the number of nodes

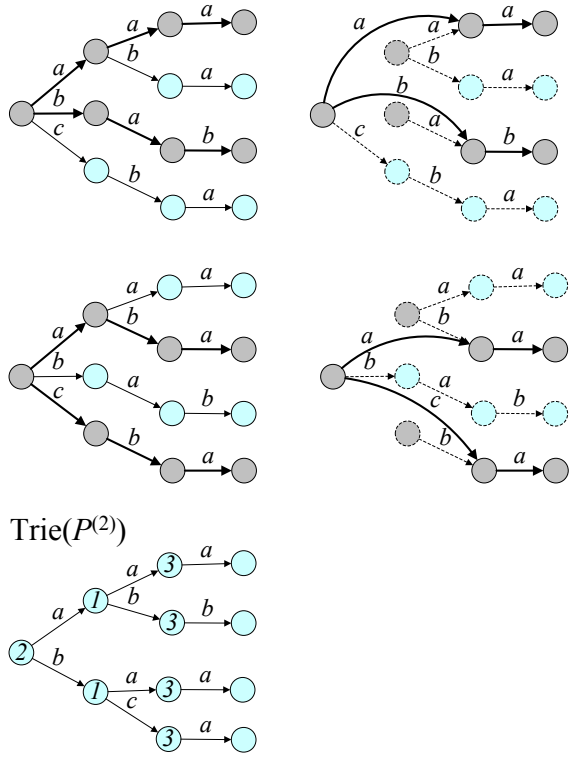

Fig. 2 The construction of Trie($P^{(2)}$). Dark nodes and edges are marked, and dotted ones are deleted. The trees in the first line are marked Trie(P)'s according to a and b. The trees in the second line are a-subtree and b-subtree of Trie($P^{(2)}$) generated from the marked tries. Trie($P^{(2)}$) is built from the two trees.

in the first l levels of $\mathscr{T}$ and $\mathscr{T}^l$. Define $\Delta(l) =$ Size($\mathscr{T}^l$) − Size($\mathscr{T}$). The minimal $\mathscr{T}^l$ corresponds to the minimal $\Delta(l)$. We count the number of nodes in the first l levels of $\mathscr{T}^l$ efficiently from $\mathscr{T}$ without building $\mathscr{T}^l$ explicitly. Given $\mathscr{T}$, the counting procedure CountT(l,a) is as the following.

Algorithm 3 Procedure CountT(l,a)
Step 1: Mark the root, $count \leftarrow 1$
Step 2: `for` each l-level node s that has a-edge `do`
`while` s is not marked `do`
1: mark s and the edge to s
2: $count \leftarrow count + 1$
3: $s \leftarrow$ parent(s)
Step 3: return $count$

CountT(l,a) first marks the root of $\mathscr{T}$. For each l-level node s that has an a-edge, it marks the path from s to the root and counts the newly marked nodes. In the procedure, after the `for` loop, the tree of the marked nodes and edges is isomorphic to the first l levels of the a-subtree of $\mathscr{T}^l$, and we know the number of nodes. We summarize the result as follows.

Proposition 1 *Given the trie $\mathscr{T}$ of P, procedure* CountT(l,a) *returns the number of nodes of the first l levels of a-subtree of $\mathscr{T}^l$.*

Proof: Each path from root to an l-level node with an a-edge spells out a prefix of length l, denote the set of such prefixes by B. Procedure CountT(l,a) marks the trie of B and returns the number of nodes of the trie. By appending a to the end of each string in B, we have the set of prefixes of P of length $l+1$ that end with a. Hence the trie of B is isomorphic to the first l levels of the a-subtree of $\mathscr{T}^l$. Thus procedure CountT(l,a) returns the number of nodes of the first l levels of the a-subtree of $\mathscr{T}^l$. □

Properties of trees used to implement ML

Procedure CountT works when $\mathscr{T}$ is given. To count the nodes in $\mathscr{T}^l$, many nodes are accessed redundantly by CountT(l,a) for many a's. We introduce a faster and space efficient algorithm to count $\mathscr{T}^l$. The method does not need $\mathscr{T}$ or backtracking. It generates at most $|P|$ nodes in the running and the time complexity for compute ML(P) is $O(\|P\|)$.

We first give some necessary notions. Let n_1 and n_2 be nodes in the l level of $\mathscr{T}$. We denote the least common ancestor (LCA) of n_1 and n_2 by $n = \text{LCA}(n_1,n_2)$. The encounter distance between n_1 and n_2 is defined as $D(n_1,n_2) = l - \text{depth}(n)$.

Definition 3 Define a partial order $<$ on the set of nodes of $\mathscr{T}$ as follows.
(i) If $n_1,\ldots,n_k$ are children of a node, then $n_1 < n_2 < \cdots < n_k$.
(ii) For nodes n, m in the same level, if parent(n) < parent(m), then $n < m$.
Let n, m be nodes in the same level and $n < m$. If there does not exist node n' such that $n < n' < m$, then n is called the predecessor of m, denoted by $n = \text{pre}(m)$.

The order in which the nodes are generated by BuildOT(P) satisfies Definition 3. The computation of the number of nodes of $\mathscr{T}^l$ and that of encounter distance are based on the following results on trees.

Proposition 2 *Let $n_1 < n_2 < n$ be nodes in the same level. We have $D(n,n_2) \leqslant D(n,n_1)$.*

Proof: If $\text{LCA}(n_1,n) = \text{LCA}(n_2,n) = q$, then $D(n,n_2) = D(n,n_1)$. Let $\text{LCA}(n_1,n) \neq \text{LCA}(n_2,n)$.

Then $\mathrm{LCA}(n_1,n_2)$ is either $\mathrm{LCA}(n_1,n)$ or $\mathrm{LCA}(n_2,n)$. Let $q_1 = \mathrm{LCA}(n,n_1)$ and $q_2 = \mathrm{LCA}(n,n_2)$. If $D(n,n_2) > D(n,n_1)$, then $q_2 = \mathrm{LCA}(n_1,n_2)$. Let q_2' be the node in the path from n_2 to m of the same depth of q_1. We have $q_1 < q_2'$, for $n_1 < n_2$. As q_1 is an ancestor of n and q_2' is an ancestor of n_2, we have $n < n_2$, which contradicts $n > n_2$. Hence $D(n,n_2) \leqslant D(n,n_1)$. □

The number of nodes in the first l levels of the a-subtree of $\mathscr{T}^l$ is computed by the encounter distances between adjacent nodes according to $<$ in the level l of the subtree.

Proposition 3 *Let $n_1 < n_2 \cdots < n_k$ be the set of l-level nodes that have an a-edge. The number of nodes in the first l levels of a-subtree of $\mathscr{T}^l$ is $l + \sum_{j=2}^{k} D(n_j, n_{j-1})$.*

Proof: In the running of $\mathrm{CountT}(l,a)$, we denote the tree of the marked nodes after the jth iteration by T_j. After backtracking n_1, we marked a path from n_1 to the root, say T_1, to have $\mathrm{Size}(T_1) = l$. After the tth iteration of $\mathrm{CountT}(l,a)$, we assume that

$$\begin{aligned}\mathrm{Size}(T_t) &= l + \sum_{j=2}^{t} D(n_j, n_{j-1}) \\ &= \mathrm{Size}(T_{t-1}) + D(n_{t-1}, n_t).\end{aligned}$$

In the $(t+1)$th iteration of $\mathrm{CountT}(l,a)$, let q be the first node in T_t in the path from n_{t+1} to the root. We have that q is the LCA of n_t and n_{t+1}, which is also the deepest among LCAs of n_j and n_{t+1}. If it is not the case, then there are some nodes on the path that is a part of T_t, contradicting the definition of q. By Proposition 2, $D(n_{t+1}, n_t)$ is the minimum among $\{D(n_{t+1}, n_j) \mid 1 \leqslant j \leqslant t\}$. Hence the number of nodes in the path that is not q is $D(n_{t+1}, n_t)$. Thus

$$\begin{aligned}\mathrm{Size}(T_{t+1}) &= \mathrm{Size}(T_t) + D(n_t, n_{t+1}) \\ &= l + \sum_{j=2}^{t+1} D(n_j, n_{j-1}).\end{aligned}$$

By Proposition 1, the number of nodes of the first l levels of $\mathscr{T}^l = l + \sum_{j=2}^{k} D(n_j, n_{j-1})$. □

The encounter distance between two nodes in the same level is computed as follows.

Proposition 4 *Let $n_1 < n_2 \cdots < n_c$ be the set of nodes in the same level. For $i < j$, we have $D(n_i,n_j) = \mathrm{Max}\{D(n_{t-1},n_t) \mid i < t \leqslant j\}$.*

Proof: For $j = i+1$, $D(n_i,n_j) = \mathrm{Max}\{D(n_{t-1},n_t) \mid i < t \leqslant j\}$. Assume that for $j > i+1$, $D(n_i,n_j) = \mathrm{Max}\{D(n_{t-1},n_t) \mid i < t \leqslant j\}$. If $D(n_i,n_j) = D(n_i,n_{j+1})$, then $D(n_i,n_{j+1}) = \mathrm{Max}\{D(n_{t-1},n_t) \mid i < t \leqslant j+1\}$. Assume that $D(n_i,n_j) < D(n_i,n_{j+1})$. Let $m = \mathrm{LCA}(n_i,n_{j+1})$. We have $\mathrm{LCA}(n_j,n_{j+1}) = m$. Hence we prove the proposition. □

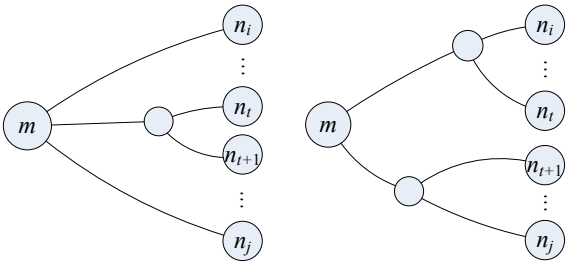

Fig. 3 There exist adjacent nodes n_t and n_{t+1}, where $i \leqslant t < j$ such that $\mathrm{LCA}(n_t,n_{t+1}) = \mathrm{LCA}(n_i,n_j)$.

An illustration of Proposition 4 is shown in Fig. 3.

Let $n_1 < n_2 \cdots < n_c$ be the set of l level nodes. Define DP^l as an array of k entries such that $DP^l[i] = \mathrm{LCA}(n_i,n_{i-1})$, $n_0 = n_1$. According to Proposition 4, the computation of $D(n_i,n_j) = l - \mathrm{depth}(\mathrm{LCA}(n_i,n_j))$, $i > j$, can be reduced to the problem of range minimum query (RMQ) on DP^l. The problem of RMQ has been studied extensively. For a sequence of n integers, the work[9–12] solve the RMQ problem in $O(1)$ query time using $O(n)$ space and $O(n)$ preprocessing time. The array DP^l can be computed from DP^{l-1} using the following observation.

Proposition 5 *Let s, s' be nodes such that $s = \mathrm{pre}(s')$. If s and s' are siblings, we have $D(s,s') = 1$, otherwise, $D(s,s') = D(\mathrm{parent}(s), \mathrm{parent}(s')) + 1$.*

A faster algorithm to compute ML

Based on Proposition 3–5, we design the algorithm Cholevel to compute $\Delta(1),\ldots,\Delta(L)$ in building $\mathscr{T}$ breadth-firstly. In contrast to procedure CountT, any node is accessed once without redundant visiting. In expanding the nodes in level l, the algorithm computes $\Delta(l)$. In the running, the algorithm only keeps the information of some nodes in level l and $l+1$, the number of which does not exceed $|P|+1$.

In round l, the algorithm generates the nodes in level $l+1$ in the increasing order defined in Definition 3, and computes the array of the depth of the LCA node of each $(l+1)$-level node and its predecessor. We build the RMQ data structure for the array. Using the RMQ structure, we compute the encounter distance between nodes (Proposition 4)

in $O(1)$ time and count the number of nodes of the first l levels of the trie $\mathscr{T}^l$ by Proposition 3. we compute $\Delta(l)$ using the number of nodes of the first l levels of the trie $\mathscr{T}$. By finding out the l with the maximum $\Delta(l)$, we implement the function ML.

Algorithm 4 Procedure Cholevel(n)

$FS \leftarrow \{n\}$; $NS \leftarrow \emptyset$; $l \leftarrow 1$; Ntrie $\leftarrow 1$; NOtrie $\leftarrow 1$; $\Delta \leftarrow 0$

`for` each $pos \in$ PosSet$[n]$ `do`
  `for` each $s \in FS$ `do`
    `for` each $p \in$ PatSet$[s]$ `do`
      `if` edge $(s, p[pos], s')$ does not exist `then`
        Create node s' and edge $(s, p[pos], s')$
        Add s' to NS
        $DP[s'] \leftarrow l$
        `if` s' is the first child of s being created
        `then` $DP[s'] \leftarrow DP[s]$
      Add p to PatSet$[s']$
  Build the RMQ structure for $\{DP[t] \mid t \in NS\}$
  Set all entries of LAST to n
  `for` each $s \in NS$ `do`
    Let a be the label of the edge to s
    NOtrie $\leftarrow$ NOtrie $+ l -$ RMQ$(s,$ LAST$[a])$
    LAST$[a] \leftarrow s$
  `if` Ntrie $+ |NS| -$ NOtrie $> \Delta$ `then`
    $\Delta \leftarrow$ Ntrie $-$ NOtrie
    OptLevl $\leftarrow pos$
  $FS \leftarrow NS$; $NS \leftarrow \emptyset$; $l \leftarrow l + 1$
  Ntrie $\leftarrow$ Ntrie $+ |NS|$; NOtrie $\leftarrow 1$
`return` OptLevl

In the algorithm, l is the number of levels processed, NOtrie is the number of nodes in the first l levels of $\mathscr{T}^l$, Ntrie is that of $\mathscr{T}$, and pos is the position of the pattern set being used. The algorithm uses FS to keep all the nodes in level l that are not expanded and NS to keep the children of nodes in level l. For array DP, $DP[s] =$ depth(LCA$(s,$ pre$(s)))$. For a node s in level l, $D(s,$ pre$(s)) = l - DP[s]$. According to Proposition 5, $DP[s]$ is computed from $DP[$parent$(s)]$. The RMQ structure is built upon the array of DP values of nodes in NS. Let $n_1 < n_2 < \cdots < n_c$ be the set of nodes in NS. Function RMQ(n_i, n_j), $i < j$, returns the minimum value in $DP[n_i], DP[n_{i+1}], \ldots, DP[n_j]$.

Let n' be the newly generated node and pointed by an edge labelled by a. Table LAST has $|\Sigma|$ entries for each character in Σ, where LAST$[a]$ is the node in NS, which is the latest node being created and pointed by an a-edge. By definition, the encounter distance between n' and LAST$[a]$ is $l -$ depth(LCA$(n',$ LAST$[a]))$. The number of nodes in the first l levels of $\mathscr{T}^l$ can be computed from Proposition 3. Cholevel returns the l with the maximum $\Delta(l)$.

Time and space complexities

Theorem 2 *Let P be a set of L-length strings. The level $1 \leqslant l \leqslant L$ such that $\mathscr{T}^l$ has the minimum number of states can be computed in $O(\|P\|)$ time and $O(|P| \log |P|)$ space.*

Proof: The algorithm Cholevel computes ML(P) by finding the maximum among $\Delta(1), \ldots, \Delta(L)$. For the running-time analysis, we divide the time into two components. The first component consists of generating NS from FS and DP values of nodes of NS from that of FS. It takes time proportional to the number of visited nodes. The total number of visited nodes is just the number of nodes in $\mathscr{T}$, which is $O(\|P\|)$.

The second component consists of building the RMQ structure for DP values of nodes in NS and computing RMQ of each node and its precedent in NS. It takes time proportional to the number of visited nodes. The total number of visited nodes is the number of nodes in $\mathscr{T}$. According to Refs. 9–12, RMQ structure is built in linear time and answers the query in $O(1)$ time. The time is $O(\|P\|)$. Thus the total time of the two components is $O(\|P\|)$.

The space used by FS and NS is not greater than $|P|$. The total number of elements in PatSet(n) of nodes in FS, NS, and DP is not greater than $|P| + 1$. Each entry in these arrays, index or value, uses $O(\log |P|)$ bits. The total used space is $O(|P| \log |P|)$ bits. □

EXPERIMENTS

We implemented the algorithms in C to compare with previous work. We conducted experiments on a machine with an Intel i7-4610M Haswell 3.0 GHz processor, 8 GB RAM, running 64 bit CentOS 7. Programs are compiled by GCC. We tested with random pattern sets that are generated using the uniform random distribution. The parameters of pattern sets include the number of patterns $|P|$, the length of each pattern $|p|$, and the size of alphabets $|\Sigma|$.

Table 1 Results of pattern sets of 4000 patterns when each pattern is of length 20. The size of alphabets ranges from 2–128.

$\|\Sigma\|$	2	4	8	16	32	64	128
Trie	36 794	59 346	66 913	70 296	72 320	72 790	75 220
Greedy	30 355	52 976	62 641	68 340	71 176	71 503	74 564
	(18%)	(11%)	(6%)	(3%)	(2%)	(2%)	(0.87%)
O-Trie	28 979	52 744	62 397	67 613	71 058	71 503	74 523
	(21%)	(11%)	(7%)	(4%)	(2%)	(2%)	(0.93%)

Table 2 Results on pattern sets of 4000 patterns when the size of alphabet is 4. The length of patterns ranges from 10–140.

$\|p\|$	10	20	40	60	80	100	140
Trie	19 280	59 271	139 477	219 216	299 516	379 305	539 307
Greedy	17 543	53 016	124 365	196 254	267 023	339 631	483 063
	(9%)	(11%)	(11%)	(10%)	(11%)	(10%)	(10%)
O-Trie	17 313	52 821	124 398	195 950	264 484	340 048	482 880
	(10%)	(11%)	(11%)	(11%)	(12%)	(10%)	(10%)

We compared the number of nodes of tires and the O-Tries generated by the greedy heuristic[7] with our method. The results are given in Table 1–Table 5, where each entry is the number of nodes of a trie or O-Trie. The improvement ratio between an O-Trie T and a trie $\mathscr{T}$ is defined as $1-|T|/|\mathscr{T}|$. We give the improvement ratios in the tables under the size of each O-Trie, each ratio is expressed as a percentage. The results for random patterns are given in Table 1–Table 3.

The results for real patterns are given in Table 4 and Table 5. The DNA sequences are from Homo sapiens chromosome 21. The English texts are from the novel 'Mona Lisa Overdrive'. The length of patterns is 40 symbols. The patterns are consecutive strings picked from texts. In all cases (Table 1–Table 5), our method generates O-Tries that are smaller than that of the greedy method. Table 1 shows that when the size of the pattern sets is fixed, both methods perform better on short alphabets. For large alphabets, a better level of space saving can be achieved on pattern sets of larger size, which is implied by Table 5.

Discussion

The results of Table 1 show the improvement achieved by the new algorithm and the greedy algorithm over the trie. The improvements decline as

Table 3 Results on pattern sets of patterns of length 20 on alphabet of size 16. The number of patterns in each set ranges from 100–12 000.

$\|P\|$	100	500	1000	2000	4000	8000	12 000
Trie	1892	9204	18 127	35 705	70 275	138 368	205 600
Greedy	1837	8925	17 592	34 649	68 320	134 243	199 197
	(3%)	(3%)	(3%)	(3%)	(3%)	(3%)	(3%)
O-Trie	1825	8878	17 515	34 463	67 544	132 523	196 695
	(4%)	(4%)	(3%)	(3%)	(4%)	(4%)	(4%)

Table 4 Results on DNA patterns of length 40.

$\|P\|$	100	500	1000	2000	4000	8000	12 000
Trie	3750	18 034	35 660	70 316	138 537	273 527	406 533
Greedy	3394	15 990	31 638	62 568	123 285	243 505	360 908
	(10%)	(11%)	(11%)	(11%)	(11%)	(11%)	(11%)
O-Trie	3312	15 969	31 594	62 286	123 193	242 922	360 388
	(12%)	(11%)	(11%)	(11%)	(11%)	(11%)	(11%)

Table 5 Results on English patterns of length 40.

$\|P\|$	100	500	1000	2000	4000	8000	12 000
Trie	3889	19 098	37 763	74 594	147 251	290 565	421 917
Greedy	3772	18 519	36 622	72 207	141 974	279 676	405 949
	(3%)	(3%)	(3%)	(3%)	(4%)	(4%)	(4%)
O-Trie	3775	18 414	36 496	71 890	141 376	278 344	403 529
	(3%)	(4%)	(3%)	(4%)	(4%)	(4%)	(4%)

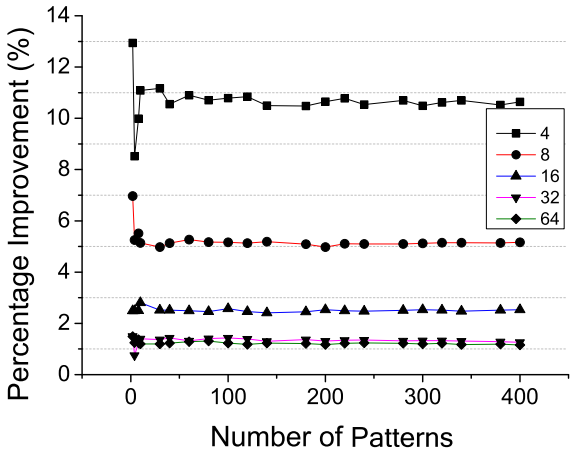

Fig. 4 Results of pattern sets of 2–400 patterns when each pattern is of length 100. The size of alphabets ranges from 4–64.

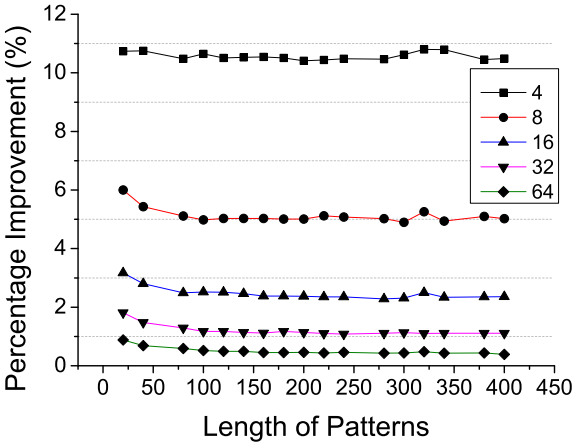

Fig. 5 Results of pattern sets of 200 patterns, the length of patterns ranges from 20–400. The size of alphabets ranges from 4–64.

$|\Sigma| = \sigma$ increases. The results in Table 2–Table 5 show that for a given alphabet the improvement ratios are close to each other, and thus unrelated to the length of patterns or the number of patterns. The ratios of different alphabets are close to ratios

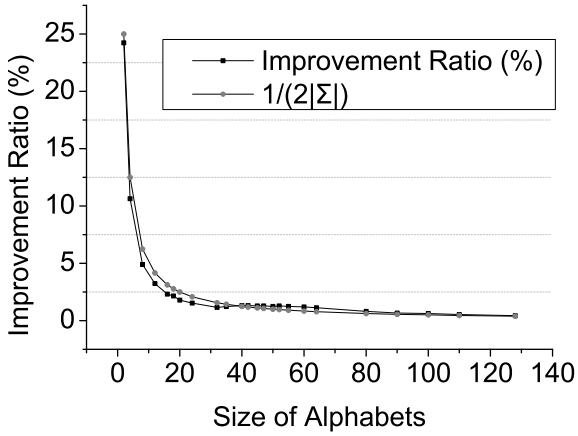

Fig. 6 Results of pattern sets of 200 patterns of length 300. The size of alphabets ranges from 2–128.

of alphabets shown in Table 1. To verify the surmise, we conducted more experiments. We give the results in Fig. 4 and Fig. 5 where each data point is the median of improvement ratios of ten trials. The results imply that the size of alphabet determines the improvement ratio, the ratio equals $1/2|\Sigma|$ in average.

When we select a level with s different letters of the current PatSet, s nodes will be generated. Denote the average number of different letters in a string of $|P|$ symbols as ac($|P|$). In the case of two patterns, the probability of two letters in the same position of the two patterns being the same is $1/\sigma$. The number of nodes of the trie of P is very close to $\|P\|$, for the average length of the common prefix of the two random strings is $1/\sigma + 1/\sigma^2 + \cdots + 1/\sigma^{|p|}$, approximately $1/(\sigma-1)$. The trie has $2|p|+1-1/(\sigma-1)$ nodes in average. But the O-Tries for two patterns are smaller, for the average number of positions with the same symbol of two random strings is $|p|/\sigma$. The O-Trie have $2|p|+1-|p|/\sigma$ nodes in average.

We give an upper bound of ac($|P|$) as follows. Partition the letters in a level to a set of disjoint pairs. In average there are $|P|/2\sigma$ pairs in which the letters

are the same. Thus $\mathrm{ac}(|P|) \leqslant |P| - |P|/2\sigma$. Hence for a node q in an O-Trie, the mean of $A(q)$ is not greater than $|\mathrm{PatSet}(q)|(1-1/2\sigma)$, here $A(q)$ is the number of children of q. In average $|A(q)|/|\mathrm{PatSet}(q)| \leqslant 1 - 1/2\sigma$. Thus the improvement ratio of O-Trie over $\|P\|$ is greater than $1/2\sigma$ with high probability. With the length of patterns grows, $|\mathrm{Trie}(P)|/\|P\|$ is near 1. The average improvement ratio of O-Tries over tries is near $1/2\sigma$ with high probability.

The experiments also show that the improvement ratio is mainly determined by the size of alphabets. We conducted experiments testing a wide range of combinations of $|P|$ and $|p|$. The result in Fig. 6 depicts the ratios of pattern sets on different sizes of alphabets with a fixed $|P|$ and $|p|$ and the curve $y = 1/2|\Sigma|$. In the experiments, the ratio is close to $1/2|\Sigma|$.

CONCLUSIONS

We have presented a new heuristic for full trie minimization based on rearranging the symbols of strings, which is an efficient method to compute the heuristic function. Further research includes a thorough analysis of the quality of solutions of the method. It is interesting to design pattern matching algorithms using O-Tries. Some orders of O-Tries may help to skip more input symbols. We can use these O-Tries to speed up the dictionary lookup or dictionary matching.

Acknowledgements: This work is supported by the fund of development and reform commission of Jilin province No. 2019C053-10 and the fund of the education Department of Jilin province No. JJKH20190162KJ.

REFERENCES

1. Fredkin E (1960) Trie memory. *Commun ACM* **3**, 490–9.
2. Morrison DR (1968) PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J ACM* **15**, 514–34.
3. Aho AV, Corasick MJ (1975) Efficient string matching: An aid to bibliographic search. *Commun ACM* **18**, 333–40.
4. Belazzougui D (2010) Succinct dictionary matching with no slowdown. In: Amir A, Parida L (eds) *Combinatorial Pattern Matching, CPM 2010*, Lecture Notes in Computer Science, **6129**, Springer, Berlin, Heidelberg, pp 88–100.
5. Rotwitt T Jr, deMaine PAD (1971) Storage optimization of tree structured files representing descriptor sets. In: *SIGFIDET '71 Proceedings of the 1971 ACM SIGFIDET (SIGMOD)*, Workshop on Data Description, Access and Control, pp 207–17.
6. Comer D, Sethi R (1977) The complexity of trie index construction. *J ACM* **24**, 428–40.
7. Comer D (1979) Heuristics for trie index minimization. *ACM Trans Database Syst* **4**, 383–95.
8. Comer D (1981) Analysis of a heuristic for trie minimization. *ACM Trans Database Syst* **6**, 513–37.
9. Harel D, Tarjan RE (1984) Fast algorithms for finding nearest common ancestors. *SIAM J Comput* **13**, 338–55.
10. Berkman O, Vishkin U (1993) Recursive star-tree parallel data structure. *SIAM J Comput* **22**, 221–42.
11. Bender MA, Farach-Colton M (2000) The LCA problem revisited. In: Gonnet GH, Viola A (eds) *LATIN 2000: Theoretical Informatics*, Lecture Notes in Computer Science, **1776**, Springer, Berlin, Heidelberg, pp 88–94.
12. Fischer J, Heun V (2006) Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In: Lewenstein M, Valiente G (eds) *Combinatorial Pattern Matching, CPM 2006*, Lecture Notes in Computer Science, **4009**, Springer, Berlin, Heidelberg, pp 36–48.