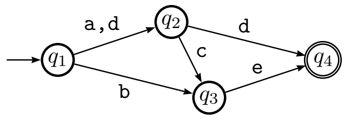
Acyclic Automata with Easy-to-Find Short Regular Expressions[*]

José João Morais, Nelma Moreira, and Rogério Reis

DCC-FC & LIACC, Universidade do Porto,
R. do Campo Alegre 823, 4150 Porto, Portugal
`jjoao@netcabo.pt, {nam, rvr}@ncc.up.pt`

Abstract. Computing short regular expressions equivalent to a given finite automaton is a hard task. We present a class of acyclic automata for which it is easy-to-find a regular expression that has linear size. We call those automata UDR. A UDR automaton is characterized by properties of its underlying digraph. We give a characterisation theorem and an efficient algorithm to determine if an acyclic automaton is UDR, that can be adapted to compute an equivalent short regular expression.

Computing a regular expression from a given finite automaton can be achieved by well-known algorithms based on Kleene's theorem. However the resulting regular expression depends on the order in which the automaton's states are considered in the conversion. In particular, this is the case if the algorithm is based on the *state elimination technique*. Consider the following automaton:



If we remove the state q_2 and then the state q_3, we obtain the regular expression $(\mathtt{a}+\mathtt{d})\mathtt{d}+((\mathtt{a}+\mathtt{d})\mathtt{c}+\mathtt{b})\mathtt{e}$. But if we remove first q_3 and then q_2, we obtain the regular expression $\mathtt{be}+(\mathtt{a}+\mathtt{d})(\mathtt{ce}+\mathtt{d})$. If our goal is to obtain a short regular expression, the order in which we consider the automaton states is of great importance. Moreover, obtaining a minimal regular expression equivalent to a given automaton is PSPACE-complete, and remains NP-complete for acyclic automata. In this work we present a characterisation of acyclic automata for which it is easy to find an order of state removal such that the resulting regular expressions have size linear in the number of the automata transitions. Given a *nondeterministic finite automaton* (NFA) $A=(Q,\Sigma,\delta,q_0,F)$, its *underlying digraph* is $D=(Q,E)$ such that $E=\{(q,q')\mid q,q'\in Q$ and $\exists a\in\Sigma\cup\{\epsilon\}$ such that $(q,a,q')\in\delta\}$. An automaton is *useful* if in its underlying digraph, every state is in a path from the initial state to a final state. An automaton is *acyclic* if its underlying digraph is acyclic. We will consider only useful acyclic automata with one final state. We say that two digraphs are homeomorphic if both can be obtained from the same digraph by a sequence of subdivisions of arcs. Let consider the digraph $\mathsf{R}^{\rightarrow}=(\{q_1,q_2,q_3,q_4\},\{(q_1,q_2),(q_1,q_3),(q_2,q_3),(q_2,q_4),(q_3,q_4)\})$. A useful

[*] Work partially funded by Fundação para a Ciência e Tecnologia and Program POSI.

acyclic NFA with one final state is *UDR* (*Unique for the Distributivity Rule*) if its underlying digraph does not contain a subgraph homeomorphic to $\mathsf{R}^{\rightarrow}$. The underlying digraph is a *UDR digraph.* The NFA represented above is not UDR, as its underlying digraph is $\mathsf{R}^{\rightarrow}$. If an automaton is not UDR, there are at least two states such that the order chosen to eliminate them leads to two different regular expressions, one that results from the application of a distributivity rule to the other. This is not the case for UDR automata. We have:[1]

Theorem 1. *Let $D = (Q, E, i, f)$ be a UDR digraph and $|Q| > 2$. Then D has at least a vertex q such that its indegree and its outdegree are 1 (i.e. $q(1;1)$).*

Theorem 2. *Let $A = (Q, \Sigma, i, \delta, f)$ be a useful acyclic NFA. We can obtain a regular expression equivalent to A using the state elimination algorithm in such way that in each step we remove a state q with $q(1;1)$ if and only if A is UDR. Moreover, that regular expression has size linear in the size of A.*

The following algorithm determines if an acyclic digraph is UDR in $O(n^2 \log n)$. If an automaton is UDR, the algorithm `udrp` can be adapted to compute an equivalent regular expression with size linear in the size of the automaton.

```
udrp {
  %AdjT(v):list of vertices adjacent to v
  %AdjF(v):list of vertices adjacent from v
  %label(u,v):for each edge (u,v), a list of vertices vn with
  %           |AdjF(vn)|> 1 that precedes v in a path from i
  % ← assignment
  % == strictly identical (\== not identical)
  % ?= unifiable, = unification a la Prolog
  for v1 in Q topological order do
     nf ← |AdjF(v1)|
     while nf > 1 do
        max ← max{|label(v,v1)|:v ∈ AdjF(v1)}
        lmax ← {v ∈ AdjF(v1):|label(v,v1)| = max}
        vi ← first(lmax)
        if v in lmax-{vi} and (label(v,v1) \== label(vi,v1))
           and (label(v,v1) ?= label(vi,v1)) then
               vp ← last(label(vi,v1))
               outdegree(vp) ← outdegree(vp) - 1
               label(v,v1) = label(vi,v1)
               if outdegree(vp) == 1 then
                  label(v,v1) ← butlast(label(v,v1))
               nf ← nf - 1
        else return 0 % is not UDR
     if v1 == i then lp ← nil
     else lp = label(first(AdjF(v1)),v1)
     for v2 in AdjT(v1) do
        if outdegree(v1) ≠ 1 then label(v1,v2) ← lp.v1
        else label(v1,v2) = lp
  return 1 % is UDR }
```

[1] For the proofs see http://www.dcc.fc.up.pt/Pubs/TR05/dcc-2005-03.ps.gz.