Advances in Automata Implementation Techniques (Abstract)

Bruce W. Watson

Department of Computer Science, University of Pretoria

1 Introduction

In this abstract, I give a brief overview of the latest issues and advances in automata (state machine) implementations. The full material (available from me by email) was originally presented in Potsdam at the FSMNLP Conference in September 2007. Contemporary automata toolkits[1] are applied in areas as diverse as computational linguistics, network security, text indexing, compression, and parallel/concurrent systems. Such implementations typically have three usage scenarios:

- *Compilation* from a regular expression to an automaton.
- *Minimization* of an automaton.
- *Execution* of the automaton on an input string.

The implementations can be improved in the following areas:

1. *Expressive power and succinctness.* Using the more exotic regular operators (such as intersection, negation, etc.) can make a regular expression exponentially more succinct.
2. *Memory consumption.*
3. *Running time.*
4. *Hardware utilization.* Contemporary CPU's include wide bit-wise operators, large memories and potentially FPGA's ('field-programmable gate arrays'), all of which can be reconfigured for highly parallel operations.

These particular opportunities are arising for several reasons: CPU's are not getting much faster (in terms of clock-speed), but they are getting *wider* (e.g. multi-core CPU's, very wide bit-wise operations); main-memory sizes are growing, but cache memory is not, meaning memory-access locality is as important as ever; reconfigurable hardware (e.g. complex multi-core graphics cards from NVidia and ATI, for FPGA's) are becoming the norm in high-end computers. All of these aspects are still underutilized in automata implementations.

Here, I pay particular attention to compilation, though minimization and execution are equally important phases. Depending on the length of the input being processed by an automaton, the execution phase consumes the majority of the time. As such, efficient implementations in hardware are within reach.

[1] Examples of such footnotes include FIRE Engine, Grail, Vaucanson, FST, etc.

2 Compiling (constructing) automata

Automata compilation algorithms fall into two categories:

- *Inductive* constructions build up an automaton based on the structure of the input regular expression: simple automata are used for the atomic regular expressions, and each expression operator gives rise to a constructive operator on automata. As a result, 'compiling' is simply constructing the homomorphic image of the input regular expression. Such constructions have a number of advantages:
  - The automaton's structure reflects the underlying expression's structure.
  - The subautomata (corresponding to subexpressions) can be constructed independently.
  - Shared subexpressions need only be constructed once.

  Similarly, they have some disadvantages:
  - Shared subexpressions do not lead to shared subautomata — often leading to nonminimal automata.
  - Dead (unreachable) states may be constructed for some regular expressions.
  - Exotic regular operators (negation, etc.) are extremely difficult to implement inductively.
- *Reachability* constructions being with a just a few states (usually only one — the start state), and use graph-reachability algorithms to construct the remainder of the automaton. Such constructions often use *derivatives* (also known as *continuations*). The advantages of reachability constructions are:
  - No dead states are constructed.
  - Shared subexpressions are handled only once, leading to automata that are smaller than with inductive constructions.
  - Exotic regular operators are handled extremely easily (indeed, all sixteen Boolean regular operators are handled 'for free' in derivative-based reachability constructions).

  The disadvantages are:
  - A constructed automaton displays virtually no structure which is recognizable from the input regular expression, especially when exotic operators are used.
  - Subautomaton sharing may occur, but is difficult to identify.

Regardless of which construction style is used, there are only a few algorithmic optimization that are applicable:

- *Incremental* algorithms involve doing minimal recomputation of the output when the input changes — usually accomplished by saving some of the intermediate computations. Numerous incremental minimization algorithms are already known, including algorithms which can be halted at any point yielding a partially-minimized automaton ([1]). Incremental inductive construction is easily implemented thanks to the homomorphic nature of the algorithm; by contrast, incremental reachability construction is an important area of future work.

– *Parallel* algorithms are a new imperative, based on trends in hardware. Recent work has yielded parallel automaton minimization algorithms. For example, the algorithm given in [2] checks equivalence of states in separate threads; with enough available threads, this algorithm can minimize in linear time. Parallel constructions are straightforward:
  - Parallel inductive construction can be done with separate threads dealing with the subexpressions. A linear speedup is possible, with enough threads.
  - Parallel reachability construction can be done with separate threads dealing with the out-transitions of each new state in the reachability graph. The performance improvement is somewhat sensitive to the structure of the transition graph (in the worst case, it leads to no speedup).

Acknowledgements: I am particularly thankful to the FSMNLP 2007 organizers who helped me tremendously in forming my thoughts on this topic.

References

1. Watson, B.W., Daciuk, J.: An Efficient Incremental DFA Minimization Algorithm. Natural Language Engineering **9** (2003) 49–64
2. Strauss, T., Kourie, D.G., Watson, B.W.: A Concurrent Specification of Brzozowski's DFA Construction Algorithm. International Journal of Foundations of Computer Science **19**(1) (2008) 125–135