An Early-Retirement Plan for the States

Bruce W. Watson[1,2] and Richard E. Watson[2]

[1]Department of Computer Science
University of Pretoria
Hillcrest 0083, Pretoria
South Africa

[2]Ribbit Software Systems Inc.
IST Technologies Research Group
Box 24040, 297 Bernard Ave., Kelowna
British Columbia, V1Y 9P9, Canada

e-mail: watson@cs.up.ac.za, {watson, rwatson}@RibbitSoft.com

Abstract. New applications of finite automata, such as NLP and asynchronous circuit simulation, can require automata of millions or even billions of states. All known construction methods (in particular, the interesting reachability-based ones that save memory, such as the subset construction, and simultaneously minimizing constructions, such as Brzozowski's) have intermediate memory usage much larger than the final automaton, thereby restricting the maximum size of the automata which can be built. In this paper, we present a reachability-based optimization which can be used in any one of the construction algorithms to reduce the intermediate memory requirements. The optimization is presented in conjunction with an easily understood (and implemented) canonical automaton construction algorithm.

Key words: finite automata, very large automata, automata construction, memory constraints, reachability algorithms

1 Introduction

Automata (in the form of acceptors or transducers) are now being heavily used in computational linguistics applications, hardware simulation, text indexing and searching applications. In contrast to their traditional use in compilers, these newer applications make use of automata that are several orders of magnitude larger (in terms of both states and transitions, and therefore memory consumption) than previously contemplated. This can lead to memory problems with constructing the automata and also to runtime inefficiencies[1].

Automata can be constructed in a number of ways, however, in this paper we restrict ourselves to building them from regular expressions (REs). Constructing an automaton from an RE proceeds (conceptually) in three phases:

[1]Addressing some of the runtime inefficiencies is the subject of [4].

1. An abstract state automaton is built, in which each state contains additional information (it is an *object* in memory), for example a position within the regular expression. This additional information allows us to determine the out-transitions from the state and whether the state is final or not. The additional information is typically encoded as one of the following:
   - a set of *items* (dots) representing positions in the RE (this is used in all of the item-set constructions [3] and these are used in this paper);
   - a set of *symbol-positions* representing the specific symbols (within the RE) which can be seen next (this is used in the Berry-Sethi, McNaughton-Yamada-Glushkov and Aho-Sethi-Ullman constructions [3]); or
   - an RE representing a *derivative* of the original RE (this is used in Antimirov's and Brzozowski's constructions [3]).
2. From the abstract state automaton, we build a concrete automaton (isomorphic to the abstract state automaton) in which each state is represented only as an integer, with a single bit devoted to indicating whether it is final or not.
3. The abstract state automaton is *retired*, freeing up the memory, leaving only the concrete one.

Of course, a real implementation would not directly implement this conceptual model. As we see in §2, the construction of the final automaton representation could be done incrementally as parts of the underlying abstract state automaton are built. Still, all of the abstract states are kept in memory until the whole of the final automaton is built, after which they are freed. In the case of a deterministic automaton of a million states, the concrete representation may require less than 32MB. Unfortunately, each abstract state can take up a lot more memory as its concrete counterpart, so the intermediate data-structures could have a peak memory requirement of up to 2GB. Clearly, this is beyond even the realistic *virtual* memory capacity of an average processor and operating system. In this paper, we present Ribbit's solution to the problem.

All of the abstract states are usually kept in memory throughout the construction process since a transition (from the state under construction) can go to any one of the other abstract states. In the optimization, we use a reachability relation to determine which abstract states are no longer reachable during the construction phase. Those abstract states may then be removed from memory (retired).

This optimization technique is quite different from the obvious (and well-understood) optimization of removing unreachable states — which yields smaller automata. In our later discussion, that optimization happens to be included (simply by our use of reachability during automata construction), but the new optimization goes much further — reducing the memory used during the construction process.

2 A canonical automata construction method

In this section, we briefly outline a canonical[2] construction method for deterministic automata. The algorithm is essentially a reachability-based version of the traditional

[2] It is a *canonical* construction because it is used as the starting point of a taxonomy in [3].

three-step algorithm outlined in §1. Since the construction will only be used to illustrate the retirement plan, we will not present it formally. See [3] for a more in-depth discussion of various construction algorithms.

In the construction method, each abstract state consists of a set of *items* where an item is a dot[3] placed within the input RE (in much the same way as the LR parsing item appears within grammar production right-hand sides). A relation, called 'dot closure', takes an item set and propagates each item through the RE without jumping over alphabet symbols within the RE. More precisely, it is the reflexive and transitive closure of the following relation:

1. A dot before the empty string (ϵ) RE yields the dot after the empty string RE. Symbolically, $\bullet\epsilon$ is mapped to $\epsilon\bullet$.

2. A dot before an alternation (union) yields dots in front of each branch of the alternation. Likewise, a dot after either branch of an alternation yields a dot after the entire alternation. Symbolically, $\bullet(E \cup F)$ is mapped to $(\bullet E \cup \bullet F)$, $(E \bullet \cup F)$ is mapped to $(E \cup F)\bullet$ and $(E \cup F\bullet)$ is mapped to $(E \cup F)\bullet$.

3. A dot before a concatenation yields a dot in front of the first operand. A dot after the left part of a concatenation yields a dot in front of the second part. A dot after the second part yields a dot after the entire concatenation. Symbolically, $\bullet(EF)$ is mapped to $(\bullet E)F$, $(E\bullet)F$ is mapped to $E(\bullet F)$, and $E(F\bullet)$ is mapped to $(EF)\bullet$.

4. A dot before a Kleene closure yields a dot after the entire Kleene closure and a dot before the (single) operand of the closure. A dot after the operand of a Kleene closure yields a dot before the same operand and a dot after the entire Kleene closure. Symbolically, $\bullet(E^*)$ is mapped to $((\bullet E)^*)\bullet$ and $(E\bullet)^*$ is mapped to $((\bullet E)^*)\bullet$.

(For simplicity in this paper, we omit the other possible RE operators such as intersection.) In the algorithm, we maintain the invariant that all of our abstract states already have the dot closure operation applied to them.

To compute the destination abstract state of a transition from an abstract state on a symbol a, do the following:

1. For every dot in front of an a in the abstract state, place a dot behind the corresponding a in the destination abstract state. Include no other dots in the destination.

2. Perform the dot closure operation on the destination abstract state.

For example, abstract state

$$\bullet(\bullet a \cup (\bullet a)b)$$

has a transition on a to

$$(a \bullet \cup (a\bullet)(\bullet b))\bullet$$

The closure of the union of the transition relation (over all alphabet symbols) with the dot closure relation constitutes a reachability relation over abstract states. This reachability relation plays an important role in the optimization.

[3] We speak of 'dots' and 'items' interchangeably.

To determine the start abstract state, place the dot before the entire RE and perform the dot closure operation. A state is final if a dot appears after the main RE.

During the construction, we use a bijective data-structure (which we call the *namer*), which maps abstract states to concrete ones. We also use either a queue or a stack of abstract states[4], called the *ready pool*. For the actual construction algorithm, we perform the following steps:

1. Create the start abstract state and the corresponding new concrete state; place the abstract state in the ready pool and use the namer to map the start state to it.

2. While the ready pool is not empty: select the next state (call it the *current state*), remove it from the pool and do the following:

   (a) Lookup the corresponding abstract state in the namer. If it is a final state, mark the current state as final too.

   (b) For each alphabet symbol a such that $\bullet a$ appears as a subitem in the abstract state, do the following:

      i. Construct the destination abstract state for the alphabet symbol, using the transitions explained earlier.

      ii. Check if the destination abstract state is in the namer. If not, create a new concrete state and map the destination to it in the namer, while placing the destination abstract state in the ready pool (so that its out-transitions will eventually be constructed).

      iii. Construct a concrete transition on the alphabet symbol from the current concrete state to the concrete state which the destination is mapped to.

3. Delete the abstract states, the namer and the ready pool.

Clearly, the contents of the namer grow to include all of the abstract states mapped to their corresponding concrete states and none of these pairs are removed until the final step. Since this is the source of the memory problem, in the next section we consider how to retire as many as possible of the abstract states on-the-fly in the second step.

3 An early-retirement plan

Some of the abstract states (appearing in the namer) may be unreachable, regardless of the sequence of transitions, from any of the abstract states still in the ready pool. Indeed, the only abstract states which will be needed in the namer are those reachable (directly or indirectly) from an abstract state whose concrete state is in the ready pool. This follows directly from our use of a reachability algorithm.

[4] Using a queue for the ready pool leads to constructing the automaton transition graph breadth-first, while a stack leads to a depth-first construction. The data-structure could just as easily contain concrete states, since we have a bijection between abstract and concrete states.

Conceptually, our solution is to compute the reachability relation (or some approximation containing it) for abstract states. After we have removed a state from the ready pool and built its out-transitions (in step two), we traverse the namer and purge any abstract states (and corresponding concrete ones) which are no longer reachable from an element of the ready pool.

Our implementation maintains the set of abstract states which are reachable from any of the states in the ready pool. Using the canonical construction, we can maintain this set, R, particularly cheaply: it is the ϵ and letter transition closure of the set of all items present in the ready pool abstract states. For example, if the ready pool contains two states $(\bullet a) \cup b$ and $a \cup (b\bullet)$, we have $R = ((\bullet a \bullet) \cup (b\bullet))\bullet$. An abstract state, q, in the namer is reachable from one in the ready pool if q's constituent items are entirely contained in the set of possibly reachable items, R. In our example, the start state would be $\bullet((\bullet a) \cup (\bullet b))$ (the dot closure of $\bullet(a \cup b)$), which is not wholly contained in R, is therefore unreachable by either of the states in the ready pool and can be removed from the namer.

For efficiency, we implement a set of items by numbering all of the possible item positions within the input RE (there are a finite number of them) and using bit-vectors to represent the sets of items. Consequently, the closure and set containment operations can be performed extremely quickly on most computer hardware using bitwise instructions.

4 Observations and performance

Benchmarking data are still being collected as this paper is submitted. Preliminary data, collected while constructing a number of very large automata, shows a reduction of required working memory by a factor of roughly two. There is also a significant running time penalty of up to a factor of ten for constructing the automata, even discounting the obvious memory paging time.

There are other variants of the reachability algorithm which are being explored. One of the most interesting possibilities is to determine the number of in-transitions to each abstract state. Once the in-transitions of the corresponding concrete state have all been built, the abstract state can immediately be purged from the namer. Unfortunately, it is difficult to efficiently count a state's in-transitions before they have all been built. It is not yet clear whether this approach will improve efficiency.

Conclusions and comments

A number of conclusions can be drawn about the approach presented here:

- This technique serves only to minimize the amount of memory consumed during the construction of the automaton. It does not optimize the running time of the automaton, or even the memory consumed by the final automaton. As such, it is only applicable to the massive automata which occur in applications such as NLP or hardware simulation. In that role, the technique is not only very effective, but it is also the *only* known technique available.

- The technique presented in this paper can significantly slow the construction process by having to evaluate the reachability relation on abstract states. This tradeoff is necessary when constructing very large automata.

- This technique has become necessary because even virtual memory has its limits. The current generation of programmers thinks in terms of a 32-bit address space (4GB), which appears boundless. Not only is the virtual address space not large enough for the construction of some automata using the older algorithms, but most systems do not have 4GB of virtual memory available (due to limited physical swap space).

- Algorithms for minimizing deterministic finite automata have a similar memory constraint. During the minimization process, the set of states are grouped into equivalence classes, which will each represent a new state in the minimized automaton. (The equivalence classes are essentially abstract states.) If the input automaton is already nearing the limits of the available memory, any reasonable representation of the equivalence classes is unlikely to fit within the memory. It appears that some of the same techniques could be applied, using an incremental minimization algorithm such as Watson's [3]. In the case of acyclic automata, this would yield an algorithm similar to the one presented in [2].

- This technique minimizes the number of abstract states present in the mapping from abstract states to concrete states. There are many potential speed optimizations which can be applied, such as minimizing the number of times an abstract state is copied. These possibilities have not yet been explored.

Acknowledgements:

We would like to thank Nanette Saes for proofreading this paper.

References

[1] Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1988.

[2] Daciuk, J., B.W. Watson and R.E. Watson. "Incremental Construction of Minimal Acyclic Finite State Automata and Transducers," also submitted to FSMNLP 98.

[3] Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995.

[4] Watson, B.W. "Practical Optimizations for Automata," Second Annual Workshop on Implementing Automata, London, Canada, 1997. Also available from `www.RibbitSoft.com/research/watson/`.