Suffix Tree Automata in State Space Search

Stefan Edelkamp

Institut für Informatik, Albert-Ludwigs-Universität,
Am Flughafen 17, D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

Abstract. An on–line learning algorithm for pruning state space search is described in this paper. The algorithm is based on a finite state machine which is both created and used in the search. The pruning technique is necessary when memory resources in searching huge problem spaces are restricted. A duplicate sequence is a generating path in the search tree that has a counterpart with smaller weight. The automaton provides the dictionary operations *Insert* and *Delete* for the duplicate sequences found in the search and *Search* for pruning the search tree.
The underlying data structure is a multi suffix tree. Given that the alphabet Σ of state transitions is bounded by a constant an optimal worst case bound of $O(|m|)$ for both insertion and deletion of a duplicate sequence $m \in \Sigma^*$ is achieved. Using the structure as a finite state machine we can incrementally accept a given sequence x in time $O(|x|)$.

1 Introduction and Background

State space problems dealing with a huge problem space are actually described implicitly and thus the transition alphabet Σ is bounded. A sequence $\delta_m(u) = \delta(\ldots\delta(\delta(u, m_1), m_2), \ldots, m_n)$ of transitions applied to a given problem state u can be identified with the string $m \in \Sigma^n$. We wish to prune the search tree at all revisited states but the storage of the search tree in a hash table is impossible. The key idea of Taylor and Korf (1993) is to regard the transition sequences instead of the problem states themselves. In the *learning phase* a breadth-first search is invoked to find m and m' in Σ^* with $\delta_m(u) = \delta_{m'}(u)$ and $w(m) > w(m')$ (w is the transition cost function) s.t. m is called *duplicate* sequence and m' *shortcut* sequence. The conflict between m and m' is found using a hash table. In the *search phase* the set of duplicate strings is applied to prune the search tree. Consider an unconstrained search space in which every transition sequence can be applied. If a node with the generating path $x = \alpha m$, $\alpha \in \Sigma^*$, is reached no further expansion is needed since a sequence $\alpha m'$ has been or will be examined in the search. One duplicate found in the learning phase can thus eliminate hundreds of strings in the search phase. We call this an *off–line* learning algorithm since the automaton is only created but not used in the learning phase. Taylor and Korf propose the algorithm of Aho and Coarsic (1975) to recognize the set of duplicate strings $m \in M$ for which there exists a shortcut m'.

A constrained search space, cancelation of the common prefix in m and m', different initial states, a cycle detection search using a heuristic heading back

to the start node or a successive repetition of some transition sequences raise the quest of a dictionary D for maintaining the different duplicates. Updating the failure function in the AC-algorithm is not very efficient (cf. Meyer (1985)) so we choose another approach. The dictionary in this paper provides *on–line* learning, since it can be used to detect duplicates parallel to the search.

2 The Algorithm and Multi Suffix Trees

The algorithm combines the function of the hash table H and the dictionary automaton D. The input is a state space problem Π and the output is the solution path for Π. A node u in the search tree consists of the problem state description itself, the automaton state q_u, the last character a_u on the generating path p_u, the heuristic estimate $h(u)$ and the weight $g(u)$ of p_u. The procedure *ccp* cancels the common prefix of the input strings and *findSuperString* returns a superstring in D if there is one. Before a state has to be searched in H the automaton state in D is determined. This might be an accepting state which prunes the search tree immediately. Otherwise suppose that a collision v' with a state v is detected in the hash table and the generating path m of v is inserted in D together with the shortcut generating path m' of v'. The duplicate strings in D are kept substring free with respect to each other since a substring m' of a duplicate m which has a shortcut also provides a shortcut for m. Thus before inserting a new duplicate sequence m in the dictionary all superstrings of m are deleted. The algorithm uses a priority queue PQ and extends A^* in the lines marked with an asterisk (*).

procedure *OnLineLearn* (Π)
  $PQ.Insert(s)$; $H.Insert(s)$; // init data structures, s start node
  while $PQ \neq \emptyset$ **do** // if $PQ = \emptyset$ then no solution is found
    $u \leftarrow PQ.Deletemin$ // u not deleted in H for reopening
    for all $v \in expand(u)$ **do** // for all sucessors v of u
      if $goal(v)$ **return** p_v // a goal is found
  (*) $q.v \leftarrow D.Search(p_v)$ // extract new automaton state, $q.v \leftarrow \delta_{p_v}(q_0)$
  (*) Let (m, m') be associated with q_v // extract duplicate/shortcut string
  (*) **if** q_v is accepting **and** $\delta_{m^{-1}m'}(v) = v$ **continue** // $m^{-1}m'$ applicable
    $v' \leftarrow H.Search(v)$ // v' is counterpart of v
    if $v' = nil$ **then** // v is not found in H
      $PQ.Insert(v)$; $H.Insert(v)$ **continue** // insert v in data structures
    if $w(p_v) < w(p_{v'})$ // $g(v) + h(v) < g(v') + h(v')$
  (*) $m' = p_v$; $m = p_{v'}$ // v''s generating path is duplicate
      $PQ.Delete(v')$; $PQ.Insert(v)$; // reopen v
  (*) **else** $m = p_v$; $m' = p_{v'}$ // v's generating path is duplicate
  (*) $ccp(m, m')$ // use least common ancestor of v and v'
  (*) **if** $D.Search(m)$ is accepting **continue** // m has substrings in D
  (*) **while** $m'' \leftarrow$*findSuperString*(m) **do** $D.Delete(m'')$
  (*) $D.Insert((m, m'))$ // new duplicate with shortcut inserted

In this algorithm the automaton only prevents us from searching a state in the hash table. Although hashing of a state is in general not available in constant time it is quite fast compared to the calculation of $Search(p_v)$ in D. On the one hand we will examine how the calculation of $\delta_{p_v}(q_0)$ (q_0 is the initial automaton state) can be done incrementally by analyzing a procedure that can perform $\delta(q_u, a_v)$ in (amortized) constant time. On the other hand notice that under memory restrictions the information that a state has been revisited may not be encountered in the hash table. Many memory restricted search algorithms that have been analyzed in the last decade can be combined with the on-line pruning method. This is a important topic of further research since searching the tree of generating paths can lead to an exponential blow up of time.

A *Patricia tree* is a compact representation of a trie where all nodes with only one successor are merged to their parents. A *suffix tree* is a Patricia tree corresponding to the suffixes of a given string. Although there are $\Theta(|m|^2)$ characters for the $|m|$ suffixes of a string m the suffix tree only needs space of size $O(|m|)$. The substring information stored at each suffix node is simply given by the indices of first and last character. If an internal node v represents $a\alpha$, $a \in \Sigma$ and $\alpha \in \Sigma^*$ then the *suffix link* points to a node representing α which has to exist in the suffix tree. Using these suffix links McCreight (1976) presents and analyses an optimal linear time algorithm to build a suffix tree ST of a given string $m\$$. His approach can be extended naturally to more than one string for example by building the suffix tree of the string $m_1\$_1 \dots m_n\$_n$. Amir et al. (1994) proved that the suffix tree ST for $m_1\$_1 \dots m_n\$_n$ is isomorphic to the compacted trie ST' (cf. Fig. 1) for all suffixes of $m_1\$_1$ up to to all suffixes of $m_n\$_n$. Furthermore, the trees are identical except for the labels of the edges incident to leaves. This fact allows to insert and delete a string into an existing suffix tree. The description and the correctness proof of the linar-time insertion and deletion scheme for multi suffix trees can be found in Amir et al. (1994).

In solving the (100×100) *Maze* with a chance of 35 percent for a square to represent a wall we count the number of pruned nodes in the *OnLineSearch* algorithm. In Fig. 2 we have depicted the first 20 random instances that need more than 1000 expansions. The competitors are two predefined automata as well as two automata learned in a breadth-first-search up to *depth* 5 and 25. In the learning phase of the last two approaches we reconstruct the finite state machine for each increase of search depth. The automata sizes measured in the number of trie nodes are: 10 for the automaton based on predecessor elimination, 14 for the one described in Taylor and Korf (1993), 17 for a learning depth of 5, 43 for a learning depth of 25 and 75 up to 135 for the on-line learning dictionary.

3 Incremental Search using State Transitions

There are two different approaches to find a substring of a given string x in the suffix tree. Amir et al. (1994) determine the longest pattern prefix h of the string stored in the suffix tree that matches x *starting* at position i, $i \in \{1, \dots, |m|\}$. In contrast to this algorithm we fix the longest substring h of the strings stored

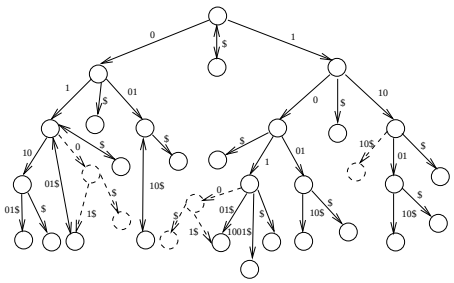

Fig. 1 The multi suffix tree ST' for (1100110), (1011001), (010101) and (11010) to be inserted.

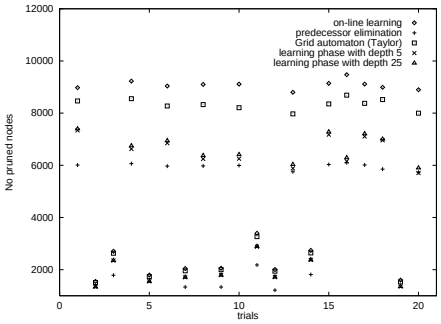

Fig. 2 Experimental results on on-line and off-line pruning the (100×100) *Maze*.

in ST' that matches x *ending* at position i. If the strings m stored in ST' are *substring free*, i.e., no string is a substring of another one, then the only thing to do in both cases is to check if h is maximal ($|h.m| = |h|$). In the general case we have to test the membership of the prefixes of h in M which is called the *dictionary prefix problem (DPP)* introduced by Amir et al. (1994). Our algorithm will be called *incremental* because it doesn't refer to characters x_j with $j < i$. This is crucial, since in the overall *OnLineLearn* algorithm we need an efficient way to determine $\delta_{p_v}(q_0) = \delta(q_u, a_v)$. To find $q_v = \delta(q_u, a_v)$ we search for a new node el and an integer offset at such that a_v corresponds to the transition stored at position $first+at$ of the string stored at el. Thus we will use the suffix links until we have achieved this task. The returned value h_j of $\delta(x_j)$ is the substring corresponding to the path from the root to the new location. Together with the properties of suffix links we can prove the following result inductively.

Theorem 1. *Let $x \in \Sigma^n$ be read from x_1 up to x_{j-1}. The returned value h_j of procedure δ invoked with x_j is the longest suffix $x(i,j)$ of $x(1,j)$ which is also substring of one $m \in M$ stored in the suffix tree ST'. The amortized time complexity for δ is $O(1)$.*

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
2. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *J. Comput. Syst. Sci.*, 49(2):208–222, 1994.
3. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
4. B. Meyer. Incremental string matching. *Inf. Process. Lett.*, 21:219–227, 1985.
5. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 756–761. AAAI Press, 1993.