Subsumption Trees for On-The-Fly Universality Checking of Finite-State Automata

Evan LeGros and Seth Fogarty
Trinity University, San Antonio TX 78212

Abstract. Subsumption is a heuristic optimization that avoids storing certain elements during the execution of an algorithm. Employing subsumption requires subsumptive inserts into a data structure: insertions that remove subsumed elements. We propose a new tree-based data structure that efficiently supports subsumptive insertion.

1 Introduction

Formal verification seeks to prove that computational systems adhere to specifications. One approach is automata-theoretic model-checking, which uses automata to model both specifications and systems [3]. Finite-state automata can be used for safety properties, more complex properties are expressed in Büchi automata. This approach reduces the question of system adherence to a question of automata containment. Checking automata containment requires searching extremely large collections of states. Subsumption is a heuristic technique that can be used to decrease the size of working collections [1]. This technique takes advantage of a subsumption relation between states, and allows an algorithm to store only the maximal elements under the subsumption relation.

Employing subsumption requires subsumptive inserts: inserting a new element into a collection while ensuring that only the maximal elements are stored. The typical data structure for storing collections is a balanced binary search tree (BST). However, subsumptive insertion into a BSTs is inefficient, as the entire tree must be searched. Thus the current state-of-the-art data structure for subsumption is a linked list: subsumptive insertion still requires searching the entire list, but the overhead of balancing the tree is avoided [4].

We present subsumption trees: a tree-based data structure which efficiently supports subsumptive inserts. While subsumption has primarily been employed with Büchi automata [1], we consider NFAs as an initial study. We describe the data structure for the specific case of subset subsumption, perform an experimental analysis, and discuss extensions for other subsumption relations.

2 Background

As a case study, we consider a forward-traversal algorithm for checking NFA universality that employs a frontier. Universality checking focuses on the algorithmically difficult components of containment checking. This algorithm traverses the powerset construction of the input automaton. States of the powerset
construction are sets of states of the input automaton. As detailed in [4], this algorithm admits subset as a subsumption relation.

The frontier algorithm requires storing a large collection of sets. The algorithm both inserts sets into collections and pulls single sets from collections. The data structure representing this collection must support subsumptive insertion. When subsumptively inserting a set $Q$ into a collection, the data structure must first check if it already contains a subset of $Q$. If a subset exists, then $Q$ is not inserted. If not, $Q$ is inserted and every superset of $Q$ is removed.

3 Subsumption Trees

A subsumption tree $T$ is a binary tree where each internal node $N$ is labeled with a single element $N.p\text{ivot}$, and each leaf $L$ is labeled with a set $L.set$. For every internal node $N$, all sets which contain $N.p\text{ivot}$ are stored in the right child $N.\text{right}$, while all sets which do not contain $N.p\text{ivot}$ are stored on the left child $N.\text{left}$. Unlike the binary search invariant, this invariant does not always allow us to limit our search to one child. However, when looking for subsets of a set $Q$ when $N.p\text{ivot} \not\in Q$, there is no need to search $N.\text{right}$. Similarly, when looking for supersets of a set $Q$, if $N.p\text{ivot} \in Q$ then no supersets of $Q$ can exist on $N.\text{left}$. While the entire subsumption tree must still be searched on a worst case, we can often avoid traversal down certain branches.

For example, Figure 1 depicts a subsumption tree. Consider searching the tree for subsets of $Q = \{r, t\}$. Starting at the root, as the pivot $p \not\in Q$, no subset of $Q$ will contain $p$. Therefore we may avoid checking the right child of the root completely. Recursively, the left child of the root has pivot $r$. As $r \in Q$, a subset of $Q$ may occur in both children and we must search both to find the subset $\{r\}$.

Inserting a set $P$ into a subsumption tree $T$ requires three operations. First, search for an existing subset of $P$, which would subsume $P$. Second, remove any supersets of $P$. Finally, insert $P$ into the appropriate location in $T$.

To ensure there are no subsets of $P$, traverse the subsumption tree as follows. For an internal node $T$, if $T.p\text{ivot} \in P$ then subsets may occur in both $T.\text{left}$ and $T.\text{right}$.
and \text{T.right}, and we must recurse down both children. However, if \text{T.pivot} \notin P, subsets of P can only occur in \text{T.left} and we only need to search \text{T.right}. Once we have reached a leaf \text{L}, if \text{L.set} \subseteq P then \text{P} is subsumed by \text{L.set} and does not need to be inserted.

To remove all supersets of \text{P}, we traverse the tree in a dual fashion. For an internal node \text{T}, if \text{N.pivot} \in P. If it is, then because all supersets of \text{P} must contain \text{N.pivot}, we only need to recurse on \text{N.right}. However, if \text{N.pivot} \notin P then we must recurse down both \text{N.left} and \text{N.right}. Once we reach a leaf \text{L}, we check to see if \text{P} \subseteq \text{L.set}. If it is, then we remove \text{L.set} from the tree.

These two operations require traversing overlapping portions of the tree. A subsumptive insert of \text{P} may perform both simultaneously. For an internal node \text{T}, the algorithm checks to see if \text{T.pivot} \in P. If it is, then the insertion algorithm continues on \text{T.left} and we removes supersets from \text{T.right}. If \text{T.pivot} \notin P, we search for subsets of \text{P} in \text{T.left}. If a subset exists, we stop insertion. Otherwise, we recursively insert \text{P} into \text{T.right}. At each point, the algorithm will continue insertion down one of \text{T}'s children, and check for either subsets or supersets in the other child.

Once we reach a leaf \text{L}, we compare \text{P} and \text{L.set}. If \text{L.set} \subseteq \text{P}, then we need not insert \text{P}. If \text{P} \subseteq \text{L.set}, then we replace \text{L.set} with \text{P}. If the two are incomparable, we find a distinguishing element \text{p}: an element contained in one set but not the other. The algorithm then changes \text{L} to be an internal node \text{N} where: \text{N.pivot} is \text{p}; \text{N.right} is a new leaf labeled with the set containing \text{p}; and \text{N.left} a new leaf labeled with the set not containing \text{p}.

4 Experimental Results

We compared subsumption trees and linked lists as an underlying subsumption data structure for NFA universality. We used a corpus of randomly generated Tabakov-Vardi automata. The Tabakov-Vardi model is parametrized by a size \( n \), a transition density \( r \), and acceptance density \( f \) [2]. Varying the parameters determines the probability that the generated automata are universal.

Unfortunately, finite automata universality turns out to be too simple a problem to providing useful running times. For automata large enough to be interesting, the running time was dominated by set generation and the underlying data structure was irrelevant. Thus we measured the number of memory word accesses each data structure performed. We counted the words accessed when comparing a pivot and a set or determining the relation between two sets\(^1\).

From initial experiments we found that the most interesting comparisons can be made when \( r = 2.5 \) and \( f = 0.9 \). To compare the scalability of the two data structures, we keep the \( r \) and \( f \) values constant, while increasing the size of the automata from 20 to 6,000 states. These experiments are not designed to directly compare the accesses made, but to contrast how the number of accesses scales.

\(^1\) As they are irrelevant to the performance of the data structure, comparisons made by the universality algorithm, for instance to check if a set is disjoint from the set of accepting states, were not counted.
with automaton size. We generated 100 automata for each configuration of size, transition density, and acceptance density, and show the median.

![Linked Lists vs Subsumption Trees](image)

**Fig. 2.** The median number of memory accesses versus size for both data structures.

Figure 2 compares subsumption trees and linked lists for universality checking. Subsumption trees perform fewer comparisons, and the gap grows with automaton size, particularly past \( n = 4000 \). This suggests that subsumption trees scale better than linked lists. Unfortunately, our data is insufficient to determine if the growth rates are asymptotically different or only differ by a constant.

## 5 Conclusion

While NFA universality turns out not to be limited by the underlying data structure, the results from our experiments are promising for other applications. However, comparing memory accesses does not account for the space overhead of subsumption trees. It is possible that the added memory usage outweighs the reduced number of comparisons. Further research into more complex applications is needed to fully determine if subsumption-trees improve scalability. Subsumption trees currently support subset subsumption, although they can be trivially extended to superset subsumption. Future work will expand subsumption trees to relations over other structures, such as the ranked sets or arc-labeled graphs used in Büchi universality checking. Finally, we can investigate a way to correct imbalanced subsumption trees.

## References